Linearity for Objects

Matthew Kehrt

Abstract

Linear type systems guarantee that no copies are made of certain program values. The Ego language is a foundational calculus which adds linearity to object oriented languages. Ego allows changes to be made to the interface of an object, such as the addition or removal of methods, as long as such an object is linear, i.e., there exists only one reference to it. However, this linearity constraint is often unwieldy and hard to program with. We extend Ego with a linguistic primitive for temporarily relaxing the linearity guarantee. Ego allows objects to be linear and enforces that only one reference exists such an object. We allow multiple references to linear objects in certain expressions by borrowing references to these objects. Borrowing annotates the type of the reference with a region, which is a unique token indicating where the reference was borrowed. We disallow references with types containing regions that are not currently borrowed. We use this to temporarily make multiple references to an object in a given expression but enforce that outside this expression only one reference exists.

1 Introduction

Linear type systems allow restrictions to be made on how program values may be copied. Linearity has been studied extensively in functional languages. [21] presents a simple functional calculus with linear types. However, linear types have been less well investigated in other settings.

One area where linear types could be effectively used is that of object oriented languages. Several uses of linearity in such a setting immediately present themselves. The first of these is memory management. Because a linear reference to an object in memory is guaranteed to be the only pointer to the object, when the pointer is no longer in scope, we know that the memory where the object resides can be safely reclaimed with no fear of creating a dangling pointer elsewhere in the program.

Another, particularly compelling use of linearity is in statically checking that an object's methods are called according to a specific protocol. Objects often require the methods they provide to be called in a given order or according to some pattern. Linear references to objects allow us to change the type of an object to reflect its current state without worrying about the type of pointers elsewhere in the code. Encoding the state of the object into its type enables enforcing that only certain methods be called in certain states. [6] presents Fugue, a tool that tracks pointer aliasing for checking protocols in this way.

Unlike linearity in functional languages, work done so far on linearity for objects has been restricted to studying higher level object oriented languages such as Java or Eiffel [2, 15]. We are not aware of any work in linear type systems for a foundational, imperative object calculus. We consider such a calculus to be a useful tool for the study of linear objects in general, both for the insight it itself offers and in that it can be used to model more complex object systems.

Previous work in [4, 5] introduced EGO , a typed, imperative object calculus for studying linearity in object oriented languages. In this paper, we propose a more foundational version of EGO based on the calculus of [1]. This new calculus removes redundancy in the original version by eliminating first class functions, leaving only objects. We then extend this version of EGO with a mechanism for temporarily relaxing linearity based on Wadler's `let!` in [21].

EGO also explores enforcing protocols through several mechanisms based on those found in SELF [20]. SELF introduced *dynamic inheritance*. Objects in SELF have delegee objects, where methods are looked up if method lookup failed on the base object. Dynamic inheritance allows objects to dynamically change this

delegee and thus the methods available to them. This can be used to ensure that methods are called in the order the object expects. SELF also allows methods to be changed or added to objects at runtime, which can also be used to enforce such protocols.

EGO also includes dynamic inheritance and method addition. Unlike SELF , however, it has a static type system that prevents runtime errors. This type system assigns objects types which reflect the current methods that can be called on them. Since the types of objects can change, the type system relies on object linearity to check programs with delegee changing and method addition and update without having to find all other references to the same object.

1.1 Contributions

The contribution of this paper is twofold. We present an object calculus based on previous work in [5]. We have simplified the original calculus by eliminating first class functions. The original work had both objects and functions. We have reworked the language to contain only objects. If needed, first class functions can be modeled with objects.

The calculus has the following properties:

- The calculus is an *object calculus*. It models a language in which objects are the primary construct. It does not have classes; instead, objects are built from primitives for creating empty objects and adding methods to them.
- The calculus is *typed*. In addition to a description of the runtime behavior, we present a type system. This type system staticly checks the type safety of programs to guarantee the absence of runtime errors.
- This calculus provides mechanisms for both *linear methods* and *linear objects*. Linear objects are those with only one pointer to them. Linear methods may only be called once.
- The calculus provides ways to change objects at runtime. We allow methods to be added to objects and delegation to be changed during program execution in a well-typed manner.

The other contribution is the addition of a mechanism for temporarily relaxing linearity. Since we use linearity to allow type changes, we can allow methods whose type is not being changed to be temporarily aliased, or *borrowed*. We use regions to track where in a program a given object is borrowed and guarantee no aliases to a borrowed object escape.

1.2 Paper Layout

The rest of the paper is arranged as followed.

- We start in section 2 by presenting a simplified version of EGO with no mechanism for relaxing linearity. We discuss the intuition behind the language and then discuss some examples in detail.
- In section 3, we present he formal system of the language. We conclude this section by sketching a proof of type safety for the the language.
- Section 4 discuses how to add regions to the language to relax linearity. We again discuss the intuition behind this and then some examples.
- In section 5 We show how the previous formalization needs to be changed to add this mechanism. We also discuss the proof of type safety.
- Section 6 discusses related work, and
- section 7 concludes.

2 Simplified EGO

This section introduces a simplified version of EGO , which is based on previous work on EGO in [5]. This simplified EGO lacks a construct for relaxing linearity. We briefly discuss the intuition behind the language. We end by discussing several examples to motivate the language.

2.1 Intuition

Intuitively, programs in EGO proceed by manipulating objects. An object consists of a record of methods and possibly a delegation pointer to another object. Methods can be added to this record or the delegation pointer changed, or methods can be invoked on an object.

A program in EGO consists of a mutable store and an expression. The store is a partial map from abstract locations to objects. Expressions are built of primitives for modifying objects, which can contain other expressions. Our primitives are based on those in [11, 12]. They consist of

- $\langle\rangle$, creates a new object on the heap and returns a reference to it.
- $e_1 \leftarrow\!\!+ m = \sigma$ adds the method σ to the object on the heap referred to by e with the name m, or changes the method named m to be σ if it already exists in the object to which e refers.
- $e_1 \leftarrow e_2$ changes the delegee 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.

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

All of the primitives return a reference to an object on the heap. The first three return a reference to the object they create or modify. The fourth executes a method body and returns what the method evaluates to; since method bodies will be composed of these four primitives, they will return a location as well.

Methods in EGO are based on [1]. A method is of the form $\varsigma(x{:}\tau).e$. A method is invoked on an object, called the method's *receiver*, which need not be the object that contains the method. When invoked on an object, a method is looked up by searching the object's record of methods for the invoked method. If the method exists, it is invoked; otherwise, the object's delegee is is searched and lookup recurses up the hierarchy of objects. Once a method is found, invocation substitutes all occurrences of x, the variable it binds, in e, its body, with a reference to the object on which it was invoked. This our only way of abstracting expressions. Lambda abstraction, if needed, can be defined in terms of objects and methods.

EGO allows both methods and objects to be *linear*. A linear object is one to which only one reference is allowed. A linear method is one which can be called only once. Calling a linear method consumes it, and it is removed from the object that contains it. To allow static typing, only linear objects are allowed to have linear methods called, add methods or change delegation, as these constitute changes to an object's type. In EGO , all objects start out linear. Changes can be made to their interface while they are linear, and their type can be changed to reflect this. When all of the necessary changes have been made, the type can be changed irreversibly to be nonlinear. No further changes can be made to the interface of the object, but the object can then be freely aliased.

2.2 Examples

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

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

$$\langle\rangle \leftarrow\!\!+ m = \varsigma(\mathit{this}{:}\text{¡}\texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow m{:}\text{¡}\texttt{obj}\mathbf{t} \rightarrow \text{¡}\texttt{obj}\mathbf{t}).\mathit{this}$$
$$\quad \leftarrow\!\!+ m = \varsigma(\mathit{this}{:}\text{¡}\texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow m{:}\text{¡}\texttt{obj}\mathbf{t} \rightarrow \text{¡}\texttt{obj}\ \mathbf{t}'.\langle\rangle \leftarrow \cdot).\langle\rangle$$

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 delegee of this second object to be the first object,

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

In the next example, we create a new object, add a linear method to it that returns the receiver, and invoke the method. This removes the method from the object, so this code fragment produces a reference to an empty object. Since the invocation removes the method from the receiving object, the type of the object the method expects does not contain the method.

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

It is not immediately obvious from the examples so far that the EGO system is flexible enough to be useful as a model for a programming language. The remaining examples demonstrate EGO 's flexibility.

