A Theory of Linear Objects

Matthew Kehrt
University of Washington
mkehrt@cs.washington.edu

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

Abstract

Recently, linearity has been proposed as a mechanism for memory management, alias control, and typestate tracking. While linear type systems have been extensively studied in functional programming, their use in object-oriented systems has been limited to useful but ad-hoc annotation systems that track unique pointers.

In this paper, we study object-oriented linearity at level of foundational object calculi. Our system tracks not only linear objects (to which there may be only one pointer), but linear methods as well (which may be called at most once). Tracking linear objects allows us to ensure type safety for imperative, type-changing update to methods and imperative, type-changing dynamic inheritance. Because some aliasing is important in practical systems, our system supports linear and non-linear objects and methods and a novel region system that permits borrowed aliases to linear objects. To enforce safety, we allow type-changing modifications only on unborrowed linear objects, but permit such changes again when these borrowed references are no longer accessible. *Note: an earlier version of this paper appeared in the proceedings of FOOL '06. The current paper includes a simpler and cleaner formalism (with only methods, no lambdas) and significantly extends the previous system with borrowing via regions.*

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Languages, Reliability

Keywords Object calculi, Linear types

1. Introduction

Linear type systems ensure that a resource is used at most once. They have been proposed to verify that clients of an abstraction call its operations in a proper protocol order [10, 11], to enforce safety of manual memory management as in Cyclone [27], and to implement functional languages using more efficient imperative operations such as destructive update [30].

Linear types have been studied extensively in functional programming languages. For example, Wadler describes an elegant lambda calculus extension in which base types and functions may be either linear or non-linear [30]. Linear objects must be used exactly once; this implies that every linear function is called exactly once. In contrast, nonlinear objects can be freely shared, and so nonlinear functions can be called as often as needed.

A number of object-oriented type systems have used a restricted form of linearity known as uniqueness [7, 6, 2, 9, 11]. A unique pointer is one that may not be aliased. Unique pointers can benefit programmers by ensuring the absence of unwanted side effects [7, 2], can aid in reasoning about operations like synchronization [6], and to enable the type system to enforce method call protocols [11]. Programmers often find uniqueness to be too restrictive, however, so ideas like borrowing [7, 2], adoption [14] and observers [22] were proposed to allow temporary aliases to unique pointers.

Despite all this activity, however, we know of no study of linearity at the more foundational level of object calculi like those of Abadi and Cardelli [1] or of Fisher et al. [15, 16]. While uniqueness has been formally modeled in simple Java-like languages such as Featherweight Java [18], we believe that studying linearity at a foundational level could potentially lead to more flexible systems, show how to apply linearity to new constructs like C#'s delegates, and allow us to more easily compare existing proposals for uniqueness.

The object-based setting is ideal for investigating linear methods in addition to linear objects. We also can explore applications of linearity to statically typing imperative update to method types or inheritance relationships, which languages such as Self [29], Python and Javascript treat dynamically. Also of interest are uses of these type changing operations to enforce protocols for object use.

1.1 Contributions

This paper studies linearity in a classless, imperative, foundational calculus called EGO, which is derived closely from existing object calculi. We extend an earlier version of this system [5] by simplifying the calculus and adding a mechanism for relaxing linearity when possible. The technical contributions of the paper are:

- We describe a linear type system for an object calculus. Our system gives an account of linear methods, which are called at most once, including the subtle interactions between linear methods and linear objects. We support imperative addition, change, and removal of methods in objects, as well as changes to the inheritance relationship between objects, features common in dynamically typed languages like Self [29].
- We demonstrate the expressiveness of our system in section 2 with a series of examples. We show how a simple example of using uniqueness to check object protocols from the literature [10] can be expressed in EGO.[1]
- We formalize the system and prove it sound in section 3.

[1] We believe other applications of linearity, such as memory management, could be added to our system with few changes, but memory management is not the focus of this paper.

FOOL '08 13 January, San Francisco, California, USA.
ACM .

- In section 4, we show how linearity can be relaxed in a generalization of borrowing by introducing regions as an extension to EGO. Our proposal extends previous work [27] to an object-oriented setting, and allows more flexible uses of borrowed pointers than in previous object-oriented research. This extension is formalized in section 5.

2. Simplified EGO

This section introduces a simplified, core version of EGO, which lacks a construct for relaxing linearity.

2.1 Intuition

Intuitively, programs in EGO proceed by manipulating objects. We follow Fisher et al. [15, 16] in modeling an object as a record of methods and possibly a delegation pointer to another object, which allows modeling imperative updates to delegation. A program in EGO consists of a mutable store and an expression which is recursively built of the following primitives for modifying objects:

- $\langle\rangle$ creates a new object on the heap and returns a reference to it.
- $e \leftarrow\!\!+ m = \sigma$ adds the method σ with the name m to the object on the heap referred to by e, or changes the method named m to be σ if it already exists.
- $e_1 \leftarrow e_2$ changes the delegatee of the object on the heap referred to by e_2 to be that referred to by e_1.
- $e.m$ invokes the method named m in the object referred to by e.
- $!e$ changes the linearity of an object, as discussed below. This only affects the type of the object; it has no dynamic effect.

We often write $\langle\rangle \leftarrow\!\!+ m_1 = \sigma_1 \cdots \leftarrow\!\!+ m_n = \sigma_n$ as $\langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle$.

EGO allows methods and objects to be either linear or nonlinear. A linear object is one to which one reference is allowed; nonlinear objects can have multiple references. To make static typing possible, only linear objects can change their interfaces by method addition or delegation change. All new objects are linear. When all needed interface changes have been made, the type of an object can be changed irrevocably to nonlinear so it can be freely aliased. This is similar to the "seal" operation of Fisher and Mitchell [16], where it allows objects to become subtypable; in our calculus objects instead become aliasable.

A linear method can be called only once, while nonlinear methods can be called multiple times. Calling a linear method removes it from the object that contains it. This is an interface change and so is only allowed on linear objects.

Methods in EGO are based on those of Abadi and Cardelli. A method is of the form $\varsigma(x{:}\tau).e$ or $¡\varsigma(x{:}\tau).e$, which are nonlinear and linear methods, respectively. A method is invoked on an object, its *receiver*, which need not be the object containing the method. When invoked, a method is found by searching the receiver and its delegatees. Once a method is found, invocation substitutes a reference to the receiver for x, the method's bound variable, in e, its body. All methods take only one argument, the self pointer of their receiver.

EGO expressions all evaluate to references to objects, and are given one of the following types.

- $\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R\rangle\!\rangle$ is a nonlinear object. This is recursive type, which binds $\mathbf{t}$ to the type of the whole object in the rest of the type. O is another object type, which the type of the object's delegatee or the distinguished type $\langle\!\rangle$, which indicates the object has no delegatee. R is a row giving types to methods in this object, of the form $m_1{:}\tau_1, \cdots, m_2{:}\tau_2$. Typechecking enforces that these τs are method types.
- $¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R\rangle\!\rangle$ is similar, but indicates that the object is linear.

Methods types, which appear in rows in object types, are of one of the two following forms.

- $\tau_1 \rightarrow \tau_2$ is the type of a nonlinear method, where τ_1 is the type the receiver must have when the method is invoked, and τ_2 is the type of the object the method returns.
- $\tau_1 \multimap \tau_2$ is the type of a linear method, which is consumed upon invocation.

2.2 Examples

We show some simple examples to demonstrate the use of EGO.

The first example shows object creation and change. First, $\langle\rangle$ creates a new object on the heap, to which is imperatively added a method, m, whose body is the identity method, which returns the receiver object. This method is then replaced by another of the same name which returns a new object when invoked.

$$\begin{aligned}&\langle\rangle \leftarrow\!\!+ m = \varsigma(this{:}¡\texttt{obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle m{:}¡\texttt{obj}\mathbf{t} \rightarrow ¡\texttt{obj}\mathbf{t}\rangle\!\rangle).this\\ &\quad \leftarrow\!\!+ m =\\ &\qquad \varsigma(this{:}¡\texttt{obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle m{:}¡\texttt{obj}\mathbf{t} \rightarrow ¡\texttt{obj}\ \mathbf{t}'.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle\cdot\rangle\!\rangle\rangle\!\rangle).\langle\rangle\end{aligned}$$

The next example shows how delegation can be changed. It creates a new object, adds the identity method to it, creates another object and changes the delegatee of this second object to be the first object,

$$\langle\rangle \leftarrow\!\!+ m = \varsigma(this{:}¡\texttt{obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle id{:}¡\texttt{obj}\mathbf{t} \rightarrow ¡\texttt{obj}\mathbf{t}\rangle\!\rangle).this \leftarrow \langle\rangle$$

In the next example, we create a new object, add a linear method to it that returns the receiver, and call the method. This removes the method, so this code produces a reference to an empty object. Since the invocation removes the method from the receiver, the type of the object the method expects is empty.

$$(\langle\rangle \leftarrow\!\!+ m = ¡\varsigma(x{:}¡\texttt{obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle\cdot\rangle\!\rangle).x).m$$

It is not obvious from these examples that EGO is powerful enough to be useful as a model for a full language. The remaining examples demonstrate EGO's power.

We exhibit an embedding of the simply typed lambda calculus. We define a lambda term of type $\tau \rightarrow \tau'$ as follows. This is based on a similar embedding shown by [1]. Subterms in double angle brackets represent recursively translated terms.

$$\begin{aligned}[\![\lambda x{:}\tau.e]\!] \stackrel{\text{def}}{=} !\langle gen = \varsigma(\_{:}[\![\tau \rightarrow \tau']\!]).\\ \langle body = \varsigma(this{:}\text{bodytype}(\tau,\tau')).\\ [this.arg/x][\![e]\!]\rangle\rangle\end{aligned}$$

where

$$\begin{aligned}&[\![\tau \rightarrow \tau']\!] \stackrel{\text{def}}{=} \texttt{obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle\\ &\quad gen{:}(\texttt{obj}\mathbf{t} \rightarrow (¡\texttt{obj}\ \mathbf{t}'.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle\\ &\qquad body : (\text{bodytype}(\tau,\tau') \rightarrow [\![\tau']\!])\rangle\!\rangle)\rangle\!\rangle\end{aligned}$$

This works by creating a new object with a single method, gen. This object is made nonlinear so that it can be aliased. gen is defined to return a new object containing a method whose body represents that of the lambda term with the lambda bound variable replaced by the invocation of a method called arg on the method's receiver.

