

# Wandering through linear types, capabilities, and regions

François Pottier

May 24th, 2007



## Abstract

Here is a pointer. Which memory blocks can it possibly point to? If I write through it, who will observe this effect? In fact, am I allowed to write through it? Does it point to a valid piece of memory? Who owns this piece of memory?

This talk is not about original work of mine. It is an attempt to present a fraction of the many type systems that answer the above questions via notions of *linearity*, *capabilities*, or *regions*.

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

## Our starting point: linearity

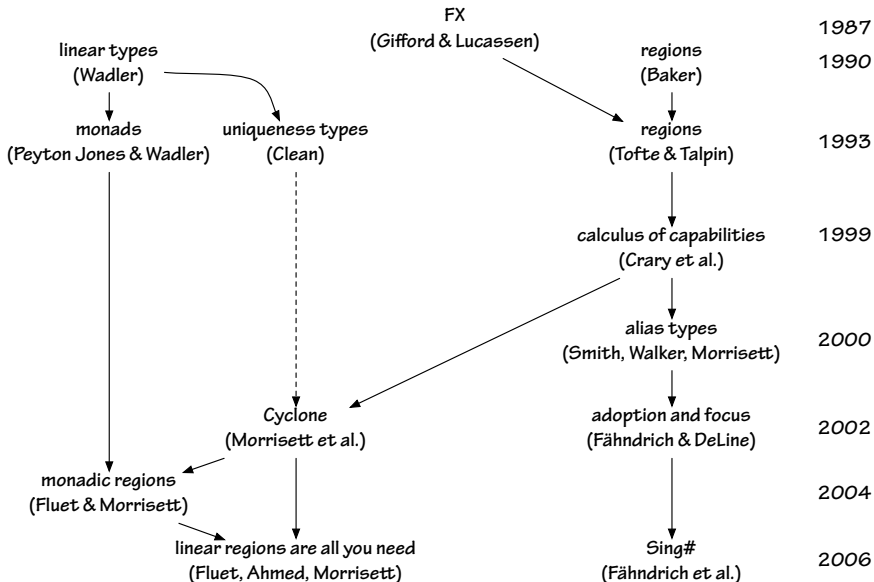
A long time ago, researchers in the programming languages community realised that *linearity* could help:

- model the physical world (input/output, ...),
- control when memory can be freed or re-used,
- reason about imperative programs.

For instance, Reynolds' "syntactic control of interference" [1978] was an affine  $\lambda$ -calculus [O'Hearn, 2003]. Wadler's linear type system [1990] was inspired by Girard's linear logic [1987].

But exactly *how* does one design a type system that helps reap these benefits? The historic path is tortuous...

# A (very partial) history



## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

Wadler [1990] noted that linearity seems to mean:

- *no duplication*: there exists at most one pointer to a linear value, so *destructive update* is safe;
- *no discarding*: every linear value has a use, which represents an explicit deallocation site, so *garbage collection is not required*;
- the real world (e.g., the state of the file system) could be modeled as a linear value.



# Linear versus nonlinear types

Wadler's type system introduces a *segregation* between linear and nonlinear types, based on their head constructor:

$$\tau ::= K \mid \tau \rightarrow \tau \quad \text{nonlinear}$$
$$iK \mid \tau \multimap \tau \quad \text{linear}$$

The base types  $K$  and  $iK$  are algebraic data types:

$$\begin{aligned} K &= C\tau \dots \tau + \dots + C\tau \dots \tau && \text{nonlinear} \\ iK &= iC\tau \dots \tau + \dots + iC\tau \dots \tau && \text{linear} \end{aligned}$$

In the linear case, the component types  $\tau$  may be any types, while in the nonlinear case they must be nonlinear: *a nonlinear data structure must not contain any linear components.*

It is worth pondering the meaning of Wadler's restriction: at any point in time, the heap consists of a *linear upper structure* (a forest), whose leaves form the boundary with a *nonlinear lower structure* (an arbitrary graph).

This restriction may seem natural, but is very limiting in practice. Time will elapse before it is lifted [[Fähndrich and DeLine, 2002](#)]...

▶ forward

In Wadler's system, a variable is well-typed only in a *singleton* environment:

$$x : \tau \vdash x : \tau$$

Function application (and other binary constructs) *split* the environment:

$$\frac{\Gamma_1 \vdash t_1 : \dots \quad \Gamma_2 \vdash t_2 : \dots}{\Gamma_1, \Gamma_2 \vdash t_1 t_2 : \dots}$$

The environment is not split at a case construct, though, since only one branch is actually executed.

*Weakening* and *contraction* are available at nonlinear types only:

$$\frac{\Gamma \vdash \dots \quad \tau \text{ is nonlinear}}{\Gamma, x:\tau \vdash \dots}$$

$$\frac{\Gamma, x:\tau, x:\tau \vdash \dots \quad \tau \text{ is nonlinear}}{\Gamma, x:\tau \vdash \dots}$$

A nonlinear closure cannot capture a linear value:

$$\frac{\Gamma, x:\tau \vdash u:\tau' \quad \Gamma \text{ is nonlinear}}{\Gamma \vdash \lambda x.u:\tau \rightarrow \tau'}$$

This is in keeping with the principle that a nonlinear value cannot contain a pointer to a linear value.

Wadler noted that the type system is *extremely restrictive*.

For instance, reading one element out of a linear array constitutes a use of the array, which forbids any further use of the array!

A workaround is to have the “get” operation return a (linear) pair of the value that was read and the (unchanged) array. This leads to a programming style where linear values are *explicitly threaded*. This style is heavy and over-sequentialised.



## Temporary aliasing with “let!”

Wadler noted that it is fine to *temporarily* view a linear value at a nonlinear type, which means that pointers to it can be duplicated, *provided only one pointer remains* in existence when the value recovers its original linear type.

For instance, a read/write, linear array can be temporarily turned into a nonlinear, read-only array; and, once the reading is over and only one pointer to the array remains, it can be turned back into a read/write, linear array.

The details of Wadler’s “let!” construct are extremely ad hoc, but the idea is essential: we will come back to it.

# A quiproquo over linearity

Is a linear value...

- a value that is *used* exactly once?
- a value to which there exists exactly one *pointer*?

## A quiproquo over linearity

The two interpretations differ: for instance, one is compatible with a *subtyping* relation that turns a nonlinear type into its linear equivalent, while the other isn't.

