# On Regions and Linear Types*
# (Preliminary Notes)

David Walker and Kevin Watkins
Carnegie Mellon University

**Abstract**

We explore how two different mechanisms for reasoning about state, linear typing and the type, region and effect discipline, complement one another in the design of a strongly typed functional programming language. The basis for our language is a simple lambda calculus containing *first-class* regions, which are explicitly passed as arguments to functions, returned as results and stored in user-defined data structures. In order to ensure appropriate memory safety properties, we draw upon the literature on linear type systems to help control access to and deallocation of regions. In fact, we use two different interpretations of linear types, one in which multiple-use values are freely copied and discarded and one in which multiple-use values are explicitly reference-counted, and show that both interpretations give rise to interesting invariants for manipulating regions. We also explore new programming paradigms that arise by mixing first-class regions and conventional linear data structures.

## 1 Introduction

One of the classic challenges in programming languages research is to design mechanisms that help programmers reason about the behavior of their code in the presence of imperative operations such as update and deallocation of memory. Over the past 15 years, three techniques for solving this problem have repeatedly found success, particularly for the domain of functional programming languages:

1. Girard's linear logic [13] and related work on linear type systems [19, 1, 37] and syntactic control of interference [29] control sharing and/or the number of uses of important computer resources such as memory. These systems make it possible to deallocate and reuse storage safely.

2. Moggi's computational lambda calculus [23] separates pure values from effectfull computations through the use of monads. This idea forms the basis for adding imperative features to pure functional languages such as Haskell [26].

*This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for System Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. the views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

3. Finally, the type and effect discipline developed by Gifford and Lucassen [12] and refined by Jouvelot, Talpin and Tofte [17, 32, 34] uses a type system and type inference techniques to track accesses to resources. In order to make their analysis feasible, resources are normally grouped into *regions*. Tofte and others [33] use regions and effects to perform all memory management operations in their ML compiler.

More recently, researchers have begun to investigate the relationships between these three fundamental mechanisms. For example, Chen and Hudak [5] have discovered a connection between linear types and monads and Wadler [38] has recently presented a correspondence between monads and effect systems. In this paper, we fill in the third side of the triangle by exploring the synergy between linear types and region, type and effect systems, specifically for the purpose of exploring new techniques in safe, static memory management.

## 1.1 A New Type System for Safe Static Memory Management

The starting point for our development is a simple functional programming language that contains explicit programmer-controlled *regions*. A region is simply an unbounded area of memory or "address space" where values such as function closures, lists or pairs may be allocated. The sole purpose of these regions is to group objects with similar lifetimes. When no object in a region is needed to complete the rest of the computation, the region (and all of the objects contained therein) may be deallocated. Experimental results indicate that this batch-style deallocation can be very efficient in practice, rivaling the best implementations of malloc and free [10].

Unlike previous approaches to region-based memory management, our regions are ordinary programming objects with no special status. In particular, like ordinary objects, references to regions are first-class: they may be stored in data structures and they may be passed explicitly to and from procedures. Since regions have no special status, we may immediately apply known techniques from linear type systems to track the number of uses of regions, just as we can use linear types to track the number of uses of functions or pairs.

This "number of uses" information can be used in a variety of ways [36], but we will concentrate on applications to memory management here. The main idea is that once any programming object has been used for the last time, that object may safely be deallocated and its memory may be reused without affecting the rest of the computation. Furthermore, if using an object within a region implies using the region itself, then in the case a region is used for the last time, both the region and its contents may safely be deallocated. The main contribution of this paper is to explore further the many ways in which regions and linear types can be used together to specify and enforce a wide range of memory management invariants.

Our main technical results arise from two further observations. First, by varying our operational interpretation of linear types, it is possible to develop radically different region-based type systems. More specifically, we contrast a language based on the *use types* of Turner and others [36, 42] where references to multiple-use objects may be freely copied or discarded with Chirimar, Gunter and Riecke's interpretation of intuitionistic linear types [7] where references to multiple-use objects are reference counted. The first interpretation gives rise to a purely static mechanism for ensuring memory safety. We believe the resulting language can be used to encode Tofte and Talpin's original region-based type system. The second interpretation gives rise to a new, more dynamic memory management system. We derived this new system directly from the work of Chirimar *et al.* and our novel core language of regions. Independently, Makholm, Niss and Henglein [21] have developed a related reference-counting system from first principles and they are currently working on type inference techniques for the language. We do not address the issue of type inference in this paper, leaving this topic and other source-language questions to future work.

The second key observation is that because we treat regions as ordinary objects and apply a linear typing discipline uniformly across the entire language, we are free to develop new programming paradigms that mix linear regions with other linear data structures. For example, when we freely mix regions with linear types, we can easily define a linear list of regions, where each region contains some other complex data structure, such as a binary tree. In this case, all the nodes in any particular tree are managed as a unit (and all such nodes may alias one another) whereas each tree is managed independently of the others (but no tree may alias any other – unless the trees are reference counted). No existing type system gives programmers the flexibility to alternate between the coarse-grained memory management used on the nodes of the trees and the fine-grained memory management used on the trees themselves. In traditional linear type systems, aliasing is disallowed and in traditional region-based type systems all objects in the same container data structure must inhabit the same region. Similar limitations arise when programming with mutable data structures: in traditional region-based type systems all objects stored in a mutable data structure must inhabit the same region, even if they have wildly different lifetimes. We have developed related techniques to handle this problem as well.

In the remainder of this article, we present a language of regions and linear types in more detail. Section 2 describes a core calculus including features for allocating and deallocating linear regions, pairs and functions. The linear types in this language are based on *use types* which are in turn derived from Girard's Logic of Unity. Although use types are the correct starting point for our exploration of this topic, they are not quite flexible for our purposes. Therefore we add a construct to our language derived from Wadler's `let`! operator [37]. With this new operator, we can encode Tofte and Talpin's original type system. Section 3 describes the abstract machine that executes programs in our language. It specifies the evaluation relation for the abstract machine and the static semantics for abstract machine states. Section 4 extends the language with reference-counted regions. Once again, Wadler's `let`! comes in handy, as it permits us to define a form of deferred reference counting. We give both static and operational semantics for these extensions. Section 5 extends the language again, this time with lists. Our main goal in this section is to demonstrate how programmers can safely mix linear types, regions, and reference counting in the implementation of complex data structures. Section 6 introduces mutable data structures and shows how they interact with with regions and linear types. Finally, section 7 discusses related work and section 8 concludes.

# 2 The Core Language

Our core language arises from the synthesis of a particular linear type system, the *use types* of Turner *et al.*[36], and a somewhat new variant of Tofte and Talpin's region type system.

## 2.1 The Types

We first explain our choice of linear type system and then proceed to augment the language of types with types for regions.

### 2.1.1 Use Types

There are many subtly different type systems that, to a first approximation, might be called "linear." Although the differences may appear small they can result in significantly different memory management properties. True linear type systems, those type systems pulled along the Curry-Howard isomorphism from Girard's linear logic [13], such as Abramsky's intuitionistic linear type system [1] contain a collection of multiplicatives, including $\tau_1 \multimap \tau_2$, a function type that requires its argument to be used exactly once, and $\tau_1 \otimes \tau_2$, a pair in which each component is used exactly once. In order to retain the expressiveness of an ordinary intuitionistic calculus, a single operator (!) is used to make it possible to duplicate arguments to a function or components of a pair.

Unfortunately, it appears that this type system cannot be given an operational semantics with satisfying memory management properties. Turner and Wadler [35] demonstrate that when working within this type system, one must make a choice: in order to do useful work on an intuitionistic object, one must either make a complete copy of the object in which case the language admits no effective way to *share* objects, or, if one does not make copy of an intuitionistic object each time it is used, then there is no way to guarantee that it is safe to deallocate objects of linear type.

For our application, we must allow sharing; regions can contain an unbounded number of objects so copying them is much too expensive. Type system support for explicit deallocation is equally important. Consequently, a true linear type system will not work here. Instead, we use a slightly different system in which the types of storable objects, such as functions and pairs, have two variants: The "linear" variant[1] classifies objects that are referenced by exactly one pointer and must be used exactly once. The intuitionistic variant classifies objects that can be used an unlimited number of times (including not at all). Since we have two sorts of functions and two sorts of pairs, we do not need the modality "!."

We write $\tau_1 \stackrel{\phi}{\rightarrow} \tau_2$ for generic functions where the qualifier $\phi$ is either $\cdot$, indicating an intuitionistic function that may be used many times, or $\wedge$, indicating a linear function that must be used exactly once.[2] After its single use, the closure containing the function's free variables will be deallocated. Likewise, we write $\tau_1 \stackrel{\phi}{\times} \tau_2$ for generic pair types. A linear pair is deallocated after its components have been projected. Normally, we will suppress the "$\cdot$" annotation above the intuitionistic types. Hence, we write $int \times int$ for an intuitionistic pair of integers.

In our formal work, we will use $()$ as a based type and assume it may be used many times. We could have introduced two variants of $()$ just as we have two variants of the other types, but instead we will assume that there is no cost to using $()$ (an implementation need not allocate it in the store) and therefore no need to define the linear variant. In our examples, we will freely use other base types, such as integers.

For simplicity, we did not include multi-argument functions in our language. However, we can simulate them easily using single-argument functions that accept linear pairs as arguments. Therefore, in our examples, rather than write

$$int \stackrel{\wedge}{\times} int \rightarrow int$$

we will often write

[1] We continue to use the terms "linear" and "intuitionistic" despite loose connections with intuitionistic linear logic.