Both the expected type of the receiver of the $body$ method, bodytype(τ,τ'), and the way in which application is done, depend on the linearity of the type the translated function expects. If the function expects something with linear type, the arg method added

to the generated object must be linear, and so be consumed when called. arg is therefore not in the receiver type for itself.

$$[\![\text{bodytype}(\tau, \tau')]\!] \stackrel{\text{def}}{=} \text{¡obj } \mathbf{t}''.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle \\ arg{:}(\text{¡obj } \mathbf{t}'''.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle body{:}(\text{¡obj}\mathbf{t}''' \rightarrow [\![\tau']\!])\rangle\!\rangle \multimap [\![\tau]\!]), \\ body{:}(\text{¡obj}\mathbf{t}'' \rightarrow [\![\tau']\!])\rangle\!\rangle$$

When e_1 is $\lambda x{:}\tau.e : \tau'$ as defined above, and $e_2{:}\tau$, then

$$[\![(e_1 : \tau \rightarrow \tau')e_2]\!] \stackrel{\text{def}}{=} \\ ((([\![e_1]\!].gen) \leftarrow\!\!+ arg = \\ \text{¡}\varsigma(\_{:}\text{¡obj } \mathbf{t}''.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle body{:}(\text{bodytype}(\tau, \tau') \rightarrow [\![\tau']\!])\rangle\!\rangle).[\![e_2]\!]) \\ .body$$

This calls gen on an object, e_1, which models a function, to create a new, linear object containing the function's body. To this linear object it adds a new linear method called arg which returns the argument of the application, e_2. Calling $body$ on the new object then simulates a β-reduction, as the function's bound variable has been replaced with a call to arg, which returns the argument. Since the function's argument is linear, arg is linear and so is consumed.

On the other hand, if the function expects something with linear type, the arg method added to the generated object must be nonlinear. In this case, since arg is not consumed, it appears in the receiver type for itself, as seen below.

$$\text{bodytype}(\tau, \tau') \stackrel{\text{def}}{=} \\ \text{obj } \mathbf{t}''.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle arg{:}(\text{obj}\mathbf{t}'' \rightarrow [\![\tau]\!]), body{:}(\text{obj}\mathbf{t}'' \rightarrow [\![\tau']\!])\rangle\!\rangle$$

and

$$[\![(e_1 : \tau \rightarrow \tau')e_2]\!] \stackrel{\text{def}}{=} \\ (!(([\![e_1]\!]).gen \leftarrow\!\!+ arg = \varsigma(\_{:}\text{bodytype}(\tau, \tau')).[\![e_2]\!])).body$$

Here, application is the same as above, but no the argument is no longer linear, so it is not removed on application.

Since we can calculate the return type, we elide it in later lambda expressions.

We can then use this to define a let binding, where we bind an expression e_1 of type τ.

$$\texttt{let } x = e_1 \texttt{ in } e_2 \stackrel{\text{def}}{=} (\lambda x{:}\tau.e_2)e_1$$

and a sequence operator

$$e_1; e_2 \stackrel{\text{def}}{=} \texttt{let } \_ = e_1 \texttt{ in } e_2$$

In a similar manner to lambda abstractions, we can also define linear lambda abstractions that are consumed when applied, as in [30].

$$[\![\text{¡}\lambda x{:}\tau.e]\!] \stackrel{\text{def}}{=} \langle body = \text{¡}\varsigma(this{:}\text{bodytype}(\tau)).[this.arg/x][\![e]\!]\rangle$$

where

$$\text{bodytype}(\tau) \stackrel{\text{def}}{=} \text{¡obj } \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle arg : \text{¡obj } \mathbf{t}'.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle\cdot\rangle\!\rangle \multimap [\![\tau]\!]\rangle\!\rangle$$

and we translate the type of a linear function by

$$[\![\tau \multimap \tau']\!] \stackrel{\text{def}}{=} \text{¡obj } \mathbf{t}''.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle body{:}(\text{bodytype}(\tau) \multimap [\![\tau']\!])\rangle\!\rangle$$

Finally, we translate application as follows.

$$[\![(e_1{:}\tau \multimap \tau')e_2]\!] \stackrel{\text{def}}{=} \\ ([\![e_1]\!] \leftarrow\!\!+ arg = \text{¡}\varsigma(\_{:}\text{¡obj } \mathbf{t}'.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle\cdot\rangle\!\rangle).[\![e_2]\!]).body$$

This example is slightly different than the one above. We simulate function consumption by having $body$ be a linear method that is consumed on invocation. Since it invokes a linear method, the object is linear, and so on invocation we add the argument directly to it, rather than calling a gen method to create a new, linear object.

This translation only allows linear arguments to linear functions, as we cannot call a linear method on a nonlinear object, so we cannot access the object carrying arg at multiple places in the function body. Later, we will show a way to avoid this restriction with borrowing.

```
typedef closedType = ¡obj t1.(¡obj t2.(⟪⟫ ← ⟪·⟫) ← ⟪·⟫)
typedef readType =
  ¡obj t1.(¡obj t2.(⟪⟫ ← ⟪read:¡objt1 → ¡objt1,
                          close:¡objt1 → closedType⟫)
          ← ⟪·⟫)
typedef openType =
  ¡obj t1.(¡obj t2.(⟪⟫ ← ⟪open:¡objt1 → readType⟫)
          ← ⟪·⟫)

let ClosedSocket = ⟨⟩ in
let ReadSocket = ⟨
  read = ς(this:readType)./*read from a socket*/; this
  close = ς(this:readType).
            /*close a socket*/; ClosedSocket ← this⟩
 in let OpenSocket = ⟨
  open = ς(this:openType).
            /*open a socket*/; ReadSocket ← this⟩
 in let Socket = OpenSocket ← ⟨⟩
 in /*More code*/
```

Figure 1. A series of objects for a network socket

This translation only allows linear arguments to linear functions, as we cannot call a linear method on a nonlinear object, so we cannot access the object carrying arg at multiple places in the function body. Later, we will show a way to avoid this restriction with borrowing.

A more complex and realistic example is that of a network socket object, given in Figure 1. In this example, we use a $typedef$ construct to simplify presentation; however, this is not part of the calculus.

This example creates an object called $Socket$ to model a network socket. The socket starts with a single method, $open$, which opens the socket and provides the socket object with two methods, $read$ and $close$. $read$ reads some data from the socket; $close$ closes the socket and removes all methods from the object.

These methods make other methods available and unavailable by changing the delegation of $Socket$, which remains linear. There are secondary objects corresponding to the three states of a socket. Each of these is delegated to at a different point in the $Socket$'s lifetime. $Socket$ starts delegated to $OpenSocket$, which contains the $open$ method. Calling $open$ changes the delegation of $Socket$ to $ReadSocket$, which contains $read$ and $close$ methods. Finally, calling $close$ changes delegation to $ClosedSocket$.

The pattern used here is of note. In this code, objects are created that correspond to states in an object's lifecycle and have a series of methods that are appropriate to the state they represent. We then create an empty object and transition from state to state by changing its delegatee to the object corresponding to the state we are entering. We therefore guarantee that only methods appropriate to the current object state exist on that object at any time.

3. Formalism for Simplified EGO

In this section we discuss the formalism we use to describe this fragment of EGO.

3.1 Syntax

A program in the fragment of EGO we present here consists of a pair, μ, e of store and an expression.

A store is a partial map $(\ell \mapsto s)^*$, where ℓ is an abstract location and s is an object descriptor of the form $loc \leftarrow \langle m_1 =$

Expressions	e	::=	$x, y \mid \langle\rangle \mid e.m \mid e \leftarrow\!\!+ m = \sigma$
		$\mid$	$e_1 \leftarrow e_2 \mid !e \mid v$
Values	v	::=	$loc \mid \sigma$
Locations	loc	::=	$\texttt{null} \mid \ell$
Stores	μ	::=	$\cdot \mid \mu, \ell \mapsto s$
Object Descriptors	s	::=	$loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle$
Methods	σ	::=	$\varsigma(x{:}\tau).e \mid ¡\varsigma(x{:}\tau).e$
Types	τ	::=	$\tau \rightarrow \tau' \mid \tau \multimap \tau' \mid O$
Object Types	O	::=	$L\mathbf{t} \mid \langle\!\langle\rangle\!\rangle \mid L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$
Linearities	L	::=	$\texttt{obj} \mid ¡\texttt{obj}$
Rows	R	::=	$\cdot \mid R, m{:}\tau$

Figure 2. Syntax of Simplified EGO

$\sigma_1, \cdots, m_n = \sigma_n\rangle$. loc is either a reference, ℓ, to an object's delegatee or `null`, which indicates the object has no delegatee, and $\langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle$, a record of the methods the object has.

A method, σ, is either $\varsigma(x{:}\tau).e$ or $¡\varsigma(x{:}\tau).e$. Both of these abstract the variable x out of the method body e. The first is a nonlinear method and the second linear.

An expression is either a variable, x or y, a new object creation, $\langle\rangle$, a method call, $e.m$, a method add or update, $e \leftarrow\!\!+ m = \sigma$, a delegation change $e_1 \leftarrow e_2$ or a linearity change $!e$. We consider methods as expressions to simplify the typing rules; however no other expression evaluates to a method. Locations are also expressions but they are intermediate forms which do not occur in user code. loc and methods are the only values.

The types, τ, of expressions are based on those used in [16]. These are either the types of methods or object types, O. The types of nonlinear and linear methods are $\tau \rightarrow \tau'$ and $\tau \multimap \tau'$.

Object types, O, are either type variables, $L\mathbf{t}$, or $\langle\!\langle\rangle\!\rangle$ or the recursive type $L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$. $\langle\!\langle\rangle\!\rangle$ is the type of `null` and $L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$ the type of ℓ. L is either $¡\texttt{obj}$ or $\texttt{obj}$ which indicate whether the object is linear or nonlinear. Type variables are prefixed with a linearity to allow them to be used with a different type than the one they are bound with. This allows us to write methods on an linear object that expect the receiver's type to be the current object type but nonlinear, which is useful as our linear objects can change to nonlinear. In $L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$, R is a row of the form $m_1{:}\tau_1, \cdots, m_n{:}\tau_n$, which specifies the method types of the methods in the object, and O is the type of the object's delegatee. As a method in an object may mention its receiver's type, it may be necessary for the type of an object to mention itself; in the object type $\mathbf{t}$ is a type variable recursively bound to the whole type. This may be used with the form $L\mathbf{t}$, which annotates the recursive variable with a linearity.

3.2 Dynamic Semantics

This section discusses how the expressions of EGO are evaluated. The operational rules defining how these expressions are evaluated are shown in Figure 3.

The simplest primitive is $\langle\rangle$. $\langle\rangle$ extends the heap with a new mapping, from some fresh location to an object descriptor with no methods and no delegatee. $\langle\rangle$ evaluates to the new location.

$e \leftarrow\!\!+ m = \sigma$ adds or updates a method on an object. Method addition modifies the store such that the object descriptor pointed to by the value of e has $m = \sigma$ added to its record of methods. Method update, on the other hand, replaces an existing $m = \sigma$ with $m = \sigma'$ in a similar way.