- if a value can be used as many times as desired, then certainly it is fine to use it exactly once;
- but, if there exist multiple pointers to a value, then it is *not sound* to pretend that there exists a unique pointer to it.

There has been some confusion about this issue in the literature, which I haven't quite sorted out. O'Hearn [[2003](#)] sheds some light.

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

## A glimpse of uniqueness types

Clean's uniqueness types [[Barendsen and Smetsers, 1995](#)] seem pretty close to Wadler's linear types.

Every type constructor carries a *mode*, which can be “unique”, “non-unique”, or a variable — avoiding duplication and paving the way to mode polymorphism and type inference.

As in Wadler's system, a nonlinear container cannot have linear components. This is expressed via constraints on modes.

# A glimpse of uniqueness types

A value that *might be accessed more than once* must have non-unique mode. (Unfortunately, the details of this constraint were apparently never published.)

There used to be an *ordering* relation on modes, so a unique value can be considered non-unique. This feature seems somewhat anecdotal, and has been recently dropped [[de Vries et al., 2007](#)].

A Clean user is given access to a unique value, the “world”, which represents the state of the entire operating system [Achten and Plasmeijer, 1995].

The world must be explicitly *threaded* by the user throughout the code, so a typical function has type  $\text{world}^\bullet \times \tau_1 \rightarrow \text{world}^\bullet \times \tau_2$ .

The world is a linear tuple of various state components (input/output channels, user interface elements, ...), so it can also be explicitly *decomposed* and *rebuilt* by the user.

The user can compose computations that operate on part of the world only, thus sometimes avoiding over-sequentialisation.

An update to a unique array can be performed *in place* by the Clean compiler.

Still, some hard limitations remain: for instance, an array of unique objects is essentially *unusable*, because reading an element duplicates it!

There is a painful workaround, based on an exchange, i.e., a swap operation. The same trick is exploited in Cyclone! [▶ forward](#)



Only *state* has to be linear: a *state transformer*, on the other hand, can safely be nonlinear.

Monads [Peyton Jones and Wadler, 1993] are a language design in which *state* is implicit and only *state transformers* are first-class values.

Monads in Haskell allow encapsulating interaction with the operating system, and allow in-place update of certain data structures, much like Clean's uniqueness types. They appear simpler, in terms of both syntax and types.

In principle, one could typecheck a monad's *implementation* using a linear type system, and typecheck its *clients* using a standard type system [Chen and Hudak, 1997]. Fluet *et al.* [2006] implement an indexed state monad in terms of linear types and regions.

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

Linearity is meant to enforce the absence of aliasing. Regions are intended to *control aliasing*: roughly speaking, they can be thought of as equivalence classes for a static, approximate “may alias” relation.

Baker [1990] noted that Hindley & Milner's type inference system provides free *aliasing information*. For instance, list concatenation:

$$(@) : \forall a. \text{list } a \rightarrow \text{list } a \rightarrow \text{list } a$$

has an inferred type whose memory representation could be written:

$$\forall a \beta_1 \beta_2 [\beta_1 = \text{list } a, \beta_2 = \text{list } a]. \beta_1 \rightarrow \beta_2 \rightarrow \beta_2$$

Baker concludes: the result of  $(@)$  cannot contain cells from its first argument (that is, unless the two arguments were already sharing, due to external constraints — we foresee the issue of *region aliasing*).

In order to make this aliasing information more explicit, Baker notes that one could annotate the list constructor with *region variables*. That would lead to:

$$(@) : \forall p_1 p_2. \text{list } p_1 a \rightarrow \text{list } p_2 a \rightarrow \text{list } p_2 a$$

Baker foresees at least two potential applications of regions, which he describes informally:

- if a function allocates intermediate data that does not alias with its result, then this data can be *collected* before the function returns;
- if a function allocates an array that does not alias with anything else, then this array can be *updated in place*.

A precise definition of when deallocation, or update in place, is safe, will be provided by later calculi, such as the calculus of capabilities and the calculus of alias types.

Baker did not clearly explain how to control or infer the *lifetime* of a region.

Drawing on ideas present in FX-87 and FX-91 [Gifford et al., 1992], Tofte and Talpin [1994] show how to infer region lifetimes that coincide with a *lexical scope*.

Roughly speaking, their analysis inserts statements of the form

$$\text{letregion } \rho \text{ in } e$$

and determines, at each memory allocation or memory access site, which region is involved.

## Two interpretations of regions

One operational interpretation of the construct “letregion  $\rho$  in  $e$ ” is to allocate a fresh region, bind it to the name  $\rho$ , evaluate  $e$ , and finally discard the region  $\rho$ , whose contents are lost.

In this interpretation, *regions exist at runtime*. Since their lifetimes coincide with lexical scopes, *regions form a stack*. No garbage collector is needed. This interpretation has been implemented and fine-tuned in the ML Kit.

In a slightly different interpretation, *regions have no existence at runtime*. A garbage collector is required. Regions are only a static mechanism for establishing non-aliasing facts. This interpretation is also useful; it is, in fact, more basic.



Since a region disappears when the scope of “letregion” is exited, one must ensure that no pointer into this region is ever accessed after this point. (This is an *escape analysis*.)

Tofte and Talpin’s system uses region-annotated types, and requires  $\rho$  to *not occur* in the environment or in the return type of “letregion  $\rho$  in  $e$ ”.

But that is *not enough*: if the result of “letregion  $\rho$  in  $e$ ” is a function, a pointer into  $\rho$  could be stored within its closure, yet would be invisible in its type.

Tofte and Talpin suggest annotating every function type with an *effect*, a set of regions that might be accessed when the function is invoked.

$$\tau ::= \tau \xrightarrow{e} \tau \mid \dots$$

This ensures that Tofte and Talpin's use of "letregion" is sound.

In order to compute this information, a typing judgement must associate with an expression not only a type, but also an effect. This yields a (by now standard) *type and effect system*.

## Strengths and weaknesses of Tofte and Talpin's work

Tofte and Talpin's system has *region polymorphism*, which, operationally, means that a function can be parameterized over one or several regions.

It also has *polymorphic recursion* in the region annotations (versus monomorphic recursion in the underlying type structure), for enhanced flexibility – each instance of a recursive function can create its own fresh, local region.

These features, and later improvements [[Aiken et al., 1995](#)], make the region inference system quite impressive.

