# On Regions and Linear Types* (Extended Abstract)

David Walker and Kevin Watkins
Carnegie Mellon University
School of Computer Science

## 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* memory 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, two techniques for solving this problem have repeatedly found success:

- Linear type systems, which have been derived from Girard's linear logic [11] and Reynolds' syntactic control of interference [24], and
- The type, region and effect discipline developed by Gifford and Lucassen [10] and refined by Jouvelot, Talpin and Tofte [17, 26, 28].

Despite the individual successes of these techniques, there has been little research that attempts to understand the relationships between the two or how to unify them in a single language. Hence, in this paper, we investigate how they may be fruitfully used together in the domain of memory management.

### 1.1 Regions

The starting point for our development is a simple functional programming language that contains 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 or exceeding memory management via malloc and free or garbage collection in many situations [8, 12].

As an example, consider the function *Pair*:[1]

$$\begin{array}{l}\lambda(x,\ gen,\ r).\\ \quad \texttt{let}\ r' = gen\ ()\ \texttt{in}\\ \quad \texttt{let}\ y = x \times x\ \texttt{at}\ r'\ \texttt{in}\\ \quad r' \times y\ \texttt{at}\ r\end{array}$$

*Pair* has three arguments: a value $x$ that will be duplicated and returned in a pair (call it $y$), a first-class function *gen* that returns the region $r'$ used to hold the pair and finally, a region $r$ that will hold the ultimate result (the pair $y$ and the region $r'$ that it was allocated in). The expression $x \times x$ `at` $r'$ allocates a pair of $x$'s in the region $r'$.

The function *Pair* has many of the features that make our language interesting. Most importantly, regions, like

*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 and by ONR grant number 1140015, "Efficient Logics for Reasoning about Network Security." 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, ONR, or the U.S. government. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.


*ICFP'01,* September 3-5, 2001, Florence, Italy.


[1]Normally, function closures, like other storage objects, are allocated in regions, but we will ignore this detail in our informal introduction.

other values, are ordinary *first-class* programming objects. They can be passed as arguments to functions, returned as results and stored in data structures. In order to program with regions, we must also be able to allocate new ones, and we do this using the `alloc` primitive. When a region is no longer needed, it can be deallocated using the `free` primitive. Given these two primitives and the expression `let` $x \times x' = e$ `in` $e'$, which projects the two components $x$ and $x'$ from the pair $e$ for use in the expression $e'$, we can write the following code, which uses the *Pair* function.

$$
\begin{array}{lll}
\texttt{let} & gen & = \lambda() \rightarrow \texttt{alloc}\ ()\ \texttt{in} \\
\texttt{let} & r & = \texttt{alloc}\ ()\ \texttt{in} \\
\texttt{let} & r' \times y & = Pair\ (17, gen, r)\ \texttt{in} \\
\multicolumn{3}{l}{\texttt{free}(r);} \\
\texttt{let} & x \times x' & = y\ \texttt{in} \\
\multicolumn{3}{l}{\texttt{free}(r');} \\
\multicolumn{3}{l}{x + x'}
\end{array}
$$

Of course, programming with regions, like other forms of explicit memory management, is fraught with danger. If a programmer accidentally deallocates a region too early, then chaos ensues as his or her program chases dangling pointers. Forgetting to deallocate a region is almost as bad since it causes a memory leak. Tofte and Talpin [28] solved this problem by developing a type-and-effect system to check the safety of programs that use regions. Unfortunately, their type system is based on the notion that regions must be used in a first-allocated/last-deallocated, stack-like fashion and moreover, that regions are intrinsically second-class objects. Other proposals for static region-based memory management [31, 14] and optimizations of Tofte and Talpin's original model [4, 2] helped to alleviate some of the expressiveness problems, but these proposals are often very complex. Moreover, none of these efforts consider regions to be first-class programming objects. As a result, the simple *Pair* function will not type check in previous systems.

## 1.2 Safety through Linear Types

Linear type systems have been used many times before to guarantee safety in the presence of explicit memory management operations for individual objects. These type systems provide information about the *last use* of a data structure, and clearly, if we are guaranteed that a data structure has been used for the last time, we can safely deallocate it. The simplest linear type systems [19, 1] actually guarantee that linear data structures are used exactly once. After this one use, the data structure is a deallocated. More sophisticated type systems [30, 6, 18, 15] make it possible to use "linear" objects several times, but still provide support for detecting the last use of such objects. The main disadvantage of memory management through linear type systems is that they restrict the amount of sharing/aliasing that can occur in linear data structures. As a result, programs are often forced to copy entire data structures or to maintain reference counts on every object, both of which can lead to excessive time and space overhead.

In this work, we take a new approach to the problem of safe, explicit memory management. In order to avoid restrictions on sharing between individual data structures and to avoid maintaining per-object reference counts, we group objects into regions. However, rather than attempting to craft our own custom region-based type system from scratch, we will take advantage of a large body of pre-existing literature designed specifically for controlling volatile resources — the literature on linear type systems. The combination of both regions and linear types has never been studied before and it is highly effective, yielding much more than the straightforward sum of the individual systems.

## 1.3 Contributions

The main contribution of this paper is to explore the synergy between linear type systems and region-based memory management. To this end, we have designed a simple lambda calculus of first-class regions in which a linear type system controls the use, reuse and deallocation of regions as well as other objects such as pairs or closures.

Because regions are, for the most part, ordinary first-class programming objects, it is relatively straightforward for us to adopt existing ideas from the literature on linear type systems. In this paper, we will actually study two such systems although we believe there are several more related type systems that can be combined effectively with regions. The first is a purely static system derived from Wadler's early work on linear type systems [30]. The derived rules for manipulating regions easily capture the effect of Tofte and Talpin's `letregion` construct. The second type system has a very different behavior from the first as it is derived from the reference-counting interpretation of linear types discovered by Chirimar, Gunter and Riecke [6]. Reference-counting adds a dynamic component to the language that increases the flexibility of the system but gives fewer static guarantees. Finally, by combining ideas from Wadler with the reference counting interpretation, we obtain new invariants that make it possible to manage deferred reference counts.

Another important component of our system is that notions of linearity are applied *uniformly* across our language: any storage object can be linear or not. This helps to contribute to the simplicity of our language. It also implies that programmers can freely mix ordinary linear data structures with regions, which gives rise to additional new memory management invariants. For example, programmers will be able to define *heterogeneous* linear lists in which every element of the list inhabits its own region and therefore may be deallocated independently of any other elements in the list. In contrast, previous region-based type systems could only represent *homogeneous* lists, where every element inhabited the same region and therefore no list elements could be deallocated until the entire list was dead. Previous region-based type systems have also had difficulty dealing with mutable data structures. Related techniques make it possible to handle mutable data structures more effectively than before. Unfortunately, due to space considerations we are unable to explain them here (See a preliminary version of this work [32] for details).

