Linearity For Objects

Matthew Kehrt[*] **Andi Bejleri**[†] **Jonathan Aldrich**[‡]

July 2006
CMU-ISRI-06-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[†]Computer Science Department, Università di Pisa, Lungarno Pacinotti, 43, 56126 Pisa, Italy
[‡]Institute for Software Research, International, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

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.

This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A and NSF grants CCR-0204047 and CCF-0546550.

Keywords: linear types, object calculi, prototyped-based languages, static type checker, type safety

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. For example, Wadler, in [25], 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. Cyclone [15, 16] has unique pointers which are tracked in this way.

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. For instance, [7] 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, 18]. 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 [5, 6] 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 Abadi and Cardelli [1]. This new calculus removes redundancy in the original version by eliminating first class functions, leaving only objects. We then extend this version of EGOwith a mechanism for temporarily relaxing linearity based on Wadler's `let!`[25].

EGO also explores enforcing protocols through several mechanisms based on those found in Self [24]. Self introduced *dynamic inheritance*. Objects in Self have delegee objects, where methods are looked up if method lookup failed on the original 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 in a similar way.

EGO also includes dynamic inheritance and method addition. Unlike Self, however, EGO 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 [6]. 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 focus. 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 modify 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 objects whose type are not being changed to be temporarily aliased, or *borrowed*. We use regions to track in which expressions in a program a given object is borrowed and guarantee no aliases to a borrowed object escape these expressions.

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 the formalization of this language. We conclude this section by sketching a proof of type safety for the the language.
- Section 4 describes how to add borrowing to the language to relax linearity. We again discuss the intuition behind this and several examples.
- In section 5 we show how the previous formalization needs to be changed to reflect the addition of the borrowing 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 [6]. This simplified EGO lacks a construct for relaxing linearity. We briefly discuss the intuition behind the language. We then discuss several examples to demonstrate 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 of Fisher et al. [12, 13]. They follow.

- $\langle\rangle$ creates a new object on the heap and returns a reference to it.
- $e \longleftarrow m = \sigma$ 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 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.
- $!e$ changes the linearity of an object, as discussed below. This only affects the type of the object; it has no dynamic effect.

For simplicity we often write $\langle\rangle \longleftarrow m_1 = \sigma_1 \cdots \longleftarrow m_n = \sigma_n$ 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 the value the method evaluates to; since method bodies will be composed of these five primitives, they will return a location as well. The fifth primitive returns the object to which it is applied.

EGO allows both methods and objects to be either *linear* or *nonlinear*. A linear object is one to which only one reference is allowed; conversely, nonlinear objects can have multiple references. To make static typing possible, only linear objects are allowed to have their interfaces changed by method addition or delegation change. Since such changes modify the type of an object, we need to change the type of any reference to this object used in an expression. However, finding these references statically is not always possible. Therefore, we only allow these changes to linear objects to guarantee we can find all references to the object.

In EGO, all new objects are linear. When all of the necessary interface changes have been made, the type of such an object can be changed irreversibly to be nonlinear, and it can then be freely aliased. This is similar to the "seal" operation of Fisher and Mitchell [13], but our in our calculus objects can change to be aliasable, while Fisher and Mitchell allow objects to become subtypable.

A linear method is one which can be called only once, while nonlinear methods can be called multiple times. Calling a linear method consumes it, and it is removed from the object that contains it. This also counts as an interface change and so is only allowed on linear objects. We do not allow nonlinear methods to contain references to linear objects; multiple invocations of such a method would constitute multiple references to any objects mentioned within its body.

Methods in EGO are based on those of Abadi and Cardelli [1]. 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, 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 a series of delegees. 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, as we describe below.

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.

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

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 delegee of this second object to be the first object,

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

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.

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

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

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

where

$$[\![\tau \to \tau']\!] \stackrel{\text{def}}{=} \texttt{obj t}.\langle\rangle \leftarrow gen{:}(\texttt{objt} \to \text{¡}\texttt{obj t}'.\langle\rangle \leftarrow body : (\text{bodytype}(\tau, \tau') \dot{\to} [\![\tau']\!]))$$

This works by creating a new object and adding a single method, gen. After adding this method to the object, it is made nonlinear so that it can be aliased. As this is the translation of a function, the translation of the type of a function, $[\![\tau \to \tau']\!]$ is the type of this object. 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, as seen below.

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

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

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

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 $body$ is called 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.