A weakness is in the formalisation, which mixes concerns of inference and soundness, so that, for instance, a sound typing rule for “letregion” is never explicitly isolated.

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

The calculus of capabilities [Crary et al., 1999] is a low-level type and effect system. It allows reasoning about the *soundness* of a piece of code that explicitly allocates and deallocates regions, without any concern of *inference*.

It can serve as the target of a translation for Tofte and Talpin's system, and is in fact significantly more expressive, because it does not impose lexical region lifetimes.

## Basic operations on regions

The calculus has static *region identifiers*  $\rho$  as well as dynamic *region handles*. (Again, the latter are really optional.)

The basic operations are:

$\text{newrgrn } \rho, x$  allocates a fresh region

$\text{freergrn } v$  destroys a region

$x = h \text{ at } v$  allocates an object within a region

There is a (singleton) type of *region handles*: “ $\text{rgnhandle } \rho$ ” is the type of a handle for a region whose static name is  $\rho$ .

There is a type of *tuples*: “ $\langle \tau, \dots, \tau \rangle$  at  $\rho$ ” is the type of a pointer to a tuple allocated within region  $\rho$ .

There is a type of *functions*: “ $\forall \Delta [C] \tau \rightarrow O$  at  $\rho$ ” is the type of a closure allocated within region  $\rho$ . The function is polymorphic with respect to the type/region/capability variables in  $\Delta$ , requires a *capability*  $C$  and an argument of type  $\tau$ , and does not return (the calculus is in CPS style).

*Possession* of a pointer of type  $\langle \tau, \dots, \tau \rangle$  at  $\rho$  does not imply *permission* to dereference that pointer. Indeed, since regions can be deallocated, there is no guarantee that dereferencing is safe.

Permission to access  $\rho$  is represented by a *capability*, written  $\{\rho\}$ . A function  $f$  can access  $\rho$  only if it holds the capability  $\{\rho\}$  – for instance, if  $\rho$  was freshly created by  $f$  itself, or if  $\{\rho\}$  was explicitly passed to  $f$ .

*Capabilities and effects are two sides of the same coin.* Capabilities are prescriptive, while effects are descriptive, but that is only a matter of presentation. (I do find the capability view somewhat more inspiring.)



To a first approximation, capabilities are as follows:

$$C ::= \epsilon \mid \emptyset \mid \{\rho\} \mid C, C$$

The “comma” operator is associative and commutative, but not idempotent: the equality  $\{\rho\} = \{\rho\}, \{\rho\}$  does not hold. There is *no weakening or contraction* of capabilities.

In other words, *capabilities are linear*: they cannot be duplicated. This is essential for soundness: duplicating  $\{\rho\}$  would allow one to destroy the region  $\rho$  and still hold a capability to it.

## Linear capabilities, nonlinear values

While *capabilities* are linear, *values are nonlinear* and can be duplicated at will.

In other words, it is fine to have multiple pointers to an object within a region, or multiple copies of a region handle, as long the right to access (and to destroy) the region remains unique.

This *key idea* leads to much greater flexibility than afforded by linear type systems, such as Wadler's, where “pointer” and “permission to dereference” are not distinguished.

Because the calculus of capabilities is in CPS style, it does not have a sequencing construct.

If it was a source-level calculus, the typing rule for sequencing would perhaps look like this:

$$\frac{\Gamma; C_1 \vdash t_1 \dashv \tau_1; C_2 \quad (\Gamma, x : \tau_1); C_2 \vdash t_2 \dashv \tau_2; C_3}{\Gamma; C_1 \vdash \text{let } x = t_1 \text{ in } t_2 \dashv \tau_2; C_3}$$

*Capabilities are threaded*, while *environments are shared*.

# Region allocation and deallocation

Region allocation binds  $\rho$  and *produces a capability*:

$$\frac{}{\Gamma; C \vdash \text{newrgn } \rho, x \dashv (\Gamma, \rho, x : \text{rgnhandle } \rho); C, \{\rho\}}$$

Conversely, region deallocation *consumes a capability*:

$$\frac{\Gamma \vdash v : \text{rgnhandle } \rho}{\Gamma; C, \{\rho\} \vdash \text{freergn } v \dashv \Gamma; C}$$

The name  $\rho$  still exists; dangling pointers into the region can still exist; but they can no longer be dereferenced.

The only capability available to a function's body is the capability transmitted by the caller. That is, *a (nonlinear) closure cannot capture a (linear) capability.*

A function call consumes the capability that is transmitted to the callee.

## What about region aliasing?

A function can be parameterized over multiple regions:

$$f = \Lambda \rho_1, \rho_2 \dots$$

Imagine  $f$  deallocates  $\rho_1$  and subsequently writes into  $\rho_2$ . *Could I break type soundness* by applying  $f$  twice to a single region  $\rho$ ?

## What about region aliasing?

The answer is *no*: in order to deallocate  $\rho_1$  and subsequently write into  $\rho_2$ ,  $f$  needs two capabilities,  $\{\rho_1\}$  and  $\{\rho_2\}$ . Because *capabilities are linear*, if  $\rho_1$  and  $\rho_2$  were instantiated with the same region, it would be impossible to provide the two capabilities that  $f$  requires. So, in such a case, *region aliasing is impossible*.

If, on the other hand,  $f$  does not require both  $\{\rho_1\}$  and  $\{\rho_2\}$ , then it is possible for  $\rho_1$  and  $\rho_2$  to be aliases. In that case, *region aliasing is possible*, and useful (see next)...

## What about region aliasing?

Let  $f$  be parameterized over two pointers in two possibly distinct regions:

$$f = \Lambda \rho_1, \rho_2. \lambda (x_1 : \langle \tau_1 \rangle \text{ at } \rho_1, x_2 : \langle \tau_2 \rangle \text{ at } \rho_2) \dots$$

Imagine  $f$  wishes to dereference both pointers. This seems to require the capabilities  $\{\rho_1\}$  and  $\{\rho_2\}$ , which, as we have seen, means that  *$f$  cannot be applied twice to a single region  $\rho$*  — yet, in this case, such an application would be *safe*.



Allocating, reading, or writing an object within a region requires the region to exist, but *does not require exclusive ownership* of the region. Only deallocation of a region requires exclusive ownership.

We introduce a weaker capability that reflects existence, but not ownership, of a region:

$$C ::= \dots \mid \{\rho^+\}$$

We set up the typing rules for memory allocation and memory access so as to require such a nonexclusive capability.