One important problem that we make no attempt to solve in this report is the issue of type inference. As a result, the current work could be viewed as a specification for an idealistic compiler intermediate language, rather than a source programming language.

In the remainder of this paper, 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. Section 3 describes the execution model for the language. Sections 4 and 5 extend the language with reference-counted regions and lists respectively. The latter demonstrates how to define heterogeneous data structures. Finally, section 6 discusses related work.

# 2. THE CORE LANGUAGE

Our core language arises by layering ideas drawn from Wadler's linear type system [30] on top of a call-by-value lambda calculus with first-class regions.

## 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 Linear Types

Our linear type system includes two different variants of every storage object: there are two forms of closure, two forms of pairs and later there will be two forms of regions. The "linear" variant classifies objects that are referenced by exactly one pointer and are "used" exactly once.[2] Linear objects are deallocated after they are used. The "intuitionistic" variant classifies objects that can be used an unlimited number of times (including not at all). In this system, by contrast with linear logic, linearity is inherent in the types themselves, rather than in the context in which they appear.

We write $\tau_1 \overset{\phi}{\to} \tau_2$ for generic functions where the qualifier $\phi$ is either $\cdot$, indicating an intuitionistic function that may be used many times, or $1$, indicating a linear function that must be used exactly once.[3] After its single use, the closure containing the function's free variables will be deallocated. Likewise, we write $\tau_1 \overset{\phi}{\times} \tau_2$ 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 $\tau_1 \times \tau_2$ for an intuitionistic pair.

In our formal work, we will use $()$ as a base 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 actual implementation need not allocate it in the store) and therefore no need to define the linear variant. In our examples, we will use other base types, such as integers, assuming they may be freely copied.

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 $int \overset{1}{\times} int \to int$ we will often write $(int, int) \to int$.

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 section 2.2. In addition, we consider intuitionistic pairs with linear component types, such as $(\tau_1 \overset{1}{\times} \tau_2) \times \tau_3$ to be syntactically ill-formed.

[2] Beware, we will later introduce an operator that temporarily converts a single-use object into a multi-use object.

[3] 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.

### 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. In particular, 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 $\tau_1 \overset{\phi}{\to} \tau_2\,\texttt{at}\,\rho$ 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 "$\texttt{at}\,\rho$," for some region $\rho$. However, in our examples we will assume there is some global top-level region named "$\_$" that is always accessible and is never deallocated. This convention allows us to simulate an ordinary linear type system simply by allocating all objects in $\_$. Whenever we omit a region annotation "$\texttt{at}\,\rho$" or (see the next section) "$\texttt{at}\,r$," assume the data structure lives in the region $\_$.

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

$$\forall[\rho].rgn(\rho) \to (int \times int\ \texttt{at}\ \rho)$$

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 is the *gen* function defined in the introduction. It takes no arguments and returns some new region $\rho$, so it is assigned the type $() \to \exists\rho.\overset{1}{rgn}(\rho)$.

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 — linear typing will ensure that deallocation is safe. 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.

### 2.1.3 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

$$
\begin{array}{l}
\Delta ::= \cdot \mid \Delta, \rho \\
\phi ::= \cdot \mid {}_{1} \\
\tau ::= L \mid I \\
L ::= \overset{1}{rgn}(\rho) \mid \tau_1 \overset{1}{\times} \tau_2 \ \texttt{at}\ \rho \mid \forall[\Delta].\tau_1 \overset{1}{\to} \tau_2 \ \texttt{at}\ \rho \mid \exists\rho.\tau \\
I ::= (\,) \mid rgn(\rho) \mid I_1 \times I_2 \ \texttt{at}\ \rho \mid \forall[\Delta].\tau_1 \to \tau_2 \ \texttt{at}\ \rho
\end{array}
$$

**Figure 1: Syntax: Types**

$$
\begin{array}{l}
\Gamma ::= \cdot \mid \Gamma, x{:}\tau \\
e ::= x \mid (\,) \mid e_1; e_2 \\
\qquad \mid e_1 \overset{\phi}{\times} e_2 \ \texttt{at}\ e_3 \mid \texttt{let}\ x_1 \times x_2 = e_1 \ \texttt{in}\ e_2 \\
\qquad \mid \lambda[\Delta]x{:}\tau \overset{\phi}{\to} e_1 \ \texttt{at}\ e_2 \mid e_1[\Delta]\ e_2 \\
\qquad \mid \texttt{pack}[\rho, e] \ \texttt{as}\ \exists\rho.\tau \mid \texttt{unpack}\ \rho, x = e_1 \ \texttt{in}\ e_2 \\
\qquad \mid \texttt{alloc}\ e \mid \texttt{free}\ e \\
\qquad \mid \texttt{let}\ x = e_1 \ \texttt{in}\ e_2 \mid \texttt{let!}\ (y)\ x = e_1 \ \texttt{in}\ e_2
\end{array}
$$

**Figure 2: Syntax: Expressions**

then the notation is undefined. The judgment $\Delta \vdash \tau$ states that the free variables in $\tau$ are contained in $\Delta$ and that intuitionistic types do not contain linear component types.

## 2.2 Expressions

Figure 2 presents the expression syntax. As usual, the syntax includes variables as well as introduction and elimination forms for each type of object. Each of the introduction forms (aside from `alloc`, the introduction form for regions, which always introduces linear regions) uses a qualifier to indicate whether a linear or non-linear value is introduced.

We also include two forms of let-expression. The first is standard; it binds the variable $x$ to the value computed by expression $e_1$ and then continues to compute the result of $e_2$. The second let-expression has the same effect as the first at run time. However, the static semantics are derived from Wadler's `let!` construct[30]. At compile time, the input region $y$, which must initially have linear type, is given intuitionistic type in the expression $e_1$ and then linear type again in $e_2$.[4] Through this device, we can use a "linear" region $y$ multiple times and later recover its linear type, which allows us to deallocate the region. In the following informal example, we allocate a linear region, use it twice to allocate two pairs and then delete it.

$$
\begin{array}{ll}
\texttt{let}\ y & = \texttt{alloc ()}\ \texttt{in} \\
\texttt{let!}\ (y)\ x & = (3 \times 5\ \texttt{at}\ y) \times 7\ \texttt{at}\ y\ \texttt{in} \\
\texttt{free}(y) &
\end{array}
$$

In order to use `let!` safely, it is necessary to ensure that no references to $y$ escape from $e_1$ and into $e_2$. In the example above, only pairs allocated in $y$ escape, not references to $y$ itself, for if a reference did escape, we could not justify $y$'s linear type in $e_2$. The type system will prevent references to $y$ from escaping by performing an analysis of the type of $e_1$. We will explain these operations more fully in conjunction with their typing rules, but before we can proceed with the formal semantics we must present a few auxiliary definitions.

[4]Note that the variable $y$ is free, not bound in this expression.