$e_1 \leftarrow e_2$ changes the delegation link on the object descriptor in the store to which the value of e_2 points to e_1.

$$\textsc{E-New}\ \frac{\ell \notin \texttt{Dom}(\mu) \qquad \mu' = [\ell \mapsto \texttt{null} \leftarrow \langle\rangle]\mu}{\mu, \langle\rangle \longrightarrow \mu', \ell}$$

$$\textsc{C-Upd}\ \frac{\mu, e \longrightarrow \mu', e'}{\mu, e \leftarrow\!\!+ m = \sigma \longrightarrow \mu', e' \leftarrow\!\!+ m = \sigma}$$

$$\textsc{E-Add}\ \frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots\rangle \quad \forall i. m \neq m_i \qquad \mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma\rangle]\mu}{\mu, \ell \leftarrow\!\!+ m = \sigma \longrightarrow \mu', \ell}$$

$$\textsc{E-Upd}\ \frac{\mu(\ell) = loc \leftarrow \langle \cdots, m = \sigma, \cdots\rangle \quad \mu' = [\ell \mapsto loc \leftarrow \langle \cdots, m = \sigma', \cdots\rangle]\mu}{\mu, \ell \leftarrow\!\!+ m = \sigma' \longrightarrow \mu', \ell}$$

$$\textsc{C-Del}_1\ \frac{\mu, e_1 \longrightarrow \mu', e_1'}{\mu, e_1 \leftarrow e_2 \longrightarrow \mu', e_1' \leftarrow e_2} \qquad \textsc{C-Del}_2\ \frac{\mu, e \longrightarrow \mu', e'}{\mu, \ell \leftarrow e \longrightarrow \mu', \ell \leftarrow e'}$$

$$\textsc{E-Del}\ \frac{\mu(\ell) = loc \leftarrow \langle \cdots\rangle \qquad \mu' = [\ell \mapsto loc' \leftarrow \langle \cdots\rangle]\mu}{\mu, loc' \leftarrow \ell \longrightarrow \mu', \ell}$$

$$\textsc{C-Inv}\ \frac{\mu, e \longrightarrow \mu', e'}{\mu, e.m \longrightarrow \mu', e'.m} \qquad \textsc{E-NLinInv}\ \frac{\texttt{mbody}(\mu, \ell, m) = \varsigma x{:}\tau.e}{\mu, \ell.m \longrightarrow \mu, [\ell/x]e}$$

$$\textsc{E-LinInv}\ \frac{\mu(\ell) = \langle \cdots, m = ¡\varsigma x{:}\tau.e, \cdots\rangle \qquad \mu' = [\ell \mapsto \langle \cdots\rangle]\mu}{\mu, \ell.m \longrightarrow \mu', [\ell/x]e}$$

$$\textsc{C-ChLin}\ \frac{\mu, e \longrightarrow \mu', e'}{\mu, !e \longrightarrow \mu', !e'} \qquad \textsc{E-ChLin}\ \mu, !v \longrightarrow \mu, v$$

$$\textsc{mbody}_1\ \frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma, \cdots\rangle}{\texttt{mbody}(\mu, \ell, m) = \sigma}$$

$$\textsc{mbody}_2\ \frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots\rangle \quad m = \sigma \notin \langle m_1 = \sigma_1, \cdots\rangle \qquad \texttt{mbody}(\mu, loc, m) = \sigma}{\texttt{mbody}(\mu, \ell, m) = \sigma}$$

Figure 3. Dynamic Semantics of Simplified EGO

$e.m$ invokes a method. It looks up the object descriptor that the value of e refers to in the heap. In the invocation of linear methods, the method body is found in the record of methods in the object descriptor by finding a method named m. In nonlinear methods, method lookup is slightly more complicated: if the method is found in the record of methods in the object descriptor referred to by e, this method body is returned. Otherwise, the method body is searched for recursively in that object descriptor's delegatee. Then, in either case, the method's receiver is substituted for the variable bound by the method in the method body. In the case of linear method invocation, the store is modified by removing the method from the object that contains it.

$!e$ has no dynamic effect. It only affects the typing of objects.

$$\text{T-NLinLoc}\ \frac{\Sigma(\ell) = \texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle}{\Sigma; A \vdash \ell{:}\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow \{\}} \qquad \text{T-LinLoc}\ \frac{\Sigma(\ell) = ¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle}{\Sigma; A \vdash \ell{:}¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow \{\ell\}}$$

$$\text{T-Null}\ \frac{}{\Sigma; A \vdash \texttt{null}{:}\langle\!\langle\rangle\!\rangle \Longrightarrow \{\}} \qquad \text{T-ChLin}\ \frac{\Sigma; A \vdash e{:}¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l}{\Sigma; A \vdash !e{:}\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l}$$

$$\text{T-NLinInv}\ \frac{\Sigma; A \vdash e{:}L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l \qquad \tau_u = L\mathbf{t}.[\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}](O \leftarrow \langle\!\langle R \rangle\!\rangle) \qquad \texttt{mtype}(\tau_u, m) = L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \to \tau}{\Sigma; A \vdash e.m{:}\tau \Longrightarrow l}$$

$$\text{T-LinInv}\ \frac{\begin{array}{c}\Sigma; A \vdash e{:}¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l \qquad \tau_u = ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}](O \leftarrow \langle\!\langle R \rangle\!\rangle) \qquad \texttt{lmtype}(\tau_u, m) = ¡\texttt{obj}\ \mathbf{t}'.O' \leftarrow \langle\!\langle R' \rangle\!\rangle \multimap \tau \\ ¡\texttt{obj}\ \mathbf{t}''.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle = \tau_u \qquad ¡\texttt{obj}\ \mathbf{t}'.[\mathbf{t}'.O' \leftarrow \langle\!\langle R' \rangle\!\rangle/\mathbf{t}'](O' \leftarrow \langle\!\langle R' \rangle\!\rangle) = ¡\texttt{obj}\ \mathbf{t}''.O'' \leftarrow \langle\!\langle R''/m{:}\tau'' \rangle\!\rangle\end{array}}{\Sigma; A \vdash e.m{:}\tau \Longrightarrow l}$$

$$\text{T-Upd}\ \frac{\begin{array}{c}\Sigma; A \vdash \sigma{:}\tau \Longrightarrow l \qquad \Sigma; A \vdash e{:}¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l' \qquad \texttt{lmtype}(¡\texttt{obj}\ \mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle, m) = \tau' \\ ¡\texttt{obj}\ \mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle = ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}](O \leftarrow \langle\!\langle R \rangle\!\rangle) \qquad ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle/\mathbf{t}](O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle) = ¡\texttt{obj}\ \mathbf{t}.O' \leftarrow \langle\!\langle [m{:}\tau/m{:}\tau']R' \rangle\!\rangle\end{array}}{\Sigma; A, A' \vdash e \leftarrow\!\!+ m = \sigma : ¡\texttt{obj}\ \mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle \Longrightarrow l, l'}$$

$$\text{T-Add}\ \frac{\begin{array}{c}\Sigma; A \vdash \sigma{:}\tau \Longrightarrow l \qquad \Sigma; A \vdash e{:}¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l' \qquad \texttt{lmtype}(¡\texttt{obj}\ \mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle, m) \neq \\ ¡\texttt{obj}\ \mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle = ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}](O \leftarrow \langle\!\langle R \rangle\!\rangle) \qquad ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle/\mathbf{t}](O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle) = ¡\texttt{obj}\ \mathbf{t}.O' \leftarrow \langle\!\langle R', m{:}\tau \rangle\!\rangle\end{array}}{\Sigma; A, A' \vdash e \leftarrow\!\!+ m = \sigma : ¡\texttt{obj}\ \mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle \Longrightarrow l, l'}$$

$$\text{T-NLinMeth}\ \frac{\Sigma; A, x{:}\tau \vdash e{:}\tau' \Longrightarrow \{\} \qquad x \notin \texttt{Dom}(A) \qquad A\ \texttt{nonlinear}}{\Sigma; A \vdash \varsigma x{:}\tau.e{:}\tau \to \tau' \Longrightarrow \{\}} \qquad \text{T-LinMeth}\ \frac{\Sigma; A, x : \tau \vdash e{:}\tau' \Longrightarrow l \qquad x \notin \texttt{Dom}(A)}{\Sigma; A \vdash ¡\varsigma x{:}\tau.e : \tau \multimap \tau' \Longrightarrow l}$$

$$\text{T-Del}\ \frac{\begin{array}{c}\Sigma; A \vdash e_2{:}¡\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l \qquad \Sigma; A' \vdash e_1{:}O'' \Longrightarrow l' \\ ¡\texttt{obj}\ \mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle = ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}]O \leftarrow \langle\!\langle R \rangle\!\rangle \qquad ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O''' \leftarrow \langle\!\langle R'' \rangle\!\rangle/\mathbf{t}](O''' \leftarrow \langle\!\langle R'' \rangle\!\rangle) = ¡\texttt{obj}\ \mathbf{t}.O'' \leftarrow \langle\!\langle R' \rangle\!\rangle\end{array}}{\Sigma; A, A' \vdash e_1 \leftarrow e_2{:}¡\texttt{obj}\ \mathbf{t}.O''' \leftarrow \langle\!\langle R'' \rangle\!\rangle \Longrightarrow l, l'}$$

$$\text{T-New}\ \frac{}{\Sigma; A \vdash \langle\rangle{:}¡\texttt{obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle \cdot \rangle\!\rangle \Longrightarrow \{\}} \qquad \text{T-Var}\ \frac{}{\Sigma; x{:}\tau \vdash x{:}\tau \Longrightarrow \{\}} \qquad \text{T-Kill}\ \frac{\Sigma; A \vdash e{:}\tau \Longrightarrow l}{\Sigma; A, x{:}\tau' \vdash e{:}\tau \Longrightarrow l} \qquad \text{T-Copy}\ \frac{\Sigma; A, x{:}\tau', x{:}\tau' \vdash e{:}\tau \Longrightarrow l \qquad \tau'\ \texttt{nonlinear}}{\Sigma; A, x{:}\tau' \vdash e{:}\tau \Longrightarrow l}$$

Figure 4. Static Semantics of Simplified EGO

$$\text{T-LMethT}\ \frac{m{:}\tau \in R}{\texttt{lmtype}(L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, m) = \tau} \qquad \text{T-MethT}_1\ \frac{\texttt{lmtype}(L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, m) = \tau}{\texttt{mtype}(L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, m) = \tau}$$

$$\text{T-MethT}_2\ \frac{\texttt{lmtype}(L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, m) \neq \qquad \texttt{mtype}(O, m) = \tau}{\texttt{mtype}(L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, m) = \tau}$$

Figure 5. Method Type Lookup in Simplified and Full EGO

3.3 Type System

EGO's type system enforces a lack of runtime errors. One important mechanism for this guarantee is maintenance of the distinction between linear and nonlinear objects.

The type system allows changes to the interface of an object only for linear objects. This is because changes to objects are imperative: they affect the object descriptor on the heap. If an object is aliased in unknown ways, it may be impossible to update the types of the other aliases to the object, making the system unsound. We avoid this unsoundness by prohibiting changes to the interface on non-linear (potentially aliased) objects.