The fact these capabilities are nonexclusive is technically reflected by making them *nonlinear*:

$$\begin{aligned} \{\rho^+\} &= \{\rho^+\}, \{\rho^+\} \\ \{\rho^+\} &\leq \emptyset \end{aligned}$$

This allows our earlier function  $f$  to access two regions without needing to care whether the two regions are aliases:

$$f = \Lambda \rho_1, \rho_2. \lambda(\{\rho_1^+\}, \{\rho_2^+\}, x_1 : \langle \tau_1 \rangle \text{ at } \rho_1, x_2 : \langle \tau_2 \rangle \text{ at } \rho_2) \dots$$

Our original, exclusive capabilities could be made *affine* instead of linear:

$$\{\rho\} \leq \emptyset$$

This would be sound, but would allow memory leaks (i.e., forgetting to call “freern”), so it really makes sense only if regions have no dynamic interpretation.

## From linear to nonlinear capabilities...

An (exclusive) capability to access and destroy  $\rho$  can be weakened and turned into a (nonexclusive) capability to access  $\rho$ :

$$\{\rho\} \leq \{\rho^+\}$$

In doing so, one renounces some power, and, in exchange, one gains some flexibility.

When  $\{\rho\}$  is turned into  $\{\rho^+\}$ , the capability to free  $\rho$  is lost (for good).

Doesn't that render this axiom useless in practice?

What we would really like is a way of *temporarily* turning an exclusive capability into a nonexclusive one, and subsequently *recovering* the original capability.

In a source-level, non-CPS-style calculus, one could introduce a construct that weakens an exclusive capability within a lexical scope and *reinstates it upon exit*:

$$\frac{\Gamma; (C, \{\rho^+\}) \vdash e \dashv \Gamma; C' \quad \rho \# C'}{\Gamma; (C, \{\rho\}) \vdash \text{rgnalias } \rho \text{ in } e \dashv \Gamma; (C', \{\rho\})}$$

This would be safe:  $\{\rho^+\}$  cannot possibly escape, because *capabilities cannot be stored*, a crucial point, to which I return later on. [▶ forward](#)

Note the connection to Wadler's "let!": a linear "thing" is temporarily turned into a nonlinear "thing". Here, the thing is a capability, as opposed to a value. This simplifies matters significantly.

In CPS style, this effect can be achieved by combining the weakening axiom  $\{\rho\} \leq \{\rho^+\}$  with *bounded quantification over capabilities* – a really nice trick.

The idea is to turn the expression  $e$  of the previous slide into a function of type

$$\forall \epsilon[\epsilon \leq \{\rho^+\}]. (\epsilon, \dots, k : (\epsilon, \dots) \rightarrow O) \rightarrow O$$

and to instantiate the capability variable  $\epsilon$  with  $\{\rho\}$  at the call site. The callee then holds  $\{\rho\}$ , under the name  $\epsilon$ , and can transmit it to its continuation  $k$ . However, because  $\epsilon$  is abstract, the callee can effectively only exploit  $\{\rho^+\}$ .

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References



## From regions to single objects

The calculus of capabilities groups objects within regions, and keeps track of *region ownership*, and *region aliasing*, via capabilities.

Alias types [Smith et al., 2000, Walker and Morrisett, 2000], which is directly inspired by the calculus of capabilities, has no regions at all, and keeps track of *object ownership*, and *object aliasing*, via capabilities.

Object allocation (resp. deallocation) will produce (resp. consume) a linear capability for a single object, whereas reading or writing an object will require a weaker, nonlinear capability.

# Incorporating types within capabilities

In the calculus of capabilities, a capability  $\{\rho\}$  or  $\{\rho^+\}$  mentions only a region's name. The type of a pointer into the region, e.g.  $\langle\tau\rangle$  at  $\rho$ , provides the type of the object that is pointed to.

With alias types,  $\rho$  is a static name for *a single location*. A capability mentions both a location *and the type of its content*, while a pointer type mentions a location only:

$$\begin{aligned} C &::= \emptyset \mid \{\rho \mapsto \tau\} \mid \{\rho \mapsto \tau\}^+ \mid C, C \\ \tau &::= \dots \mid \text{ptr } \rho \end{aligned}$$

Why is it useful to move the type of the location into the capability?

Because the type is in the capability, a linear capability can be viewed as permission not only to deallocate the object, but also to *change its type*.

An operation that modifies an object's type is known as a *strong update*.

# Weak versus strong update

There are two typing rules for writing to a memory block:

Weak Update

$$\frac{\Gamma \vdash v_1 : \text{ptr } \rho \quad \Gamma \vdash v_2 : \tau \quad C \leq \{\rho \mapsto \tau\}^+}{\Gamma; C \vdash v_1 := v_2 \dashv \Gamma; C}$$

Strong Update

$$\frac{\Gamma \vdash v_1 : \text{ptr } \rho \quad \Gamma \vdash v_2 : \tau_2}{\Gamma; \{\rho \mapsto \tau_1\} \vdash v_1 := v_2 \dashv \Gamma; \{\rho \mapsto \tau_2\}}$$

Strong update *modifies a (linear) capability*. There can be *multiple values* of type  $\text{ptr } \rho$  around. Their type remains  $\text{ptr } \rho$ , but the meaning of that type changes.

Again, with linear capabilities and nonlinear values, there is *no direct restriction* on the use or copying of pointers.

Strong update allows:

- non-atomic initialization of memory blocks;
- delayed initialization; destination-passing style [Minamide, 1998];
- recycling memory blocks [Sobel and Friedman, 1998];
- perhaps most important: *changing one's view of memory*, without actually writing to a block (developed in several of the forthcoming slides).

# Changing one's view of memory

Here are typing rules for sums that exploit strong update  
[Walker and Morrisett, 2000]:

$$\{\rho \mapsto \tau_1\} \leq \{\rho \mapsto \tau_1 \cup \tau_2\}$$

$$\frac{\begin{array}{c} \Gamma \vdash v : \text{ptr } \rho \\ \Gamma; C, \{\rho \mapsto \langle \text{int } 1, \tau_1 \rangle\} \vdash i_1 \\ \Gamma; C, \{\rho \mapsto \langle \text{int } 2, \tau_2 \rangle\} \vdash i_2 \end{array}}{\Gamma; C, \{\rho \mapsto \langle \text{int } 1, \tau_1 \rangle \cup \langle \text{int } 2, \tau_2 \rangle\} \vdash \text{case } v \text{ of } i_1 \mid i_2}$$

One can similarly introduce or eliminate a recursive type, an existential type, or a pair of a capability and a type (what are those? read on).