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

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

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

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

Once again, we elide types because we can calculate them from the terms.

In a similar manner to lambda abstractions, we can also define linear lambda abstractions that are consumed when applied, as in [25].

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

where

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

and

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

Finally, we translate application as follows.

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

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 this example invokes a linear method on an object, the object is linear, and so on invocation we can add the argument directly to it, rather than calling a gen method to generate 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.

Finally, for completeness, we translate any base types and variables to themselves.

$$[\![x]\!] \stackrel{\text{def}}{=} x$$

$$[\![\tau_{base}]\!] \stackrel{\text{def}}{=} \tau_{base}$$

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 object 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 make other methods available and unavailable by changing delegation on $Socket$, which remains linear throughout the example. 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 fragment, 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 an empty 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 therefore 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 dynamic semantics and the type system. Finally, we sketch 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 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 $\texttt{null}$, 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 $\text{¡}\varsigma(x{:}\tau).e$. Both of these abstract the variable x out of the method body e. The first is a nonlinear method and the second a linear method.

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

```
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

Expressions	e	::=	$x, y \mid \langle\rangle \mid e.m \mid e \leftarrow\!\!+ m = \sigma$
		|	$e_1 \leftarrow e_2 \mid !e \mid v$
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	::=	$\varsigma(x{:}\tau).e \mid \text{¡}\varsigma(x{:}\tau).e$
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	::=	$\texttt{obj} \mid \text{¡obj}$
Rows	R	::=	$\cdot \mid R, m{:}\tau$

Figure 2: Syntax of Simplified EGO

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. We also count locations, loc, which are either `null` or ℓ, 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 [13]. 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 the recursive type $L\mathbf{t}.O \leftarrow R$. $\langle\rangle$ 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 original 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 recursively mention itself; in the object type **t** is a recursive type variable bound to the whole type. This may be referred to in the rest of the type by the form $L\mathbf{t}$, which annotates the recursive variable with a linearity.

Figure 3: Dynamic Semantics of Simplified EGO

$$\frac{\ell \notin \mathtt{Dom}(\mu) \quad \mu' = [\ell \mapsto \mathtt{null} \leftarrow \langle\rangle]\mu}{\mu, \langle\rangle \longrightarrow \mu', \ell} \text{ E-NEW}$$

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, e \leftarrow\!\!+ m = \sigma \longrightarrow \mu', e' \leftarrow\!\!+ m = \sigma} \text{ C-UPD} \qquad \frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots\rangle \quad \forall i. m \neq m_i \\ \mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma\rangle]\mu\end{array}}{\mu, \ell \leftarrow\!\!+ m = \sigma \longrightarrow \mu', \ell} \text{ E-ADD}$$

$$\frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle \cdots, m = \sigma, \cdots\rangle \\ \mu' = [\ell \mapsto loc \leftarrow \langle \cdots, m = \sigma', \cdots\rangle]\mu\end{array}}{\mu, \ell \leftarrow\!\!+ m = \sigma' \longrightarrow \mu', \ell} \text{ E-UPD} \qquad \frac{\mu, e_1 \longrightarrow \mu', e_1'}{\mu, e_1 \leftarrow e_2 \longrightarrow \mu', e_1' \leftarrow e_2} \text{ C-DEL}_1$$

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, \ell \leftarrow e \longrightarrow \mu', \ell \leftarrow e'} \text{ C-DEL}_2 \qquad \frac{\mu(\ell) = loc \leftarrow \langle \cdots\rangle \quad \mu' = [\ell \mapsto loc' \leftarrow \langle \cdots\rangle]\mu}{\mu, loc' \leftarrow \ell \longrightarrow \mu', \ell} \text{ E-DEL}$$

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, e.m \longrightarrow \mu', e'.m} \text{ C-INV} \qquad \frac{\mathtt{mbody}(\mu, \ell, m) = \varsigma x{:}\tau.e}{\mu, \ell.m \longrightarrow \mu, [\ell/x]e} \text{ E-NLININV}$$

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

$$\frac{}{\mu, !v \longrightarrow \mu, v} \text{ E-CHLIN} \qquad \frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma, \cdots\rangle}{\mathtt{mbody}(\mu, \ell, m) = \sigma} \text{ MBODY}_1$$