[2] Notice that *the function* is used once or many times. Unlike type systems based directly on linear logic, these function types say nothing about how often their arguments are used. The number of uses of an argument is determined exclusively by the argument's type.

$$(int, int) \rightarrow int$$

In order to preserve the single-use invariant of linear objects, it is necessary to ensure that intuitionistic objects do not contain linear objects. The term formation rules help maintain this invariant by preventing linear assumptions from being captured in intuitionistic closures. These rules are discussed in more detail in the following section. In addition, we consider intuitionistic pairs with linear component types, such as $(\tau_1 \overset{\wedge}{\times} \tau_2) \times \tau_3$ to be syntactically ill-formed.

### 2.1.2 Regions

Regions are unbounded extents of memory that hold groups of objects. Every region has a unique name, denoted using the meta-variable $\rho$, that can be used to identify the region and the objects it contains. For most purposes, regions are just like any other storage objects. A region with name $\rho$ has a type that may be qualified as either linear or intuitionistic: $\overset{\phi}{rgn}\ (\rho)$. When a region has linear type, it may be deallocated.

When a value is allocated in a region with name $\rho$, the type of the value is tagged with $\rho$. For example, a closure in $\rho$ has type $\tau_1 \overset{\phi}{\rightarrow} \tau_2\ \mathtt{at}\ \rho$ and similarly with pairs. For the sake of uniformity in our formal language we will assume that all stored objects are allocated in some region and therefore that all function and product types are annotated "$\mathtt{at}\,\rho$," for some region $\rho$. However, in our examples we will assume there is some global top-level region named "_" that is never deallocated and we will normally omit the "at _" annotations.

In order to use functions in many contexts, they must be polymorphic with respect to the names of their region arguments[3]. A polymorphic function is considered linear (intuitionistic), if the underlying monomorphic function is linear (intuitionistic). For example, the intuitionistic function `pair`, which allocates a pair of integers in its argument region $\rho$, could be given the type

$$\forall[\rho].(int, rgn(\rho)) \rightarrow (int \times int\ \mathtt{at}\ \rho)$$

Sometimes, we will wish to define functions that return new regions they have allocated. For this purpose, we will use an existential type. The simplest such function takes no argument and returns some new region $\rho$:

$$() \rightarrow \exists\rho.\ \overset{\wedge}{rgn}\ (\rho)$$

Traditional region-based type systems disallow objects of existential type as existentials allow regions to escape the scope of their definition, and, normally, deallocation is linked to the scope of region definition. Our system is similar in that if we want to be able to deallocate intuitionistic regions, we must place some constraints on the way they flow through programs. However, we do not have to restrict the flow of linear regions, we must simply ensure references to linear regions are not duplicated. Therefore, an existential type is permitted to hide the name of a linear region but is not permitted to hide the name of an intuitionistic region. Moreover, existential types are themselves linear, meaning that they may be opened exactly once. We will explain the rules for manipulating existentials in more detail in section 2.2.2.

[3]It is fairly straightforward to make our functions polymorphic over types as well as regions, but for simplicity we omit this degree of freedom in this paper.

| | | | |
|---|---|---|---|
| *type contexts* | $\Delta$ | $::=$ | $\cdot \mid \Delta, \rho$ |
| *qualifiers* | $\phi$ | $::=$ | $\cdot \mid \wedge$ |
| *types* | $\tau$ | $::=$ | $(\,) \mid \overset{\phi}{rgn}\ (\rho) \mid \forall[\Delta].\tau_1 \overset{\phi}{\to} \tau_2 \texttt{ at } \rho \mid \tau_1 \overset{\phi}{\times} \tau_2 \texttt{ at } \rho \mid \exists\rho.\tau$ |
| *intuitionistic types* | $I$ | $::=$ | $(\,) \mid rgn(\rho) \mid \forall[\Delta].\tau_1 \to \tau_2 \texttt{ at } \rho \mid I_1 \times I_2 \texttt{ at } \rho$ |
| *linear types* | $L$ | $::=$ | $\overset{\wedge}{rgn}\ (\rho) \mid \forall[\Delta].\tau_1 \overset{\wedge}{\to} \tau_2 \texttt{ at } \rho \mid \tau_1 \overset{\wedge}{\times} \tau_2 \texttt{ at } \rho \mid \exists\rho.\tau$ |

Figure 1: Syntax: Types

#### 2.1.3 Summary of Type Syntax

Figure 1 summarizes the syntax of the type language. It also documents a subset of the types, ranged over by the meta-variable $I$, that we refer to as "intuitionistic" and a disjoint subset, the linear types, ranged over by the meta-variable $L$. Types (and later terms) are considered equivalent up to renaming of bound variables. We implicitly assume that type contexts, $\Delta$, contain no repeated region names. We concatenate two type contexts using the notation $\Delta\Delta'$. If $\Delta$ and $\Delta'$ have any region names in common then the notation is undefined.

Figure 2 summarizes the well-formedness conditions on types. These conditions are given by a judgment with the form $\Delta \vdash \tau$.

### 2.2 Expressions

Figure 3 presents the expression syntax. As usual, the syntax includes variables as well as introduction and elimination forms for each type of object. We also include two forms of let-expression. The first is standard, but the second is special and will be explained later. The expressions are best explained in conjunction with their typing rules, but before we can proceed with the typing rules we must present a few auxiliary definitions.

#### 2.2.1 Type Checking Contexts

The typing rules for expressions have the form $\Delta; \Gamma \vdash e : \tau$ where $\Gamma$ is a list of assumptions concerning the free type variables in $e$. We assume that no variable is repeated in $\Gamma$. Rather than using the explicit structural rules exchange, contraction and weakening to control reordering, duplication and discarding of assumptions, our type system relies upon a nondeterministic operation ($\bowtie$) that splits the linear assumptions in $\Gamma$ between the contexts $\Gamma_1$ and $\Gamma_2$. The splitting operation is defined below. We will often write $\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3$ as an abbreviation for $\Gamma = \Gamma_1 \bowtie \Gamma'$ and $\Gamma' = \Gamma_2 \bowtie \Gamma_3$.

$$\frac{}{\cdot = \cdot \bowtie \cdot}$$

$$\frac{\Gamma = \Gamma' \bowtie \Gamma''}{\Gamma, x{:}I = (\Gamma', x{:}I) \bowtie (\Gamma'', x{:}I)}$$

$$\frac{\Gamma = \Gamma' \bowtie \Gamma''}{\Gamma, x{:}L = (\Gamma', x{:}L) \bowtie \Gamma''}$$

$$\frac{}{\Delta \vdash (\,)}$$

$$\frac{}{\Delta \vdash \overset{\phi}{rgn}\ (\rho)}\ (\rho \in \Delta)$$

$$\frac{\Delta\Delta' \vdash \tau_1 \quad \Delta\Delta' \vdash \tau_2}{\Delta \vdash \forall[\Delta'].\tau_1 \xrightarrow{\phi} \tau_2\ \texttt{at}\ \rho}\ (\rho \in \Delta)$$

$$\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \overset{\wedge}{\times} \tau_2\ \texttt{at}\ \rho}\ (\rho \in \Delta)$$

$$\frac{\Delta \vdash I_1 \quad \Delta \vdash I_2}{\Delta \vdash I_1 \times I_2\ \texttt{at}\ \rho}\ (\rho \in \Delta)$$

$$\frac{\Delta, \rho \vdash \tau}{\Delta \vdash \exists\rho.\tau}$$

Figure 2: Well-formed Types

$$
\begin{array}{lcll}
\textit{value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\textit{expressions} & e & ::= & x \mid (\,) \mid e_1; e_2 \mid \lambda[\Delta]x{:}\tau \xrightarrow{\phi} e_1\ \texttt{at}\ e_2 \mid e_1[\Delta]\ {}^{\phi}\ e_2\ \texttt{at}\ e_3 \\
 & & & \mid e_1 \overset{\phi}{\times} e_2\ \texttt{at}\ e_3 \mid \texttt{let}\, x \overset{\phi}{\times} y = e_1\ \texttt{at}\ e_2\ \texttt{in}\, e_3 \\
 & & & \mid \texttt{pack}[\rho, e]\ \textit{as}\ \exists\rho.\tau \mid \texttt{unpack}\ \rho, x = e_1\ \texttt{in}\, e_2 \\
 & & & \mid \texttt{alloc}\, e \mid \texttt{free}\ [\rho]e \\
 & & & \mid \texttt{let}\, x = e_1\ \texttt{in}\, e_2 \mid \texttt{let}\,(y =! e_1)x = e_2\ \texttt{in}\, e_3
\end{array}
$$

Figure 3: Syntax: Expressions

$$\frac{\Gamma = \Gamma' \bowtie \Gamma''}{\Gamma, x{:}L = \Gamma' \bowtie (\Gamma'', x{:}L)}$$

We also use the notation $\overset{\phi}{\Gamma}$. When $\phi$ is $\cdot$, then all the types in $\Gamma$ must be intuitionistic. When $\phi$ is $\wedge$ then $\Gamma$ is unrestricted. This notation is used to prevent intuitionistic object from containing linear objects.

### 2.2.2 Typing Rules for Expressions

The typing rules for expressions are derived from consideration of three main invariants:

1. An object of linear type must be "used" exactly once.
2. Any access to a region (*i.e.* allocation within a region or use of an object within a region) must be accompanied by proof that the region is still live.
3. References to intuitionistic regions must not escape the scope in which the intuitionistic region is introduced.