Again, a key point is that *one* capability is modified, but *all* values of type  $\text{ptr } \rho$  are (indirectly) affected.

## Towards storing capabilities

So far, capabilities can be manipulated only in limited ways. There is a notion of a *current set of capabilities*, which is transformed by primitive operations such as allocation, deallocation, reading and writing, and transmitted from caller to callee, and back, at function invocation sites.

But (in the absence of regions) a finite set of capabilities can only describe *a finite portion* of the store!

How do we describe, say, a linked list, or a binary tree?

How about something along these lines?

$$\text{list } a = \langle \text{int } 1 \rangle \cup \exists \rho. [\{\rho \mapsto \text{list } a\}] \langle \text{int } 2, a, \text{ptr } \rho \rangle$$

$$\text{tree } a = \langle \text{int } 1 \rangle \cup \exists \rho_1, \rho_2. [\{\rho_1 \mapsto \text{tree } a\}, \{\rho_2 \mapsto \text{tree } a\}] \\ \langle \text{int } 2, a, \text{ptr } \rho_1, \text{ptr } \rho_2 \rangle$$

The type  $\text{list } a$  describes a *linked list cell*. The type  $\text{ptr } \rho$ , together with the capability  $\{\rho \mapsto \text{list } a\}$ , describes a *pointer* to such a cell.

Every list cell *contains a full capability* to the next cell. In other words, perhaps more intuitive, *each cell owns its successor cell*.

These definitions require *recursive types*, *existential types*, and *pairs of a capability and a type*.



## Towards storing capabilities

So far, I have insisted on the flexibility offered by the distinction between *capabilities*, which can be linear or nonlinear, and *values*, which are nonlinear.

Sometimes, though, it is useful to *package a capability and a value together*, and possibly to *store* such a package within a memory cell.

In addition to the encodings of lists and trees (already shown), this allows *recovering standard linear types* (à la Wadler, for instance), via the following encoding:

$$\tau_1 \otimes \tau_2 = \exists \rho. [\{\rho \mapsto \langle \tau_1, \tau_2 \rangle\}] \text{ptr } \rho$$

A standard “linear pair” is a pointer to a memory block that holds a pair, packaged together with a unique capability to access that block.

How do I construct or destruct a pair of a capability and a value?  
*One beautiful axiom* is enough:

$$\{\rho \mapsto \tau\}, C = \{\rho \mapsto [C]\tau\}$$

Such a capability rearrangement has no operational significance. It is a *memory view change*: we are switching between “*I own capability C*” and “*capability C is owned by block  $\rho$* ”.

This requires  $\tau ::= \dots \mid [C]\tau$ . Simple, right? (Hmm...)

How do I hide or reveal a region name? Again, just one axiom:

$$\exists \rho'. \{ \rho \mapsto \tau \} = \{ \rho \mapsto \exists \rho'. \tau \}$$

This can be informally understood as switching between “I know about a memory block, which I call  $\rho'$ ” and “block  $\rho$  knows about a memory block, which it (privately) calls  $\rho'$ ”. Again, this is a memory view change.

This requires  $C ::= \dots \mid \exists \rho. C$ . Simple, right? (Yes. Really.)

## Exploiting a linear value: borrowing

Imagine  $r$  is a pointer to a ref cell, allocated in region  $\rho$ , containing a pointer to a linear pair, encoded as before:

	type environment	capability
	$\rho; r : \text{ptr } \rho$	$\{\rho \mapsto \exists \rho'. [\{\rho' \mapsto \langle \tau_1, \tau_2 \rangle\}] \text{ptr } \rho'\}$
(unpack)	$\rho; r : \text{ptr } \rho; \rho'$	$\{\rho \mapsto [\{\rho' \mapsto \langle \tau_1, \tau_2 \rangle\}] \text{ptr } \rho'\}$
(fetch)	$\rho; r : \text{ptr } \rho; \rho'$	$\{\rho \mapsto \text{ptr } \rho'\}, \{\rho' \mapsto \langle \tau_1, \tau_2 \rangle\}$

At this point,  $!r$  has type  $\text{ptr } \rho'$ , a *nonlinear* pointer type: *the address of the linear pair can be read and copied at will.*

This offers a mechanism for *turning a linear value into a nonlinear one*, and back, since the axioms are reversible. This serves the same purpose as Wadler's "let!" [◀ back](#)

When the linear pair is re-packaged, the capability  $\{\rho' \mapsto \langle \tau_1, \tau_2 \rangle\}$  disappears, so any remaining aliases become unusable.

It is essential that  $C$  is moved into, or out of, a linear capability.  
The following variant of the axiom is *unsound* (why?):

$$\{\rho \mapsto \tau\}^+, C = \{\rho \mapsto [C]\tau\}^+$$

## Storing capabilities: linearity caveat 2

Furthermore, and less obviously, it is necessary to add a side condition (not previously shown) that  $C$  itself is linear. The following scenario is *unsound*:

- allocate an object, producing  $\{\rho \mapsto \tau\}$ ;
- temporarily weaken this capability to  $\{\rho \mapsto \tau\}^+$ ;
- store the weakened capability into memory;
- exit the temporary weakening and reinstate  $\{\rho \mapsto \tau\}$ ;
- deallocate the object, consuming  $\{\rho \mapsto \tau\}$ ;
- fetch the previously stored, weakened capability and attempt to exploit it.

This risk was mentioned earlier. [◀ back](#)

## Storing capabilities: linearity caveat 3

If  $C$  is a linear capability, then  $[C]\tau$  should be considered a linear type.