$$\frac{\begin{array}{c}\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots\rangle \\ m = \sigma \notin \langle m_1 = \sigma_1, \cdots\rangle \quad \mathtt{mbody}(\mu, loc, m) = \sigma\end{array}}{\mathtt{mbody}(\mu, \ell, m) = \sigma} \text{ MBODY}_2$$

3.2 Dynamic Semantics

This section discusses how the expressions of EGO are evaluated and the effect this 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 $\texttt{null} \leftarrow \langle\rangle$, that is, an object descriptor with no methods and no delegee. $\langle\rangle$ 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 $m = \sigma$ in a similar way.

$e_1 \leftarrow e_2$ 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_2 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 find the called method 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 that object descriptor's delegee. Then, in either case, ℓ, the method 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, as invocation of a linear method consumes the method.

$!e$ is defined by E-CHLIN. C-CHLIN reduces e to a value. $!e$ has no dynamic effect. It only effects the typing of objects.

3.3 Type System

EGO's type system prevents a 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 give a type to 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. 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, 13] 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 receive 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 uses gives recursive types of the form $L\mathbf{t}.O \leftarrow R$ to objects. Here, $\mathbf{t}$ is bound recursively to the whole type. Any time we change the type of the object, this whole type will change so we must unfold the type by substituting the type in for the recursive variable in itself. We then make the necessary changes to the type and refold this type by abstracting out

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

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

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

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

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

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

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

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

Figure 4: Static Semantics of Simplified EGO

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

$$\frac{\texttt{lmtype}(L\mathbf{t}.O \leftarrow R, m) = \tau}{\texttt{mtype}(L\mathbf{t}.O \leftarrow R, m) = \tau}\ \text{T-MethT}_1$$

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

Figure 5: Method Type Lookup in Simplified and Full EGO

occurrences of the new type the object will have. This ensures that the recursive type variable will always refer to the current type of the object. 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. 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 is 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.

`null` is typed using the rule T-NULL, which gives it the type $\langle\rangle$.

We can turn a linear location as a nonlinear location with $!e$, as described by T-CHLIN. 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. This is safe as it can only go one way: we cannot turn a linear location into a nonlinear one.

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 $L\mathbf{t}.O \leftarrow R$, 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 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 of the unfolded object type, which describes the types of the methods contained in the original object. These lookup rules are defined in Figure 5. 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 delegee type section of the unfold object type if the method type is not found in the row. For nonlinear methods, we then check that the type the method expects is the type the of the receiver. The invocation is then given the return type of the method. For linear 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 type the method expects with the receiver type as we do with nonlinear types, we check equality of the type the method expects type with that of the receiver type refolded with the invoked method removed, as this is what will be substituted into the method body. We once again give the whole expression the return type of the method.

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

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, nonlinear methods are not allowed to mention linear 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 changed is unfolded. The expression is given this type with the type of the delegee changed and the object type 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 $\texttt{¡obj}\ \mathbf{t}.\langle\rangle \leftarrow \cdot$.

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. This is a difference between our system and Wadler's: we allow linear objects to leave scope. 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 [25], modeled on the same technique from Girard's linear logic [14]. 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 a nonlinear variable's binding in the context.

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