Interface changes are also allowed only to the object on which they are performed, not its delegatees, for similar reasons. This is because we can delegate to nonlinear objects, so we have no guarantee that this object is not aliased elsewhere. Thus, the same problems exist with changing the interface of a delegatee as do with changing the interface of a nonlinear object. Pragmatically, the effect this has is disallowing method update unless we have a linear pointer to the object descriptor containing the method.

Our method types, $\tau \to \tau'$ and $\tau \multimap \tau'$ have receiver types as part of them. This contrasts with many other object calculi [1, 16] where the receiver type is left out, as it is known to be the type of the object or a subtype. However, since we allow changes to the type of objects, the receiver type may be different when the method is called, so we must include it.

Since objects may contain methods whose types contain the type of the object, the EGO type system assigns recursive types of the form $L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$ to objects. Here, $\mathbf{t}$ is bound recursively to the whole type.

In general, when we change the interface of an object, we need to unfold the type, do the interface change, and refold the new type. This maintains that any use of the outermost recursive type variable in the type will always correspond to the current type, rather than a type from before the interface change. We also need to unfold the type of an object before looking up a method in it to check that

the receiver type the method expects is the type the whole object currently has.

To unfold our objects types, we substitute the type, with the initial linearity indicator removed, in for the bound type variable in the whole type, not just the body of the recursive type. Since every type variable usage is prefixed by a linearity indicator, this gives us a type of the form $L\mathbf{t}.O \leftarrow \langle R \rangle$, with the variable binding and and linearity indicator still at the head of the type. This makes the type syntactically valid, and allows to compare the linearities of two unfolded types.

Now, we discuss our typing rules, shown in Figure 4, in more detail. Our typing judgment looks like

$$\Sigma; A \vdash e{:}\tau \Longrightarrow l$$

Here, Σ is the store typing, which consists of a mapping, $(\ell \mapsto \tau)^*$ from locations to types. A is the type context. e is the expression to be typechecked and τ is the type given to it. l is a list of linear locations in e. This is a technical device used in the type safety proof for proving that linear locations are never aliased.

Locations are typed by looking them up in the store. The rule that types linear locations, T-LINLOC, also puts the location it types into the list of linear locations it returns, l.

`null` is typed by giving it the type $\langle\!\langle\rangle\!\rangle$.

We can turn a linear location into a nonlinear location with $!e$. All new objects are linear so that methods can be added and other interface changes can be made. We get a nonlinear object by turning a linear object into a nonlinear one. This is safe as it can only go one way: we cannot turn a nonlinear location into a linear one.

Thus, to type a method invocation, we first type the receiver with a type of the form $L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$, which is the folded type of the object. This type is unfolded by substituting it into its recursively bound variable, and the type of the method is looked up in this new type. Since invocation of a linear method changes the type of the receiver object as described below, and we cannot change the interface of delegatees, we only look up linear method types in the row of the unfolded object type, which describes the methods contained in the object. However, invocation of nonlinear methods on an object does not change the interface of the object, so we can look such a method up recursively in the delegatee type on the unfolded object type if the method type is not found in the row. These lookup rules are defined in Figure 5, where we use the judgment $\texttt{lmtype}(L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, m) \neq$ to indicate that no such method is found in the current row. For nonlinear methods, we then check that the type the method expects is the type of the receiver. The invocation is then given the return type of the method.

For linear methods, we must do more. Since invocation of linear methods removes them from the object containing them, the interface of the object is changed by such invocations. Therefore, an object invoking a linear method must be linear. Also, rather than checking equality of the type the method expects with the receiver type as we do with nonlinear types, we check equality of the type the method expects with that of the receiver type refolded with the invoked method removed, as this is what will be substituted into the method body when the method is invoked.

Method addition and update are checked by typing the object as some linear object type $\texttt{¡obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$. This type is unfolded, and we give this expression the type found by adding or updating the appropriate method and folding the object type back up.

Methods themselves are typed similarly to lambda calculus functions. The abstracted variable is placed in the typing context with type which which it is bound and the method body is typed under this new context. However, nonlinear methods are not allowed to mention linear objects; otherwise, multiple calls to the method would result in multiple occurrences of the linear object.

$$\text{T-STORE}\quad \frac{\forall \ell \in \texttt{Dom}(\mu).\Sigma; \cdot \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_\ell \qquad \texttt{Dom}(\mu) = \texttt{Dom}(\Sigma)}{\Sigma \vdash \mu\ \texttt{ok} \Longrightarrow \text{concat}\ l_\ell}$$

$$\text{T-ODESCR}\quad \frac{\forall i \in 1..n.(\Sigma; A \vdash \sigma_i{:}\tau_i \Longrightarrow l_i) \qquad [\tau/\mathbf{t}]R = m_1{:}\tau_1, \cdots, m_n{:}\tau_n \quad \Sigma; A \vdash loc{:}O \Longrightarrow l_{loc} \qquad l = l_{loc}, l_1, \cdots, l_n \qquad \tau = L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle}{\Sigma; A \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle{:}\tau \Longrightarrow l}$$

Figure 6. Store and Object Typing for Simplified EGO

Delegation is typed similarly to addition and update. The type of the expression whose delegation is being changed is unfolded. The expression is given this type with the type of the delegatee changed and the object type folded back up.

$\langle\rangle$ is given the type of an empty linear object with no delegatee. This is the type $\texttt{¡obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle \cdot \rangle\!\rangle$.

Finally, we type variables by looking them up in the type context, A, according to T-VAR. This rule expects the context to contain only one binding. However, we can use T-KILL to eliminate extra bindings, as in Wadler's calculus [30]. This makes our type system more similar to affine logic than linear logic.

We enforce linearity by splitting the context when we type subexpressions. This is the approach taken by Wadler in [30], modeled on the same technique from Girard's linear logic [17]. We allow aliases to nonlinear objects through T-COPY, which makes copies of a nonlinear variable's binding in the context.

We also have a store typing judgment, defined in T-STORE. This checks that each object stored in the heap has the type the store type, Σ, gives it by checking that all the methods in each object have the type the object type gives them and that the delegatees have the correct type. The object type has been folded but the types calculated for a method types are not, so before comparing them, we must unfold the object type. The store typing judgment also produces a list of all linear locations mentioned in the store. This is used to prove that no linear objects are mentioned more than once in the heap and currently executing expression. These rules are shown in Figure 6.

3.4 Safety Proof

We have no formal proof of safety for the fragment of EGO presented so far, only a proof for the system with the extensions described below. However, we sketch a hypothetical proof of safety for this fragment below. Type safety consists of standard progress and preservation lemmas.

THEOREM 3.1 (Progress). *If* $\Sigma; \cdot \vdash e{:}\tau \Longrightarrow l$ *and* $\Sigma \vdash \mu\ \texttt{ok} \Longrightarrow l$ *then* $\mu, e \longrightarrow \mu', e'$ *or* e *is a value.*

The proof of this is by induction on the derivation of $\Sigma; A \vdash e{:}\tau \Longrightarrow l$ with a canonical forms lemma.

Preservation states in general that an evaluation step maintains program invariants. Specifically, we wish to maintain three invariants.

1. The expression has some type τ.
2. The heap is well typed.
3. All linear locations are used at most once in the expression and store.

To do this, we define a relation on store types, $\Sigma \geq_\ell \Sigma'$, which says that a new store type is related to an old store type by either extending it or changing a single location with only a linear reference.

THEOREM 3.2 (Preservation). *If*

i. $\Sigma; \cdot \vdash e{:}\tau \Longrightarrow l_e$
ii. $\Sigma \vdash \mu \ \ \texttt{ok} \Longrightarrow l_s$
iii. there are no duplicates in l_e, l_s*, and*
iv. $\mu, e \longrightarrow \mu', e'$

then for some $\Sigma' \geq_\ell \Sigma$

i. $\Sigma'; \cdot \vdash e'{:}\tau \Longrightarrow l'_e$
ii. $\Sigma' \vdash \mu' \ \ \texttt{ok} \Longrightarrow l'_s$*, and*
iii. there are no duplicates in l'_e, l'_s.

The proof of this is by induction on the evaluation judgment and uses the following substitution lemma.

LEMMA 3.1 (Substitution). *If* $\Sigma; A, x{:}\tau' \vdash e{:}\tau \Longrightarrow l$ *and* $\Sigma; \cdot \vdash v{:}\tau' \Longrightarrow l'$*, then* $\Sigma; A \vdash [v/x]e{:}\tau \Longrightarrow l''$ *and* $l'' \subseteq l, l'$.

We also need a lemma that says that if we have a well typed store, and we replace a linear object with a new object and suitably modify the store typing, the store remains well typed.

LEMMA 3.2 (Store Change). *If*

i. $\Sigma \vdash \mu \ \ \texttt{ok} \Longrightarrow l_s$
ii. $\Sigma; A \vdash \ell_L{:}\texttt{¡obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l_e$
iii. there are no duplicates in l_s, l_e
iv. $\mu(\ell) = s$
v. $\Sigma; \cdot \vdash s{:}\tau \Longrightarrow l_o$*, and*
vi. $\Sigma; \cdot \vdash s'{:}\tau' \Longrightarrow l'_o$

then $[\ell \mapsto \tau']\Sigma \vdash [\ell \mapsto s']\mu \ \ \texttt{ok} \Longrightarrow l_s - l_o, l'_o$

4. Relaxing Linearity

The fragment of the EGO language presented so far is powerful but has a significant drawback. The aliasing restriction of linearity is often too restrictive. It is sometimes useful to be able to temporarily give up the ability to change an object's interface in order to make short-lived aliases of it.

One example of this is the use of the network socket example above. A socket here is a linear object which contains an *open* method. Calling this method opens a socket and changes the interface, as the *open* method is removed and $read$ and $close$ methods are added. At this point, assuming the $read$ method is reentrant, there would be no reason to prevent aliasing the object to allow several sections of the program to read from it simultaneously, as long as no interface changes are made. After all of these reads are done, if no aliases of the object exist, a $close$ call could close the socket and remove the $read$ method.

The hitherto presented fragment of EGO does not allow this. We now introduce a construct based on `let!` as presented by Wadler in [30], to allow us to make temporary aliases of linear objects.

4.1 Additions to the Calculus

Intuitively, our new construct allows us to evaluate three expressions in sequence. The first two each bind a new variable to be used in the successive expressions. The first expression evaluates to a possibly linear object which is bound to a variable. In the second expression, this variable is bound with a borrowed object type. It can be freely aliased but no changes can be made to its interface. This borrowed object is similar to a linear object acting temporarily as a nonlinear object. This second expression is evaluated to a value and bound to a second variable. In the third expression, both variables are bound with the first being bound with to its original type, as it is no longer borrowed.