We can define a lambda term of type $\tau \rightarrow \tau'$ as follows. This is based on a similar embedding shown by [1].

$$\lambda x{:}\tau.e \stackrel{\text{def}}{=} \langle gen = \varsigma(\_{:}\texttt{obj t}.\langle\rangle \leftarrow gen{:}\texttt{objt} \rightarrow ¡\texttt{obj t}.\langle\rangle \leftarrow body{:}\texttt{objt} \rightarrow \tau').\langle body = \varsigma(y{:}O).[y.arg/x]e\rangle\rangle$$

where

$$O \stackrel{\text{def}}{=} \texttt{obj t}.\langle\rangle \leftarrow body{:}\texttt{objt} \rightarrow \tau', arg{:}\texttt{objt} \rightarrow \tau$$

This works by creating a new object and adding a single method, gen. This object can be freely aliased, as after creating it we no longer change its type. 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.

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

$$e_1 e_2 \stackrel{\text{def}}{=} (e_1.gen \leftarrow\!\!+ arg = \varsigma(\_{:}O).e_2).body$$

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 method called arg which simply returns the argument of the application, e_2. Then $body$ is called on the new linear object. Since the bound variable in the function body is modeled as calling arg on the receiver and arg on the receiver now returns the application argument, this substitutes the argument into the function as we would expect a lambda calculus reduction to do.

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.

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

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.

$$¡\lambda x{:}\tau.e \stackrel{\text{def}}{=} \langle body = ¡\varsigma(y{:}O').[y.arg/x]e, arg = \varsigma(z{:}O').z.arg\rangle$$

where

$$O' \stackrel{\text{def}}{=} \texttt{obj t}.\langle\rangle \leftarrow arg{:}\texttt{objt} \rightarrow \tau$$

When e_1 is $¡\lambda x{:}\tau.e : \tau'$ and $e_2{:}\tau$

$$e_1 e_2 \stackrel{\text{def}}{=} (e_1 \leftarrow\!\!+ arg = \varsigma(\_{:}O').e_2).body$$