### 2.2.1 Notation

The typing rules for expressions have the form $\Delta; \Gamma \vdash e : \tau$ where $\Gamma$ is a finite map from variables to types. The domain of $\Gamma$ will include all the free variables in $e$. We assume bound variables are appropriately alpha-converted before being entered into the context. As for type contexts, the notation $\Gamma, \Gamma'$ is undefined unless the domains of $\Gamma$ and $\Gamma'$ are disjoint. Our type system relies upon a nondeterministic operation $\Gamma = \Gamma_1 \bowtie \Gamma_2$ that splits the linear assumptions in $\Gamma$ between the contexts $\Gamma_1$ and $\Gamma_2$. The intuitionistic assumptions in $\Gamma$ appear in both $\Gamma_1$ and $\Gamma_2$. We will often write $\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3$ as an abbreviation for $\Gamma = \Gamma_1 \bowtie \Gamma'$ and $\Gamma' = \Gamma_2 \bowtie \Gamma_3$.

We also use the notation $\overset{\phi}{\Gamma}$. When $\phi$ is $\cdot$, all the types in $\Gamma$ must be intuitionistic. When $\phi$ is ${}_{1}$, $\Gamma$ is unrestricted. This notation is used to prevent intuitionistic objects from containing linear objects. Since $\Gamma$ is just a finite map, we implicitly allow exchange of any two assumptions in the context. Weakening and contraction will be admissible on intuitionistic components of the context, but not on linear ones.

We use the notation $e[x_1/x_2]$ and $e[\rho_1/\rho_2]$ to denote standard capture-avoiding substitution of expression variables and regions into expressions.[5] The notation $e[\Delta_1/\Delta_2]$ extends region substitution pointwise to region contexts, and is only defined if $\Delta_1$ and $\Delta_2$ have the same number of elements.

### 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. If an object contains a reference to an intuitionistic region, the region must appear in its type.

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 conditions on the formation of closures and existential packages, which otherwise could capture references to an intuitionistic region without its being mentioned in the type. This final invariant ensures it is possible to perform a type-based analysis to prevent stored intuitionistic regions from escaping the scope of a `let!` expression.

Figure 3 presents the typing rules for expressions. The first three rules do not involve regions so they are the normal typing rules for a linear lambda calculus. The rule for variables requires that the context $\Gamma$ contain 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 context splitting operator

[5]Because of the way we have defined our operational semantics (see section 3), only variables are subsituted into expressions—arbitrary expressions are never substituted.

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma = (\overset{\phi}{\Gamma_1}, \dot{\Gamma}_2) \bowtie \Gamma_3 \qquad \Delta, \Delta' \vdash \tau \\ \Delta, \Delta'; \overset{\phi}{\Gamma_1}, x{:}\tau \vdash e_1 : \tau_1 \\ \Delta;\Gamma_3 \vdash e_2 : rgn(\rho)\end{array}}{\Delta;\Gamma \vdash \lambda[\Delta']x{:}\tau \xrightarrow{\phi} e_1 \ \mathtt{at}\ e_2 : \forall[\Delta'].\tau \xrightarrow{\phi} \tau_1 \ \mathtt{at}\ \rho} \ (closed(\Gamma_1))$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \\ \Delta;\Gamma_1 \vdash e_1 : \forall[\Delta_2].\tau_1 \xrightarrow{\phi} \tau_2 \ \mathtt{at}\ \rho \\ \Delta;\Gamma_2 \vdash e_2 : \tau_1[\Delta_1/\Delta_2] \\ \Delta;\Gamma_3 \vdash x : rgn(\rho) \qquad \text{(for some } x\text{)}\end{array}}{\Delta;\Gamma \vdash e_1[\Delta_1]\ e_2 : \tau_2[\Delta_1/\Delta_2]} \ (\Delta_1 \subseteq \Delta)$$

$$\frac{\Delta;\Gamma \vdash e : \tau[\rho_0/\rho]}{\Delta;\Gamma \vdash \mathtt{pack}[\rho_0, e]\ \mathtt{as}\ \exists\rho.\tau : \exists\rho.\tau} \ (closed_\rho(\tau),\ \rho_0 \in \Delta)$$

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

$$\frac{\Delta;\Gamma \vdash e : (\,)}{\Delta;\Gamma \vdash \mathtt{alloc}\ e : \exists\rho.\overset{1}{rgn}(\rho)}$$

$$\frac{\Delta;\Gamma \vdash e : \overset{1}{rgn}(\rho)}{\Delta;\Gamma \vdash \mathtt{free}\ e : (\,)}$$

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

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \quad \Delta;\Gamma_1 \vdash y : \overset{1}{rgn}(\rho) \\ \Delta;\Gamma_2, y{:}rgn(\rho) \vdash e_1 : \tau_1 \\ \Delta;\Gamma_3, y{:}\overset{1}{rgn}(\rho), x{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Delta;\Gamma \vdash \mathtt{let!}\ (y)\ x = e_1 \ \mathtt{in}\ e_2 : \tau_2} \ (closed_\rho(\tau_1))$$

**Figure 3: Well-formed Expressions**

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 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 a 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.[6] In a moment, we will define an 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, `let` $x_1 \times x_2 = e_1$ `in` $e_2$, projects the two components of the pair $e_1$ and binds them to $x_1$ and $x_2$ before continuing with the expression $e_2$. If $e_1$ inhabits region $\rho$ then we must ensure that this region is still live. Otherwise, this access is a memory error. A reference $y$ to the region is extracted from the context to witness that the region is still live.

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

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, breaking invariant 3 listed above. Therefore, we require all functions to be closed with respect to intuitionistic regions. 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, a concept found in standard effect systems [17, 28], is represented as part of the type of the function argument. The closure requirement is enforced by the predicate $closed_\rho(\tau)$ (pronounced "$\tau$ is region-closed with respect to $\rho$").

$$\begin{array}{lcll}
closed_\rho(rgn(\rho)) & = & \text{false} & \\
closed_\rho(\tau_1 \overset{\phi}{\times} \tau_2 \ \mathtt{at}\ \rho') & = & closed_\rho(\tau_1) \wedge closed_\rho(\tau_2) & \\
closed_\rho(\exists\rho'.\tau) & = & closed_\rho(\tau) & (\text{if } \rho' \neq \rho) \\
closed_\rho(\tau) & = & \text{true} & (\text{otherwise})
\end{array}$$

In clause two above, $\rho'$ may or may not be equal to $\rho$ and the pair is still closed if its components are. The predicate is used to rule out references to intuitionistic regions (with

[6] 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. This would essentially require that programs be written in A-normal form.

type $rgn(\rho)$) which carry with them the privilege to access a region, but it does not rule out references to objects (such as pairs or closures) within an intuitionistic region. Notice that linear regions are always closed. 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 3). 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. Therefore, the function closure cannot contain references to intuitionistic regions (although it can contain pairs and other functions that inhabit intuitionistic regions). Section 2.3 explains how to lift the region-closed restriction.

The rule for function application ensures the region name arguments ($\Delta'$) match the expected region name parameters and that the argument has the expected type. As in the elimination form for pairs, the existence of a reference to the region containing the function ($x$) serves as proof that the region is still live.