To prevent aliases of a borrowed object escaping the expression in which it is borrowed, we annotate the types of borrowed objects with *regions*. A region is a unique tag generated every time an object is borrowed which indicates where it is borrowed. We keep track of which regions are currently annotating borrowed objects and do not allow typing of an object with a region not in scope.

Our region system differs from previous systems such as [28] in two ways. First, much previous work focuses on using regions for statically verifying explicit memory management; we use regions instead to track aliases. Second, we enforce a weaker invariant. While previous systems often prohibit references to any pointer whose region is out of scope, we allow such references but prohibit their use. This gives our system additional flexibility.

Our `let!` looks like this:

$$\texttt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}$$

Here, ρ is a region variable bound to the region generated when a location is borrowed with this expression. The value of e_1 is bound to x_1 in e_2 and e_3 and the value of e_2 is bound to x_2 in e_3.

Our `let!` differs from Wadler's in that Wadler does not use regions to contain aliases. Instead, he places restrictions on the type of the expression in which a value is borrowed to prevent it from escaping. We allow these expressions to have any type as long as it does not contain the region under which the location is borrowed. Since we are not relying on the type of the value of this expression to restrict aliases, just the regions it contains, our system also works in an imperative setting where values may be stored on the heap.

Similar work to Wadler's by Odersky [22], allows creation of read only aliases to a value. Other work, such as Fändrich and DeLine's work on adoption[14], also allows temporary aliases to be stored on the heap.

Several other modifications to the existing calculus need to be made to accommodate regions. The first of these is the addition of region polymorphism. Methods can only be added to linear objects. However, we may wish to call methods on borrowed objects. We therefore need to add methods to objects before they are borrowed which expect receivers that are borrowed under regions not yet in scope. We do this by allowing methods to be abstracted over a number of regions, which are instantiated when the method is called. To accomplish this, `let!` also binds a region variable which is in scope where the object is borrowed and which refers to the region at which the object is borrowed so that regions may be referred to in code and instantiated.

The following example uses region polymorphism. It binds to x an object containing a single method that returns a new object. The method is polymorphic in the region of its receiver. In the type the method expects its receiver to have, the method itself has a polymorphic type to reflect this. In the second subexpression, where this object is borrowed, the method is invoked. When this happens, the polymorphic variable is instantiated with the region in which the object is borrowed to allow the method to be called. The value of this method call is bound to y, which is, in the the third subexpression, returned as the value of the entire expression.

$$\begin{aligned}
&\texttt{let!}\ (\rho_1)\, x = \langle\rangle \leftarrow\!\!+ new = \\
&\qquad \Lambda\rho_2.\varsigma(this{:}\rho_2\mathbf{t_1}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle \\
&\qquad\qquad new{:}(\forall\rho_3.\rho_3\mathbf{t_1} \dot{\rightarrow} \texttt{¡obj}\ \mathbf{t_2}.(\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle\cdot\rangle\!\rangle\rangle\!\rangle)\rangle\!\rangle).\langle\rangle \\
&\qquad y = x.new[\rho_1] \\
&\texttt{in}\ y\ \texttt{end}
\end{aligned}$$

We also now annotate method types with a list of regions used by the method. This is because it will not always be apparent otherwise from the arrow type what regions are used in a given method. We only allow invoking methods with regions that are in scope.

The next example uses this type. It is similar to the above example, but the added method uses the receiver in its body before returning an empty object. Since it uses a region in its body, the type of this method in the receiver type on the method must mention the

$$[\![¡\lambda x{:}\tau.e]\!] \stackrel{\text{def}}{=} \langle body = ¡\varsigma(this{:}\text{bodytype}(\tau)).\ \texttt{let!}\ (\rho)\, this_{bor} = this\ result = [this_{bor}.arg[\rho]/x][\![e]\!]\ \texttt{in}\ result\ \texttt{end}\rangle$$

$$\text{bodytype}(\tau) \stackrel{\text{def}}{=} ¡\texttt{obj}\ \mathbf{t}.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle arg : \forall\rho'.\rho'\mathbf{t} \stackrel{\cdot}{\rightarrow} [\![\tau]\!] \rangle\!\rangle$$

$$[\![(e_1{:}\tau \multimap \tau')e_2]\!] \stackrel{\text{def}}{=} ([\![e_1]\!] \leftarrow\!\!+ arg = \Lambda\rho'.\varsigma(\_{:}\text{argtype}(\tau,\rho')).[\![e_2]\!]).body$$

$$\text{argtype}(\tau,\rho') \stackrel{\text{def}}{=} \rho'\mathbf{t}'.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle arg : \forall\rho''.\rho''\mathbf{t}' \stackrel{\cdot}{\rightarrow} [\![\tau]\!] \rangle\!\rangle$$

$$[\![\tau \multimap \tau']\!] \stackrel{\text{def}}{=} ¡\texttt{obj}\ \mathbf{t}''.\langle\!\langle\rangle\!\rangle \leftarrow \langle\!\langle body{:}(\text{bodytype}(\tau) \stackrel{\cdot}{\multimap} [\![\tau']\!]) \rangle\!\rangle$$

Figure 7. Linear functions with nonlinear arguments

region. Note that the annotation on the arrow is polymorphic. It, as well as the region of the receiver, is instantiated when the method is invoked.

```
let! (ρ1) x = ⟨⟩ ←+ new =
        Λρ2.ς(this:
          ρ2t1.⟪⟫ ← ⟪
            new:∀ρ3.ρ3t1 -ρ3-> ¡obj t2.(⟪⟫ ← ⟪·⟫)⟫).
          (this; ⟨⟩)
      y = x.new[ρ1]
in y end
```

The final modification arises from a similar problem to that which gave rise to region polymorphism. We may add methods to objects that refer to regions which are later no longer in scope. We cannot call these methods, but we allow other methods to be invoked on such an object. This means that any type annotations we write on methods must be able to be the type of objects with methods that mention regions that are not in scope. Since the region variable we bound to the region is no longer in scope, we cannot write such a type. We solve this problem by having a type, $\top$, which is the type of uncallable methods. This type is a supertype of a normal method type, so can be used on method type annotations which accept objects whose types contain unknown regions.

This example shows the use of this type. We create an object, x with one method which returns a new object. Then we borrow a new object and bind it to the variable y. We then add a new method to x, which is still linear, that mentions y while y is still borrowed. After we leave the subexpression where y is borrowed, we call the first method on x. x now contains a method that mentions a region no longer in scope. To allow this, on the expected type of the receiver on the method we call, we give this method the type $\top$. This means we never can call the method, but, as it mentions regions no longer in scope, we would not be able to in any case.

```
typedef receivertype =
        objt.⟪⟫ ← ⟪meth1:objt -> (¡objt.⟪⟫ ← ⟪·⟫),
                   meth2:⊤⟫
let x = ⟨⟩ ←+ meth1 = ς(this:receivertype).⟨⟩ in
let! (ρ) y = ⟨⟩
        z = x ←+ meth2 = ς(this:receivertype).(y; ⟨⟩)
in z.meth1 end
```

We now have the necessary linguistic mechanisms to model linear lambda abstractions which take nonlinear arguments. As with those

```
typedef closedType = ¡obj t1.(¡obj t2.(⟪⟫ ← ⟪·⟫) ← ⟪·⟫)
typedef readType(L) = Lt1.(¡obj t2.(⟪⟫ ← ⟪
            read:∀ρ2.ρ2t1 -> ρ2t1,
            close:¡objt1 -> closedType⟫) ← ⟪·⟫)
typedef openType =
      ¡obj t1.(¡obj t2.(⟪⟫ ← ⟪open:¡objt1 -> readType⟫)
            ← ⟪·⟫)

let ClosedSocket = ⟨⟩ in
let ReadSocket = ⟨
      read = Λρ1.ς(this:readType(ρ1))./*read from a socket*/;
            this
      close = ς(this:readType(¡obj))./*close a socket*/;
            ClosedSocket ← this⟩
 in let OpenSocket = ⟨
      open = ς(this:openType)./*open a socket*/;
            ReadSocket ← this⟩
 in let Socket = OpenSocket ← ⟨⟩
 in /*More code*/
Socket.open;
let! (ρ2) Socket' = Socket
      SomeData = /*Code that aliases Socket'*/
in /*More Code*/
      Socket'.close
end
```

Figure 8. A socket using `let!`

taking linear arguments, we do this by creating an object with a linear *body* method which expects to be called on an object with an arg method representing the argument to the function. Since the method will be consumed, it must be called on a linear object. Now, however, we can duplicate the object within the body of the expression to access arg multiple times. We do this by borrowing the object within the *body* method, so it s borrowed in the function's body. The arg method must therefore be abstracted over the region it expects to be called in. This is shown in Figure 7.

We can also implement the socket example described at the beginning of this section. In fact, this example is fairly simple; it is given in Figure 8. It is based on the previous socket example in Figure 1. The main differences are that the $read$ method is now parametrized over a region. After defining the objects, we open the socket and then borrow it as $Socket'$. Now we can freely alias this object. We can use the borrowed socket by calling $read$ on any reference to the object as long as such calls to $read$ instantiate the polymorphic variable ρ_1 with the bound region variable ρ_2. After leaving this expression, we no longer have access to any aliases of $Socket'$, so we can close the socket.

5. Formalism

In this section we present extensions to the previous formalism that implement `let!` and regions as we have described them.

5.1 Additions to the Syntax

The differences from the syntax of the original fragment of EGO are presented below and appear in Figure 9.

We add the `let!` construct itself to the expressions of the language. This is of the form $\texttt{let!}\ (\varrho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}$. Here, ϱ is either ρ, a region variable, or r, a region. ϱ is added to the linearities, L. However, we do not allow users to write down regions, r, so $\texttt{let!}\ (r)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}$ is an intermediate form generated during program execution.