$typedef\ closedType = \text{¡obj } \mathbf{t_1}.\text{¡obj } \mathbf{t_2}.\langle\rangle \leftarrow \cdot \leftarrow \cdot$
$typedef\ readType = \text{¡obj } \mathbf{t_1}.\text{¡obj } \mathbf{t_2}.\langle\rangle \leftarrow read{:}\text{¡obj}\mathbf{t_1} \rightarrow \text{¡obj}\mathbf{t_1}, close{:}\text{¡obj}\mathbf{t_1} \rightarrow closedType \leftarrow \cdot$
$typedef\ openType = \text{¡obj } \mathbf{t_1}.\text{¡obj } \mathbf{t_2}.\langle\rangle \leftarrow open{:}\text{¡obj}\mathbf{t_1} \rightarrow readType \leftarrow \cdot$

```
let ClosedSocket = ⟨⟩ in
let ReadSocket = ⟨
      read = ς(this:readType)./*read from a socket*/
      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 example is slightly different than the one above. Since this example invokes a linear method on an object, the object is linear. Because the object is linear, we can add the argument directly to it, rather than calling a gen method to generate a new linear object to which we can add methods. Application, then, no longer calls gen to get an object representing the function body, but does the method addition and invocation to simulate application directly to e_1.

Note that a linear method contained in the object simulating a function was called in application. Since the linear method is consumed, the object must be linear. However, the $body$ method expects a nonlinear object so that its argument can appear in the body multiple times. This is allowed, as we allow linear invocation to change the linearity of the method's argument for cases such as this one.

A more complex and realistic example is that of a a network socket object, given in Figure 1. In this example, we use a $typedef$ construct to simplify presentation; however, this construct is not part of the calculus. The example also used let and the sequence operator as defined above.

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

The methods in this example add and remove methods by changing delegation. There are secondary objects corresponding to the three states a socket can be in (able to be opened, open and closed). Each of these is delegated to at a different point in the object's lifetime. $Socket$ starts as an empty object which is delegated to $OpenSocket$, an object containing the $open$ method. Calling this method changes the delegation of $Socket$ to $ReadSocket$, which contains $read$ and $close$ methods. Finally, calling $close$ changes delegation to $ClosedSocket$, which is empty.

The pattern used here is of note. In this code, objects are created that correspond to states in the lifecycle of an object. Each object representing a state the object might be in has a series of methods added to it that are appropriate to that state. We then create a base object and transition from state to state by changing delegation of this object to the object corresponding to the state we are entering. This pattern allows us to create and enforce protocols on method use. We can therfore guarantee that only methods appropriate to the current object state exist on that object at a given time.

3 Formalism for Simplified EGO

In this section we discuss the formalism we use to describe this fragment of EGO . We first present the syntax of the language. Then we present the operational semantics and the type system. Finally, we sketch

Expressions	e	::=	$x, y \mid \langle\rangle \mid e.m \mid e \leftarrow\!\!+ m = \sigma \mid e_1 \leftarrow e_2 \mid v$
Values	v	::=	$loc \mid \sigma$
Locations	loc	::=	$\texttt{null} \mid \ell$
Stores	μ	::=	$\cdot \mid \mu, \ell \mapsto s$
Object Descriptors	s	::=	$\langle\rangle \mid loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle$
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	::=	$L\mathbf{t} \mid \langle\rangle \mid L\mathbf{t}.O \leftarrow R$
Linearities	L	::=	$\texttt{obj} \mid ¡\texttt{obj}$
Rows	R	::=	$\cdot \mid R, m{:}\tau$

Figure 2: Syntax of Simplified EGO

a proof of type safety.

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 of the form $(\ell \mapsto s)*$, where ℓ is an abstract location and s is an object descriptor. An object descriptor is either the empty object descriptor, $\langle\rangle$, or of the form $loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle$. Here loc is either a reference, ℓ, to an object's delegee or `null`, which indicates the object has no delegee, and $\langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle$ is 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 the linear method.

An expression is either a variable, x, a new method creation, $\langle\rangle$, a method add or update, $e \leftarrow\!\!+ m = \sigma$, a delegation change $e_1 \leftarrow e_2$ or a variable x or y. We also count locations, loc, which are either `null` or ℓ, and methods as expressions but they are intermediate forms which do not occur in user code. loc and methods are the only values; all programs evaluate to a one of these expressions or diverge.

The types, τ, of expressions are based on those used in [12]. These are either the types of methods or object types, O. The types of methods are of the form $\tau \rightarrow \tau'$ or $\tau \multimap \tau'$. Here τ represents the type of the the object the method will be called on and τ represents the type of the object to which it evaluates.

Object types O are either type variables $L\mathbf{t}$ or of the form $\langle\rangle$ or $L\mathbf{t}.O \leftarrow R$. $\langle\rangle$ is the type of `null` and $L\mathbf{t}.O \leftarrow R$ the type of ℓ. L is either `obj` or `¡obj` which indicate whether the object is linear or nonlinear respectively. In $L\mathbf{t}.O \leftarrow R$, R is a row of the form $m_1{:}\sigma_1, \cdots, m_n{:}\sigma_n$, which specify the method types of the methods in the base object, and O is the type of the object's delegee. As a method in an object may mention the object in the form of the variable bound by the method, it may be necessary for the type of an object to mention itself; `obj`$\mathbf{L}t$ is a type variable which is bound here to represent that type.

3.2 Operational Semantics

This section discusses how the expressions of EGO are evaluated and the effect evaluation has on the heap and the objects it contains. The rules defining how these expressions are evaluated are in Figure 3.

The simplest primitive is $\langle\rangle$, whose semantics are defined by E-NEW. $\langle\rangle$ extends the heap with a new mapping, from some fresh ℓ to `null` $\leftarrow \langle\rangle$, that is, an empty object descriptor. $\langle\rangle$ evaluates to ℓ: it returns a pointer to the new location.

$e \leftarrow\!\!+ m = \sigma$ is defined by E-ADD and E-UPD and the rule C-UPD, which reduces e to a value. The first allows the addition of methods to an existing object that does not already contain a method with that name. It modifies the store such that the location pointed to by the value of e has $m = \sigma$ added to its record of methods. E-UPD, on the other hand, replaces an existing $m = \sigma'$ with $m = \sigma$.

Figure 3: Dynamic Semantics of Simplified EGO

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, e.m \longrightarrow \mu', e'.m}\ C-Inv$$

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

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

$$\frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma, \cdots \rangle}{\texttt{mbody}(\mu, \ell, m) = \sigma}\ mbody_1$$

$$\frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots \rangle \\ m = \sigma \notin \langle m_1 = \sigma_1, \cdots \rangle \\ \texttt{mbody}(\mu, loc, m) = \sigma\end{array}}{\texttt{mbody}(\mu, \ell, m) = \sigma'}\ mbody_2$$

$$\frac{\ell\ \texttt{fresh} \quad \mu' = [\ell \mapsto \texttt{null} \leftarrow \langle\rangle]\mu}{\mu, \langle\rangle \longrightarrow \mu', \ell}\ E-New$$

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

$$\frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle \\ \forall i. m \neq m_i \\ \mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n, m = \sigma \rangle]\mu\end{array}}{\mu, \ell_L \leftarrow\!\!+ m = \sigma \longrightarrow \mu', \ell}\ E-Add$$

$$\frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma, \cdots, m_n = \sigma_n \rangle \\ \mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma', \cdots, m_n = \sigma_n \rangle]\mu\end{array}}{\mu, \ell \leftarrow\!\!+ m = \sigma' \longrightarrow \mu', \ell}\ E-Upd$$

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

$$\frac{\mu, e_2 \longrightarrow \mu', e_2'}{\mu, e_1 \leftarrow e_2 \longrightarrow \mu', e_1' \leftarrow e_2}\ C-Deleg_0$$

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, \ell \leftarrow e \longrightarrow \mu', \ell \leftarrow e'}\ C-Deleg_1$$

$e_1 \leftarrow e_2$ is defined by E-DEL. C-DEL$_1$ and C-DEL$_2$ reduce e_1 and e_2, respectively, to values. E-DEL reduces $\ell_1 \leftarrow \ell_2$ to ℓ_2 and modifies the store. It changes the delegation link on the object descriptor in the store to which ℓ_1 points to ℓ_1. That is, if ℓ_2 maps to some $loc \leftarrow \langle \cdots \rangle$, this expression changes this to $\ell_1 \leftarrow \langle \cdots \rangle$.

$e.m$ invokes a method. Its semantics are defined by E-NLININV and E-LININV. C-INV reduces e to a value. The two invocation rules takes something of the form $\ell.m$ and look up the called method in the heap as follows. They look up the object descriptor that ℓ refers to in the heap. In the case of invocation of linear methods, the method body is found in the record of methods in the object descriptor. In the case of invocation of nonlinear methods, method lookup is slightly more complicated: if it is found in the record of methods in the object descriptor pointed to by ℓ, this method body is returned. Otherwise, the method body is searched for recursively in the delegee of ℓ. Then, in either case, ℓ, the method receiver, is substituted for the variable bound by the method into the method body. In the case of linear method invocation, the store is modified by removing the method from the object that contains it simultaneously with the substitution, as invocation of a linear method consumes the method.

3.3 Type System

EGO 's type system prevents an EGO program from getting into a state from which the dynamic semantics do not define an reduction. One specific stuck state we wish to avoid is that in which a method is called which does not exist on the method's receiver or the receiver's delegees. To prevent this and other stuck states, we type an expression only if the dynamic semantics define a reduction for it.

One important function of the type system is that it maintains the distinction between linear and nonlinear objects. Object types are annotated with linearities. We allow aliases only to be made of nonlinear references.

The type system allows changes to the interface of an object, such as method add or update and delegation change, only to linear objects. This is because changes to objects are imperative: they affect the object descriptor on the heap. Since an object's type reflects its interface, a local change to the interface of an object has global changes on the type of the object. If we allow pointers to be aliased, it becomes impossible to keep track of the global changes of their types. Thus, we allow changes to an object's interface only on linear objects.

Interface changes are also allowed only to the object on which they are performed, not its delegees, for similar reasons. Specifically, a method on an object cannot be changed through a reference to an object that delegates to the object containing the method. Instead, a direct reference to the containing object is needed. 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 delegee as do with changing the interface of a nonlinear object.

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

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

.

Here, Σ is the store typing. For an expression to be well-typed, every use of a location in the expression must use it with the same type. Σ consists of a mapping, $(\ell \mapsto \tau)*$ from locations to types. This mapping is used to look up the types of locations, which guarantee that all uses of a location in an expression have the same type. A the type context, which gives the type of all free variables in the expression. 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, as shown in T-LINLOC and T-NLINLOC. T-LINLOC also puts the location it types into the list of linear locations, l, as this is a linear location used in the expression.