Existential types pose difficulties similar to those already described for function closures, and the solution we have adopted is the same. In fact, given Minamide, Morrisett and Harper's interpretation of function closures as existential packages [20], existential types may be viewed as the real source of the problem. To ensure that 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. The elimination form for existentials is the standard unpack expression.

### 2.2.4 Region Allocation and Deallocation

The `alloc` primitive returns a new, linear region. It naturally has type $() \to \exists\rho.\overset{1}{rgn}(\rho)$. The `free` primitive consumes a linear region and has the type $\exists\rho.\overset{1}{rgn}(\rho) \to ()$. However, we do not treat these primitives as constants with these types because our operational semantics is slightly more elegant if we treat them as expressions with their own typing rules (see Figure 3). For programmer convenience, it is unnecessary to pack the argument to `free` as an existential (the region name in the premiss of the typing rule for `free` may be viewed as implicitly existentially quantified).

Intuitionistic regions are introduced and eliminated using `let!` as explained earlier. One of the key constraints in the typing rule is that the type of $e_1$ should be region-closed with respect to $\rho$. This prevents intuitionistic references to $\rho$ from escaping from $e_1$ into $e_2$.

We have introduced `let!` as an orthogonal programming construct so that the central concept may be understood in isolation from other expressions in the language. However, it is useful to be able to make a linear region temporarily intuitionistic in many different program scopes, not just those connected with a `let!` expression. A more general treatment would permit expressions of the form `let!` $(y)$ *pattern* $=$ $e_1$ `in` $e_2$. We use the following instance of the more general construct in the example we are about to present:[7]

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \bowtie \Gamma_4 \\ \Delta; \Gamma_1 \vdash y : \overset{1}{rgn}(\rho_1) \\ \Delta; \Gamma_2, y{:}rgn(\rho_1) \vdash e_1 : \tau_1 \overset{\phi}{\times} \tau_2 \ \texttt{at}\ \rho_2 \\ \Delta; \Gamma_3, y{:}\overset{1}{rgn}(\rho_1), x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau_2 \\ \Delta; \Gamma_4, y{:}rgn(\rho_1) \vdash z : rgn(\rho_2) \quad \text{(for some } z\text{)}\end{array}}{\Delta; \Gamma \vdash \texttt{let!}\ (y)\ x_1 \times x_2 = e_1\ \texttt{in}\ e_2 : \tau_3} \ (closed_{\rho_1}(\tau_1, \tau_2))$$

*Example.* Now we can look at how to type the example given in the introduction. The text of the example has only been changed to add typing annotations, pack and unpack instructions, and linearity annotations (! and $\hat{1}$).

$$\begin{array}{l}\lambda[\rho](x{:}int, gen{:}() \to \exists\rho'.\overset{1}{rgn}(\rho'), r{:}rgn(\rho)) \to \\ \quad \texttt{unpack}\ \rho', r' = gen\ ()\ \texttt{in} \\ \quad \texttt{let!}\ (r')\ y = x \times x\ \texttt{at}\ r'\ \texttt{in} \\ \quad \texttt{pack}[\rho', r' \overset{1}{\times} y\ \texttt{at}\ r]\ \texttt{as}\ \tau_{res}\end{array}$$

The function $gen$ generates fresh linear regions, and therefore it has type $() \to \exists\rho'.\overset{1}{rgn}(\rho')$. The region argument $r$ is given intuitionistic type because it is used by $Pair$, but is not deallocated by it. Therefore, the context calling $Pair$ must retain an alias to $r$ in order to deallocate it. The function returns a value of type $\tau_{res} = \exists\rho'.\overset{1}{rgn}(\rho') \overset{1}{\times} (int \times int\ \texttt{at}\ \rho')\ \texttt{at}\ \rho$. The calling context may be typed as follows.

$$\begin{array}{ll}\texttt{let}\ gen & = (\lambda() \to \texttt{alloc}\ ())\ \texttt{in} \\ \texttt{unpack}\ \rho, r & = \texttt{alloc}\ ()\ \texttt{in} \\ \texttt{let!}\ (r)\ x_{res} & = Pair[\rho](17, gen, r)\ \texttt{in} \\ \texttt{unpack}\ \rho', z & = x_{res}\ \texttt{in} \\ \texttt{let!}\ (r)\ r' \times y & = z\ \texttt{in} \\ \texttt{free}(r); & \\ \texttt{let!}\ (r')\ x \times x' = y\ \texttt{in} & \\ \texttt{free}(r'); & \\ x + x' & \end{array}$$

## 2.3 Relation to the Tofte-Talpin Language

There are close connections between our `let!` and Tofte and Talpin's `letregion`. Both constructs use a type-based escape analysis to ensure safety. When Wadler first introduced `let!` into his linear lambda calculus, he had no notion of a region name, so his analysis was very imprecise. Since a region type contains a unique region name, it is a form of singleton type, a very precise classifier that makes the modified construct much more effective. In fact, it is possible to define a `letregion` construct in our calculus:

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

A general translation of the Tofte-Talpin language into our calculus is not possible without significant run-time overhead. The primary barrier is that (simplified) Tofte-Talpin

[7] This construct can be defined within the language without overhead if the pair that is accessed is allocated in some region other than $y$. Otherwise, we must incur the cost of an allocation and immediate deallocation of a linear pair:

$\texttt{let!}\ (y)\ z = (\texttt{let}\ x_1 \times x_2 = e_1\ \texttt{in}\ x_1 \overset{1}{\times} x_2\ \texttt{at}\ \_)\ \texttt{in}$
$\texttt{let}\ x_1 \times x_2 = z\ \texttt{in}\ e_2.$

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

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

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

$$\frac{\Delta,\Delta' \vdash \tau \quad \Delta,\Delta';\overset{\phi}{\Gamma}_1, x{:}\tau \vdash e : \tau'}{\Delta;\overset{\phi}{\Gamma}_1,\dot{\Gamma}_2 \vdash \langle\lambda[\Delta']x{:}\tau \overset{\phi}{\to} e\rangle_\rho : \forall[\Delta'].\tau \overset{\phi}{\to} \tau' \;\texttt{at}\; \rho} \quad (\rho\in\Delta,\ \mathit{closed}(\Gamma_1))$$

$$\frac{\Delta;\Gamma \vdash x : \tau[\rho_0/\rho]}{\Delta;\Gamma \vdash \texttt{pack}[\rho_0, x] \;\texttt{as}\; \exists\rho.\tau : \exists\rho.\tau} \quad (\rho_0\in\Delta,\ \mathit{closed}_\rho(\tau))$$

$$\frac{}{\Delta;\dot{\Gamma} \vdash \overset{\phi}{\mathit{data}}(\rho) : \overset{\phi}{\mathit{rgn}}(\rho)} \quad (\rho\in\Delta)$$