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

Figure 6: Store and Object Typing for Simplified EGO

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 delegees have the correct type with. 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 instruction. 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. Rather we have a proof of the safety of the entire system. However, we sketch a hypothetical proof of safety for this fragment 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.

Theorem 1 (Progress) *If $\Sigma;\cdot \vdash e{:}\tau \Longrightarrow l$ and $\Sigma \vdash \mu \ \ \mathsf{ok} \Longrightarrow l$ 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 either is a value or can be reduced. To do this, we need a canonical forms lemma.

Lemma 1 (Canonical Forms) *1. If a value has the type $\tau_1 \to \tau_2$, 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 $L\mathbf{t}.O \leftarrow R$, it has the form ℓ.

4. If a value has the type $\langle\rangle$, it has the form $\mathtt{null}$.

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.

$$\frac{}{\Sigma \geq_l \Sigma}\ \text{S-REFL}$$

$$\frac{\text{Dom}(\Sigma') = \text{Dom}(\Sigma) \cup \{\ell\} \quad \forall \ell' \in \text{Dom}(\Sigma).\Sigma(\ell') = \Sigma'(\ell')}{\Sigma' \geq_l \Sigma}\ \text{S-GROW}$$

$$\frac{\text{Dom}(\Sigma') = \text{Dom}(\Sigma) \quad \forall \ell'_{L'} \in \text{Dom}(\Sigma) - \{\ell\}.\Sigma(\ell') = \Sigma'(\ell') \quad \Sigma(\ell) = \texttt{¡obj t}.O \leftarrow R \leftarrow \quad \Sigma'(\ell) = \texttt{obj t}.O \leftarrow R \leftarrow}{\Sigma' \geq_l \Sigma}\ \text{S-CHLIN}$$

$$\frac{\text{Dom}(\Sigma') = \text{Dom}(\Sigma) \quad \Sigma(\ell) = \texttt{¡obj t}.O \leftarrow R \leftarrow \quad \forall \ell' \in \text{Dom}(\Sigma) - \{\ell\}.\Sigma(\ell') = \Sigma'(\ell')}{\Sigma' \geq_l \Sigma}\ \text{S-LOBJ}$$

Figure 7: Syntax of EGO

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 more easily prove that a changed store is still well typed, as it changes in a limited number of known ways. This is defined in Figure 7.

Theorem 2 (Preservation) *If*

i. $\Sigma; \cdot \vdash e{:}\tau \Longrightarrow l_e$

ii. $\Sigma \vdash \mu \ \ \mathsf{ok} \Longrightarrow l_s$

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. $\Sigma' \vdash \mu' \ \ \mathsf{ok} \Longrightarrow l'_s$*, and*

iii. 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, two important lemmas are needed. Since methods do substitution, we need to show that the substitution of well typed expressions into well typed expressions maintains well typedness.

Lemma 2 (Substitution) *If* $\Sigma; A, x{:}\tau' \vdash e{:}\tau \Longrightarrow l$ *and* $\Sigma; \cdot \vdash e'{:}\tau' \Longrightarrow l'$*, then* $\Sigma; A \vdash [e'/x]e{:}\tau \Longrightarrow l''$ *and* $l'' \subseteq l, l'$*.*

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

We also need to show that if we have a well typed store, if we replace a linear object type in the store typing with a new type and the object at that location in the store with an object of that type, the store remains well typed.

Lemma 3 (Store Change) *If*

i. $\Sigma \vdash \mu \ \ \mathsf{ok} \Longrightarrow l_s$

ii. $\Sigma; A \vdash \ell_L{:}\text{¡}\texttt{obj}\ \texttt{t}.O \leftarrow R \Longrightarrow l_e$

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 restrictions on linear objects are often counterintuitive and sometimes overrestrictive. 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 [25]. 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. 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 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 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 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 of an object with a region not in scope.

Our region system differs from previous systems such as [23]. Such work has been in using regions for memory management. There, the region is an actual block of memory where data resides, allowing information about when memory can be freed can be statically inferred. We do not have blocks of memory; instead our system instead uses similar techniques to track aliases. 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 in that Wadler does not use regions to contain aliases. Instead, he places restrictions on the type of the expression in which a linear value is borrowed to prevent values containing the type of the borrowed value 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, our system also works in an imperative setting where values may be stored on the heap.

The following example demonstrates a very simple use of `let!`. In this example, x is bound to a reference to an object containing only a new method that returns a new object. This object to is borrowed in the second subexpression but not used, as we have not yet introduced the necessary mechanisms to do so. The second expression evaluates to an object containing a single id method that returns the receiver. This new object is bound to y. Finally, outside the scope of the borrowing, both methods are called. The whole expression evaluates to an empty object, as this is what x's new method returns.