We also can type any linear location as a nonlinear location with the rule T-CHLIN. $\langle\rangle$ produces a new linear object to which interface changes can be made. We get nonlinear objects by typing this as nonlinear. This is safe as it we can only go one way: we cannot type a linear location as a nonlinear one. We allow

$$\frac{}{\Sigma; x{:}\tau \vdash x{:}\tau \Longrightarrow \{\}}\ T-Var$$

$$\frac{\Sigma; A, x{:}\tau \vdash e{:}\tau' \Longrightarrow l \quad x \notin \mathtt{Dom}(A) \quad A\ \mathtt{nonlinear}}{\Sigma; A \vdash \varsigma x{:}\tau.e{:}\tau \to \tau' \Longrightarrow l}\ T-NLinMeth$$

$$\frac{\Sigma; A, x{:}\tau \vdash e{:}\tau' \Longrightarrow l \quad x \notin \mathtt{Dom}(A)}{\Sigma; A \vdash ¡\varsigma x{:}\tau.e{:}\tau \multimap \tau' \Longrightarrow l}\ T-LinMeth$$

$$\frac{\Sigma(\ell) = \mathtt{obj}\ \mathbf{t}.O \leftarrow R}{\Sigma; A \vdash \ell{:}\tau \Longrightarrow \{\}}\ T-NLinLoc$$

$$\frac{\Sigma(\ell) = ¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R}{\Sigma; A \vdash \ell{:}\tau \Longrightarrow \{\ell\}}\ T-LinLoc$$

$$\frac{}{\Sigma; A \vdash \mathtt{null}{:}\langle\rangle \Longrightarrow \{\}}\ T-Null$$

$$\frac{\Sigma; A \vdash e{:}\tau \Longrightarrow l}{\Sigma; A, x{:}\tau' \vdash e{:}\tau \Longrightarrow l}\ T-Kill$$

$$\frac{\Sigma; A, x{:}\tau', x{:}\tau' \vdash e{:}\tau \Longrightarrow l \quad \tau'\ \mathtt{nonlinear}}{\Sigma; A, x{:}\tau' \vdash e{:}\tau \Longrightarrow l}\ T-Copy$$

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

$$\frac{\begin{array}{c}\Sigma; A \vdash \sigma{:}\tau \Longrightarrow l \\ \Sigma; A' \vdash e{:}¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R \Longrightarrow l' \\ \mathtt{lmtype}(¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R, m) = \mathtt{DNE}\end{array}}{\Sigma; A, A' \vdash e \leftarrow\!\!+ m = \sigma{:}¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R, m{:}\tau \Longrightarrow l, l'}\ T-Add$$

$$\frac{\begin{array}{c}\Sigma; A \vdash e{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l \\ \mathtt{mtype}(L\mathbf{t}.O \leftarrow R, m) = L'\mathtt{objt}'.O' \leftarrow R' \to \tau \\ L\mathbf{t}.O \leftarrow R = [\mathbf{t}'O' \leftarrow R'/\mathbf{t}']L'\mathtt{objt}'.O' \leftarrow R'\end{array}}{\Sigma; A \vdash e.m{:}\tau \Longrightarrow l}\ T-NLinInv$$

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

$$\frac{\Sigma; A \vdash e{:}¡\mathtt{obj}\ \mathbf{t}.R \leftarrow\!\!\Longrightarrow l}{\Sigma; A \vdash e{:}\mathtt{obj}\ \mathbf{t}.R \leftarrow\!\!\Longrightarrow l}\ T-ChLin$$

$$\frac{}{\Sigma; A \vdash \langle\rangle{:}¡\mathtt{obj}\ \mathbf{t}.\langle\rangle \leftarrow \cdot \Longrightarrow \{\}}\ T-New$$

$$\frac{\Sigma; A \vdash e_2{:}¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R \Longrightarrow l \quad \Sigma; A' \vdash e_1{:}O' \Longrightarrow l'}{\Sigma; A, A' \vdash e_1 \leftarrow e_2{:}L\mathbf{t}.O' \leftarrow R \Longrightarrow l, l'}\ T-Del$$

Figure 4: Static Semantics of Simplified EGO

$$\frac{m{:}\tau \in R}{\texttt{lmtype}(L\mathbf{t}.O \leftarrow R, m) = \tau}\ T - LMethT$$

$$\frac{\texttt{lmtype}(L\mathbf{t}.O \leftarrow R, m) = \tau}{\texttt{mtype}(L\mathbf{t}.O \leftarrow R, m) = \tau}\ T - MethT_1$$

$$\frac{\texttt{lmtype}(L\mathbf{t}.O \leftarrow R, m) = \texttt{DNE} \quad \texttt{mtype}(O, m) = \tau}{\texttt{mtype}(L\mathbf{t}.O \leftarrow R, m) = \tau}\ T - MethT_2$$

Figure 5: Method Type Lookup in Simplified and Full EGO

interface changes until we alias an object, at which point it becomes nonlinear, and we can no longer make interface changes.

The typing of method invocation is by the rules T-LININV and T-NLinInv. The receiver object is typed with a type of the form $L\mathbf{t}.O \leftarrow R$. The type of the method is looked up in this type. Since the invocation of a linear method changes the type of the receiver object as described below, and we cannot change the interface of delegees, we only look up linear method types in the row, R, which describes the types of the methods contained in the base object. However, invocation of nonlinear methods on an object does not change the interface of the object, so we can look such method up recursively in O if it is not in R. After this, we check that the recursive object type the method expects unfolds to the type the receiver will have invocation. We do this for nonlinear methods by checking that if we unfold the type the method expects by substituting this type into the recursively bound variable we get the receiver type.

For nonlinear methods, we must do a little 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 unfolded type the method expects with the receiver type as we do with nonlinear types, we check equality of the unfolded type with that of the receiver type minus the invoked method, as this is what will be substituted into the method body.

Method addition and update is checked with T-UPD and T-ADD. These rules check the type of the object as some $\texttt{¡obj t}.O \leftarrow R$. They unfold this type by substituting this type into $\texttt{¡objt}$ in $O \leftarrow R$. Then they give this expression the type found by adding or updating the appropriate method and folding the object back up. This maintains the recursive type, as the type of the object is folded at itself.

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

Delegation is typed similarly to addition and update. The type of the expression whose delegation is being changes is unfolded. The expression is given this type with the type of the delegee changed and folded back up. This keeps the object type folded at itself.

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

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.

We enforce linearity by splitting the context when we type subexpressions. This is the approach taken by [21] modeled on the same technique from Girard's linear logic [13]. Since, for a given expression, we can only use a variable in one subexpression, the variable can only be used once, and so anything substituted in for the variable can only be used once. We allow aliases to nonlinear objects through T-COPY, which makes copies of the variable's binding in the context.

We also have a store typing Judgement, 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 delegees have the correct type. It also returns 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 store.

$$\frac{\forall \ell \in \mathtt{Dom}(\mu).\Sigma;\cdot \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_\ell \quad \mathtt{Dom}(\mu) = \mathtt{Dom}(\Sigma)}{\Sigma;\vdash \mu \ \ \mathtt{ok} \Longrightarrow \text{concat } l_\ell}\ \mathit{Store}$$

$$\frac{\begin{array}{c}\forall i \in 1..n.\Sigma; A \vdash \sigma_i{:}\tau_i \Longrightarrow l_i \\ R = m_1{:}\tau_1, \cdots, m_n{:}\tau_n \\ \Sigma; A \vdash \mathit{loc}{:}O \Longrightarrow l\end{array}}{\Sigma; A \vdash \mathit{loc} \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle{:}L\mathtt{objt}.O \leftarrow R \leftarrow\!\!\Longrightarrow l_{loc}, l_1, \cdots, l_n}\ T - ODescr$$

Figure 6: Store and Object Typing for Simplified EGO

3.4 Safety Proof

We have no formal proof of safety for the fragment of EGO presented so far. Rather we have a proof of the safety of the entire system. However, we sketch such a proof below.

Type safety consists of two lemmas. The first is progress. Progress states that a program that consists of the pair of a well typed store and a well typed expression can always be reduced to a new program if the expression is not already a value.

Progress: If $\Sigma; A \vdash e{:}\tau \Longrightarrow l$ and $\Sigma;\vdash \mu \ \ \mathtt{ok} \Longrightarrow l$ then $\mu, e \longrightarrow \mu', e'$ or e is a value.

The proof of this is by induction on the derivation of $\Sigma; A \vdash e{:}tau \Longrightarrow l$. For each case, we show that if the expression is correctly typed, it has a form which is either a value, or some subexpression of which can reduce, or which itself reduces. To do this, we need a canonical forms lemma.

Canonical Forms:

1. If a value has the type $\tau_1 \rightarrow \tau_2$, it has the form $\sigma x{:}\tau.e$.
2. If a value has the type $\tau_1 \multimap \tau_2$, it has the form ¡$\sigma x{:}\tau.e$.
3. If a value has the type $L\mathtt{t}.O \leftarrow R \leftarrow$, it has the form ℓ.
4. If a value has the type $\langle\rangle$, it has the form `null`.

The proof of this is by case analysis on the typing rules.

Preservation states in general that if a pair of store and expression have certain properties we wish to maintain invariant, and this program reduces to another, the new program will also maintain these 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 in one of three ways. The first is that the store type is extended with a new location, ℓ. The second is that the linearity of location ℓ has been changed from linear to nonlinear. The third is that the type of the linear location ℓ has been arbitrarily changed. By limiting the store change to these three changes, we can show that a changed store is still well typed, as it changes in a limited number of ways.

Preservation If $\Sigma;\cdot \vdash e{:}\tau \Longrightarrow l_e$, $\Sigma;\vdash \mu \ \ \mathtt{ok} \Longrightarrow l_s$, there are no duplicates in l_e, l_s and $\mu, e \longrightarrow \mu' e'$, then for some $\Sigma' \geq_\ell \Sigma$, $\Sigma';\cdot \vdash e'{:}\tau \Longrightarrow l'_e$, $\Sigma';\vdash \mu' \ \ \mathtt{ok} \Longrightarrow l'_s$ and there are no duplicates in l'_e, l'_s.

The proof of this is by induction on the derivation of $\mu, e \longrightarrow \mu', e'$. For each of possible way to derive this, we show that if the program on the left maintains our invariants, so does the program on the right.

To prove this, one important lemma is needed. Since methods do substitution, we need to show that the substitution of well typed expressions into well typed expressions maintains well typedness.

Substitution If $\Sigma; A, x{:}\tau' \vdash e{:}\tau \Longrightarrow l$ and $\Sigma;\cdot \vdash e'{:}\tau' \Longrightarrow l'$, then $\Sigma; A \vdash [e'/x]e{:}\tau \Longrightarrow l, l'$.

The proof of this is by induction on the typing rules.

4 Relaxing Linearity

The fragment of the EGO language presented so far is powerful but has a significant drawback. The restrictions on linear objects are often counterintuitive. Linearity guarantees that we can make changes to the interface of an object while retaining the ability to statically check its type. However, often it is useful to be able to make temporary aliases on an object and then to regain the ability to change its interface once all of these aliases are no longer available. 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. 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. Methods can be added and later removed by changing delegation, but we cannot allow temporary aliases. We now introduce a construct based on `let!` as presented in [21]. This construct allows 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. Each one binds a new variable to be used in the successive expressions. The first expression evaluates to a linear object which is bound to a variable. In the seconds expression, this variable is bound with a *borrowed* object type. This borrowed type can be freely aliased but no changes can be made to its interface. 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 linear type, as it is no longer borrowed.

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

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.

In the following example, x is bound to a reference to an object containing only an id method that returns the receiver. Then this object is borrowed in the second subexpression which evaluates to an object containing a single method that returns an empty object. This new object is bound to y. Note that this second subexpression does not mention x. Finally, outside the scope of the borrowing, both methods are called. The whole expression evaluates to an empty object.