**Figure 4: Well-Formed Stored Values**

$\boxed{\Delta;\Gamma \vdash S : \tau_1 \Rightarrow \tau_2}$

$$\frac{}{\Delta;\dot{\Gamma} \vdash \cdot : \tau \Rightarrow \tau}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta;\Gamma_1, x{:}\tau_1 \vdash E[x] : \tau_2 \quad \Delta;\Gamma_2 \vdash S : \tau_2 \Rightarrow \tau_3}{\Delta;\Gamma \vdash E, S : \tau_1 \Rightarrow \tau_3}$$

$$\frac{\Gamma = \Gamma_1 \bowtie (\Gamma_2, \dot{\Gamma}_3) \quad \Delta;\Gamma_1 \vdash y : \mathit{rgn}(\rho) \quad \Delta;\Gamma_2, x{:}\overset{1}{\mathit{rgn}}(\rho) \vdash S : \tau_1 \Rightarrow \tau_2}{\Delta;\Gamma \vdash \texttt{let!}\; x = y \;\texttt{in}\; S : \tau_1 \Rightarrow \tau_2} \quad (\mathit{closed}_\rho(\tau_1),\ \mathit{closed}_\rho(\Gamma_2))$$

**Figure 5: Well-Formed Stacks**

closures have type $\tau_1 \overset{\psi}{\to} \tau_2 \;\texttt{at}\; \rho$ where $\psi$ is the set of regions $\{\rho_1, \ldots, \rho_n\}$ that the function accesses. Equality on these types is modulo equality of sets of regions. One might try to translate Tofte-Talpin closures into closures with the form $\mathit{rgn}(\rho_1) \times \cdots \times \mathit{rgn}(\rho_n) \times ((\mathit{rgn}(\rho_1) \times \cdots \times \mathit{rgn}(\rho_n), \tau_1) \to \tau_2 \;\texttt{at}\; \rho) \;\texttt{at}\; \rho$, but such a translation does not preserve equality (pairs are not associative, commutative, etc.). Therefore, if a translation from Tofte-Talpin is desired one must generalize our function types to include an effect on the arrow and use the following rule where $\mathit{closed}_{\overline{\psi}}(\Gamma)$ requires $\Gamma$ be closed with respect to all regions other than those in $\psi$:

$$\frac{\Gamma = (\overset{\phi}{\Gamma}_1, \dot{\Gamma}_2) \bowtie \Gamma_3 \quad \Delta,\Delta' \vdash \tau \quad \psi \subseteq \Delta \quad \Delta,\Delta';\overset{\phi}{\Gamma}_1, x{:}\tau \vdash e_1 : \tau_1 \quad \Delta;\Gamma_3 \vdash e_2 : \mathit{rgn}(\rho)}{\Delta;\Gamma \vdash \lambda[\Delta']x{:}\tau \overset{\phi,\psi}{\to} e_1 \;\texttt{at}\; e_2 : \forall[\Delta'].\tau \overset{\phi,\psi}{\to} \tau_1 \;\texttt{at}\; \rho} \quad (\mathit{closed}_{\overline{\psi}}(\Gamma_1))$$

In addition, the definition of $\mathit{closed}_\rho(\tau)$ must account for the new closure types:

$$\mathit{closed}_\rho(\forall[\Delta'].\tau \overset{\phi,\psi}{\to} \tau_1 \;\texttt{at}\; \rho') = \text{false (if } \rho \in \psi)$$

With these modifications we can capture the simple Tofte-Talpin closures. However, capturing the Tofte-Talpin notion of effect polymorphism is not trivial and we differ it to future work. For the remainder of this paper, we concentrate on the closure types defined earlier in the paper, as they are all we need to explore the relationship between regions and linear types.

# 3. THE ABSTRACT MACHINE

Programs in our language execute on an abstract machine. An abstract machine state ($\Sigma$) includes the list of live regions ($\Delta$), a description of the store ($H$), a stack ($S$) representing the current continuation and, finally, the expression to be evaluated.

The store maps variables to stored values ($s$), which may be unit, a function closure allocated in region $\rho$, a pair allocated in region $\rho$, an existential package, or the data structure associated with a region ($\overset{\phi}{\mathit{data}}(\rho)$).[8] The stack contains a list of evaluation contexts $E$, which are expressions with a hole $\Box$. The notation $E[e]$ denotes the expression formed by filling the hole in $E$ with $e$. A stack can also contain the special instruction $\texttt{let!}\; x = y \;\texttt{in}\; S$, which is used to represent the action of the `let!` expression in the static language. We will discuss this construct in further detail in the next section.

$$\begin{array}{rcl}
s & ::= & (\,) \mid \langle x_1 \overset{\phi}{\times} x_2\rangle_\rho \mid \langle\lambda[\Delta]x : \tau \overset{\phi}{\to} e\rangle_\rho \\
 & & \mid \texttt{pack}[\rho, x] \;\texttt{as}\; \exists\rho.\tau \mid \overset{\phi}{\mathit{data}}(\rho) \\
H & ::= & \cdot \mid H, x \mapsto s \\
S & ::= & \cdot \mid E, S \mid \texttt{let!}\; x = y \;\texttt{in}\; S \\
E & ::= & \Box; e \mid \Box \overset{\phi}{\times} e_1 \;\texttt{at}\; e_2 \mid x \overset{\phi}{\times} \Box \;\texttt{at}\; e \\
 & & \mid x_1 \overset{\phi}{\times} x_2 \;\texttt{at}\; \Box \mid \texttt{let}\; x_1 \times x_2 = \Box \;\texttt{in}\; e_2 \\
 & & \mid \lambda[\Delta]x{:}\tau \overset{\phi}{\to} e \;\texttt{at}\; \Box \mid \Box[\Delta]\; e \mid x[\Delta]\; \Box \\
 & & \mid \texttt{pack}[\rho, \Box] \;\texttt{as}\; \exists\rho.\tau \mid \texttt{unpack}\; \rho, x = \Box \;\texttt{in}\; e \\
 & & \mid \texttt{alloc}\; \Box \mid \texttt{free}\; \Box \mid \texttt{let}\; x = \Box \;\texttt{in}\; e \\
\Sigma & ::= & (\Delta; H; S; e)
\end{array}$$

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, stored values and stacks. These rules is to guarantee the following simple facts:

- There is exactly one region data structure in the store for each live region.
- All stored values are well-formed with appropriate types.
- The expression to be executed and the stack are well-formed with respect to the current store.

Aside from the typing rule for the special `let!`, which is discussed in more detail below, the typing rules for the abstract machine are quite intuitive. The rules are shown in Figures 4 through 6.

[8] In the ML Kit, the data associated with a region includes a pointer to the beginning of the region in memory and a pointer to the current allocation point within the region [27].