Expressions	e	::=	$\cdots \mid e.m[\varrho_1, \cdots, \varrho_n] \mid \cdots$
		|	$\texttt{let!}\ (\varrho)\ x_1 = e_1\ x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}$
Locations	loc	::=	$\texttt{null} \mid \ell_L$
Methods	σ	::=	$\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma(x{:}\tau).e$
		|	$\Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma(x{:}\tau).e$
Types	τ	::=	$\cdots \mid \forall\rho_1.\cdots\forall\rho_n.\tau \xrightarrow{P} \tau'$
		|	$\forall\rho_1.\cdots\forall\rho_n.\tau \multimap^{P} \tau' \mid \top$
Linearities	L	::=	$o \mid \varrho$
Object Linearities	o	::=	$\texttt{obj} \mid ¡\texttt{obj}$
Regions	ϱ	::=	$\rho \mid r$
Region Effects	P	::=	$\cdot \mid P, \varrho$

Figure 9. Additions to the Syntax of EGO

$$\text{C-LET}_1\ \frac{\mu, e_1 \longrightarrow \mu', e_1'}{\mu, \texttt{let!}\ (\rho)\ x_1 = e_1\ x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu', \texttt{let!}\ (\rho)\ x_1 = e_1'\ x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}}$$

$$\text{E-LET}_1\ \frac{r\ \texttt{fresh}}{\mu, \texttt{let!}\ (\rho)\ x_1 = \ell_L\ x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu, \texttt{let!}\ (r)\ x_1 = \ell_L\ x_2 = [r, \ell_r/\rho, x_1]e_2\ \texttt{in}\ e_3\ \texttt{end}}$$

$$\text{C-LET}_2\ \frac{\mu, e_2 \longrightarrow \mu', e_2'}{\mu, \texttt{let!}\ (r)\ x_1 = v_1\ x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu', \texttt{let!}\ (r)\ x_1 = v_1\ x_2 = e_2'\ \texttt{in}\ e_3\ \texttt{end}}$$

$$\text{E-LET}_2\ \frac{}{\mu, \texttt{let!}\ (r)\ x_1 = v_1\ x_2 = v_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu, [v_1, v_2/x_1, x_2]e_3}$$

$$\text{C-INV}\ \frac{\mu, e \longrightarrow \mu', e'}{\mu, e.m[\varrho_1, \cdots, \varrho_n] \longrightarrow \mu', e'.m[\varrho_1, \cdots, \varrho_n]}$$

$$\text{E-NLININV}\ \frac{\texttt{mbody}(\mu, \ell_L, m) = \Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma x{:}\tau.e}{\mu, \ell_L.m[\varrho_1, \cdots, \varrho_n] \longrightarrow \mu, [\ell_L, \varrho_1, \cdots, \varrho_n/x, \rho_1, \cdots, \rho_n]e}$$

$$\text{E-LININV}\ \frac{\mu(\ell) = \langle \cdots, m = \Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma x{:}\tau.e, \cdots\rangle \qquad \mu' = [\ell_l \rightarrow \langle\cdots\rangle]\mu}{\mu, \ell_L.m\varrho_1, \cdots, \varrho_n \longrightarrow \mu', [\ell_L, \varrho_1, \cdots, \varrho_n/x, \rho_1, \cdots, \rho_n]e}$$

$$\text{E-CHLIN}\ \frac{}{\mu, !\ell_{¡\texttt{obj}} \longrightarrow \mu, \ell_{\texttt{obj}}}$$

Figure 10. Additions to the Dynamic Semantics of EGO

We also make changes to methods. Methods are now of the form $\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma(x{:}\tau).e$, or $\Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma(x{:}\tau).e$. These are parametrized over some number of regions.

Method types are are now polymorphic. We also add annotations to method types indicating the regions mentioned by the methods they type. Our method types now look like $\forall\rho_1.\cdots\forall\rho_n.\tau \xrightarrow{P} \tau'$ or $\forall\rho_1.\cdots\forall\rho_n.\tau \multimap^{P} \tau'$. We use $\top$ a separate type for methods which mention regions no longer in scope.

We add region instantiation to method invocations. $e.m[\varrho_1, \cdots, \varrho_n]$ invokes a method and instantiates its region arguments. We often write $e.m[]$ as $e.m$.

Finally, our locations are now annotated with linearities. They are now of the form ℓ_L. This allows us to give types to objects in settings where their linearities may change, as discussed below.

5.2 Dynamic Semantics

The rules of the operational semantics of EGO that differ from the previous system are shown in Figure 10.

The first change is the addition of several rules for `let!`: C-LET$_1$, E-LET$_1$, C-LET$_2$ and E-LET$_2$. Given an expression of the form $\texttt{let!}\ (\varrho)\ x_1 = e_1\ x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}$, these proceed by first evaluating e_1 to a value, of the form ℓ_L. Then a new region, r, is generated for this borrowing, and r and ℓ_L are substituted into e_2 for ρ and x_1 with its linearity annotation changed to reflect that the location is borrowed. Next, e_2 is evaluated. Finally, ℓ_L and v_2 are substituted into e_3 for x_1 and x_2, and the entire expression steps to e_3 with these substitutions.

Another change to the dynamic semantics is that method invocation now instantiates regions over which the method is abstracted. Methods now can be prefixed by a series of abstractions of the form $\Lambda\rho$ which abstract these region variables. An invocation of the form $e.m[\varrho_1, \cdots, \varrho_n]$ looks up this method as before. Upon finding the method, each of these regions or region variables is substituted for the appropriate region variable in e, in addition to a reference to the receiver being substituted for the method's bound variable.

Finally, as locations are now annotated with linearities, $!e$ must change this annotation from a linear object to a nonlinear object.

5.3 Type System

The most significant additions to the calculus are in the type system. The type system is expanded with several mechanisms for `let!` and regions.

The first of these is the change made to object types. We add two new linearities in addition to `obj` and `¡obj`. These are region variables, ρ, and regions, r. Both of these indicate that an object whose type is annotated with this linearity has been borrowed. We allow the same operations to be performed on borrowed objects as on nonlinear objects. This is in line with the motivating intuition that borrowed objects are linear objects that have been made temporarily nonlinear.

The typing judgment is now

$$\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$$

Here, Σ, A, e, τ and l remain the same as before. However, we add two new contexts. The first, P (read as a capital ρ), is a list of regions and region variables which are currently in scope. The second, S, is a partial map from regions to locations which indicates what locations are borrowed at what regions anywhere in the program, and is used for checking the well-typedness of the store. All the typing rules are updated to use the new typing judgment. Most simply pass P and S up the derivation. The exceptions to this are discussed here and shown in Figure 11.

We add rules for the typing of `let!`, T-LET$_1$ and T-LET$_2$. For some expression, $\texttt{let!}\ (\varrho)\ x_1 = e_1\ x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}$, these rules

type `let!` by first finding the type of e_1. The type of e_1 is required to be some object type $L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$. x_1 is now bound with the type $\varrho\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$ in e_2, where ϱ is either ρ, a region variable, or r, a region. ϱ is added to the region context, P to check the type of e_2. This means that the object is borrowed in e_2 and can be aliased but no changes can be made to its interface. If we are typechecking a `let!` currently being evaluated, so locations annotated with this borrowing's region, ℓ_ϱ, have already been substituted into e_2, the presence of ϱ in the region context will let this location be typechecked. Under these contexts, we check the type for e_2 and that it does not contain ϱ, as this would allow aliased locations to be returned as part of the value to which e_2 reduces. Finally, we bind x_1 to its original type, $L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle$, and x_2 to the type of e_2, and we check the type of e_3 under these assumptions to find the type of the whole expression.

In the case where ϱ is some region r, we also check that $r = \ell$ is in the map of borrowings, S. These checks build S up over all `let!` typings in a given derivation to give us a map of all borrowings in a given program state which we use in checking the heap.

We have also added region annotations to method types which indicate the regions that a method body uses, as a method's type does not always show all the regions used in the method body. This allows us to tell from its type the regions invoking a method would use and determine when it is safe to invoke.

In a related vein is the addition of region polymorphism to methods. As mentioned above, this affects method types by prefixing them with a series of region variable bindings of the form $\forall\rho$.

These two changes are reflected in new method typing rules, T-NLINMETH and T-LINMETH. To type a method a binding of the method's bound variable is added to the type context, and the method body is checked under some region context. This region context is a nondeterministic subset of the current region context with the bound type variables added as the method may mention these regions. The method is then given the type $\forall\rho_1.\cdots\forall\rho_n.\tau \xrightarrow{P} \tau'$ or $\forall\rho_1.\cdots\forall\rho_n.\tau \overset{P}{\multimap} \tau'$, where P is the region context under which the method's body was typechecked.

As discussed above, we may want the receiver of a method to contain methods that use regions no longer in scope, as long as these methods are never called. We cannot write down these types, but since these methods can never be called, we can simply give them the type $\top$. This type is supertypes of arrow and lolly method types. We have a series of standard subtyping rules in Figure 12 which show how types which differ only by method type annotations are subtyped. For simplicity, we do not have a subsumption rule. Instead, where it is necessary we be able to type some expression at a supertype, we explicitly allow subtyping. This is needed in method invocation and location typing.

With the above changes, typing invocation is more complex than before. As before, the receiver object is typed and its type unfolded before the method type is looked up. Now, however, the method type will possibly be polymorphic. In this case, we substitute the regions or region variables with which the invocation is instantiated for the abstracted region variables in the method type before comparing it with the receiver type. We no longer check that the types match exactly, but that the receiver is a subtype of the expected type. We also check to make sure that the regions the method call uses are in scope after a similar substitution is done on the method type's region list to guarantee that any region which is used by the method is in scope after instantiation. Finally, we give the invocation the return type of the method after appropriate substitutions for abstracted region variables in this type.

One advantage of the `let!` we have is that it allows borrowed locations to be referenced by methods added to objects on the heap. Since we only check that the regions in the current expression are in scope, we can leave these methods on an object even after the region is out of scope if we do not call these methods. To prevent these methods from being called, we do not allow objects in the executing expression to have methods whose type contains regions out of scope. Instead, any such method types are replaced with $\top$ by typing locations by looking them up in the store typing and giving them a supertype of this type such that no arrows in it are annotated with regions not in scope. Any method types on the object type given the location by the store type that have arrow annotations with regions out of scope are thus replaced by $\top$.

Since the linearity of a borrowed location can be different in different places in an expression, typing locations is slightly more involved. We now find the linearity of a location from the subscript on it, rather than from the type it has in the heap. This is apparent in T-NLINLOC and T-LINLOC. This also arises in typing borrowed locations, as shown in T-BORLOC. To do so, we first type it as either a linear or nonlinear object and then replace the linearity with the region subscripted on the location. This gives it the same type as the location had before it was borrowed with the linearity replaced to indicate that it has been borrowed. We also discard the list of linear locations we get from typechecking the region as an unborrowed pointer because this pointer does not count towards the count of linear locations as it is borrowed.

The final alteration made to the calculus is in typing the store. We still check to make sure every object on the heap has the type the store typing gives it, but typing each object is more complex. Methods on objects in the heap may now contain regions that are not in scope anywhere in the current expression. However, we know that if these regions are not in scope anywhere, these methods cannot ever be called. Because of this, we do not actually care about their type. When checking an entire program state, we have S which contains all regions at which objects are borrowed in the program. We use this to typecheck objects. If a method can be typechecked under the region context defined by S, it could be possibly used in the future and so we check that the method has the type expected by the object's type. Otherwise, we ignore the method while typechecking the object, as it does not matter. This lets methods in the heap mention any region, even if the region is not in scope anywhere in the current program.

5.4 Safety Proof

A proof of type safety for the full version of EGO presented here is in [19]; we describe it here. The proof is similar to the proof we sketch for the earlier EGO fragment. As is standard, type safety consists of two theorems, progress and preservation.

Progress for the full EGO language is slightly different from earlier, updated to use the full typing judgment.