```
let! (ρ)
   x = ⟨⟩ ←+ id = ς(this:¡obj t.⟨⟩ ← id:¡objt →̇ ¡objt).this
   y = ⟨⟩ ←+ new = ς(this′:¡obj t.⟨⟩ ← new:¡objt →̇ ¡obj t.⟨⟩ ← ·).⟨⟩
in
   y.id; x.new
end
```

Several other modifications the existing calculus need to be made to accomodate regions.

The first modification 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 have type annotations that refer to regions not in scope. We do this

by region abstraction. Methods may 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. This allows regions to be referred to in code and to be instantiated.

The following example uses region polymorphism. It binds an object containing a single method that returns a new object to x. The method is polymorphic in the region of its receiver. In the type the method expects its receiver to have, the method itseld 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 the method call is bound to y, which is, in the the third subexpression, returned as the value of the entire expression.

```
let! (ρ1)
   x = ⟨⟩ ←+ new = Λρ2.ς(this:ρ2objt.⟨⟩ ← new:∀ρ3.ρ3t →̇ ¡obj t.⟨⟩ ← ·).⟨⟩
   y = x.new[ρ1]
in
   y
end
```

Another modification is that we 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 method uses the receiver in its body before returning an empty object. Since it uses a region in its body, the type of the method in the type annotation must mention the 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:∀ρ3.ρ3t⟨⟩new:objt →^ρ3 ¡obj t.⟨⟩ ← ·).(this; ⟨⟩)
   y = x.new[ρ1]
in
   y
end
```

The final modification arises from a smilar 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 allowing methods to have types which are simply unannotated arrows. This indicates that the method is uncallable. These unannotated arrows are supertypes of annotated arrows, so can be used on method type annotations which will then accept objects whose types contain unknown regions.

This example shows the use of such an unannotated arrow. 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 stil 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 type annotation for the method we call, we give this method a type which has no region annotations. 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.

$typedef\ closedType = \text{¡}\texttt{obj}\ \mathbf{t}.\text{¡}\texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow \cdot \leftarrow \cdot$
$typedef\ readType = \text{¡}\texttt{obj}\ \mathbf{t_1}.\text{¡}\texttt{obj}\ \mathbf{t_2}.\langle\rangle \leftarrow read{:}\forall\rho_2.\rho_2\mathbf{t_1} \dot{\rightarrow} \rho_2\mathbf{t_1}, close{:}\text{¡}\texttt{obj}\mathbf{t_1} \dot{\rightarrow} closedType \leftarrow \cdot$
$typedef\ openType = \text{¡}\texttt{obj}\ \mathbf{t_1}.\text{¡}\texttt{obj}\ \mathbf{t_2}.\langle\rangle \leftarrow open{:}\text{¡}\texttt{obj}\mathbf{t_1} \dot{\rightarrow} readType \leftarrow \cdot$

```
let ClosedSocket = ⟨⟩ in
let ReadSocket = ⟨
      read = Λρ1.ς(this:readType)./*read from a socket*/
      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*/

Socket.open;
let! (ρ2)
      Socket′ = Socket
      SomeData = /*Code that aliases Socket′*/
in
      /*More Code*/
      Socket′.close
end
```