The first invariant is enforced mainly through careful manipulation of the type checking context and the use of the nondeterministic splitting operator. The second invariant is enforced by requiring that the program present a reference to a region every time the region is accessed. We subsequently ensure that there is a reference to a region if and only if the region is still live. The third invariant is enforced by ensuring that intuitionistic regions always appear in the type of the data structure that contains them. This final invariant ensures it is possible to perform a type-based analysis to prevent stored intuitionistic regions from escaping the scope of their definitions.

Figures 4 and 5 present the typing rules for expressions. The first three rules do not involve regions so they are normal natural deduction-style typing rules from the linear lambda calculus. The rule for variables requires that the contexts $\Gamma_1$ and $\Gamma_2$ contain the only intuitionistic variables – we must not let linear variables go unused. The rule for unit is similar. The last of the three is the rule for sequencing. It uses the splitting operator to divide the linear variables between the first and second expressions in the sequence.

The rules for pairs and functions are more complex since we must worry about accessing regions. Pairs are the simpler of the two so we will explain them first. Pairs are allocated using the expression $e_1 \overset{\phi}{\times} e_2$ `at` $e_3$ where $e_1$ and $e_2$ compute values that form the components of the pair. The pair is allocated into the region denoted by expression $e_3$. As in the typing rule for sequencing, the splitting operator divides the linear variables between the three expressions. There are two further details to notice in this rule. First, the third expression should have type $rgn(\rho)$, the type of an intuitionistic region. We do not allow allocation into linear region because we do not want an allocation to be the single use of a linear region. What would be the point of allocating an object in a region that could not be used in the future? It would be impossible to use the object itself.[4] In a moment, we will define an

[4]There are other ways we could organize our language so that access to linear regions is allowed and yet access does not constitute the single use of a linear region. For example, an allocation operation could return a pair of the allocated object and the reference to the region. However, this solution and others we have considered lead to a more complicated operational semantics.

operation that temporarily converts linear regions into intuitionistic regions in order to allow access to linear regions without having to deallocate them.

A second subtle but important aspect to this rule is that it explicitly maintains the invariant that intuitionistic objects (in this case intuitionistic pairs) do not contain linear objects. It does so through the well-formedness judgment on the result type of the expression. If the pair's qualifier $\phi$ is $\cdot$ then this constraint specifies that the component types must not be linear.

The elimination form for pairs, $\texttt{let}\, x_1 \overset{\phi}{\times} x_2 = e_1 \;\texttt{at}\; e_2 \;\texttt{in}\, e_3$, projects of the two components of the pair $e_1$ and binds them to $x_1$ and $x_2$ before continuing with the expression $e_3$. The pair must inhabit the region computed by expression $e_2$. This region is not needed at run time to implement projection function. However, at compile time, it serves as a witness to the continued existence of the region and the pair contained therein. An implementation can optimize away the runtime overhead of passing around these region references using the "ignore region" optimization proposed by Birkedal *et al.*[2], but we will not concern ourselves with such details here. As in the introduction form for pairs, the type of the accessed region is required to be intuitionistic.

#### 2.2.3 Escaping Regions, Function Closures and Existential Packages

Before we can explain the typing rules for functions or existentials, we must clarify the invariants that govern intuitionistic and linear regions. In a typical intuitionistic linear lambda calculus, it is impossible to reclaim the resources used to construct intuitionistic objects, unless one resorts to meta-linguistic tools such as a garbage collector. In our language, it is possible to reclaim intuitionistic functions and pairs if we place them in linear regions. However, if we would like to ensure that all data structures are eventually collected, we must also find some way to collect intuitionistic regions.

In principle, our solution is very similar to the original solution proposed by Tofte and Talpin. The key idea is to prevent usable references to intuitionistic regions from escaping a particular program scope by forcing every data structure that contains a reference to a region to declare the names of these regions in its type. When all region references appear in the types of the data structures that contain them, it is possible to detect escaping references by analyzing the type of the data structure. Moreover, if we can guarantee that no reference to a region escape a particular scope then it will be safe to deallocate the region when control exits that scope – we have constructed the language so that every region access requires a reference to the region as proof that the region is still live.

Unless we are careful, function closures will be able to capture references to intuitionistic regions without revealing these references in the type of the closure. Tofte and Talpin solve this problem by isolating regions in a separate syntactic class from other values and annotating functions with a "latent effect" that includes regions stored in the function closure. Our approach is similar except that we do not define a separate syntactic class of regions. Instead, we require all functions to be closed with respect to intuitionistic regions. Therefore, if a function wants to access a value in an intuitionistic region, that region must be explicitly passed as an argument to the function. Hence, the "latent effect" of the function is represented as part of the type of the function argument. Since regions are ordinary first-class values, this is a natural and elegant design. The closure requirement is enforced by the predicate $\mathit{closed}_\rho(\tau)$ (pronounced "$\tau$ is region-closed with respect to $\rho$"), defined below.

$$
\begin{array}{lcll}
closed_\rho(()) & = & \text{true} & \\
closed_\rho(rgn(\rho)) & = & \text{false} & \\
closed_\rho(rgn(\rho')) & = & \text{true} & (\rho' \neq \rho) \\
closed_\rho(\hat{rgn}\ (\rho')) & = & \text{true} & \\
closed_\rho(\forall[\Delta].\tau_1 \xrightarrow{\phi} \tau_2 \ \texttt{at}\ \rho') & = & \text{true} & \\
closed_\rho(\tau_1 \overset{\phi}{\times} \tau_2 \ \texttt{at}\ \rho') & = & closed_\rho(\tau_1) \wedge closed_\rho(\tau_2) & \\
closed_\rho(\exists\rho'.\tau) & = & closed_\rho(\tau) & (\rho' \neq \rho)
\end{array}
$$

We use the notation $closed(\tau)$ (pronounced "$\tau$ is region-closed") when $closed_\rho(\tau)$ for all regions $\rho$. We lift the definition of region-closed pointwise to contexts $\Gamma$.

Given these definitions we can now interpret the typing rules for functions (see figure 4). As before, the splitting operator partitions the linear assumptions between the context used to check the function body and the computation that generates the region into which the closure is allocated. If the closure is an intuitionistic object then following our rule about no linear objects inside intuitionistic objects, the context used to check the function body can contain no linear variables. Finally, this context must also be region-closed. The rule for function application ensures the region name arguments ($\Delta'$) match the expected region name parameters[5] and that the argument has the expected type. As before, the presence of the region $e_3$ attests to the fact that the region containing the function closure has not yet been deallocated.

Existential types pose similar difficulties and have similar solutions as function closures. In fact, due to Minamide, Morrisett and Harper's encoding of function closures as existential packages [22],existential types may be viewed as the real source of the problem of escaping regions. To ensure intuitionistic regions can be restricted to a particular program scope, we require the type $\tau$ to be closed with respect to intuitionistic regions named $\rho$ when we form an existential of type $\exists\rho.\tau$ using the `pack` expression. Well-formed existentials normally contain linear regions, which are not restricted to any particular scope. The elimination form for existentials is the standard unpack expression.

### 2.2.4 Region Allocation and Deallocation

We have covered the introduction and elimination forms for all of the standard types, only regions remain (see figure 5 for the typing rules). The `alloc` primitive generates a new linear region with a fresh name. Intuitively, it has the type $() \rightarrow \exists\rho.\ \hat{rgn}\ (\rho)$ and formally, we could make it a special constant with this type, but for the sake of convenience[6] we add allocation as a primitive. The `free` operation consumes a linear region. It naturally has the type $\forall[\rho].\ \hat{rgn}\ (\rho) \rightarrow ()$, but again we add it as a primitive to the language.

Intuitionistic regions are introduced and eliminated using a single syntactic form, $\texttt{let}\,(y = !e_1) x = e_2\ \texttt{in}\, e_3$, that was inspired by Wadler's `let!` construct [37]. Operationally, we evaluate $e_1$ expecting a linear region named $\rho$. That region is bound to $y$ and may be used in $e_2$. The result of evaluating $e_2$ is bound to $x$ and both $x$ and $y$ may be used in $e_3$. Since the linear region bound to $y$ is potentially used

[5]In this rule, we rely on alpha-conversion of the bound region names in the function type.

[6]If we made it a constant, we would need to find a region to hold the function. We could use the "_" region, but then we would have to add this region to our system formally. Nevertheless, this is a possibility and it is satisfying to know that the type structure captures region allocation exactly.

multiple times, we must take great care to ensure that there is no way the region can be deallocated too early. In the first expression, $y$ is given the intuitionistic type $rgn(\rho)$ in $e_2$. However, the typing rule constrains the type of $e_2$ to be region-closed with respect to $\rho$ so intuitionistic references to $\rho$ cannot escape from $e_2$ into $e_3$. In $e_3$, $y$ is once again given the linear type $\widehat{rgn}\,(\rho)$. Since no references to this region can flow from $e_2$ to $e_3$, $y$ is the *only* reference to $\rho$ in $e_3$, justifying its linear type. The complete typing rule for this construct can be found in figure 5.