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

$$\frac{\begin{array}{c}\Delta \vdash H \text{ live} \\ \Delta' \vdash H : \Gamma \text{ store} \quad (\text{for some } \Delta' \supseteq \Delta) \\ \Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta'; \Gamma_1 \vdash e : \tau_1 \\ \Delta'; \Gamma_2 \vdash S : \tau_1 \Rightarrow \tau\end{array}}{\vdash (\Delta; H; S; e) : \tau \text{ program}}$$

$\boxed{\Delta \vdash H \text{ live}}$

$$\frac{}{\cdot \vdash \cdot \text{ live}}$$

$$\frac{\Delta_1, \Delta_2 \vdash H \text{ live}}{\Delta_1, \rho, \Delta_2 \vdash H, x \mapsto \overset{\phi}{data}(\rho) \text{ live}}$$

$$\frac{\Delta \vdash H \text{ live}}{\Delta \vdash H, x \mapsto s \text{ live}} \; (s \neq \overset{\phi}{data}(\rho))$$

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

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

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

**Figure 6: Well-Formed Machine States**

## 3.1 Operational Semantics

In order to define the operational semantics, we will need to define some additional notation. We require that no variable appear more than once in the domain of the store. Thus, the notation $H, x \mapsto s$ implicitly requires that $x$ not be in the domain of $H$. Similarly, the notation $H_1, H_2$ for the concatenation of two stores is undefined unless the domains of $H_1$ and $H_2$ are disjoint. The operation $H(x)$ selects the object at address $x$ from store $H$. If $x$ does not appear in the store then the operation is undefined.

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 $\overset{1}{-}$ for linear objects) implement this behavior.

$$\begin{array}{rcl} H \dot{-} x &=& H \\ (H, x \mapsto s, H') \overset{1}{-} x &=& H, H' \end{array}$$

The operational semantics for the language is given by a mapping from machine states to machine states. This mapping is presented in Figure 7. In general, an introduction form is evaluated by choosing a fresh address[9] 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 penultimate rule in Figure 7 explains how to evaluate a `let!` expression. It removes the linear copy of the data structure associated with region $\rho$ from the store and replaces it with an intuitionistic copy at a fresh address $z$. At the same time, the current stack $S$ is extended with the evaluation context for a let expression, and this new stack is wrapped with the special `let!` stack form. In summary, the final stack is:

$$\texttt{let!}\; y = z \;\texttt{in}\; (\texttt{let}\; x = \Box \;\texttt{in}\; e_2, S)$$

The purpose of this construction is to preserve the information that intuitionistic references to $\rho$ do not appear in the stack ($\texttt{let}\; x = \Box \;\texttt{in}\; e_2, S$). The special `let!` construct does this by preserving the information that the stack is well-formed in a context that is region-closed with respect to $\rho$. The typing rule for the `let!` stack form makes this idea precise. The closure condition on the stack justifies the removal of the intuitionistic region data structure from the store once the current expression has been evaluated.

The last operational rule eliminates the *intuitionistic* region $\rho$ from the store. It replaces the reference to $\rho$ with a dummy value (we use unit) and extends the store with a fresh reference to a linear copy of $\rho$:

$$\begin{array}{c}(\Delta; H_1, z \mapsto data(\rho), H_2; \texttt{let!}\; y = z \;\texttt{in}\; S; x) \longrightarrow \\ (\Delta; H_1, z \mapsto (\,), H_2, y \mapsto \overset{1}{data}(\rho); S; x)\end{array}$$

Ordinarily, if we were to replace an intuitionistic value with another value of a different type (say, if we replaced a function value with unit), there would be no guarantee that the resulting store would be well-formed. However, due to the closure conditions on the formation of function values and existential types, we can guarantee that this replacement is sound. The resulting store type is related to the original store type through the erasure function:

$$\begin{array}{lcll} erase_\rho(rgn(\rho)) &=& (\,) & \\ erase_\rho(\tau_1 \overset{\phi}{\times} \tau_2 \;\texttt{at}\; \rho') &=& erase_\rho(\tau_1) \overset{\phi}{\times} erase_\rho(\tau_2) \;\texttt{at}\; \rho' & \\ erase_\rho(\exists \rho'.\tau) &=& \exists \rho'.erase_\rho(\tau) & (\text{if } \rho' \neq \rho) \\ erase_\rho(\tau) &=& \tau & (\text{otherwise}) \end{array}$$

Notice that the structure of $erase_\rho(\tau)$ follows the structure of $closed_\rho(\tau)$ exactly and that neither need recurse into the structure of function types (due to the closure requirements on function formation). We lift the definition of erasure pointwise to contexts $\Gamma$. Now we can prove that the potentially dangerous replacement of $rgn(\rho)$ by $(\,)$ does lead to a well-formed store, albeit one with type $erase_\rho(\Gamma)$:

LEMMA 1. *If* $\Delta' \vdash (H, y \mapsto data(\rho), H') : \Gamma$ store *and* $\Delta \vdash (H, y \mapsto data(\rho), H')$ live *and* $\Delta' \supseteq \Delta$ *then* $\Delta' \vdash (H, y \mapsto (\,), H') : erase_\rho(\Gamma)$ store.

This lemma allows us to show that the store remains well-formed when the operational rule for `let!` is executed. However, we must also show that the stack and variable $x$ remain well-formed. The key to this proof is that when a type $\tau$ (or context $\Gamma$) is region-closed, it is equal to its erasure:

LEMMA 2. *If* $closed_\rho(\tau)$ *then* $erase_\rho(\tau) = \tau$.

Therefore, using the closure conditions on the stack $S$ and the variable $x$ implied by the typing rule for `let!` we

[9] 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.

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

$$(\Delta; H; S; E[e]) \longrightarrow (\Delta; H; E, S; e)$$
$$\text{if } e \text{ not a variable}$$

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

$$(\Delta; H; S; (\,)) \longrightarrow (\Delta; H, x \mapsto (\,); S; x)$$

$$(\Delta; H; S; (x; e)) \longrightarrow (\Delta; H; S; e)$$
$$\text{if } H(x) = (\,)$$

$$(\Delta; H; S; x_1 \stackrel{\phi}{\times} x_2 \texttt{ at } x_3) \longrightarrow (\Delta; H, y \mapsto \langle x_1 \stackrel{\phi}{\times} x_2 \rangle_\rho; S; y)$$
$$\text{if } H(x_3) = \mathit{data}(\rho) \text{ and } \rho \in \Delta$$

$$(\Delta; H; S; \texttt{let } x_1 \times x_2 = y \texttt{ in } e) \longrightarrow$$
$$(\Delta; H \stackrel{\phi}{-} y; S; e[x'_1, x'_2/x_1, x_2])$$
$$\text{if } H(y) = \langle x'_1 \stackrel{\phi}{\times} x'_2 \rangle_\rho \text{ and } \rho \in \Delta$$

$$(\Delta; H; S; \lambda[\Delta']x{:}\tau \stackrel{\phi}{\to} e \texttt{ at } y) \longrightarrow$$
$$(\Delta; H, z \mapsto \langle \lambda[\Delta']x{:}\tau \stackrel{\phi}{\to} e \rangle_\rho; S; z)$$
$$\text{if } H(y) = \mathit{data}(\rho) \text{ and } \rho \in \Delta$$

$$(\Delta; H; S; x[\Delta_a]\ x_a) \longrightarrow (\Delta; H \stackrel{\phi}{-} x; S; e[\Delta_a/\Delta_f][x_a/x_f])$$
$$\text{if } H(x) = \langle \lambda[\Delta_f]x_f{:}\tau \stackrel{\phi}{\to} e \rangle_\rho \text{ and } \rho \in \Delta$$