Figure 7: A socket used with `let!`

$typedef\ receivertype = \texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow (meth_1{:}\texttt{objt} \dot{\rightarrow} \text{¡}\texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow \cdot, \texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow meth_2{:}\texttt{objt} \rightarrow \text{¡}\texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow \cdot)$

```
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 implement the socket example described at the begining of this section. In fact, this example is fairly simple; it is given in Figure 7. It is based on the previous socket example in Figure 1. The main differences are that the $read$ method is now parameterized over a region. After defining the objects, we open the socket and then borrow it as $Socket'$. Now we can freely alias $Socket'$. We can use the borrowed socket by calling $read$ with 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 the expression, we no longer have access to any aliases of $Socket'$, so we can close the socket.

5 Formalism

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

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x, y \mid v \mid e.m[L_1, \cdots, L_n] \mid e \leftarrowtail m = \sigma \mid e_1 \leftarrow e_2 \\
 & & & \mid \langle\rangle \mid \texttt{let!}\ (\varrho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end} \\
\text{Values} & v & ::= & loc \mid \sigma \\
\text{Locations} & loc & ::= & \texttt{null} \mid \ell_L \\
 & & & \\
\text{Stores} & \mu & ::= & \cdot \mid \mu, \ell \mapsto s \\
\text{Object Descriptors} & s & ::= & \langle\rangle \mid loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle \\
\text{Methods} & \sigma & ::= & \Lambda\rho_1. \cdots \Lambda\rho_n.\varsigma(x{:}\tau).e \mid \Lambda\rho_1. \cdots \Lambda\rho_n.¡\varsigma(x{:}\tau).e \\
 & & & \\
\text{Types} & \tau & ::= & O \mid \forall\rho_1. \cdots \forall\rho_n.\tau \xrightarrow{P} \tau' \mid \forall\rho_1. \cdots \forall\rho_n.\tau \overset{P}{\multimap} \tau' \\
 & & & \mid \forall\rho_1. \cdots \forall\rho_n.\tau \to \tau' \mid \forall\rho_1. \cdots \forall\rho_n.\tau \multimap \tau' \\
\text{Object Types} & O & ::= & L\mathbf{t} \mid \langle\rangle \mid L\mathbf{t}.O \leftarrow R \\
\text{Linearities} & L & ::= & o \mid \varrho \\
\text{Object Linearities} & o & ::= & \texttt{obj} \mid ¡\texttt{obj} \\
\text{Regions} & \varrho & ::= & \rho \mid r \\
\text{Rows} & R & ::= & \cdot \mid R, m{:}\tau
\end{array}
$$

Figure 8: Syntax of EGO

5.1 Additions to the Syntax

The changes to the calculus involved in adding `let!` cause some changes to the syntax. The first such change is the addition of 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.

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 nonlinear and linear methods, respectively, which are parameterized over some number of regions.

Methods types are changed to reflect the fact that they are now polymorphic, which prefixes method types with a series of the form $\forall\rho_1. \cdots \forall\rho_n$. We also add annotations to method types indicating the regions the methods they type mention, which give us method types of the form $\forall\rho_1. \cdots \forall\rho_n.\tau \xrightarrow{P} \tau'$ and $\forall\rho_1. \cdots \forall\rho_n.\tau \overset{P}{\multimap} \tau'$. We introduce unannotated method types as a seperate type, which gives us types of the form $\forall\rho_1. \cdots \forall\rho_n.\tau \to \tau'$ and $\forall\rho_1. \cdots \forall\rho_n.\tau \to \tau'$.

We also add region instantiation to method invocations. $e.m[L_1, \cdots, L_n]$ 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 solves allows us to give types to linear regions in settings where there linearities may change, as discussed below.

The final syntax of EGO appears in Figure 8.

5.2 Dynamic Semantics

The addition of new expressions to the language and the alteration of existing expressions necessitates the addition to and alteration of the operational semantics of the calculus.

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_r are substituted into e_2 for ρ and x_1. Note that the linearity annotation on the location is changed to reflect that the location is borrowed. Next, e_2 is evaluated to a value. 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 after these substitutions.

The other change to the dynamic semantics is that method invocation now instantites any regions over which the invoked method is abstracted. This is reflected in the new C-INV, E-NLININV and E-LINLINV rules. Methods now can be prefixed by a series of abstractions of the form $\Lambda\rho_1.\cdots\Lambda\rho_n.$ which abstract these region variables. An invocation of the form $e.m[L_1,\cdots,L_n]$ looks up this method in the base object in the case of a linear object or recursively up the object hierarchy in the case of nonlinear objects. Upon finding the method, each linearity in the series $L_1,\cdot,L_n$ is substituted for the appropriate region variable in e, as well as a reference to the receiver being substitued for the method's bound variable.

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. They may be aliased freely but no change may be made to their interface. This is in line with the motivating intuition that borrowed objects model linear objects that have been made temporarily nonlinear.

The typing judgement is now

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

.

Here, Σ, A, e, τ and l remain the same as before. However, we add two new contexts. The first, R, is a list of regions and region variable which are currently in scope. The second, S, is a partial map from regions to locations. This indicates what locations are borrowed at what regions in the whole expression currently being typechecked, and is used for checking the well-typedness of the store, as discussed below. All our typing rules are updated to use the new typing judgement. Most simply pass P and S up the derivation. The exceptions to this are discussed below.

We add rules for the typing of `let!`, T-LET$_1$ and T-LET$_2$. For some expression, `let!` $(\varrho)\, x_1 = e_1\, x_2 = e_2$ `in` e_3 `end`, these rules type `let!` by first finding the type of e_1. The type of e_1 is required to be object type of the form $L\mathbf{t}.O \leftarrow R$.

The variable x_1 is now bound with the type of ϱ`obj`$\mathbf{O}.R \leftarrow$ in e_2, and 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 freely aliased but no changes can be made to its interface. If we are typechecking a `let!` that is currently being evaluated and so locations annotated with this borrowing's region, ℓ_ϱ, have already been substituted into e_2, the presence of the region in the region context will allow this location to be typechecked. Under these contexts, we check the type for e_2. We check that the type of e_2 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, $L\mathbf{t}.O \leftarrow R$, 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 to make sure $r = \ell$ 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. These annotations indicate the regions that a method body uses. This is because a method's type does not always contain the types of every expression used in the method body. This allows us to tell during typechecking the regions invoking a method would use. We can therefore determine what methods it is safe to invoke.

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

These two changes are reflected in new method typing rules, T-METH and T-LMETH. For a method $\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma(x{:}\tau).e$ or $\Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma(x{:}\tau).e$ the appropriate rule binds the method's bound variable with the type the method is annotated with and check the method under some region context. This region context is checked to be less than the region context the method is checked under with the bound type variables

$$\frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma, \cdots \rangle}{\texttt{mbody}(\mu, \ell_L, m) = \sigma}\ mbody_1$$

$$\frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots \rangle \quad m = \sigma \notin \langle m_1 = \sigma_1, \cdots \rangle \quad \texttt{mbody}(\mu, loc, m) = \sigma}{\texttt{mbody}(\mu, \ell_L, m) = \sigma'}\ mbody2_1$$

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

$$\frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle \\ \forall i. m \neq m_i \\ \mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n, m = \sigma \rangle]\mu\end{array}}{\mu, \ell_L \leftarrow\!\!+ m = \sigma \longrightarrow \mu', \ell_L}\ E - Add$$

$$\frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma, \cdots, m_n = \sigma_n \rangle \\ \mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma', \cdots, m_n = \sigma_n \rangle]\mu\end{array}}{\mu, \ell_L \leftarrow\!\!+ m = \sigma' \longrightarrow \mu', \ell_L}\ E - Upd$$

$$\frac{\ell\ \texttt{fresh} \quad \mu' = [\ell \mapsto \texttt{null} \leftarrow \langle\rangle]\mu}{\mu, \langle\rangle \longrightarrow \mu', \ell_o}\ New$$

$$\frac{\mu, e_1 \longrightarrow \mu', e_1'}{\mu, \texttt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu', \texttt{let!}\ (\rho)\, x_1 = e_1'\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}}\ C - Let_1$$

$$\frac{r\ \texttt{fresh}}{\mu, \texttt{let!}\ (\rho)\, x_1 = \ell_L\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu, \texttt{let!}\ (r)\, x_1 = v_1\, x_2 = [r, \ell_r/\rho, x_1]e_2\ \texttt{in}\ e_3\ \texttt{end}}\ E - Let_1$$

$$\frac{\mu, e_2 \longrightarrow \mu', e_2'}{\mu, \texttt{let!}\ (r)\, x_1 = v_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu', \texttt{let!}\ (r)\, x_1 = v_1\, x_2 = e_2'\ \texttt{in}\ e_3\ \texttt{end}}\ C - Let_2$$

$$\frac{}{\mu, \texttt{let!}\ (r)\, x_1 = v_1\, x_2 = v_2\ \texttt{in}\ e_3\ \texttt{end} \longrightarrow \mu, [v_1, v_2/x_1, x_2]e_3}\ E - Let_2$$

$$\frac{\mu(\ell) = loc \leftarrow \langle \cdots \rangle \quad \mu' = [\ell \mapsto loc' \leftarrow \langle \cdots \rangle]\mu}{\mu, loc' \leftarrow \ell_L \longrightarrow \mu', \ell_L}\ E - Deleg$$

$$\frac{\mu, e_2 \longrightarrow \mu', e_2'}{\mu, e_1 \leftarrow e_2 \longrightarrow \mu', e_1' \leftarrow e_2}\ C - Deleg_1$$

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, \ell_L \leftarrow e \longrightarrow \mu', \ell_L \leftarrow e'}\ C - Deleg_1$$

Figure 9: Dynamic Semantics of EGO part 1

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, e.m[L_1, \cdots, L_n] \longrightarrow \mu', e'.m[L_1, \cdots, L_n]}\ C - Inv$$

$$\frac{\texttt{mbody}(\mu, \ell_L, m) = \Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma x{:}\tau.e}{\mu, \ell_L.m[L_1, \cdots, L_n] \longrightarrow \mu, [\ell_L, L_1, \cdots, L_n/x, \rho_1, \cdots, \rho_n]e}\ E - NLinInv$$