THEOREM 5.1 (Progress). *If* $\Sigma;\cdot;P;S \vdash e{:}\tau \Longrightarrow l_e$ *and* $\Sigma;S \vdash \mu$ `ok` $\Longrightarrow l_s$ *then either* $\mu, e \longrightarrow \mu', e'$ *for some* μ' *and some* e'*, or* e *is a value.*

As before, this is proven by induction on the typing judgments, using a canonical forms lemma.

Preservation is somewhat more complex. It still shows that those properties we want to maintain invariant remain true when a program state steps. The invariants we wish to enforce have changed, however. We now wish to maintain four invariants.

1. The expression has some type τ or a subtype of τ.
2. The heap is well typed.
3. All linear locations are used at most once in the expression, store and the list, S, of aliased locations.
4. All regions in the expression appear in the current region context or are bound by region abstraction or a `let!`.

T-LET!$_1$

$$\frac{\Sigma; A_1; P; S \vdash e_1{:}L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l_1 \qquad \Sigma; A_2, x_1{:}\rho\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle; P, \rho; S \vdash e_2{:}\tau_2 \Longrightarrow l_2 \\ \Sigma; A_3, x_1{:}L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3 \qquad \rho \notin \texttt{tregions}(\tau_2) \qquad \rho \notin P \qquad x_1 \notin \texttt{Dom}(A) \qquad x_2 \notin \texttt{Dom}(A) \qquad x_1 \neq x_2}{\Sigma; A_1, A_2, A_3; P; S \vdash \quad \texttt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ x_3\ \texttt{end}{:}\tau_3 \Longrightarrow l_1, l_2, l_3}$$

T-LET!$_2$

$$\frac{\Sigma; A_1; P; S \vdash \ell_L{:}L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l_1 \\ \Sigma; A_2, x_1{:}r\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle; P, r; S \vdash e_2{:}\tau_2 \Longrightarrow l_2 \qquad \Sigma; A_3, x_1{:}L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3 \\ r \notin \texttt{tregions}(\tau_2) \qquad r \notin P \qquad x_1 \notin \texttt{Dom}(A) \qquad x_2 \notin \texttt{Dom}(A) \qquad x_1 \neq x_2 \qquad r = \ell_L \in S}{\Sigma; A_1, A_2, A_3; P; S \vdash \quad \texttt{let!}\ (r)\, x_1 = \ell_L\, x_2 = e_2\ \texttt{in}\ x_3\ \texttt{end}{:}\tau_3 \Longrightarrow l_1, l_2, l_3}$$

T-NLINMETH

$$\frac{\Sigma; A, x{:}\tau; P'; S \vdash e{:}\tau' \Longrightarrow \{\} \qquad x \notin \texttt{Dom}(A) \qquad A\ \texttt{nonlinear} \qquad P' \subseteq P, \rho_1, \cdots \qquad \rho_1, \cdots \notin P}{\Sigma; A; P; S \vdash \Lambda\rho_1.\cdots\varsigma x{:}\tau.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau \xrightarrow{P'} \tau' \Longrightarrow \{\}}$$

T-LINMETH

$$\frac{\Sigma; A, x{:}\tau; P'; S \vdash e{:}\tau' \Longrightarrow l \qquad x \notin \texttt{Dom}(A) \qquad P' \subseteq P, \rho_1, \cdots \qquad \rho_1, \cdots \notin P}{\Sigma; A; P; S \vdash \Lambda\rho_1.\cdots\Lambda_{\text{¡}}\varsigma x{:}\tau.e{:}\forall\rho_1.\cdots\forall\tau \xrightarrow{P'}\!\!\circ\ \tau' \Longrightarrow l}$$

T-NLINVINV

$$\frac{\Sigma; A; P; S \vdash e{:}L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l \qquad \tau_u = L\mathbf{t}.[\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}]O \leftarrow \langle\!\langle R \rangle\!\rangle \\ \texttt{mtype}(\tau_u, m) = \forall\rho_1.\cdots L\mathbf{t}'.O' \leftarrow \langle\!\langle R' \rangle\!\rangle \xrightarrow{P'} \tau \qquad \cdot \vdash L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \leq [\varrho_1, \cdots/\rho_1, \cdots]L\mathbf{t}'.O' \leftarrow \langle\!\langle R' \rangle\!\rangle \qquad [\varrho_1, \cdots/\rho_1, \cdots]P' \subseteq P}{\Sigma; A; P; S \vdash e.m[\varrho_1, \cdots]{:}[\varrho_1, \cdots/\rho_1, \cdots]\tau \Longrightarrow l}$$

T-LININV

$$\frac{\Sigma; A; P; S \vdash e{:}\text{¡}\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l \\ \text{¡}\texttt{obj}\ \mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle = \text{¡}\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}](O \leftarrow \langle\!\langle R \rangle\!\rangle) \qquad \texttt{lmtype}(\text{¡}\texttt{obj}\ \mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle, m) = \forall\rho_1.\cdots\text{¡}\texttt{obj}\ \mathbf{t}'.O' \leftarrow \langle\!\langle R' \rangle\!\rangle \xrightarrow{P'}\!\!\circ\ \tau \\ \text{¡}\texttt{obj}\ \mathbf{t}.[\mathbf{t}.O''' \leftarrow \langle\!\langle R''' \rangle\!\rangle/\mathbf{t}](O''' \leftarrow \langle\!\langle R''' \rangle\!\rangle) = \text{¡}\texttt{obj}\ \mathbf{t}.(O'' \leftarrow \langle\!\langle R''/m{:}\tau'' \rangle\!\rangle) \\ \cdot \vdash \text{¡}\texttt{obj}\ \mathbf{t}.O''' \leftarrow \langle\!\langle R''' \rangle\!\rangle \leq [\varrho_1, \cdots/\rho_1, \cdots](\text{¡}\texttt{obj}\ \mathbf{t}'.O' \leftarrow \langle\!\langle R' \rangle\!\rangle) \qquad [\varrho_1, \cdots/\rho_1, \cdots]P' \subseteq P}{\Sigma; A; P; S \vdash e.m[\varrho_1, \cdots]{:}[\varrho_1, \cdots/\rho_1, \cdots]\text{¡}\texttt{obj}\ \mathbf{t}.O''' \leftarrow \langle\!\langle R''' \rangle\!\rangle \Longrightarrow l}$$

T-NLINLOC

$$\frac{\Sigma(\ell) = \texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \qquad \cdot \vdash \texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \leq \tau \qquad \texttt{eregions}(\tau) \subseteq P}{\Sigma; A; P; S \vdash \ell_{\text{obj}}{:}\tau \Longrightarrow \{\}}$$

T-LINLOC

$$\frac{\cdot \vdash \text{¡}\texttt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \leq \tau \qquad \texttt{eregions}(\tau) \subseteq P}{\Sigma; A; P; S \vdash \ell_{\text{¡obj}}{:}\tau \Longrightarrow \{\ell\}}$$

T-BORLOC

$$\frac{\Sigma; A; P; S \vdash \ell_o{:}o\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l \qquad \cdot \vdash o\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \leq o\mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle \qquad \texttt{eregions}(\varrho\mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle) \subseteq P}{\Sigma; A; P; S \vdash \ell_\varrho{:}\varrho\mathbf{t}.O' \leftarrow \langle\!\langle R' \rangle\!\rangle \Longrightarrow \{\}}$$

Figure 11. Additions to the Static Semantics of EGO

We formalize the idea of all regions free in a given expression by defining a function, $\texttt{eregions}(e)$, which recursively examines an expression.

THEOREM 5.2 (Preservation). *If*

i. $\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow l_s$
ii. $\Sigma; \cdot; P; S \vdash e{:}\tau \Longrightarrow l_e$
iii. $\texttt{eregions}(e) \subseteq P$
iv. there are are no duplicates in $l_e, l_s, \texttt{Range}(S)$*, and*
v. $\mu, e \longrightarrow \mu', e'$

then for some $\Sigma' \geq_\ell \Sigma$

i. $\Sigma'; S' \vdash \mu'\ \ \texttt{ok} \Longrightarrow l'_s$
ii. $\Sigma'; \cdot; P; S \vdash e'{:}\tau' \Longrightarrow l'_e$
iii. $\texttt{eregions}(e') \subseteq P$
iv. $\cdot \vdash \tau' \leq \tau$*, and*
v. there are no duplicates in $l'_s, l'_e, \texttt{Range}(S)$.

Here $\texttt{eregions}(e)$ recursively walks the expression and returns the set of regions that locations are borrowed at in the expression.

$\Sigma' \geq_\ell \Sigma$ is the same as above: either a new location ℓ was added or the type mapped to by ℓ has changed.

To prove this, we need two substitution lemmas. The first is similar to the one above but now shows that the type produced is a subtype of the expression substituted into if the substituted expression is a subtype of that expected.

LEMMA 5.1 (Substitution). *If* $\Sigma; A, x{:}\tau_1; P; S \vdash e{:}\tau'_1 \Longrightarrow l_e$*,* $\Sigma; \cdot; P; S \vdash v{:}\tau_2 \Longrightarrow l_x$ *and* $\cdot \vdash \tau_2 \leq \tau_1$ *then* $\Sigma; A; P; S \vdash [v/x]e{:}\tau'_2 \Longrightarrow l$*,* $\cdot \vdash \tau'_2 \leq \tau'_1$ *and* $l \subseteq l_e, l_x$.

We also need a similar lemma for region substitution, as both polymorphic instantiation and evaluating `let!` do region substitution. As regions appear in types, the lemma states that substituting a region in for a region variable in an expression substitutes the region in for the region variable in the expression's type.

LEMMA 5.2 (Region Substitution). *If* $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$ *then* $\Sigma; A; [r/\rho]P; S \vdash [r/\rho]e{:}[r/\rho]\tau \Longrightarrow l$.