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

$$(\Delta; H; S; \texttt{unpack } \rho, y = x \texttt{ in } e) \longrightarrow$$
$$(\Delta; H \stackrel{1}{-} x; S; e[\rho'/\rho][y'/y])$$
$$\text{if } H(x) = \texttt{pack}[\rho', y'] \texttt{ as } \exists\rho.\tau$$

$$(\Delta; H; S; \texttt{alloc } x) \longrightarrow$$
$$(\Delta, \rho; H, y \mapsto \stackrel{1}{\mathit{data}}(\rho), z \mapsto \texttt{pack}[\rho, y] \texttt{ as } \exists\rho.\stackrel{1}{\mathit{rgn}}(\rho); S; z)$$
$$\text{if } H(x) = (\,) \text{ and } \rho \notin \Delta \cup \mathrm{FV}(H) \cup \mathrm{FV}(S)$$

$$(\Delta_1, \rho, \Delta_2; H; S; \texttt{free } x) \longrightarrow (\Delta_1, \Delta_2; H \stackrel{1}{-} x, y \mapsto (\,); S; y)$$
$$\text{if } H(x) = \stackrel{1}{\mathit{data}}(\rho)$$

$$(\Delta; H; S; \texttt{let } x = x' \texttt{ in } e) \longrightarrow (\Delta; H; S; e[x'/x])$$

$$(\Delta; H; S; \texttt{let! } (y)\ x = e_1 \texttt{ in } e_2) \longrightarrow$$
$$(\Delta; H \stackrel{1}{-} y, z \mapsto \mathit{data}(\rho);$$
$$\texttt{let! } y = z \texttt{ in } (\texttt{let } x = \Box \texttt{ in } e_2, S); e_1[z/y])$$
$$\text{if } H(y) = \stackrel{1}{\mathit{data}}(\rho)$$

$$(\Delta; H_1, z \mapsto \mathit{data}(\rho), H_2; \texttt{let! } y = z \texttt{ in } S; x) \longrightarrow$$
$$(\Delta; H_1, z \mapsto (\,), H_2, y \mapsto \stackrel{1}{\mathit{data}}(\rho); S; x)$$

**Figure 7: Operational Semantics**

are able to prove that $S$ and $x$ are still well-typed in the new machine state, and equally importantly, have the same type. Thus, the well-formedness of the abstract machine is preserved during this operational step.

## 3.2 Properties of the Core Language

We have proven a type soundness theorem for our core language. Given the recent research on proving soundness of Tofte and Talpin's region calculus [31, 13, 5] it should come as no surprise that we were able to apply syntactic techniques to the problem.

To state our Type Soundness theorem, we will define the *stuck states*. A state $\Sigma$ is *stuck* if $\Sigma$ is not a terminal state of the form $(\Delta; H; \cdot; x)$ and there is no state $\Sigma'$ such that $\Sigma \longrightarrow \Sigma'$. We also use the notation $\stackrel{*}{\longrightarrow}$ to denote the reflexive and transitive closure of $\longrightarrow$.

Theorem 3 (Type Soundness). *If* $\vdash \Sigma : \tau$ program *and* $\Sigma \stackrel{*}{\longrightarrow} \Sigma'$ *then* $\Sigma'$ *is not stuck.*

# 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. 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 [6] 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.[10] Here, for simplicity, we concentrate exclusively on reference-counted regions, which we give type $\stackrel{\#}{\mathit{rgn}}(\rho)$. The new type of reference-counted regions belongs to the class $L$ of linear objects — implicit contraction or weakening of assumptions with this type is not admissible.

We extend the language of expressions with operations to allocate reference-counted regions, explicitly increment reference counts, and explicitly decrement the count (and

[10]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 techniques similar to those of previous sections which ensure that only intuitionistic objects appear inside of intuitionistic objects.

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

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

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Delta; \Gamma_1 \vdash e : \overset{\#}{rgn}(\rho) \quad \Delta; \Gamma_2, x{:}\overset{\#}{rgn}(\rho), y{:}\overset{\#}{rgn}(\rho) \vdash e' : \tau'}{\Delta; \Gamma \vdash \texttt{let}\ x, y = \texttt{inc}\ e\ \texttt{in}\ e' : \tau'}$$

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

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 \\ \Delta; \Gamma_1 \vdash y : \overset{\#}{rgn}(\rho) \\ \Delta; \Gamma_2, y{:}rgn(\rho) \vdash e_1 : \tau_1 \\ \Delta; \Gamma_3, y{:}\overset{\#}{rgn}(\rho), x{:}\tau_1 \vdash e_2 : \tau_2\end{array}}{\Delta; \Gamma \vdash \texttt{let!}\ (y)\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ (closed_\rho(\tau_1))$$

**Figure 8: Reference Counting Constructs**

deallocate the region when the count reaches zero):

$$e \ ::= \ \cdots \mid \texttt{alloc}^{\#}\ e \mid \texttt{let}\ x, y = \texttt{inc}\ e\ \texttt{in}\ e' \mid \texttt{dec}\ e$$

Figure 8 defines additional rules for type checking expressions.

In the previous sections, the `let!` 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. 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.

*Example.* To demonstrate our new reference counting operations, we will reuse our previous *Pair* example, but this time rather than allocating two linear regions, we will only allocate a single reference-counted region. The *Pair* function itself is unchanged, except for its type, which specifies that it expects the *gen* function to be linear and to return a reference-counted region. The code for the function follows. We use the abbreviation $\tau_\#$ for the type $\exists \rho'. \overset{\#}{rgn}(\rho')$ and $\tau_{res}$ for $\exists \rho'. \overset{\#}{rgn}(\rho') \overset{1}{\times} (int \times int\ \texttt{at}\ \rho')\ \texttt{at}\ \rho$.

$$\begin{array}{l}\lambda[\rho](x{:}int, gen{:}() \overset{1}{\to} \tau_\#, r{:}rgn(\rho)) \to \\ \quad \texttt{unpack}\ \rho', r' = gen\ ()\ \texttt{in} \\ \quad \texttt{let!}\ (r')\ y = x \times x\ \texttt{at}\ r'\ \texttt{in} \\ \quad \texttt{pack}[\rho', r' \overset{1}{\times} y\ \texttt{at}\ r]\ \texttt{as}\ \tau_{res}\end{array}$$