$$\frac{\mu(\ell) = \langle \cdots, m = \Lambda\rho_1.\cdots\Lambda\rho_n.\text{¡}\varsigma x{:}\tau.e, \cdots\rangle \quad \mu' = [\ell_\text{ı} \rightarrow \langle\cdots\rangle]\mu}{\mu, \ell_L.m[L_1, \cdots, L_n] \longrightarrow \mu', [\ell_L, L_1, \cdots, L_n/x, \rho_1, \cdots, \rho_n]e}\ E - LinInv$$

Figure 10: Dynamic Semantics of EGO part 2

appended. This shows that the method can be checked with region variables in scope. 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. This type then shows the region variables which are abstract in the type and the regions which are used by the method's body.

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. This means that we would like the region lists on method types in certain types that appear in our programs to contain region variables not in scope. However, as these are not in scope, we cannot write them as part of the type. But because these methods will never be called, we do not actually need to know these regions, just that the methods cannot be called.

To do this, we introduce two types which are the types of linear methods and nonlinear methods that cannot be called. These are written as the types of methods without annotations. They therefore have the form $\forall\rho_1.\cdots\forall\rho_n.\tau \rightarrow \tau'$ or $\forall\rho_1.\cdots\forall\rho_n.\tau \multimap \tau'$.

These types are supertypes of the equivalent annotated method types. We have a series of standard subtyping rules in Figure 13 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.

With the above changes, invocation has new rules, T-LININV and T-NLinInv, to type an expression of the form $e.m[L_1, \cdots, L_2]$. As before, these type the receiver object and unfold its type before looking up method types either in the base object's row of method types or recursively, as appropriate. Now, however, the unfolded method type will possibily be polymorphic. In this case, we substitute the types with which the invocation is instantiated, $L_1, \cdots, L_2$, for the abstracted region variables in the method type annotation before comparing it with the actual receiver type. We no longer check that the types match exactly, but instead that the receiver is a subtype of the expected type, as we can use a subtype where we expect a supertype. 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. This guarantees that any region which is used by the method is in scope even after instantiation. Finally, we give the invocation expression the type the method types gives as the return type of the method after appropriate substitutions of linearities for 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. However, this means that we need to prevent these methods from ever being called. We therefore do not allow locations with methods whose types contain region annotations not in scope to appear in the currently executing expression, as in T-NLINLOC and LINLOC. Instead, any such method types are replaced with bare arrows. We do this by looking the types of objects up in the store typing and typing the object with a supertype of this type such that no arrows in it are annotated with regions not in scope. Any method types on the type given the location by the store type that have annotations with types out of scope are thus replaced by bare arrows.

Since the linearity of a location can be different in different places in an expression, typing borrowed

locations is slightly more involved, 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. In this rule, 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 because it is borrowed.

The final alteration made to the calculus is in typing the heap. We still check to make sure every object on the heap has the type the store typing gives it, but typing each individual 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 which is the regions contained in 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. This lets methods in the heap mention any region, even if the region is not in scope anywhere in the current program.

5.4 Type Safety Proof

We have a proof of type safety for the full version of EGO presented here. The proof is similar to the proof we sketched for the earlier EGO fragment. As is standard, type safety consists of two theorems, progress and preservation.

The statement of progress for the full EGO language is only slightly different from that presented earlier. It states states that a program that consists of the pair of a well typed store and a well typed expression can always be reduced to a new program if the expression is not already a value. The only difference from the earlier theorem is the typing judgement we use to type stores and expressions. We add a region context, P, to the expression typing judgement, and we add a map, S, of regions to the locations which are borrowed at them to the expression typing judgement. We use the same S in both judgements, as we need to know which regions are borrowed in the current expression to type the store. This gives us the following statement of progress.

Progress: If $\Sigma; A; P; S \vdash e{:}\tau \implies l_e$ and $\Sigma; \cdot; P; S \vdash \mu{:}\Sigma \implies l_s$ 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 judgements. For each case, we show that if the expression is correctly typed, it is of a form which is either a value, or some subexpression of which can reduce, or which itself reduces. To do this, we need a canonical forms lemma similar to the previous one.

Canonical Forms:

1. If a value has the type $\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P} \tau_2$, it has the form $\Lambda\rho_1.\cdots\Lambda\rho_n.\sigma x{:}\tau.e$.
2. If a value has the type $\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P}\!\!\circ\ \tau_2$, it has the form $\Lambda\rho_1.\cdots\Lambda\rho_n.¡\sigma x{:}\tau.e$.
3. If a value has the type $L\mathbf{t}.O \leftarrow R \leftarrow$, it has the form ℓ_L.
4. If a value has the type $\langle\rangle$, it has the form `null`.

This is once again proven by case analysis on the typing rules.

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 mainain four invariants.

1. The expression has some type τ or a subtype of τ. Unlike the earlier EGO fragment, we have subtyping. This means that an expression may evaluate to a new expression whose type is a more specific than the type of the original expression.
2. The heap is well typed.

$$\frac{}{\Sigma; x{:}\tau; P; S \vdash x{:}\tau \Longrightarrow \{\}}\ T-Var$$

$$\frac{\Sigma; A, x{:}\tau; P'; S \vdash e{:}\tau' \Longrightarrow l \quad x \notin \mathtt{Dom}(A) \quad A\ \mathtt{nonlinear} \quad P' \subseteq P, \rho_1, \cdots, \rho_n}{\Sigma; A; P; S \vdash \Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma x{:}\tau.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau \xrightarrow{P'} \tau' \Longrightarrow l}\ T-NLinMeth$$

$$\frac{\Sigma; A, x{:}\tau; P'; S \vdash e{:}\tau' \Longrightarrow l \quad x \notin \mathtt{Dom}(A) \quad P' \subseteq P, \rho_1, \cdots, \rho_n}{\Sigma; A; P; S \vdash \Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma x{:}\tau.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau \overset{P'}{\multimap} \tau' \Longrightarrow l}\ T-LinMeth$$

$$\frac{\Sigma(\ell) = \mathtt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow \quad \mathtt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow \leq \tau \quad \mathtt{freeregions}(\tau) \subseteq P}{\Sigma; A; P; S \vdash \ell_o{:}\tau \Longrightarrow \{\}}\ T-NLinLoc$$

$$\frac{\Sigma(\ell) = ¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow \quad ¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow \leq \tau \quad \mathtt{freeregions}(\tau) \subseteq P}{\Sigma; A; P; S \vdash \ell_o{:}\tau \Longrightarrow \{\ell\}}\ T-LinLoc$$

$$\frac{\Sigma; A; P; S \vdash \ell_o{:}o\mathbf{t}.O \leftarrow R \leftarrow \Longrightarrow l \quad o\mathbf{t}.O \leftarrow R \leftarrow \leq o\mathbf{t}.O' \leftarrow R' \leftarrow \quad \mathtt{freeregions}(\varrho\mathbf{t}.O' \leftarrow R' \leftarrow) \subseteq P}{\Sigma; A; P; S \vdash \ell_\varrho{:}\varrho\mathbf{t}.O' \leftarrow R' \leftarrow \Longrightarrow \{\}}\ T-BorLoc$$

$$\frac{}{\Sigma; A; P; S \vdash \mathtt{null}{:}\langle\rangle \Longrightarrow \{\}}\ T-Null$$

$$\frac{\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l}{\Sigma; A, x{:}\tau'; P; S \vdash e{:}\tau \Longrightarrow l}\ T-Kill$$

$$\frac{\Sigma; A, x{:}\tau', x{:}\tau'; P; S \vdash e{:}\tau \Longrightarrow l \quad \tau'\ \mathtt{nonlinear}}{\Sigma; A, x{:}\tau'; P; S \vdash e{:}\tau \Longrightarrow l}\ T-Copy$$

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

$$\frac{\begin{array}{l}\Sigma; A; P; S \vdash \sigma{:}\tau \Longrightarrow l \quad \Sigma; A; P; S \vdash e{:}¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow \Longrightarrow l' \\ \mathtt{lmtype}(¡\mathtt{obj}\ \mathbf{t}.O' \leftarrow R' \leftarrow, m) = \mathtt{DNE} \quad ¡\mathtt{obj}\ \mathbf{t}.O' \leftarrow R' \leftarrow = ¡\mathtt{obj}\ \mathbf{t}.[\mathbf{t}O \leftarrow R/\mathbf{t}]O \leftarrow R \leftarrow \\ \tau' = ¡\mathtt{obj}\ \mathbf{t}.[\mathbf{t}/\tau'']O' \leftarrow R', m{:}\tau \leftarrow\end{array}}{\Sigma; A, A'; P; S \vdash e \leftarrow\!+ m = \sigma{:}\tau' \Longrightarrow l, l'}\ T-Add$$

$$\frac{\begin{array}{c}\Sigma; A; P; S \vdash e{:}L\mathbf{t}.O \leftarrow R \leftarrow \Longrightarrow l \\ \mathtt{mtype}(L\mathbf{t}.[\mathbf{t}O \leftarrow R/\mathbf{t}]O \leftarrow R \leftarrow, m) = \forall\rho_1.\cdots, \forall\rho_n.L'\mathtt{obj}\mathbf{t}'.O' \leftarrow R' \leftarrow \overset{P'}{\rightarrow} \tau \\ L\mathbf{t}.O \leftarrow R \leftarrow \leq [L_1, \cdots, L_n/\rho_1, \cdots, \rho_n]L'\mathtt{obj}\mathbf{t}'.O' \leftarrow R' \leftarrow \\ [L_1, \cdots, L_n]P' \subseteq P \cup \{\mathtt{obj}, ¡\mathtt{obj}\}\end{array}}{\Sigma; A; P; S \vdash e.m[L_1, \cdots, L_n]{:}[L_1, \cdots, L_n/\rho_1, \cdots, \rho_n]\tau \Longrightarrow l}\ T-NLinInv$$

$$\frac{\begin{array}{c}\Sigma; A; P; S \vdash e{:}¡\mathtt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow \Longrightarrow l \\ \mathtt{lmtype}(¡\mathtt{obj}\ \mathbf{t}.[\mathbf{t}O \leftarrow R/\mathbf{t}]O \leftarrow R \leftarrow, m) = \forall\rho_1.\cdots\forall_n\rho.L'\mathtt{obj}\mathbf{t}'.O' \leftarrow R' \leftarrow \overset{P'}{\multimap} \tau \\ [¡]\mathtt{obj}\ \mathbf{t}.O \leftarrow [/m{:}\tau]R \leftarrow \leq [L_1, \cdots, L_n/\rho_1, \cdots, \rho_n]L'\mathtt{obj}\mathbf{t}'.O' \leftarrow R' \leftarrow \\ [L_1, \cdots, L_n]P' \subseteq P \cup \{\mathtt{obj}, ¡\mathtt{obj}\}\end{array}}{\Sigma; A; P; S \vdash e.m[L_1, \cdots, L_n]{:}[L_1, \cdots, L_n/\rho_1, \cdots, \rho_n]\tau \Longrightarrow l}\ T-LinInv$$

Figure 11: Static Semantics of EGO part 1