That is, the following rules (among others) are *unsound* when  $\tau$  is the type of a capability-value package:

$$\begin{array}{c} \text{Var} \\ x : \tau \vdash x : \tau \end{array} \qquad \begin{array}{c} \text{Read} \\ \Gamma \vdash v : \text{ptr } \rho \quad C \leq \{\rho \mapsto \tau\}^+ \\ \hline \Gamma; C \vdash x = !v \dashv \Gamma, x : \tau; C \end{array}$$



# Linear versus nonlinear types

A solution is to draw a distinction between *nonlinear types*  $\tau$  and *possibly linear types*  $\sigma$ :

$$\begin{aligned} C &::= \emptyset \mid \{\rho \mapsto \sigma\} \mid \{\rho \mapsto \sigma\}^+ \mid C, C \mid \exists \rho. C \\ \tau &::= \top \mid \text{ptr } \rho \mid \sigma \rightarrow \sigma \\ \sigma &::= \langle \tau, \dots, \tau \rangle \mid [C]\sigma \mid \exists \rho. \sigma \end{aligned}$$

*Values have nonlinear types:* a type environment maps  $x$ 's to  $\tau$ 's.

*Memory blocks have possibly linear types:* a capability can be of the form  $\{\rho \mapsto \sigma\}$ . Reading and writing to memory is restricted to nonlinear types. Switching between  $\{\rho \mapsto \sigma\}$  and  $\{\rho \mapsto \langle \tau, \dots, \tau \rangle\}$  is made possible by the axioms already studied: [◀ back](#)

$$\begin{aligned} \{\rho \mapsto \sigma\}, C &= \{\rho \mapsto [C]\sigma\} \\ \exists \rho'. \{\rho \mapsto \sigma\} &= \{\rho \mapsto \exists \rho'. \sigma\} \end{aligned}$$

(Phew! If you survived this far, you should be good from now on.)

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

The calculus of alias types, especially in its more advanced variant [Walker and Morrisett, 2000], is quite complex and powerful. Is there anything it *cannot* do?

Fähndrich and DeLine [2002] noted two problems, the first of which is:

- paradoxically, *aliasing is disallowed*, that is, two pointers to two distinct memory blocks cannot possibly have the same type! In other words, ptr  $\rho$  is a singleton type.

As a solution, Fähndrich and DeLine propose *adoption*.

## Why is aliasing disallowed?

When an object is allocated, a fresh location  $\rho$ , as well as a fresh capability, are produced:

$$\text{malloc} : () \rightarrow \exists \rho. [\{\rho \mapsto \langle T \rangle\}] \text{ptr } \rho$$

(This is allocation without initialization:  $T$  is the type of junk.)

Allocating two objects, and performing the unpacking administration, yields two capabilities  $\{\rho_1 \mapsto \langle T \rangle\}$  and  $\{\rho_2 \mapsto \langle T \rangle\}$  and *two pointers of distinct types*  $\text{ptr } \rho_1$  and  $\text{ptr } \rho_2$ .

There is no way of coercing these pointers to a common (nonlinear) type!

If one is willing to perform allocation and initialization atomically, then one can interpret  $\rho$ 's as *regions*, instead of *locations*, and *allocate new objects within an existing region*:

$$\text{ref} : \forall \rho. [\{\rho \stackrel{\omega}{\mapsto} \sigma\}] \sigma \rightarrow [\{\rho \stackrel{\omega}{\mapsto} \sigma\}] \text{ptr } \rho$$

This is saying, roughly: “show me a live region  $\rho$  where every block contains and owns a (possibly linear) content of type  $\sigma$ ; give me a capability-value package of type  $\sigma$ ; then I will allocate a new block within that region, initialize it, and return a pointer to it”.

## Shifting from locations to regions

Introducing this primitive operation into the calculus of alias types means that two distinct pointers can have a common type: *ptr is no longer a singleton type*, and aliasing becomes permitted.

## Shifting from locations to regions

This change also means that  $\rho$  now denotes a region, rather than a single location. This is reflected by adapting the language of capabilities:

$\{\rho \mapsto \sigma\}$	exclusive access to a single location	(linear)
$\{\rho \overset{\omega}{\mapsto} \sigma\}$	exclusive access to a region	(linear)
$\{\rho \overset{\omega}{\mapsto} \sigma\}^+$	shared access to a region	(nonlinear)

In the first case,  $\rho$  is both a location and a region: a *singleton region*. In the second and third cases,  $\rho$  is a region that possibly holds multiple objects.

The form  $\{\rho \mapsto \sigma\}^+$  would offer the same privileges as  $\{\rho \overset{\omega}{\mapsto} \sigma\}^+$ .



One can explain these new capabilities in terms of the *privileges* that they represent:

$\{\rho \mapsto \sigma\}$  allocate, read, (strongly) write, deallocate object  
 $\{\rho \overset{\omega}{\mapsto} \sigma\}$  allocate, read, (weakly) write objects, deallocate region  
 $\{\rho \overset{\omega}{\mapsto} \sigma\}^+$  allocate, read, (weakly) write objects

This connection between formal capabilities and actual privileges is wired in the typing rules.

Other interpretations are possible, depending on one's exact purpose.

Fähndrich and DeLine offer a finer-grained mechanism, in which *objects are born unique* and uninitialized and are later *adopted* by a region, which presumably was *created empty*:

$$\text{malloc} : () \rightarrow \exists \rho. [\{\rho \mapsto \langle \top \rangle\}] \text{ptr } \rho$$

$$\text{adopt} : \forall \rho_1, \rho_2. [\{\rho_1 \xrightarrow{\omega} \sigma\}, \{\rho_2 \mapsto \sigma\}] \text{ptr } \rho_2 \rightarrow [\{\rho_1 \xrightarrow{\omega} \sigma\}] \text{ptr } \rho_1$$

$$\text{newrgn} : () \rightarrow \exists \rho. [\{\rho \xrightarrow{\omega} \sigma\}] ()$$

Adoption consumes a singleton region. Its semantics is the identity.

*Adoption is forever.* Once an object is adopted, it can become aliased, so one abandons the ability to deallocate it — only the region can be deallocated as a whole.

The second problem that we now face, as noted by Fähndrich and DeLine [2002], is:

- a capability to a region of objects, each of which packs a linear capability, such as  $\{\rho \overset{\omega}{\mapsto} \sigma\}$ , is syntactically allowed, but *unusable*.

Indeed, an axiom of the form

$$\{\rho \overset{\omega}{\mapsto} \sigma\}, C = \{\rho \overset{\omega}{\mapsto} [C]\sigma\}$$

would make no sense at all (why?).

Suppose we hold the capability  $\{\rho \stackrel{\omega}{\mapsto} \sigma\}$  as well as a pointer  $x : \text{ptr } \rho$ .

Then, the object  $x$  *owns* a piece of memory, described by  $\sigma$ , and we would like to gain access to it, that is, to somehow “extract  $\sigma$  out of  $x$ ”.