We also need a Store Change Lemma similar to the one we saw earlier, which says that if have a well typed store, and we change the

$$\text{S-REFL}\quad \frac{}{S \vdash \tau \le \tau} \qquad \text{S-TRANS}\quad \frac{S \vdash \tau_1 \le \tau_2 \qquad S \vdash \tau_2 \le \tau_3}{S \vdash \tau_1 \le \tau_3}$$

$$\text{S-TVAR}\quad \frac{L\mathbf{t_1} \le L\mathbf{t_2} \in S}{S \vdash L\mathbf{t_1} \le L\mathbf{t_2}}$$

$$\text{S-LOC}\quad \frac{S, L\mathbf{t_1} \le L\mathbf{t_2} \vdash O_1 \le O_2 \qquad S, L\mathbf{t_1} \le L\mathbf{t_2} \vdash R_1 \le R_2}{\cdot \vdash L\mathbf{t_1}.O_1 \leftarrow \langle\!\langle R_1 \rangle\!\rangle \le L\mathbf{t_1}.O_2 \leftarrow \langle\!\langle R_2 \rangle\!\rangle}$$

$$\text{S-ROW}_1\quad \frac{}{S \vdash \cdot \le \cdot} \qquad \text{S-ROW}_2\quad \frac{S \vdash \tau_1 \le \tau_2 \qquad S \vdash R_1 \le R_2}{S \vdash R_1, m{:}\tau_1 \le R_2, m{:}\tau_2}$$

$$\text{S-NLINMETH}\quad \frac{S \vdash \tau_1' \le \tau_1 \qquad S \vdash \tau_2 \le \tau_2'}{S \vdash \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P} \tau_2 \le \forall\rho_1.\cdots\forall\rho_n.\tau_1' \xrightarrow{P} \tau_2'}$$

$$\text{S-LINMETH}\quad \frac{S \vdash \tau_1' \le \tau_1 \qquad S \vdash \tau_2 \le \tau_2'}{S \vdash \forall\rho_1.\cdots\forall\rho_n.\tau_1 \overset{P}{\multimap} \tau_2 \le \forall\rho_1.\cdots\forall\rho_n.\tau_1' \overset{P}{\multimap} \tau_2'}$$

$$\text{S-ARROW}\quad \frac{}{S \vdash \forall\rho_1.\cdots\forall\rho_n.\tau \xrightarrow{P} / \overset{P}{\multimap} \tau' \le \forall\rho_1.\cdots\forall\rho_n.\top}$$

Figure 12. Subtyping Rules

$$\text{T-STORE}\quad \frac{\forall\ell \in \mathtt{Dom}(\mu).\Sigma; \cdot; \mathtt{Dom}(S); S \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_\ell \qquad \mathtt{Dom}(\mu) = \mathtt{Dom}(\Sigma)}{\Sigma; S \vdash \mu\ \ \mathtt{ok} \Longrightarrow \text{concat } l_\ell}$$

$$\text{T-ODESCR}\quad \frac{\begin{array}{c}\forall i \in 1..n.\Sigma; A; P'; S \vdash \sigma_i{:}\tau_i \Longrightarrow l_i \text{ if } P' \subseteq P \\ [\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle/\mathbf{t}]R = m_1{:}\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_1', \cdots \\ \sigma_i = \Lambda\rho_1.,\cdots\Lambda\rho_m.[\mathfrak{i}]\varsigma(x{:}\tau_1).e \\ \tau_1 =, \forall\rho_1.\cdots\forall\rho_n.\tau_i' \overset{P',\rho_1,\cdots,\rho_n}{\rightarrow} / \overset{P',\rho_1,\cdots,\rho_n}{\multimap} \tau_i'' \\ \Sigma; A; P; S \vdash loc{:}L'\mathbf{t}'.O' \leftarrow \langle\!\langle R' \rangle\!\rangle \Longrightarrow l_{loc} \\ O = L'\mathbf{t}'.[\mathbf{t}'O'R'/\mathbf{t}'](O' \leftarrow \langle\!\langle R' \rangle\!\rangle) \\ [\mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle/\mathbf{t}]\mathfrak{i}\mathtt{obj}\ \mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle = L\mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle\end{array}}{\begin{array}{c}\Sigma; A; P; S \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle{:} \\ \mathfrak{i}\mathtt{obj}\ \mathbf{t}.O'' \leftarrow \langle\!\langle R'' \rangle\!\rangle \Longrightarrow l_{loc}, l_1, \cdots, l_n\end{array}}$$

Figure 13. Store and Object Typing of EGO

type of a linear location in the store typing and replace the object at that location in the store with an object of this type, the store remains well typed.

LEMMA 5.3 (Store Change). *If*

i. $\Sigma; S \vdash \mu\ \ \mathtt{ok} \Longrightarrow l_s$
ii. $\Sigma; A; P; S \vdash \ell_L{:}\mathfrak{i}\mathtt{obj}\ \mathbf{t}.O \leftarrow \langle\!\langle R \rangle\!\rangle \Longrightarrow l_e$
iii. *there are no duplicates in* $l_s, l_e, \mathtt{Range}(S)$
iv. $\mu(\ell) = s$
v. $\Sigma; \cdot; P; S \vdash s{:}\tau \Longrightarrow l_o$, *and*
vi. $\Sigma; \cdot; P; S \vdash s'{:}\tau' \Longrightarrow l_o'$

then $[\ell \mapsto \tau']\Sigma; S \vdash [\ell \mapsto s']\mu\ \ \mathtt{ok} \Longrightarrow l_s - l_o, l_o'$

6. Related Work

This section gives an overview of previous work in object calculi, linearity, protocol checking and regions.

Our calculus is derived from features of the calculi of Abadi and Cardelli [1] and Fisher, Honsell and Mitchell [15, 16]. Like other object calculi [20, 23, 24], these are focused on modeling issues of inheritance and subtyping. Most of the work studying method addition and delegation is in a functional context, unlike our imperative calculus. Abadi and Cardelli discuss an imperative variant of their calculus, but when a method is imperatively updated it must match the type of the original method, whereas we allow changes to the type of the object as a result of method update.

Our imperative method addition and update, and delegation change are inspired by the prototype-based Self language [29]. Self is dynamically typed, meaning that programs may experience runtime type errors, which our static type system prohibits.

The most closely related work is Anderson et al.'s application of Alias Types to the problem of statically checking imperative method and delegation updates [3]. Compared to EGO, their design achieves precision through singleton types and effects, at a cost of great complexity: the type of a method includes not just the type of the arguments and body, but also the effects of the method and the environment where it was typed. EGO's goal, in contrast, is to support many useful cases of method and delegation update in a comparatively simple and practical type system based on linearity.

Re-classification in Fickle [13] can change an object's class at runtime in class-based OO languages. In this manner class-based OO languages can achieve the same effect as changing delegation at runtime. Fickle is more limited than our system because it restricts re-classification to a fixed set of state classes rather than supporting arbitrary changes to the methods and inheritance hierarchy of an object. Furthermore, because it does not track aliasing of fields, Fickle cannot track the state of an object in a field as EGO does.

Wadler introduced linear type systems in a functional setting in [30]. This work was based on Girard's linear logic [17].

Unique pointers were proposed for Eiffel and C++ in [21], and for Java in [7]. The concept of borrowing was present in Wadler's original `let!` construct, but Wadler used a restrictive typing discipline to ensure that the borrowed reference did not leak; in contrast, we allow the reference to leak but ensure it cannot be used after the region goes out of scope. Unlike Boyland's borrowing proposal [7], regions allow us to store borrowed pointers in the heap.

Odersky [22], extends Wadler's work by allowing observer types as temporary, read-only aliases to mutable, linear objects. Similarly, Fändrich and DeLine [14] introduce adoption and focusing as mechanisms for relaxing linearity. Our work differs from these in that we work with type changing operations, rather than just mutation.

Smith et al. [25] discuss a system for tracking aliasing at a low level for uses in Typed Assembly Language. They can specify a rich

set of constraints on aliasing, and can reuse locations at different types.

Several papers describe research into ways to model objects in linear logic [4, 8, 12]. In [8] methods are characterized as resources that reside within objects, and are consumed after being invoked. We apply this intuition in a more concrete setting (i.e., operational semantics instead of an encoding in logic) for our linear methods.

Typestates were introduced in [26]. DeLine and Fähndrich discuss typestates for objects, especially in the presence of subtyping, in [11]. Their system allows an object to specify which state it is in before and after method calls, and so enforce an ordering on method calls. We model this by modifying delegation to change what methods are available, or by adding and removing methods.

Regions have been proposed for memory management, either using type inference to infer the scopes of regions [28] or with explicit types as in Cyclone [27].

7. Conclusion

We have presented EGO, an object calculus for studying linearity in objects. Our calculus contains powerful mechanisms for creating and using linear objects. We have demonstrated the expressiveness of our calculus with several examples.

We have shown how linearity allows us to manipulate objects so as to enforce protocols in a well-typed way. We can add methods to objects, remove linear methods by invoking them and change delegation at run time in a statically checkable way. We have shown that such abilities can be used to enforce protocols.

We have also shown a way of temporarily relaxing linearity to create short lived aliases. We have shown how to maintain type safety while doing so.

Acknowledgments

This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A and NSF grants CCR-0204047 and CCF-0546550. Thanks to Kevin Bierhoff for an earlier version of the Socket example, to Jason Reed for pointing out a flaw in an earlier version of the calculus and to Karl Crary for suggesting the use of $\top$. Thanks also to the CMU POP group for helpful comments on the calculus and to Timothy Wismer for proofreading.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[3] C. Anderson, F. Barbanera, and M. Dezani-Ciancaglini. Alias and Union Types for Delegation. *Ann. Math., Comput. & Teleinformatics*, 1(1), 2003.

[4] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proc. 7th International Conference on Logic Programming, Jerusalem*, May 1990.

[5] A. Bejleri, J. Aldrich, and K. Bierhoff. Ego: Controlling the power of simplicity. In *Proceedings of the Workshop on Foundations of Object Oriented Languages (FOOL/WOOD '06)*, January 2006.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[7] J. Boyland. Alias Burying: Unique Variables Without Destructive Reads. *Software Practice and Experience*, 6(31):533–553, May 2001.

[8] M. Bugliesi, G. Delzanno, L. Liquori, and M. Martelli. Object calculi in linear logic. *Journal of Logic and Computation*, 10(1):75–104, 2000.

[9] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *European Conference on Object-Oriented Programming*, July 2003.

[10] R. DeLine and M. Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Programming Language Design and Implementation*, pages 59–69. ACM Press, 2001.

[11] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*. Springer-Verlag, 2004.

[12] G. Delzanno and M. Martelli. Objects in forum. In *International Logic Programming Symposium*, pages 115–129, 1995.

[13] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle : Dynamic object re-classification. In *European Conference on Object-Oriented Programming*, pages 130–149, 2001.

[14] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 2002. ACM.

[15] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing*, 1:3–37, 1994.

[16] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Fundamentals of Computation Theory*, 1995.

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

[18] A. Igarashi, B. Pierce, and P. Wadler. Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[19] M. Kehrt, A. Bejleri, and J. Aldrich. Linearity for objects. Technical Report CMU-ISRI-06-115, Carngie Mellon, 2006.

[20] L. Liquori. An extended theory of primitive objects: First order system. In *European Conference on Object-Oriented Programming*, 1997.

[21] N. Minsky. Towards alias-free pointers. In *European Conference on Object-Oriented Programming*, pages 189–209. Springer, 1996.

[22] M. Odersky. Observers for linear types. In *ESOP'92: Symposium proceedings on 4th European symposium on programming*, pages 390–407, London, UK, 1992. Springer-Verlag.

[23] D. Rémy. From classes to objects via subtyping. In *Proc. of European Symposium on Programming*, 1998.

[24] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999.

[25] F. Smith, D. Walker, and J. G. Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381, London, UK, 2000. Springer-Verlag.

[26] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

[27] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe Manual Memory Management in Cyclone. *Sci. Comput. Programming*, October 2006.

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

[29] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242. ACM Press, 1987.

[30] P. Wadler. Linear types can change the world! In *M. Broy and C. Jones, editors, Programming Concepts and Methods*. North Holland, 1990.