### 2.3 Tofte and Talpin's `letregion`

There are close connections between the `let`! introduced by Wadler and modified here and Tofte and Talpin's `letregion`. Both constructs use a type-based escape analysis to ensure safety. When Wadler first introduced this idea into his linear lambda calculus, he had no notion of a region name, so his analysis was very imprecise. Region names are a form of singleton type, a very precise classifier for regions that makes our modified construct much more effective. In fact, we believe it is possible to use this idea to encode Tofte and Talpin's `letregion` construct in our calculus. Informally, the translation is quite straightforward:

$$
\begin{array}{l}
\texttt{letregion}\,\rho\;\texttt{in}\,e = \\
\quad \texttt{unpack}\;\rho, x = \texttt{alloc}\;(\,)\;\texttt{in} \\
\quad \texttt{let}\,(x = !x)y = e'\;\texttt{in} \\
\quad \texttt{free}\;[\rho]x; y
\end{array}
$$

where the expression $e'$ is the translation of $e$. We conjecture a formal translation from Tofte and Talpin's calculus[7] into our own will be straightforward, but we have not worked through the exercise yet.

## 3 The Abstract Machine

Programs in our language execute on an abstract machine. An abstract machine state includes the region names that may be in use ($\Delta$), a description of the store ($S$), which includes a collection of allocated regions ($R$) and a collection of values inhabiting these regions ($H$) and finally, the expression to be evaluated. The syntax of abstract machine states is presented in Figure 6.

In order to facilitate the proof that our type system is sound, we extend the source language type system to the abstract machine, giving well-formedness conditions for machine states, the store and stored values. The inference rules for the well-formed machine states can be found in figure 8. The main purpose of these rules is to guarantee the following simple facts:

- There is at most one region with a given region name.
- All stored values are well-formed.
- The expression to be executed is well-formed with respect to the current store.

The typing rules for stored values are derived directly from the corresponding source-level expressions. The formal rules may be found in figure 7.

$$\boxed{\Delta; \Gamma \vdash e : \tau}$$

$$\frac{}{\Delta; \dot{\Gamma}_1, x{:}\tau, \dot{\Gamma}_2 \vdash x : \tau}$$

$$\frac{}{\Delta; \dot{\Gamma} \vdash (\,) : (\,)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta; \Gamma_1 \vdash e_1 : (\,) \quad \Delta; \Gamma_2 \vdash e_2 : \tau}{\Delta; \Gamma \vdash e_1; e_2 : \tau}$$

$$\frac{\Gamma = \overset{\phi}{\Gamma_1} \bowtie \Gamma_2 \quad \Delta\Delta' \vdash \tau \quad \Delta, \Delta'; \overset{\phi}{\Gamma_1}, x{:}\tau \vdash e_1 : \tau_1 \quad \Delta, \Gamma_2 \vdash e_2 : \mathit{rgn}(\rho)}{\Delta; \Gamma \vdash \lambda[\Delta'] x{:}\tau \xrightarrow{\phi} e_1 \ \mathtt{at} \ e_2 : \tau \xrightarrow{\phi} \tau_1 \ \mathtt{at} \ \rho} \ (\mathit{closed}(\Gamma_1))$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \qquad \Delta; \Gamma_1 \vdash e_1 : \forall[\Delta'].\tau_1 \xrightarrow{\phi} \tau_2 \ \mathtt{at} \ \rho \\ \Delta; \Gamma_2 \vdash e_2 : \tau_1 \qquad \Delta; \Gamma_3 \vdash e_3 : \mathit{rgn}(\rho)\end{array}}{\Delta; \Gamma \vdash e_1[\Delta'] \ ^{\phi} \ e_2 \ \mathtt{at} \ e_3 : \tau_2} \ (\Delta' \subseteq \Delta)$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \qquad \Delta; \Gamma_1 \vdash e_1 : \tau_1 \\ \Delta; \Gamma_2 \vdash e_2 : \tau_2 \qquad \Delta \vdash \tau_1 \overset{\phi}{\times} \tau_2 \ \mathtt{at} \ \rho \qquad \Delta; \Gamma_3 \vdash e_3 : \mathit{rgn}(\rho)\end{array}}{\Delta; \Gamma \vdash e_1 \overset{\phi}{\times} e_2 \ \mathtt{at} \ e_3 : \tau_1 \overset{\phi}{\times} \tau_2 \ \mathtt{at} \ \rho}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \qquad \Delta; \Gamma_1 \vdash e_1 : \tau_1 \overset{\phi}{\times} \tau_2 \ \mathtt{at} \ \rho \\ \Delta; \Gamma_2 \vdash e_2 : \mathit{rgn}(\rho) \qquad \Delta; \Gamma_3, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_3 : \tau_3\end{array}}{\Delta; \Gamma \vdash \mathtt{let}\, x_1 \overset{\phi}{\times} x_2 = e_1 \ \mathtt{at} \ e_2 \ \mathtt{in}\, e_3 : \tau_3}$$

$$\frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \mathtt{pack}[\rho, e] \ \mathit{as} \ \exists\rho.\tau} \ (\mathit{closed}_\rho(\tau))(\rho \in \Delta)$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta; \Gamma_1 \vdash e_1 : \exists\rho.\tau \quad \Delta, \rho; \Gamma_2, x{:}\tau \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \mathtt{unpack}\ \rho, x = e_1 \ \mathtt{in}\, e_2 : \tau_2} \ (\rho \notin \mathrm{FV}(\tau_2))$$

Figure 4: Well-formed Expressions

$$\frac{\Delta;\Gamma \vdash e : ()}{\Delta;\Gamma \vdash \texttt{alloc}\, e : \exists\rho.\ \widehat{rgn}\ (\rho)}$$

$$\frac{\Delta;\Gamma \vdash e : \widehat{rgn}\ (\rho)}{\Delta;\Gamma \vdash \texttt{free}\,[\rho]e : ()}\ (\rho \in \Delta)$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta;\Gamma_1 \vdash e_1 : \tau_1 \quad \Delta;\Gamma_2, x{:}\tau_1 \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2}$$

$$\frac{\begin{array}{cc}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 & \Delta;\Gamma_1 \vdash e_1 : \widehat{rgn}\ (\rho) \\ \Delta;\Gamma_2, y{:}rgn(\rho) \vdash e_2 : \tau_2 & \Delta;\Gamma_3, y{:}\ \widehat{rgn}\ (\rho), x{:}\tau_2 \vdash e_3 : \tau_3\end{array}}{\Delta;\Gamma \vdash \texttt{let}\,(y = !e_1)x = e_2\, \texttt{in}\, e_3 : \tau_3}\ (closed_\rho(\tau_2))$$

Figure 5: Well-formed Expressions, continued

| | | | |
|---|---|---|---|
| *stored values* | $s$ | $::=$ | $() \mid \langle\lambda[\Delta]x : \tau \xrightarrow{\phi} e\rangle_\rho \mid \langle x_1 \overset{\phi}{\times} x_2\rangle_\rho \mid \texttt{pack}[\rho, x]\ as\ \exists\rho.\tau \mid !x$ |
| *expressions* | $e$ | $::=$ | $\cdots \mid \texttt{let}\,(y = !z, H)x = e_1\, \texttt{in}\, e_2$ |
| *region heaps* | $R$ | $::=$ | $\cdot \mid R, x \mapsto rgn(\rho)$ |
| *value heaps* | $H$ | $::=$ | $\cdot \mid H, x \mapsto s$ |
| *stores* | $S$ | $::=$ | $R, H$ |
| *machine states* | $\Sigma$ | $::=$ | $(\Delta; S; e)$ |

Figure 6: Abstract Machine

$\boxed{\Delta; \Gamma \vdash s : \tau}$

$$\frac{}{\Delta; \dot{\Gamma} \vdash (\,) : (\,)}$$

$$\frac{\Delta\Delta' \vdash \tau \quad \Delta\Delta'; \overset{\phi}{\Gamma}, x{:}\tau \vdash e : \tau'}{\Delta; \overset{\phi}{\Gamma} \vdash \langle \lambda[\Delta'] x{:}\tau \overset{\phi}{\rightarrow} e \rangle_\rho : \forall[\Delta'].\tau \overset{\phi}{\rightarrow} \tau' \ \mathtt{at}\ \rho} \ (\rho \in \Delta)$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta; \Gamma_1 \vdash x_1 : \tau_1 \quad \Delta; \Gamma_2 \vdash x_2 : \tau_2 \quad \Delta \vdash \tau_1 \overset{\phi}{\times} \tau_2 \ \mathtt{at}\ \rho}{\Delta; \Gamma \vdash \langle x_1 \overset{\phi}{\times} x_2 \rangle_\rho : \tau_1 \overset{\phi}{\times} \tau_2 \ \mathtt{at}\ \rho} \ (\rho \in \Delta)$$

$$\frac{\Delta; \Gamma \vdash x : \tau}{\Delta; \Gamma \vdash \mathtt{pack}[\rho, x] \ as \ \exists\rho.\tau : \exists\rho.\tau} \ (\rho \in \Delta)$$

$$\frac{\Delta; \Gamma \vdash x : \widehat{rgn}\ (\rho)}{\Delta; \Gamma \vdash !x : rgn(\rho)}$$

Figure 7: Well-Formed Stored Values

$\boxed{\vdash \Sigma : \tau \text{ program}}$

$$\frac{\Delta \vdash S : \Gamma \text{ store} \quad \Delta; \Gamma \vdash e : \tau}{\vdash (\Delta; S; e) : \tau \text{ program}}$$

$\boxed{\Delta \vdash S : \Gamma \text{ store}}$

$$\frac{\Delta \vdash R : \Gamma \quad \Delta; \Gamma \vdash H : \Gamma'}{\Delta \vdash R, H : \Gamma' \text{ store}}$$