```
let! (ρ)
   x = ⟨⟩ ←+ new =
      ς(this:¡obj t.⟨⟩ ← new:¡objt1 →̇ ¡obj t2.⟨⟩ ← ·).⟨⟩
   y = ⟨⟩ ←+ id =
      ς(this′:¡obj t.⟨⟩ ← id:¡objt →̇ ¡objt).this′
in
   y.id; x.new
end
```

Several other modifications to 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 expect receivers that are borrowed under regions not yet 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 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.

```
let! (ρ1)
   x = ⟨⟩ ←+ new = Λρ2.ς(this:ρ2t1.⟨⟩ ← new:(∀ρ3.ρ3t1 →̇ ¡obj t2.(⟨⟩ ← ·))).⟨⟩
   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 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 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; ⟨⟩)
```

$y = x.new[\rho_1]$
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 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 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.

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

let $x = \langle\rangle \leftarrow\!\!+ meth_1 = \varsigma(this{:}receivertype).\langle\rangle$ in
let! (ρ)
$y = \langle\rangle$
$z = x \leftarrow\!\!+ meth_2 = \varsigma(this{:}receivertype).(y; \langle\rangle)$
in
$z.meth_1[]$
end

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

let $x = \langle\rangle \leftarrow\!\!+ meth_1 = \varsigma(this{:}receivertype).\langle\rangle$ in
let! $(\rho)\, y = \langle\rangle$
$z = x \leftarrow\!\!+ meth_2 = \varsigma(this{:}receivertype).(y; \langle\rangle)$
in $z.meth_1$ end

We now have the necessary linguistic mechanisms to model linear lambda abstractions which take nonlinear arguments. As with those 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 mutltiple 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 8.

We can also now implement the socket example described at the begining of this section. In fact, this example is fairly simple; it is given in Figure 9. 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.

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

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

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

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

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

Figure 8: Linear functions with nonlinear arguments

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

```
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 9: A socket using `let!`

Expressions	e	$::=$	$x, y \mid \langle\rangle \mid e.m\boxed{[\varrho_1, \cdots, \varrho_n]} \mid e \leftarrow\!\!+ m = \sigma$
		$\mid$	$\boxed{\texttt{let!}\ (\varrho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}}$
Values	v	$::=$	$loc \mid \sigma$
Locations	loc	$::=$	$\texttt{null} \mid \ell\boxed{L}$
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	$::=$	$\boxed{\Lambda\rho_1.\cdots\Lambda\rho_n.}\varsigma(x{:}\tau).e \mid \boxed{\Lambda\rho_1.\cdots\Lambda\rho_n.}¡\varsigma(x{:}\tau).e$
Types	τ	$::=$	$O \mid \boxed{\forall\rho_1.\cdots\forall\rho_n.}\tau \xrightarrow{\boxed{P}} \tau' \mid \boxed{\forall\rho_1.\cdots\forall\rho_n.}\tau \multimap^{\boxed{P}} \tau' \mid \boxed{\top}$
Linearities	L	$::=$	$\boxed{o} \mid \boxed{\varrho}$
Object Linearities	o	$::=$	$\texttt{obj} \mid ¡\texttt{obj}$
Regions	ϱ	$::=$	$\boxed{\rho} \mid \boxed{r}$

Figure 10: Syntax of EGO

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 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.

Method types are changed to reflect the fact that they 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 seperate type for methods which mention regions no longer in scope.

We also 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.

The syntax of EGOappear in Figure 10, with differences from the previous system highlighted.

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 dynamic semantics of the calculus. The dynamic semantics of EGOare shown in Figure 11, with additions highlighted.

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.

Another 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 $e.m[\varrho_1,\cdots,\varrho_n]$ looks up this method in the original 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 region or region variable in the series $\varrho_1,\cdots,\varrho_n$ 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.

Finally, as locations are now annotated with linearities, $!e$ now 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. This is shown in Figures 12 and 13, with additions highlighted.

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

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

.

Here, Σ, A, e, τ and l remain the same as before. However, we add two new contexts. The first, P, is a list of regions and region variables 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 here.

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 $\varrho\mathbf{t}.O \leftarrow R$ 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 variable bindings of the form $\forall\rho$.

These two changes are reflected in new method typing rules, T-NLINMETH and T-LINMETH. 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 adds a binding of the method's bound variable

$$\frac{\ell \notin \mathtt{Dom}(\mu) \quad \mu' = [\ell \mapsto \mathtt{null} \leftarrow \langle\rangle]\mu}{\mu, \langle\rangle \longrightarrow \mu', \ell\boxed{L}} \text{ E-NEW} \qquad \frac{\mu, e \longrightarrow \mu', e'}{\mu, e \leftarrow\!\!+ m = \sigma \longrightarrow \mu', e' \leftarrow\!\!+ m = \sigma} \text{ C-UPD}$$