However, *we cannot modify the capability*, because it also describes objects other than  $x$  ( $\rho$  denotes a region, not a single location).

So, *how do we prevent extracting  $\sigma$  twice* out of  $x$ ?

Fähndrich and DeLine note that, although we cannot modify the capability  $\{\rho \stackrel{\omega}{\mapsto} \sigma\}$ , we can *temporarily revoke it* while we are working with  $x$  and *reinstate* it afterwards.

Meanwhile, *a capability for exclusive access to  $x$* , under a fresh name  $\rho'$ , can be temporarily provided.

This is known as *focusing on  $x$* :

$$\text{focus} : \forall \rho. \left[ \{\rho \stackrel{\omega}{\mapsto} \sigma\} \right] \quad \text{ptr } \rho \rightarrow \\ \exists \rho'. \left[ \{\rho' \mapsto \sigma\}, (\{\rho' \mapsto \sigma\}) \multimap (\{\rho \stackrel{\omega}{\mapsto} \sigma\}) \right] \quad \text{ptr } \rho'$$

The capability  $\{\rho' \mapsto \sigma\}$  allows *all forms of strong update* at  $x$ , including grabbing the memory described by  $\sigma$ . However, *the region  $\rho$  remains inaccessible* until  $\{\rho' \mapsto \sigma\}$  is restored and surrendered.

Focus allows temporarily turning a nonlinear (shared) object into a linear object, at the cost of abandoning the right to access any of its potential aliases.

This seems rather closely related to the *restrict* keyword in C99 – a promise that an object's aliases, if they exist, will not be accessed [Foster and Aiken, 2001].

Thanks to focus, capabilities of the form  $\{\rho \stackrel{\omega}{\mapsto} \sigma\}$  become usable — that is, *a shared object can point to a linear object*, yet the system remains sound and expressive.

This eliminates one of the most fundamental restrictions of standard linear type systems! [◀ back](#)

For instance, the capability  $\{\rho \stackrel{\omega}{\mapsto} \text{node}\}$  and the definition:

$$\text{node} = \exists \rho'. [\{\rho' \mapsto \sigma\}] \langle \text{list}(\text{ptr } \rho), \text{ptr } \rho' \rangle$$

mean that the region  $\rho$  contains a set of nodes, each of which holds a list of (successor) nodes and a pointer to a per-node private block.

# What about type inference?

It might seem that, at this point, any hopes of type inference should be long abandoned.

In fact, Vault and Cyclone achieve a reasonable level of succinctness using syntactic defaults and local type inference.

CQual [[Foster et al., 2002](#)] performs an impressive amount of type and qualifier inference using advanced algorithmic techniques.



## Is there an application?

Have a look at Singularity OS, an experimental operating system written in *Sing#* [Fähndrich et al., 2006], an extension of C# with capabilities, alias types, focus, etc. (much of it under the hood).

The type system keeps track of:

- *memory ownership*, allowing components to safely interact via shared memory, without hardware protection;
- *simple, finite-state protocols*, preventing (for instance) a thread from attempting twice to acquire a single lock.

Quite exciting!

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

The programming language Cyclone [Swamy et al., 2006] provides programmers with fine-grained control over memory allocation (*where* are objects allocated?) and deallocation (*when* are objects deallocated?), without sacrificing safety. It is a *safe C*.

As far as Cyclone's static type system is concerned, *a region is a logical container* for objects.

At runtime, *a region is a physical container*, implemented by:

- a garbage-collected heap,
- a stack frame,
- a LIFO arena, i.e., a dynamically-sized region with lexical lifetime;
- a dynamic arena, i.e., a dynamically-sized region with unbounded lifetime.

Cyclone is a complex programming language. Simplified calculi [Fluet and Morrisett, 2006, Fluet et al., 2006] describe its foundations and discuss interesting connections between linearity, regions, and monads.

## A difference in perspective

The “alias types” line of work considers *static, linear capabilities* and *dynamic, non-linear values* as primitive, and, if desired, defines dynamic, linear values through an encoding [◀ back](#).

Cyclone adopts a different, more classic approach, somewhat analogous to Clean’s uniqueness types. In this approach, *dynamic, linear values* are considered primitive as well.

IMHO, the former approach seems more economical and elegant.

As in Clean, every pointer type carries a *qualifier*, which can be “unique”, “aliasable”, or a variable.

An analysis checks that *a unique pointer is consumed at most once*:

- copying or deallocating a pointer consumes it;
- reading or writing through a pointer does not consume it.

The analysis is automated within each procedure, and relies on user annotations (or defaults) at procedure boundaries.

# Borrowing a unique pointer

Because copying a pointer consumes it, local aliases of a unique pointer are not permitted.

In order to work around this limitation, a unique pointer can be temporarily *borrowed*, i.e., made non-unique.

Technically, this is done by *temporarily consuming* the unique pointer and *introducing a fresh region variable and capability*. There is a strong connection with “borrowing as unpacking”. [◀ back](#)

For convenience, borrowing can be inferred at function calls.



A handle to a dynamic region is encoded as a unique pointer. A dedicated “*open*” construct *temporarily consumes* the handle and *introduces a fresh capability*, making the region accessible.

“borrow” and “open” are clearly related — in fact, both correspond to unpacking in the “alias types” approach.

Contrary to standard linear type systems, [◀ back](#) Cyclone allows storing a unique pointer within a shared data structure.

Soundness is guaranteed via a quite horrible restriction: such a pointer can be read or written only via a *swap* operation, so no duplication occurs.

There is no analogue of focus! [◀ back](#)

For a number of reasons, the consumption analysis is *affine*, as opposed to linear, which means that it sometimes allows *memory leaks*.

*Even unique pointers can belong to regions*, which can serve as *safety nets* with respect to leaks: if, either intentionally or by mistake, a unique object is not individually freed, it is reclaimed when the region is deallocated.

Regions that support a choice between *individual* and *massive* deallocation are known as *reaps*.

In addition to unique and aliasable objects, Cyclone supports *reference counted* objects.

Copying or dropping a reference counted pointer is only permitted via *explicit operations*, which increment or decrement the counter.

The idea that linearity can enforce correct use of these operations is classic [[Walker, 2005](#)]. In Cyclone, unfortunately, the memory leak issue means that incorrect use is in fact possible!