$\boxed{\Delta \vdash R : \Gamma}$

$$\frac{}{\Delta \vdash \cdot : \cdot}$$

$$\frac{\Delta, \Delta' \vdash R : \Gamma}{\Delta, \rho, \Delta' \vdash R, x \mapsto \mathit{rgn}(\rho) : \Gamma, x : \widehat{\mathit{rgn}}\ (\rho)}$$

$\boxed{\Delta; \Gamma \vdash S : \Gamma'}$

$$\frac{}{\Delta; \Gamma \vdash \cdot : \Gamma}$$

$$\frac{\Delta; \Gamma \vdash H : \Gamma' \quad \Gamma' = \Gamma_1 \bowtie \Gamma_2 \quad \Delta; \Gamma_1 \vdash s : \tau}{\Delta; \Gamma \vdash H, x \mapsto s : \Gamma_2, x : \tau}$$

Figure 8: Well-Formed Machine States

In order to facilitate our proof of type soundness, we have also added one run-time expression to the language. The runtime expression $\texttt{let}\,(y = !z, H)x = e_2 \,\texttt{in}\, e_3$ is a natural extension of the programming construct $\texttt{let}\,(y = !e_1)x = e_2 \,\texttt{in}\, e_3$. As indicated by the operational semantics below, once the abstract machine has evaluated the expression $e_1$ and produced an address $z$, it continues with the evaluation of $e_2$. If $e_2$ allocates new objects, these new objects will be stored in the local heap $H$. Once $e_2$ has evaluated to a value, the local heap $H$ is promoted to the global store (or the next enclosing local store). This organization facilitates the proof that references to $y$ do not escape the computation $e_2$. The typing rule for this runtime expression extends the earlier rule for the special let construct to account for the local heap $H$:

$$\frac{\begin{array}{cc}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 & \Delta; \Gamma_1 \vdash y \mapsto !z, H : y{:}\mathit{rgn}(\rho), \Gamma' \\ \Delta; \Gamma_2, y{:}\mathit{rgn}(\rho), \Gamma' \vdash e_2 : \tau_2 & \Delta; \Gamma_3, y{:}\,\hat{\mathit{rgn}}\,(\rho), x{:}\tau_2 \vdash e_3 : \tau_3\end{array}}{\Delta; \Gamma \vdash \texttt{let}\,(y = !z, H)x = e_2 \,\texttt{in}\, e_3 : \tau_3}\ (\mathit{closed}_\rho(\tau_2))$$

## 3.1 Operational Semantics

This subsection defines the operational semantics of the abstract machine. The operational semantics is really quite straightforward for such a powerful language, but we need to define a fair amount of notation to give a concise specification of the various operations on regions as well as linear and intuitionistic objects.

We use the following notation to add a binding to the store. The notation is only defined if $x$ does not already appear in the domain of the store.

$$(R, H), x \mapsto \mathit{rgn}(\rho) = (R, x \mapsto \mathit{rgn}(\rho), H)$$

$$(R, H), x \mapsto s = (R, H, x \mapsto s)$$

We extend this notation in the natural way to allow sequences of bindings to be added to the store as in $S, H$ which extends $S$ with $H$ or $S, S'$ which extends $S$ with $S'$.

The operation $S(x)$ selects the object at address $x$ from store $S$. If $x$ does not appear in the store then the operation is undefined. The operation is defined below.

$$\begin{array}{lcll} S, x \mapsto \mathit{rgn}(\rho), S'(x) & = & \mathit{rgn}(\rho) & \\ S, x \mapsto s, S'(x) & = & s & (s \neq !y) \\ S, x \mapsto !y, S'(x) & = & S(y) & \end{array}$$

When an intuitionistic object is used, it remains in the store. However, when a linear object is used, it is deallocated. The following two operations ($\dot{-}$ for intuitionistic objects and $\hat{-}$ for linear objects) implement this behavior.

$$S \,\dot{-}\, x = S$$

[7]If we add universal polymorphism over types to our language, we believe we can encode the entire language. Without universal polymorphism over types in our language, we cannot encode the polymorphism over types or effects in the Tofte and Talpin calculus, but all the other constructs appear straightforward.

$$
\begin{array}{lcl}
S, x \mapsto rgn(\rho), S' \stackrel{\wedge}{-} x & = & S, S' \\
S, x \mapsto s, S' \stackrel{\wedge}{-} x & = & S, S'
\end{array}
$$

Finally, before we can define the operational semantics, we need to define the evaluation contexts. This definition is mostly standard. Notice, however, that there is no evaluation context of the form $\texttt{let}\,(y = !z, H)x = E\,\texttt{in}\,e$. The operational semantics makes use of this fact.

$$
\begin{array}{lcl}
E & ::= & [\,] \mid E; e \mid x; E \mid \lambda[\Delta]x{:}\tau \xrightarrow{\phi} e\;\texttt{at}\;E \mid E[\Delta]\;{}^{\phi}\;e_1\;\texttt{at}\;e_2 \mid x[\Delta]\;{}^{\phi}\;E\;\texttt{at}\;e \mid x_1[\Delta]\;{}^{\phi}\;x_2\;\texttt{at}\;E \\
 & & \mid E \stackrel{\phi}{\times} e_1\;\texttt{at}\;e_2 \mid x \stackrel{\phi}{\times} E\;\texttt{at}\;e \mid x_1 \stackrel{\phi}{\times} x_2\;\texttt{at}\;E \\
 & & \mid \texttt{let}\,x \stackrel{\phi}{\times} y = E\;\texttt{at}\;e_1\;\texttt{in}\,e_2 \mid \texttt{let}\,x \stackrel{\phi}{\times} y = z\;\texttt{at}\;E\;\texttt{in}\,e \\
 & & \mid \texttt{pack}[\rho, E]\;as\;\exists\rho.\tau \mid \texttt{unpack}\;\rho, x = E\;\texttt{in}\,e \mid \texttt{alloc}\;E \mid \texttt{free}\;[\rho]E \\
 & & \mid \texttt{let}\,x = E\;\texttt{in}\,e \mid \texttt{let}\,(y = !E)x = e_1\;\texttt{in}\,e_2
\end{array}
$$

The operational semantics for the language is given by a mapping from machine states to machine states. This mapping is presented in figure 9. In general, an introduction form is evaluated by choosing a fresh address[8] and extending the store with the appropriate value allocated at that address. When allocating in a region, the operational semantics verifies that there exists a live region with that name. An elimination form such as a projection or function call is evaluated by looking the pair or function up in the store, ensuring that the region inhabited by the pair or function is still alive and finally taking the appropriate action. The only unusual evaluation rule is the one for the second `let` form. Evaluation under one of these `let` forms has the effect of adding the local heap to the global store for evaluation of the subterm.

### 3.2 Properties of the Core Language

We intend to prove a type soundness result for our language. Recent research [39, 14, 3] indicates that we should be to obtain our result using syntactic techniques. In fact, we have intentionally organized our operational semantics so that the hierarchical nature of the region store is implicit, following the insights of Calcagno, Helsen and Thiemann [14, 3] and we believe this decision will make the proof quite straightforward. We are currently investigating the possibility of formalizing the result in a linear logical framework [4].

## 4 Reference Counting

So far, our implementation of the intuitionistic linear type system allows objects of intuitionistic type to be shared (*i.e.* there may be many pointers to these objects). Objects of linear type, on the other hand, are always unshared and therefore they may be collected immediately after they are used. These decisions lead to a completely static memory management discipline. Unfortunately, the lack of aliasing for reusable (linear) objects has its disadvantages: it is necessary to copy linear objects in some situations to preserve the single pointer invariant and this copying can lead to unnecessary memory use.

[8] By fresh address, we mean an address that does not already appear in the domain of the store. The freshness constraint is implicit in the formal rules.

$\boxed{\Sigma \longrightarrow \Sigma'}$

$$(\Delta; S; E[()]) \longrightarrow (\Delta; S, x \mapsto (); E[x])$$

$$(\Delta; S; E[x; e]) \longrightarrow (\Delta; S; E[e]) \qquad \text{if:} \qquad S(x) = ()$$

$$(\Delta; S; E[\lambda[\Delta']x{:}\tau \xrightarrow{\phi} e \; \mathtt{at} \; y]) \longrightarrow (\Delta; S, z \mapsto \langle \lambda[\Delta']x{:}\tau \xrightarrow{\phi} e \rangle_\rho; E[z]) \qquad \text{if:} \qquad S(y) = \mathit{rgn}(\rho)$$

$$(\Delta; S; E[x_1[\Delta'] \;\; ^{\phi} \;\; x_2 \; \mathtt{at} \; x_3]) \longrightarrow (\Delta; S \overset{\phi}{-} x_1; E[e]) \qquad \text{if:} \qquad \begin{array}{l} S(x_1) = \lambda[\Delta']x_2{:}\tau \xrightarrow{\phi} e \\ S(x_3) = \mathit{rgn}(\rho) \end{array}$$

$$(\Delta; S; E[x_1 \overset{\phi}{\times} x_2 \; \mathtt{at} \; x_3]) \longrightarrow (\Delta; S, y \mapsto \langle x_1 \overset{\phi}{\times} x_2 \rangle_\rho; E[y]) \qquad \text{if:} \qquad S(x_3) = \mathit{rgn}(\rho)$$

$$(\Delta; S; E[\mathtt{let} \, x_1 \overset{\phi}{\times} x_2 = y \; \mathtt{at} \; x_3 \; \mathtt{in} \, e]) \longrightarrow (\Delta; S \overset{\phi}{-} y; E[e]) \qquad \text{if:} \qquad \begin{array}{l} S(y) = \langle x_1 \overset{\phi}{\times} x_2 \rangle_\rho \\ S(x_3) = \mathit{rgn}(\rho) \end{array}$$