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

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

$$\frac{\mu, e_1 \longrightarrow \mu', e_1'}{\mu, e_1 \leftarrow e_2 \longrightarrow \mu', e_1' \leftarrow e_2} \text{ C-DEL}_1 \qquad \frac{\mu, e \longrightarrow \mu', e'}{\mu, \ell\boxed{L} \leftarrow e \longrightarrow \mu', \ell\boxed{L} \leftarrow e'} \text{ C-DEL}_2$$

$$\frac{\mu(\ell\boxed{L}) = loc \leftarrow \langle \cdots\rangle \quad \mu' = [\ell\boxed{L} \mapsto loc' \leftarrow \langle \cdots\rangle]\mu}{\mu, loc' \leftarrow \ell\boxed{L} \longrightarrow \mu', \ell\boxed{L}} \text{ E-DEL} \qquad \frac{\mu, e \longrightarrow \mu', e'}{\mu, e.m\boxed{[\varrho_1, \cdots, \varrho_n]} \longrightarrow \mu', e'.m\boxed{[\varrho_1, \cdots, \varrho_n]}} \text{ C-INV}$$

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

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

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

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

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

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

$$\frac{\mu, e \longrightarrow \mu', e'}{\mu, !e \longrightarrow \mu', !e'} \text{ C-CHLIN} \qquad \frac{}{\mu, !\ell\boxed{¡\mathtt{obj}} \longrightarrow \mu, \ell\boxed{\mathtt{obj}}} \text{ E-CHLIN}$$

$$\frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma, \cdots\rangle}{\mathtt{mbody}(\mu, \ell, m) = \sigma} \text{ MBODY}_1 \qquad \frac{\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots\rangle \quad m = \sigma \notin \langle m_1 = \sigma_1, \cdots\rangle \quad \mathtt{mbody}(\mu, loc, m) = \sigma}{\mathtt{mbody}(\mu, \ell, m) = \sigma} \text{ MBODY}_2$$

Figure 11: Dynamic Semantics of EGO

with the type the method expects its receiver to have to the typing context and checks the method body under some region context. This region context is checked to be a subset of the region context the method is checked under with the bound type variables 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. 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 14 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, invocation has new rules, T-LININV and T-NLININV, to type an expression of the form $e.m[\varrho_1,\cdots,\varrho_n]$. As before, these type the receiver object and unfold its type before looking up method types either in the original object's row of method types or recursively, as appropriate. Now, however, the method type will possibily be polymorphic. In this case, we substitute the regions or region variables with which the invocation is instantiated, $\varrho_1,\cdots,\varrho_2$, 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 after instantiation. Finally, we give the invocation expression the return type of the method after appropriate substitutions of regions and region variables for abstracted region variables.

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 currently executing 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 up 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. 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 additions of regions makes it necessary to make a change to the typing of variables. Now we check to make sure that the regions in a variable's type are all in scope. This ensures that an expression requires the same regions to be in scope to type both before and after substitution.

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.

$$\frac{\Sigma(\ell) = \texttt{obj}\ \mathbf{t}.O \leftarrow R \qquad \boxed{\texttt{obj}\ \mathbf{t}.O \leftarrow R \leq \tau} \qquad \boxed{\texttt{eregions}(\tau) \subseteq P}}{\Sigma; A \boxed{;P;S} \vdash \ell \boxed{\text{obj}} : \boxed{\tau} \Longrightarrow \{\}}\ \text{T-NLinLoc} \qquad \frac{\Sigma(\ell) = \text{¡}\texttt{obj}\ \mathbf{t}.O \leftarrow R \qquad \boxed{\text{¡}\texttt{obj}\ \mathbf{t}.O \leftarrow R \leq \tau} \qquad \boxed{\texttt{eregions}(\tau) \subseteq P}}{\Sigma; A \boxed{;P;S} \vdash \ell \boxed{\text{¡obj}} : \boxed{\tau} \Longrightarrow \{\ell\}}\ \text{T-LinLoc}$$

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

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

$$\frac{\Sigma; A \boxed{;P;S} \vdash e{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l \qquad \tau_u = L\mathbf{t}.[\mathbf{t}O \leftarrow R/\mathbf{t}]O \leftarrow R \qquad \texttt{mtype}(\tau_u, m) = \boxed{\forall \rho_1. \cdots L\mathbf{t}'.O' \leftarrow R' \overset{P'}{\rightarrow} \tau} \qquad \boxed{L\mathbf{t}.O \leftarrow R \leq [\varrho_1, \cdots/\rho_1, \cdots]L\mathbf{t}'.O' \leftarrow R'} \qquad \boxed{[\varrho_1, \cdots/\rho_1, \cdots]P' \subseteq P}}{\Sigma; A \boxed{;P;S} \vdash e.m \boxed{[\varrho_1, \cdots]} : \boxed{[\varrho_1, \cdots/\rho_1, \cdots]}\tau \Longrightarrow l}\ \text{T-NLinInv}$$