A reference counted pointer can be “borrowed” too. This enables *deferred reference counting*, i.e., creating temporary aliases without updating the counters.

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types

Adoption and focus

Cyclone

## ③ Closing

## ④ References

# An impressive type-theoretic toolbox

The literature offers a variety of *elegant type-theoretic tools* for reasoning about aliasing, sharing, and ownership of memory, and, more generally, for *reasoning about imperative programs*.

This was only a very partial overview!

## What about separation logic?

There is a strong analogy between a *capability* and a separation logic *formula* [Reynolds, 2002].

A singleton capability  $\{\rho \mapsto \sigma\}$  is very much like a singleton heap formula. Capability conjunction is very much like separating conjunction. A function that *accepts* or *produces* a capability is very much like a function that carries a precondition or postcondition expressed in separation logic.

# What about separation logic?

Perhaps the choice is between:

- distinguishing a *capability-based type system*, possibly equipped with a form of *type inference*, on the one hand, and a *standard logic for proving programs*, possibly equipped with an *off-the-shelf theorem prover*, on the other hand;
- unifying the two, via *separation logic*, and equipping the logic either with a *dedicated theorem prover* or with an *encoding* towards a *standard logic*.



Thank you for your *focused attention!*

You may now *separate.*

## ① Introduction

## ② A tour

Wadler's linear types

Uniqueness types

Basic insights about regions

The calculus of capabilities

Alias types




Adoption and focus




Cyclone





## ③ Closing




## ④ References





(Most titles are clickable links to online versions.)





-  [Achten, P. and Plasmeijer, M. J. 1995.](#)  
The ins and outs of Clean I/O.  
*Journal of Functional Programming* 5, 1, 81–110.
-  [Aiken, A., Fähndrich, M., and Levien, R. 1995.](#)  
Better static memory management: improving region-based analysis of higher-order languages.  
*ACM SIGPLAN Notices* 30, 6 (June), 174–185.
-  [Baker, H. G. 1990.](#)  
Unify and conquer (garbage, updating, aliasing, ...) in functional languages.  
In *ACM Symposium on Lisp and Functional Programming (LFP)*.  
218–226.

-  Barendsen, E. and Smetsers, S. 1995.  
Uniqueness type inference.  
In *Programming Languages: Implementations, Logics, and Programs (PLILP)*. Lecture Notes in Computer Science, vol. 982. Springer Verlag, 189–206.
-  Chen, C.-P. and Hudak, P. 1997.  
Rolling your own mutable ADT—a connection between linear types and monads.  
In *ACM Symposium on Principles of Programming Languages (POPL)*. 54–66.
-  Crary, K., Walker, D., and Morrisett, G. 1999.  
Typed memory management in a calculus of capabilities.  
In *ACM Symposium on Principles of Programming Languages (POPL)*. 262–275.




-  de Vries, E., Plasmeijer, R., and Abrahamson, D. 2007.  
Equality based uniqueness typing.  
*In Trends in Functional Programming (TFP)*.
-  Fluet, M. and Morrisett, G. 2006.  
Monadic regions.  
*Journal of Functional Programming* 16, 4–5, 485–545.
-  Fluet, M., Morrisett, G., and Ahmed, A. 2006.  
Linear regions are all you need.  
*In European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 3924. Springer Verlag, 7–21.
-  Foster, J. S. and Aiken, A. 2001.  
Checking programmer-specified non-aliasing.  
Tech. Rep. UCB//CSD-01-1160, University of California, Berkeley.  
Oct.




-  Foster, J. S., Terauchi, T., and Aiken, A. 2002.  
Flow-sensitive type qualifiers.  
In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 1–12.
-  Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R., and Levi, S. 2006.  
Language support for fast and reliable message-based communication in Singularity OS.  
In *EuroSys*. 177–190.
-  Fähndrich, M. and DeLine, R. 2002.  
Adoption and focus: practical linear types for imperative programming.  
In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 13–24.

-  Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, J. W. 1992.  
Report on the FX-91 programming language.  
Tech. Rep. MIT/LCS/TR-531, Massachusetts Institute of Technology. Feb.
-  Girard, J.-Y. 1987.  
Linear logic.  
*Theoretical Computer Science* 50, 1, 1–102.
-  Minamide, Y. 1998.  
A functional representation of data structures with a hole.  
In *ACM Symposium on Principles of Programming Languages (POPL)*. 75–84.
-  O'Hearn, P. 2003.  
On bunched typing.  
*Journal of Functional Programming* 13, 4, 747–796.

-  Peyton Jones, S. L. and Wadler, P. 1993.  
Imperative functional programming.  
In *ACM Symposium on Principles of Programming Languages (POPL)*. 71–84.
-  Reynolds, J. C. 1978.  
Syntactic control of interference.  
In *ACM Symposium on Principles of Programming Languages (POPL)*. 39–46.
-  Reynolds, J. C. 2002.  
Separation logic: A logic for shared mutable data structures.  
In *IEEE Symposium on Logic in Computer Science (LICS)*. 55–74.
-  Smith, F., Walker, D., and Morrisett, G. 2000.  
Alias types.  
In *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 1782. Springer Verlag, 366–381.



-  Sobel, J. and Friedman, D. P. 1998.  
Recycling continuations.  
In *ACM International Conference on Functional Programming (ICFP)*.  
251–260.
-  Swamy, N., Hicks, M., Morrisett, G., Grossman, D., and Jim, T.  
2006.  
Safe manual memory management in Cyclone.  
*Science of Computer Programming* 62, 2 (Oct.), 122–144.
-  Tofte, M. and Talpin, J.-P. 1994.  
Implementation of the typed call-by-value  $\lambda$ -calculus using a  
stack of regions.  
In *ACM Symposium on Principles of Programming Languages (POPL)*. 188–201.

-  Wadler, P. 1990.  
Linear types can change the world!  
In *Programming Concepts and Methods*, M. Broy and C. Jones,  
Eds. North Holland.
-  Walker, D. 2005.  
Substructural type systems.  
In *Advanced Topics in Types and Programming Languages*, B. C.  
Pierce, Ed. MIT Press, Chapter 1, 3–43.
-  Walker, D. and Morrisett, G. 2000.  
Alias types for recursive data structures.  
In *Workshop on Types in Compilation (TIC)*. Lecture Notes in  
Computer Science, vol. 2071. Springer Verlag, 177–206.