$$\frac{\Sigma; A; P; S \vdash e{:}¡\texttt{obj}\ \mathbf{t}.R \leftarrow\!\!\Longrightarrow l}{\Sigma; A; P; S \vdash e{:}\texttt{obj}\ \mathbf{t}.R \leftarrow\!\!\Longrightarrow l}\ T-ChLin$$

$$\frac{}{\Sigma; A; P; S \vdash \langle\rangle{:}¡\texttt{obj}\ \mathbf{t}.\langle\rangle \leftarrow \cdot \leftarrow\!\!\Longrightarrow \{\}}\ T-New$$

$$\frac{\begin{array}{c}\Sigma; A; P; S \vdash e_2{:}¡\texttt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow\!\!\Longrightarrow l \quad \Sigma; A'; P; S \vdash e_1{:}O'' \Longrightarrow l' \\ ¡\texttt{obj}\ \mathbf{t}.O' \leftarrow R' \leftarrow\!\!= ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}O \leftarrow R/\mathbf{t}]O \leftarrow R \leftarrow \quad \tau = ¡\texttt{obj}\ \mathbf{t}.[\mathbf{t}/\tau]O'' \leftarrow R' \leftarrow\end{array}}{\Sigma; A, A'; P; S \vdash e_1 \leftarrow e_2{:}\tau \Longrightarrow l, l'}\ T-Del$$

$$\frac{\begin{array}{c}\Sigma; A_1; P; S \vdash e_1{:}L\mathbf{t}.O \leftarrow R \leftarrow\!\!\Longrightarrow l_1 \\ \Sigma; A_2, x_1{:}\rho\mathbf{t}.O \leftarrow R \leftarrow; P, \rho; S \vdash e_2{:}\tau_2 \Longrightarrow l_2 \\ \Sigma; A_3, x_1{:}L\rho.t \leftarrow O \leftarrow R, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3 \\ \rho \notin \texttt{regions}(\tau_2) \quad \rho \notin P \quad x_1 \notin \texttt{Dom}(A) \quad x_2 \notin \texttt{Dom}(A) \quad x_1 \neq x_2\end{array}}{\Sigma; A_1, A_2, A_3; P; S \vdash \texttt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ x_3\ \texttt{end}{:}\tau_3 \Longrightarrow l_1, l_2, l_3}\ T-Let!_1$$

$$\frac{\begin{array}{c}\Sigma; A_1; P; S \vdash e_1{:}L\mathbf{t}.O \leftarrow R \leftarrow\!\!\Longrightarrow l_1 \\ \Sigma; A_2, x_1{:}r\mathbf{t}.O \leftarrow R \leftarrow; P, r; S \vdash e_2{:}\tau_2 \Longrightarrow l_2 \\ \Sigma; A_3, x_1{:}L\mathbf{r}.t \leftarrow O \leftarrow R, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3 \\ r \notin \texttt{regions}(\tau_2) \quad r \notin P \quad x_1 \notin \texttt{Dom}(A) \quad x_2 \notin \texttt{Dom}(A) \quad x_1 \neq x_2 \quad r = \ell \in S\end{array}}{\Sigma; A_1, A_2, A_3; P; S \vdash \texttt{let!}\ (r)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ x_3\ \texttt{end}{:}\tau_3 \Longrightarrow l_1, l_2, l_3}\ T-Let!_2$$

Figure 12: Static Semantics of EGO part 2

3. All linear locations are used at most once in the expression, store and the list of locations aliased in the whole current expression. This proves that linear locations can be used only once in the expression and heap, or not at all if currently borrowed.

4. All regions in the expression appear in the current region context. This proves that no aliased locations can escape the expression in which they are borrowed, as the region at which they are borrowed appears on the location in the expression.

This gives us the following theorem.

Preservation If $\Sigma; \vdash S\ \texttt{ok} \Longrightarrow \mu l_s$, $\Sigma; \cdot; P; S \vdash e{:}\tau \Longrightarrow l_e$, all regions in e are in P and there are are no duplicates in $l_e, l_s, \texttt{Range}(S)$, then for some $\Sigma' \geq_\ell \Sigma$, $\Sigma'; \vdash S'\ \texttt{ok} \Longrightarrow \mu' l'_s$, $\Sigma'; \cdot; P; S \vdash e'{:}\tau' \Longrightarrow l'_{e'}$, and all regions in e' are in P, $\tau' \leq \tau$ and that there are no duplicates in $l'_s, l'_e, \texttt{Range}(S)$.

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

The proof is similar to the one sketeched above for progress on simplified EGO . It is by induction on the derivation of $\mu, e \longrightarrow \mu', e'$. For each way of reducing the program on the left to the program on the right we show that the invariants are maintained on the right if they were true on the left.

To prove this, we need two substitution lemmas. The first is similar to the one above which showed that substitution is type preserving. Now, however, we need to show that the type produced is a subtype of the expression substituted into if the substituted expression is a subtype of that expected. This gives us following the lemma.

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

$$\frac{}{\tau \leq \tau}\ T-SubRefl$$

$$\frac{O_1 \leq O_2 \quad R_1 \leq R_2}{L\mathbf{t_1}.O_1 \leftarrow R_1 \leftarrow\!\leq L\mathbf{t_2}.O_2 \leftarrow R_2 \leftarrow}\ T-SubLoc$$

$$\frac{}{\cdot \leq \cdot}\ T-SubRow_1$$

$$\frac{\tau_1 \leq \tau_2 \quad R_1 \leq R_2}{R_1, m{:}\tau_1 \leq R_2, m{:}\tau_2}\ T-SubRow_2$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \overset{P}{\rightarrow} \tau_2 \leq \tau_1' \overset{P}{\rightarrow} \tau_2'}\ T-SubNLinMeth$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \overset{P}{\multimap} \tau_2 \leq \tau_1' \overset{P}{\multimap} \tau_2'}\ T-SubLinMeth$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'}\ T-SubUNLinMeth$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \multimap \tau_2 \leq \tau_1' \multimap \tau_2'}\ T-SubULinMeth$$

$$\frac{}{\tau \overset{P}{\rightarrow} \tau' \leq \tau \rightarrow \tau'}\ T-SubArrow$$

$$\frac{}{\tau \overset{P}{\multimap} \tau' \leq \tau \multimap \tau'}\ T-SubLolly$$

$$\frac{\tau \leq \tau'}{\forall \rho.\tau \leq \forall \rho.\tau'}\ T-SubRegAbs$$

Figure 13: Subtyping Rules

$$\frac{\forall \ell \in \mathsf{Dom}(\mu).\Sigma; \cdot; \mathsf{Dom}(S); S \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_\ell \quad \mathsf{Dom}(\mu) = \mathsf{Dom}(\Sigma)}{\Sigma; \vdash S\ \ \mathsf{ok} \Longrightarrow \mu\text{concat}\ l_\ell}\ Store$$

$$\frac{\begin{array}{c}\forall i \in 1..n | P \subseteq \mathsf{Dom}(S).\Sigma; A; P; S \vdash \sigma_i{:}\tau_i \overset{P}{\rightarrow} / \overset{P}{\multimap} \tau_i' \Longrightarrow l_i \\ [\mathbf{t}.O \leftarrow R/\mathbf{t}]R = m_1{:}\tau_1, \cdots, m_n{:}\tau_n \\ \Sigma; A; P; S \vdash loc{:}O \Longrightarrow l_{loc}\end{array}}{\Sigma; A; P; S \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle{:}L\mathtt{obj}\mathbf{t}.O \leftarrow R \leftarrow\!\!\Longrightarrow l_{loc}, l_1, \cdots, l_n}\ T-ODescr$$

Figure 14: Store and Object Typing

We also need a similar lemma for region substitution, as 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 a nexpression substitutes the region in for the region variable in the expressio's type. The lemma follows.

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

The proof of both of these lemmas is by induction on the typing rules.

6 Related Work

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

An earlier version of EGO was presented in [4, 5]. This version differed from ours in its lack of a mechanism for relaxing linearity and in its inclusion of first class functions in addition to to first class objects.

Abadi and Cardelli, in [1], discuss foundational object calculi in great detail, including in their discussion both typed and untyped and both functional and imperative calculi. Several calculi exploring the relationship between method addition and subtyping in calculi based on Abadi and Cardelli's are presented in [14, 16, 17]. Fisher, Honsell and Mitchell present an object calculus with delegation and subtyping in [11, 12].

Ungar and Smith describe SELF in [20]. SELF is a prototype based language with method addition and update and delegation change similar to EGO 's.

In [10], Drossopoulou et al. present Fickle. Fickle is a typed, class based language which allows objects to change their class at runtime. Our system is more flexible than that of Fickle in that we allow arbitrary method adds and delegation changes while Fickle only allows changes to a few previously defined classes.

Wadler introduced linear type systems in a functional setting in [21]. This work was based on Girard's linear logic as described in [13]. Wadler's work contains a `let!` similar to ours; however, it lacks a region system and so preserves linearity by strongly restricting the types of values which expressions that contain borrowed values may return.

Several papers, especially [3, 7, 9], describe research into ways to model objects in linear logic. In general these associate fields or methods in objects with linear hypothesis. Invocation of a method consumes it in a manner similar to our linear methods.

A fair amount of research has been done in tracking aliasing in higher level object oriented languages. In [15], Minsky discusses how to add unique, unaliased pointers to Eiffel and C++. Aldrich et al. present AliasJava in [2], which enforces reference aliasing policies in Java. In [6], Boyland discusses unique pointers in higher level languages, especially Java.

Typestates were introduced in [18]. DeLine and Fähndrich discuss typestates for objects, especially in the presence of subtyping, in [8].

Regions were presented by Tofte and Talpin in[19]. Their regions differ from ours in that they were used to manage memory rather than enforce scoping of aliases.

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, including linear methods and the possibility of temporarily relaxing linearity to create short lived aliases. We have demonstrated the expressiveness of our calculus showing how the lambda calculus can be imbedded in it.

We have also shown how linearity allows us to manipulate objects so as to enforce potocols in a well typed way. We can add methods to objects, remove linear methods by invoking them and change delegation at run time and still staticly check that out programs are safe. We have shown that such abilities can be used to guarantee that methods are called on objects in a correct manner.

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] 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.

[4] A. Bejleri. A type checked prototype-based model with linearity. Draft senior thesis, published as a Carnegie Mellon Technical Report CMU-ISRI-04-142, December 2004.

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

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

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

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

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

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

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

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

[14] L. Liquori. An extended theory of primitive objects: First order system. In *European Conference on Object-Oriented Programming*, pages 146–??, 1997.

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

[16] D. R'emy. From classes to objects via subtyping, 1998.

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

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

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

[20] 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.

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