$$\frac{\Sigma; A \boxed{;P;S} \vdash e{:}\text{¡}\texttt{obj}\ \mathbf{t}.O \leftarrow R \Longrightarrow l \qquad \tau_u = \text{¡}\texttt{obj}\ \mathbf{t}.[\mathbf{t}O \leftarrow R/\mathbf{t}](O \leftarrow R) \qquad \texttt{lmtype}(\tau_u, m) = \boxed{\forall \rho_1. \cdots \text{¡}\texttt{obj}\ \mathbf{t}'.O' \leftarrow R' \overset{P'}{\multimap} \tau} \qquad \text{¡}\texttt{obj}\ \mathbf{t}.O'' \leftarrow R'' = \tau_u \qquad \tau_f = \text{¡}\texttt{obj}\ \mathbf{t}.[\mathbf{t}/\tau_f](O'' \leftarrow [/m{:}\tau'']R'') \qquad \boxed{\tau_f \leq [\varrho_1, \cdots/\rho_1, \cdots](\text{¡}\texttt{obj}\ \mathbf{t}'.O' \leftarrow R')} \qquad \boxed{[\varrho_1, \cdots/\rho_1, \cdots]P' \subseteq P}}{\Sigma; A \boxed{;P;S} \vdash e.m \boxed{[\varrho_1, \cdots]} : \boxed{[\varrho_1, \cdots/\rho_1, \cdots]}\tau \Longrightarrow l}\ \text{T-LinInv}$$

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

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

Figure 12: Static Semantics of EGO

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

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

$$\frac{\Sigma; A \boxed{;P;S} \vdash e_2{:}\text{¡}\mathtt{obj}\ \mathbf{t}.O \leftarrow R \Longrightarrow l \qquad \Sigma; A' \boxed{;P;S} \vdash e_1{:}O'' \Longrightarrow l' \qquad \text{¡}\mathtt{obj}\ \mathbf{t}.O' \leftarrow R' = \text{¡}\mathtt{obj}\ \mathbf{t}.[\mathbf{t}O \leftarrow R/\mathbf{t}]O \leftarrow R \qquad \tau = \text{¡}\mathtt{obj}\ \mathbf{t}.[\mathbf{t}/\tau]O'' \leftarrow R'}{\Sigma; A, A' \boxed{;P;S} \vdash e_1 \leftarrow e_2{:}\tau \Longrightarrow l, l'}\ \text{T-Del}$$

$$\frac{}{\Sigma; A \boxed{;P;S} \vdash \langle\rangle{:}\text{¡}\mathtt{obj}\ \mathbf{t}.\langle\rangle \leftarrow \cdot \Longrightarrow \{\}}\ \text{T-New} \qquad \frac{\mathtt{tregions}(\tau) \subseteq P}{\Sigma; x{:}\tau \boxed{;P;S} \vdash x{:}\tau \Longrightarrow \{\}}\ \text{T-Var}$$

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

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

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

Figure 13: Static Semantics of EGO

$$\frac{}{\tau \leq \tau}\ \text{S-REFL}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \text{S-TRANS}$$

$$\frac{O_1 \leq O_2 \quad R_1 \leq R_2}{L\mathbf{t_1}.O_1 \leftarrow R_1 \leftarrow\!\leq L\mathbf{t_2}.O_2 \leftarrow R_2 \leftarrow}\ \text{S-LOC}$$

$$\frac{}{\cdot \leq \cdot}\ \text{S-ROW}_1$$

$$\frac{\tau_1 \leq \tau_2 \quad R_1 \leq R_2}{R_1, m{:}\tau_1 \leq R_2, m{:}\tau_2}\ \text{S-ROW}_2$$

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

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

$$\frac{}{\forall\rho_1.\cdots\forall\rho_n.\tau \xrightarrow{P} \tau' \leq \forall\rho_1.\cdots\forall\rho_n.\top}\ \text{S-ARROW}$$

$$\frac{}{\forall\rho_1.\cdots\forall\rho_n.\tau \overset{P}{\multimap} \tau' \leq \forall\rho_1.\cdots\forall\rho_n.\top}\ \text{S-LOLLY}$$

Figure 14: Subtyping Rules

T-STORE

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

T-ODESCR

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

Figure 15: Store and Object Typing

5.4 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 sketch for the earlier EGO fragment. The full proof is included as an appendix; we describe it here. 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 whole expression to type the store. This gives us the following statement of progress.