$$(\Delta; S; E[\mathtt{pack}[\rho, x] \; \mathit{as} \; \exists \rho.\tau]) \longrightarrow (\Delta; S, y \mapsto \mathtt{pack}[\rho, x] \; \mathit{as} \; \exists \rho.\tau; E[y])$$

$$(\Delta; S; E[\mathtt{unpack} \; \rho, y = x \; \mathtt{in} \, e]) \longrightarrow (\Delta; S \overset{\phi}{-} x; E[e]) \qquad \text{if:} \qquad S(x) = \mathtt{pack}[\rho, y] \; \mathit{as} \; \exists \rho.\tau$$

$$(\Delta; S; E[\mathtt{alloc} \; x]) \longrightarrow (\Delta, \rho; S, y \mapsto \mathit{rgn}(\rho); E[\mathtt{pack}[\rho, y] \; \mathit{as} \; \exists \rho. \; \overset{\wedge}{\mathit{rgn}} \; (\rho)]) \qquad \text{if:} \qquad S(x) = ()$$

$$(\Delta; S; E[\mathtt{free} \; [\rho]x]) \longrightarrow (\Delta; S \overset{\wedge}{-} x; E[()]) \qquad \text{if:} \qquad S(x) = \mathit{rgn}(\rho)$$

$$(\Delta; S; E[\mathtt{let} \, x = x \; \mathtt{in} \, e]) \longrightarrow (\Delta; S; E[e])$$

$$\frac{(\Delta; S, y \mapsto !z, H; E_2[e_1]) \longrightarrow (\Delta'; S', y \mapsto !z, H'; e_1')}{(\Delta; S; E_1[\mathtt{let} \, (y = !z, H)x = E_2[e_1] \; \mathtt{in} \, e_2]) \longrightarrow (\Delta'; S'; E_1[\mathtt{let} \, (y = !z, H')x = e_1' \; \mathtt{in} \, e_2])}$$

$$(\Delta; S; E[\mathtt{let} \, (y = !y, H)x = x \; \mathtt{in} \, e]) \longrightarrow (\Delta; S, H; E[e])$$

Figure 9: Operational Semantics

$$
\begin{array}{llll}
\textit{types} & \tau & ::= & \cdots \mid \overset{\#}{rgn}\ (\rho) \\
\textit{linear types} & L & ::= & \cdots \mid \overset{\#}{rgn}\ (\rho) \\
\textit{expressions} & e & ::= & \cdots \mid \overset{\#}{\texttt{alloc}}\ e \mid \texttt{let}\ x, y = \texttt{inc}[\rho]e\ \texttt{in}\ e' \mid \texttt{dec}[\rho]e \\
\textit{contexts} & E & ::= & \cdots \mid \overset{\#}{\texttt{alloc}}\ E \mid \texttt{let}\ x, y = \texttt{inc}[\rho]E\ \texttt{in}\ e \mid \texttt{dec}[\rho]E \\
\textit{regions} & R & ::= & \cdots \mid R, x \mapsto \langle n, rgn(\rho)\rangle, x_1 \mapsto \#x, \ldots, x_n \mapsto \#x
\end{array}
$$

Figure 10: Syntax for Reference Counting Constructs

Alternatively, it is necessary to convert linear regions into intuitionistic regions for significant portions of a program and to delay region deallocation beyond the point at which a region is semantically dead.

Chirimar, Gunter and Riecke [7] proposed an entirely different model of linear logic. They used reference counting to keep track of the number of pointers to an object. The linear type system ensures that reference counts are maintained accurately. Reference counts add a dynamic component to the memory management system that complements a purely static approach. Rather than having to copy objects or convert linear regions into intuitionistic regions, it is possible to manipulate reference counts.

In general, one can augment the calculus of previous sections with a third qualifier (#) and manage regions, pairs, closures or other heap-allocated objects by reference counting.[9] Here, for simplicity, we concentrate exclusively on reference-counted regions. The new language constructs are presented in Figure 10. The new type of reference-counted regions is considered linear – assumptions with this type may not be implicitly duplicated or discarded. The reference counts are explicitly duplicated using the `inc` function and explicitly decremented and freed when the count reaches zero using the `dec` function. Figure 11 defines additional rules for the well-formed types and expressions.

In the previous sections, the $!e$ operator made it possible to temporarily treat linear regions as intuitionistic ones to avoid costly copying. Here, we can use the same construct to temporarily increase reference counts without the runtime cost of having to do the actual increment operation. In other words, we use the more conventional interpretation of intuitionistic types in conjunction with reference counting to obtain a form of deferred reference counting. This trick also conveniently allows us to reuse all the allocation and access rules for pairs and closures for both reference-counted regions and other sorts of regions.

## 4.1 Operational Semantics

The operational semantics for the reference counting expressions is presented in the figure 12. Notice that the semantics for increment and decrement operations relies upon two auxiliary functions. These auxiliary functions are undefined if the store does not have the proper form.

The other operations in the language remain essentially unchanged. In order to allow access to reference counted regions, we need only extend the store access function slightly:

[9] One does have to be careful to ensure that reference-counted objects contain intuitionistic objects only, not linear objects or other reference counted objects. This may be accomplished using identical techniques to those of previous sections which ensure that only intuitionistic objects appear inside of intuitionistic objects. Alternatively, one could allow linear or reference counted objects inside other reference counted objects at the expense of a more complex run-time system that recursively deallocates subcomponents of a reference-counted data structure.

$\boxed{\Delta \vdash \tau}$

$$\frac{}{\Delta \vdash \overset{\#}{rgn}\,(\rho)}\ (\rho \in \Delta)$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Delta; \Gamma \vdash e : (\,)}{\Delta; \Gamma \vdash \overset{\#}{\texttt{alloc}}\, e : \exists \rho.\ \overset{\#}{rgn}\,(\rho)}$$

$$\frac{\Delta; \Gamma \vdash e : \overset{\#}{rgn}\,(\rho) \quad \Delta; \Gamma, x : \overset{\#}{rgn}\,(\rho), y : \overset{\#}{rgn}\,(\rho) \vdash e' : \tau'}{\Delta; \Gamma \vdash \texttt{let}\, x, y = \texttt{inc}[\rho] e\, \texttt{in}\, e' : \tau'}\ (\rho \in \Delta)$$

$$\frac{\Delta; \Gamma \vdash e : \overset{\#}{rgn}\,(\rho)}{\Delta; \Gamma \vdash \texttt{dec}[\rho] e : (\,)}\ (\rho \in \Delta)$$

$$\frac{\begin{array}{cc} \Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 & \Delta; \Gamma_1 \vdash e_1 : \overset{\#}{rgn}\,(\rho) \\ \Delta; \Gamma_2, y{:}rgn(\rho) \vdash e_2 : \tau_2 & \Delta; \Gamma_3, y{:}\ \overset{\#}{rgn}\,(\rho), x{:}\tau_2 \vdash e_3 : \tau_3 \end{array}}{\Delta; \Gamma \vdash \texttt{let}\, (y = !e_1) x = e_2\, \texttt{in}\, e_3 : \tau_3}\ (closed_\rho(\tau_2))$$

$$\frac{\Delta \vdash R : \Gamma}{\Delta \vdash R, x \mapsto \langle n, rgn(\rho) \rangle, x_1 \mapsto \#x, \ldots, x_n \mapsto \#x : \Gamma, x_1{:}\ \overset{\#}{rgn}\,(\rho), \ldots, x_n{:}\ \overset{\#}{rgn}\,(\rho)}$$

Figure 11: Well-Formed Reference Counting Constructs

$$(\Delta; S; E[\overset{\#}{\texttt{alloc}}\, y]) \longrightarrow (\Delta, \rho; S, x \mapsto \langle 1, \mathit{rgn}(\rho)\rangle, x_1 \mapsto \#x; E[\texttt{pack}[\rho, x_1]\ \mathit{as}\ \exists\rho.\ \overset{\#}{\mathit{rgn}}\ (\rho)])$$

$$(\Delta; S; E[\texttt{let}\, x, y = \texttt{inc}[\rho]x\ \texttt{in}\, e]) \longrightarrow (\Delta; \mathit{inc}(S; x; y); E[e])$$

$$\begin{array}{ll} \text{where:} & \mathit{inc}(S, x \mapsto \langle n, \mathit{rgn}(\rho)\rangle, x_1 \mapsto \#x, \ldots, x_n \mapsto \#x, S'; x_i; x_{n+1}) \\ & \quad = S, x \mapsto \langle n+1, \mathit{rgn}(\rho)\rangle, x_1 \mapsto \#x, \ldots, x_{n+1} \mapsto \#x, S' \end{array}$$

$$(\Delta; S; E[\texttt{dec}[\rho]x]) \longrightarrow (\Delta; \mathit{dec}(S; x); E[(\,)])$$

$$\begin{array}{ll} \text{where:} & \mathit{dec}(S, x \mapsto \langle n, \mathit{rgn}(\rho)\rangle, x_1 \mapsto \#x, \ldots, x_n \mapsto \#x, S'; x_i) \\ & \quad = S, x \mapsto \langle n-1, \mathit{rgn}(\rho)\rangle, x_1 \mapsto \#x, \ldots, x_{i-1} \mapsto \#x, x_{i+1} \mapsto \#x, \ldots, x_n \mapsto \#x \\ & \mathit{dec}(S, x \mapsto \langle 1, \mathit{rgn}(\rho)\rangle, x_1 \mapsto \#x, S'; x_1) \\ & \quad = S, S' \end{array}$$

Figure 12: Reference Counting Constructs: Operational Semantics

$$(S, x \mapsto \langle n, \mathit{rgn}(\rho)\rangle, x_1 \mapsto \#x, \ldots, x_n \mapsto \#x, S')(x_i) = \mathit{rgn}(\rho)$$

We must also extend the definition of region-closed:

$$\mathit{closed}_\rho(\overset{\#}{\mathit{rgn}}\ (\rho')) = \text{true}$$

# 5 Container Data Structures

One of the primary weaknesses of region based memory management on its own is that all container data structures are *homogeneous* with respect to the regions that their elements inhabit. In other words, all elements of a list, tree, or other recursive datatype are required to inhabit the same region. Consequently, all elements of any given list or tree must have the same lifetime. For long-lived containers for which both insertions and deletions are common, this strategy can incur quite a cost as none of the objects that are removed from the collection can be deallocated until the entire collection is deallocated.