The code that calls the *Pair* function allocates a reference-counted region $r$ and then increments the reference count, creating a second reference $r'$. This second reference is stored in *gen*'s closure. When the *Pair* function is called, we use the `let!` operator to temporarily allow more references to $r$ then there are reference counts. At this point, there is a reference count of two (due to the single `inc` instruction), but three references to $r$: one reference to $r$ is in *gen*'s closure, a second reference is an argument to *Pair* and a third reference is retained by the calling context. When *Pair* returns, the reference count is decremented to 0 and the region is deallocated.

$$\begin{array}{ll}\texttt{unpack}\ \rho, r & = \texttt{alloc}^{\#}\ ()\ \texttt{in} \\ \texttt{let}\ r, r' & = \texttt{inc}\ (r)\ \texttt{in} \\ \texttt{let}\ gen & = (\lambda()\overset{1}{\to} \texttt{pack}[\rho, r']\ \texttt{as}\ \tau_\#)\ \texttt{in} \\ \texttt{let!}\ (r)\ x_{res} & = Pair[\rho](17, gen, r)\ \texttt{in} \\ \texttt{unpack}\ \rho', z & = x_{res}\ \texttt{in} \\ \texttt{let!}\ (r)\ r' \overset{1}{\times} y & = z\ \texttt{in} \\ \texttt{let!}\ (r')\ x \times x' & = y\ \texttt{in} \\ \texttt{dec}\ (r'); & \\ \texttt{dec}\ (r); & \\ x + x' & \end{array}$$

## 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 [27] 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 [33] 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 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.

To demonstrate these ideas, we introduce a type for lists: $\tau \overset{\phi}{list}\ \texttt{at}\ \rho$. Like other data structures such as pairs and closures, intuitionistic lists are constrained so that they do not contain linear objects.

There are three lists expressions. The expression $\overset{\phi}{[\,]}_\tau\ \texttt{at}\ e$ introduces an empty list with type $\tau$ in the region designated by $e$. The expression $\overset{\phi}{\texttt{cons}}(e_1, e_2)\ \texttt{at}\ e_3$ prepends $e_1$ to the list $e_2$, in the region designated by $e_3$. The case construct $\texttt{case}\ e_1\ \texttt{of}\ [\,] \Rightarrow e_2 \mid (x, y) \Rightarrow e_3$ follows the first branch if $e_1$

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

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

$$\frac{\begin{array}{cc} \Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3 & \Delta;\Gamma_1 \vdash e_1 : \tau \\ \Delta;\Gamma_2 \vdash e_2 : \tau \overset{\phi}{list} \texttt{ at } \rho & \Delta;\Gamma_3 \vdash e_3 : rgn(\rho) \end{array}}{\Delta;\Gamma \vdash \overset{\phi}{\texttt{cons}}(e_1, e_2) \texttt{ at } e_3 : \tau \overset{\phi}{list} \texttt{ at } \rho}$$

$$\frac{\begin{array}{cc} \multicolumn{2}{c}{\Gamma = \Gamma_1 \bowtie \Gamma_2 \bowtie \Gamma_3} \\ \Delta;\Gamma_1 \vdash e_1 : \tau' \overset{\phi}{list} \texttt{ at } \rho & \Delta;\Gamma_2 \vdash z : \overset{\phi}{rgn}(\rho) \\ \Delta;\Gamma_3 \vdash e_2 : \tau & \Delta;\Gamma_3, x{:}\tau', y{:}\tau' \overset{\phi}{list} \vdash e_3 : \tau \end{array}}{\Delta;\Gamma \vdash \texttt{case } e_1 \texttt{ of } [\,] \Rightarrow e_2 \mid (x,y) \Rightarrow e_3 : \tau}$$

**Figure 9: Well-Formed List Constructs**

is the empty list and the second branch otherwise. Figure 9 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.However, the elements of the list may inhabit different regions. For example, a linear list of lists might be given the following type:

$$\exists \rho. \overset{1}{rgn}(\rho) \overset{1}{\times} ((\,) \overset{\cdot}{list} \texttt{ at } \rho) \overset{1}{list}$$

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 can 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:

$$\exists \rho. \overset{\#}{rgn}(\rho) \overset{1}{\times} ((\,) \overset{\cdot}{list} \texttt{ at } \rho) \overset{1}{list}$$

# 6. 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 [11] and Reynolds' syntactic control of interference [24]. Linear type systems were later studied by many researchers [19, 30, 1, 3, 6, 29, 34, 15]. Type and effect systems were introduced by Gifford and Lucassen [10] and they too have been explored by many others [17, 26, 28, 21].

More recently, a number of new linear type systems, or more generally, "substructural type theories," have been developed such as Kobayashi's quasi-linear types [18], Polakow and Pfenning's ordered type theory [22, 23]. There is also renewed interest in developing new logics that facilitate Hoare-style reasoning about heap-allocated data structures. Reynolds [25] 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.

There are close connections between this work and Walker, Crary and Morrisett's capability calculus [31]. The capability calculus used a notion of linearity to control region aliasing. Our current work has the advantage of being more expressive in a number of ways (it accommodates first-class regions, heterogeneous data structures and reference counting). However, the bounded quantifiers of the capability calculus make it possible to write continuation-passing programs that we cannot write with the lexically-scoped `let!` operator (see [31] for a detailed explanation). It seems likely that there is a way to combine the two approaches.

Makholm, Niss and Henglein [14] have had similar insights with respect to reference-counted regions as we have and are developing successful type inference techniques for a language with (second-class) reference-counted regions. Gay and Aiken [8, 9] 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 [7] are developing a new type-safe variant of C called Vault. They use a form of the capabilities mentioned above to control access to all sorts of program resources including memory regions. They have also developed effective local type inference techniques and have experience using their type system to enforce safety properties in device drivers. Currently, Vault tracks linear resources only and it might benefit from our `let!` operation to temporarily make linear resources intuitionistic.

Dan Grossman, Trevor Jim and Greg Morrisett are currently developing a second type-safe variant of C, called Cyclone. Currently, Cyclone relies upon a conservative garbage collector. However, together with Grossman *et al.*, we are exploring ways to incorporate some of the ideas described here into Cyclone.


# Acknowledgments

Many of the ideas in this paper arose from discussions with Greg Morrisett. We would like to thank Manuel Fähndrich, Dan Grossman and the anonymous reviewers for their comments on earlier versions of this work.


# 7. REFERENCES


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

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

[3] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In Shyamasundar, editor, *Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51, Bombay, 1993. Springer-Verlag.

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

[5] 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.

[6] 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.

[7] Rob DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, 2001. To appear.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *ACM Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, September 2001.

[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] 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.

[21] 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.

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

[23] 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.

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

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

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

[27] 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.

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

[29] 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.

[30] 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.

[31] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

[32] David Walker and Kevin Watkins. On linear types and regions. In *Workshop on Semantics, Program Analysis and Computing Environments For Memory Management (SPACE 2001)*, London, UK, January 2001. Available at http://www.cs.cmu.edu/-dpw/papers/.

[33] 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.

[34] 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.