Tofte and others have developed clever programming techniques to avoid this problem in many cases. In essence, they manually mimic the action of the copying garbage collector. More specifically, they periodically copy the container data structure from one region to another. After the copy, they cease to use the data in the old region so it may safely be deallocated. Dan Wang and Andrew Appel [41] have exploited similar ideas to write a complete copying garbage collector in a type safe language that uses the regions.

Although copying is highly effective solution in many situations, it is not without its own overhead. If the container data structure is large, the extra space and time required to copy the live data from one region to another may not be acceptable. In our language, programmers have many more choices. On the one hand, they may employ the copying solution that we have just discussed. On the other

$$
\begin{array}{llll}
\textit{types} & \tau & ::= & \cdots \mid \tau \overset{\phi}{\textit{list}} \; \texttt{at} \, \rho \\
\textit{expressions} & e & ::= & \cdots \mid \overset{\phi}{[\,]}_{\tau} \; \texttt{at} \; e \mid \overset{\phi}{\texttt{cons}} \, (e_1, e_2) \; \texttt{at} \; e_3 \mid \overset{\phi}{\texttt{case}} \; e_1 \; \texttt{at} \; e_2 \; ([\,] \Rightarrow e_3 \mid (x, y) \Rightarrow e_4)
\end{array}
$$

Figure 13: Lists

hand, programmers can mix linear types with regions to solve this problem in new ways. In particular, programmers can define *heterogeneous* data structures. In other words, containers may hold elements stored in different regions and therefore individual objects may be deallocated independently of the other objects in the container.

Figure 13 presents the syntax of an extension to our language with lists. Like other data structures such as pairs and closures, intuitionistic lists are constrained so that they do not contain linear objects. Figure 14 presents the well-formedness rules for list types.

There are three lists expressions. The $\overset{\phi}{[\,]}_{\tau}$ $\texttt{at}$ $e$ expression introduces an empty list with type $\tau$ in the region designated by $e$ and $\overset{\phi}{\texttt{cons}}\,(e_1, e_2)\,\texttt{at}\; e_3$ prepends $e_1$ to the list $e_2$, in the region designated by $e_3$. The case construct follows the first branch if $e$ is the empty list and the second branch otherwise. The typing rules for these constructs extend the typing rules for the core language specified in previous sections in the natural way. Figure 14 also presents the well-formedness rules for list expressions.

These typing rules (in particular, the rule for cons) require that the spine of the list inhabits a single region.[10] However, the elements of the list may inhabit different regions. For example, a linear list of lists might be given the following type.

$$\exists \rho.\; \overset{\wedge}{rgn}\,(\rho) \overset{\wedge}{\times} ((\,)\; \overset{\cdot}{\textit{list}}\; \texttt{at}\, \rho)\; \overset{\wedge}{\textit{list}}$$

In this case, each element of the list is an existential package containing a pair of a reference to a region and a list inhabiting that region. Each of these inner lists may be processed and deallocated independently of any of the other inner lists. However, since the regions are linear they do not alias one other. If a programmer requires a data structure that involves aliasing between the lists then a reference counting solution could be used:

$$\exists \rho.\; \overset{\#}{rgn}\,(\rho) \overset{\wedge}{\times} ((\,)\; \overset{\cdot}{\textit{list}}\; \texttt{at}\, \rho)\; \overset{\wedge}{\textit{list}}$$

The dynamic nature of the reference counts makes it unnecessary to copy the elements of the outer list.

## 6 Mutable Data Structures

Mutable data structures pose many of the same problems for traditional region-based memory management schemes as containers like lists do. Any object that is stored in a reference must live in the same

[10] If the language revealed the structure of the implementation of lists in terms of sum types and recursive types, then we could choose how to implement the spine – either as a homogenous or a heterogeneous data structure.

$\boxed{\Delta \vdash \tau}$

$$\frac{\Delta \vdash \tau}{\Delta \vdash \tau \ \overset{\wedge}{\mathit{list}} \ \mathtt{at}\ \rho} \ (\rho \in \Delta)$$

$$\frac{\Delta \vdash I}{\Delta \vdash I \ \overset{\cdot}{\mathit{list}} \ \mathtt{at}\ \rho} \ (\rho \in \Delta)$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Delta; \Gamma \vdash e : \mathit{rgn}(\rho) \quad \Delta \vdash \tau \ \overset{\phi}{\mathit{list}} \ \mathtt{at}\ \rho}{\Delta; \Gamma \vdash \overset{\phi}{[\,]}_{\tau} \ \mathtt{at}\ e : \tau \ \overset{\phi}{\mathit{list}} \ \mathtt{at}\ \rho}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \quad \Delta; \Gamma_1 \vdash e_1 : \tau \quad \Delta; \Gamma_2 \vdash e_2 : \tau \ \overset{\phi}{\mathit{list}} \ \mathtt{at}\ \rho \quad \Delta; \Gamma_3 \vdash e_3 : \mathit{rgn}(\rho)}{\Delta; \Gamma \vdash \overset{\phi}{\mathtt{cons}} \ (e_1, e_2) \ \mathtt{at}\ e_3 : \tau \ \overset{\phi}{\mathit{list}} \ \mathtt{at}\ \rho}$$

$$\frac{\begin{array}{c} \Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \bowtie \Gamma_4 \\ \Delta; \Gamma_1 \vdash e_1 : \tau' \ \overset{\phi}{\mathit{list}} \ \mathtt{at}\ \rho \qquad \Delta; \Gamma_2 \vdash e_2 : \overset{\phi}{\mathit{rgn}} \ (\rho) \\ \Delta; \Gamma_3 \vdash e_3 : \tau \qquad \Delta; \Gamma_4, x{:}\tau', y{:}\tau' \ \overset{\phi}{\mathit{list}} \vdash e_4 : \tau \end{array}}{\Delta; \Gamma \vdash \overset{\phi}{\mathtt{case}} \ e_1 \ \mathtt{at}\ e_2 \ ([\,] \ \Rightarrow \ e_3 \mid (x, y) \ \Rightarrow \ e_4) : \tau}$$

Figure 14: Well-Formed List Constructs

$$\begin{array}{llll} \mathit{types} & \tau & ::= & \cdots \mid \tau \ \mathit{option}\ \mathit{ref}\ \mathtt{at}\ \rho \\ \mathit{expressions} & e & ::= & \cdots \mid \mathtt{refsome}(e_1)\ \mathtt{at}\ e_2 \mid \mathtt{deref}\ e_1\ \mathtt{at}\ e_2\ (\mathtt{Null} \Rightarrow e_3 \mid \mathtt{Some}\ x \Rightarrow e_4) \\ & & & \mid \mathtt{store}\ e_1 := e_2\ \mathtt{at}\ e_3\ (\mathtt{Null} \Rightarrow e_4 \mid \mathtt{Some}\ x \Rightarrow e_5) \end{array}$$

Figure 15: Mutable Data

$\boxed{\Delta \vdash \tau}$

$$\frac{\Delta \vdash L}{\Delta \vdash L\ \mathit{option}\ \mathit{ref}\ \mathtt{at}\ \rho}\ (\rho \in \Delta)$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta; \Gamma_1 \vdash e_1 : \tau \quad \Delta; \Gamma_2 \vdash e_2 : \mathit{rgn}(\rho)}{\Delta; \Gamma \vdash \mathtt{refsome}(e_1)\ \mathtt{at}\ e_2 : \tau\ \mathit{option}\ \mathit{ref}\ \mathtt{at}\ \rho}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \bowtie \Gamma_4 \qquad \Delta; \Gamma_1 \vdash e_1 : \tau\ \mathit{option}\ \mathit{ref}\ \mathtt{at}\ \rho \qquad \Delta; \Gamma_2 \vdash e_2 : \mathit{rgn}(\rho) \\ \Delta; \Gamma_3 \vdash e_3 : \tau' \qquad \Delta; \Gamma_4, x{:}\tau \vdash e_4 : \tau'\end{array}}{\Delta; \Gamma \vdash \mathtt{deref}\ e_1\ \mathtt{at}\ e_2\ (\mathtt{Null} \Rightarrow e_3 \mid \mathtt{Some}\ x \Rightarrow e_4) : \tau'}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \bowtie \Gamma_4 \bowtie \Gamma_5 \qquad \Delta; \Gamma_1 \vdash e_1 : \tau\ \mathit{option}\ \mathit{ref}\ \mathtt{at}\ \rho \qquad \Delta; \Gamma_2 \vdash e_2 : \tau \\ \Delta; \Gamma_3 \vdash e_3 : \mathit{rgn}(\rho) \qquad \Delta; \Gamma_4 \vdash e_4 : \tau' \qquad \Delta; \Gamma_5, x{:}\tau \vdash e_5 : \tau'\end{array}}{\Delta; \Gamma \vdash \mathtt{store}\ e_1 := e_2\ \mathtt{at}\ e_3\ (\mathtt{Null} \Rightarrow e_4 \mid \mathtt{Some}\ x \Rightarrow e_5)}$$

Figure 16: Well-formedness for Mutable Data Structures

region as all other objects that are ever stored in that region. Once again, objects and their resources may not be reused on individual basis and again, linear invariants can help.

In this section, we define a new sort of reference that may be pointed to by many objects, but which holds the lone pointer to the object it contains. We use a dynamic check to ensure that a linear object is not extracted from such a reference multiple times. More precisely, the object stored in the reference may be null or an address. The dereference operation extracts the object, be it null or an address, and continues with one of two branches depending on the result. If the extracted object is an address then the second branch is executed and the address is bound to $x$. The assignment operator stores an object into the reference. If the reference contained null before the store operation was attempted then control continues with the first branch and otherwise control continues with the second branch.

The new reference type ($\tau\ \mathit{option}\ \mathit{ref}\ \mathtt{at}\ \rho$) belongs to the set of intuitionistic types ($I$) but unlike other intuitionistic objects, it may contain objects of linear type. Figure 16 contains the well-formedness rules for the new types and expressions.

There is a significant cost to using this mechanism. At compile time, there is no way to distinguish between a reference that contains null and a reference that contains an object. Consequently, although the extended type system is safe in the sense that it prevents access to dangling pointers, it does not ensure that all data structures are eventually collected. Since references are intuitionistic, it is possible to forget all pointers to a reference cell and thereby to lose access to any linear object it may contain. If the linear object in question is a region then there is the potential to leak an unbounded amount of space. It may be possible to pursue a dynamic solution to this memory leak problem, but we will leave it for future work.

## 7 Related and Future Work

This paper draws together two different branches of type theory designed for managing computer resources. Research on linear types originated with Girard's linear logic [13] and Reynolds' syntactic control of interference [29]. Linear type systems were later studied by many researchers [19, 37, 1, 20, 6, 36, 42, 15]. Type and effect systems were introduced by Gifford and Lucassen [12] and they too have been explored by many others [17, 32, 34, 24].

More recently, a number of new linear type systems, or more generally, "substructural type theories," have been developed. For example, Kobayashi's quasi-linear types [18], Polakow and Pfenning's ordered type theory [27, 28], O'Hearn's bunched typing [25], and Smith, Walker and Morrisett's alias types [31, 40] fall into this category. There is also renewed interest in developing new logics that facilitate Hoare-style reasoning about heap-allocated data structures. Reynolds [30] and Ishtiaq and O'Hearn [16] have developed substructural logics for just this purpose. An interesting line of research is to investigate how these other systems for alias control interact with region-based memory management. We suspect that the grouping aspect of regions is largely orthogonal to the reasoning principles used in these logics and type theories, and we hope that further study of combined systems will lead to interesting new programming invariants.

The initial inspiration for this work comes from Walker, Crary and Morrisett's capability calculus [8, 39]. The capability calculus uses a notion of "static capability" to control access to regions. Capability aliasing was controlled through a combination of bounded quantification and a form of syntactic control of interference. Our current work has the advantage of being both conceptually simpler and more expressive in a number of ways (although there are also certain continuation-passing style programs that can be written in the capability calculus, but not here). The principal reason for these improvements is that we have taken standard linear type systems and applied them uniformly across a language in which regions are ordinary first-class objects rather than special, second-class constructs.

There are several other ongoing projects that are exploring new implementation techniques and applications of regions. Makholm, Niss and Henglein [21] have had the same insights with respect to reference-counted regions as we have. They are currently designing a strongly-typed imperative language with (second-class) reference-counted regions. Gay and Aiken [10, 11] have developed run-time libraries and language support for reference-counted regions in C. Their reference-counting scheme is somewhat different than the one we have introduced here as they count the number of pointers that cross region boundaries rather than the number of pointers to the region data structure itself. Deallocation is allowed when there are no more pointers to values in a particular region and safety is checked mainly at run time.

Deline and Fähndrich [9] are developing a new type-safe variant of C called Vault. They use a combination of the capabilities and alias types mentioned above to control access to all sorts of program resources including memory regions. They have also developed effective local type inference techniques that minimize programmer annotations. Their source language design appears to be quite practical – they have already demonstrated that it is possible to port device drivers written in C to Vault and to use capabilities to verify that the drivers obey important safety properties. We hope that our formal work provides greater confidence in the correctness of Vault and serves as a source of further ideas. For example, our `let` operation would allow Vault to relax some of its restrictions on aliasing temporarily.

Dan Grossman, Trevor Jim and Greg Morrisett are currently developing a second type-safe variant of C, called Cyclone, which, like Vault, gives low-level programmers control over data structure layout, powerful mechanisms for type abstraction and strong safety guarantees. Currently, Cyclone relies upon a conservative garbage collector. However, together with Grossman *et al.*, we are exploring ways to incorporate the memory management techniques described here into Cyclone. Certain features of Cyclone, including existential polymorphism over types, abstract types and exceptions require further thought, but none of these challenges appear to be insurmountable. We feel confident that we will soon be able to give low-level programmers a variety of options when it comes to choosing their own safe memory management policies.

# 8 Conclusions

We have developed a new framework for safe, mostly-static memory management. The framework draws its power from the fact that it combines two well-studied paradigms for controlling computer resources, one based on linear typing and the other based on regions. One of the important aspects of our development is that we make a clean separation between the role played by regions and the role played by linear typing:

- Regions group objects with related lifetimes. An operation on regions, such as deallocation, simultaneously affects all objects within the group.
- Linear types control the number of uses of any object. Regions themselves are considered ordinary program objects so linear types can control the number of uses of each region.

A second important component of our system is that we freely mix different interpretations of linear types for maximum programmer flexibility. For example, when the number of uses of a particular region is easy to determine at compile-time, it is usually possible to employ a purely static memory management solution based on the conventional interpretation of linear types. However, if the number of uses is unknown, then a static solution may be overly restrictive. In this case, programmers can choose a more dynamic solution to their memory management problems involving reference counting.

# Acknowledgments

Many of the ideas in this paper arose from discussions with Greg Morrisett. In particular, the mechanism we use to handle mutable data structures was developed in collaboration with Greg. We have also benefited from technical insights provided by Frank Pfenning.

# References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] 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.

[3] Cristiano Calcagno. Stratified operational semantics for safety and correctness of region calculus. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 155–165, London, UK, January 2001.

[4] Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Information and Computation*, July 2000. To appear.

[5] Chih-Ping Chen and Paul Hudak. Rolling your own mutable adt – a connection between linear types and monads. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 54–66, Paris, January 1997.

[6] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, April 1992.

[7] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, March 1996.

[8] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, January 1999.

[9] Rob Deline and Manuel Fähndrich. The Vault project. Presented at the Carnegie Mellon University principles of programming languages seminar. See http://research.microsoft.com/vault/ for more information., November 2000.

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

[11] David Gay and Alex Aiken. Language support for regions. In *Workshop on semantics, program analysis and computing environments for memory management (SPACE 2001)*, London, UK, January 2001.

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

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

[14] Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In *workshop on higher order operational techniques in semantics*, pages 1–19, September 2000.

[15] Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. In Gert Smolka, editor, *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, Berlin, March 2000.

[16] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 14–26, London, UK, January 2001.

[17] 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.

[18] Naoki Kobayashi. Quasi-linear types. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 29–42, San Antonio, January 1999.

[19] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[20] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 1992.

[21] Henning Makholm, Henning Niss, and Fritz Henglein. Towards a more flexible region type system. In *Workshop on Semantics, program analysis and computing environments for memory management (SPACE 2001)*, London, UK, January 2001.

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

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

[24] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 84–97, January 1994.

[25] Peter O'Hearn. On bunched typing. Unpublished manuscript, July 2000.

[26] Simon Peyton Jones and John Hughes (ed.). Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, February 1999.

[27] Jeff Polakow. Logic programming with an ordered context. In *Conference on Principles and Practice of Declarative Programming*, Montreal, September 2000.

[28] Jeff Polakow and Frank Pfenning. Properties of terms in continuation-passing style in an ordered logical framework. In *Workshop on Logical Frameworks and Meta-Languages*, Santa Barbara, June 2000.

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

[30] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial perspectives in computer science*, Palgrove, 2000.

[31] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, March 2000.

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

[33] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report 98/25, Computer Science Department, University of Copenhagen, 1998.

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

[35] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999. Special issue on linear logic.

[36] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *ACM International Conference on Functional Programming and Computer Architecture*, San Diego, CA, June 1995.

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

[38] Philip Wadler. The marriage of effects and monads. In *ACM International Conference on Functional Programming*, pages 63–74, Baltimore, September 1998.

[39] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 2000. To appear.

[40] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Montreal, September 2000.

[41] Daniel C. Wang and Andrew Appel. Type-preserving garbage collectors. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 166–178, London, UK, January 2001.

[42] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 15–28, San Antonio, January 1999.