Theorem 3 (Progress) *If* $\Sigma; \cdot; P; S \vdash e{:}\tau \implies l_e$ *and* $\Sigma; S \vdash \mu \ \texttt{ok} \implies l_s$ *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.

Lemma 4 (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* $\forall\rho_1.\cdots\forall\rho_n.\tau_1 \multimap^{P} \tau_2$*, it has the form* $\Lambda\rho_1.\cdots\Lambda\rho_n.¡\sigma x{:}\tau.e$*.*
3. *If a value has the type* $L\mathbf{t}.O \leftarrow R$*, it has the form* ℓ_L*.*
4. *More specifically, if a value has the type* $¡\texttt{obj}\ \mathbf{t}.O \leftarrow R \leftarrow$*, it has the form* $\ell_{¡\texttt{obj}}$*.*
5. *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.
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 or are bound by region abstraction or a `let!`. 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.

We formalize the idea of all regions free in a given expression by defining a function, `eregions`(e), which recursively examines an expression. This is defined in the appendix. Thus we get the following theorem.

Theorem 4 (Preservation) *If*

i. $\Sigma; S \vdash \mu \ \texttt{ok} \implies l_s$

ii. $\Sigma; \cdot; P; S \vdash e{:}\tau \implies l_e$

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. $\tau' \leq \tau$*, and*

v. *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.

Lemma 5 (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* $\Sigma; A; P; S \vdash [e'/x]e{:}\tau'_2 \Longrightarrow l$*,* $\tau'_2 \leq \tau'_1$ *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 expressio's type. The lemma follows.

Lemma 6 (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.

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 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 7 (Store Change) *If*

i. $\Sigma; S \vdash \mu \ \texttt{ok} \Longrightarrow l_s$

ii. $\Sigma; A; P; S \vdash \ell_L{:}\texttt{¡obj t}.O \leftarrow R \Longrightarrow l_e$

iii. *there are no duplicates in* $l_s, l_e, \texttt{Range}(S)$

iv. $\mu(\ell) = s$

v. $\Sigma; \cdot; \texttt{Dom}(S); S \vdash s{:}\tau \Longrightarrow l_o$*, and*

vi. $\Sigma; \cdot; \texttt{Dom}(S); S \vdash s'{:}\tau' \Longrightarrow l'_o$

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

6 Related Work

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

An earlier version of EGO was presented in [5, 6]. 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 objects. Our removal of first class functions simplified this system while retaining expressive power, and our addition of `let!` adds to its usability.

Our calculus is derived from features of the calculi of Abadi and Cardelli [1] and Fisher, Honsell and Mitchell [12, 13]. Like other object calculi [17, 19, 20], 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 [24]. 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 [11] 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 [25]. This work was based on Girard's linear logic [14]. Unique pointers were proposed for Eiffel and C++ in [18], 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.

Several papers describe research into ways to model objects in linear logic [4, 8, 10]. 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 [21]. DeLine and Fähndrich discuss typestates for objects, especially in the presence of subtyping, in [9]. 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 [23] or with explicit types as in Cyclone [22]. Compared to Cyclone, our regions are more flexible in that objects may refer to out-of-scope regions as long as these regions are not used; but Cyclone gains flexibility from region subtyping which our system does not support.

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 changing the objects a method has available at runtime. We have demonstrated the expressiveness of our calculus showing how the lambda calculus can be embedded in it.

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 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.

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.

Acknowledgements

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 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] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig 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. A type checked prototype-based model with linearity. Draft senior thesis, published as a Carnegie Mellon Technical Report CMU-ISRI-04-142, December 2004.

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

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

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

[9] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*. Springer-Verlag, 2004.

[10] Giorgio Delzanno and Maurizio Martelli. Objects in forum. In *International Logic Programming Symposium*, pages 115–129, 1995.

[11] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In *European Conference on Object-Oriented Programming*, pages 130–149, 2001.

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

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

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

[15] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th internation symposium on Memory management*, pages 73–84, 2004.

[16] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c, 2002.

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

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

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

[20] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999.

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

[22] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe Manual Memory Management in Cyclone. *Science of Computer Programming*, October 2006.

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

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

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

A Safety Proof

Progress

The first part of type safety is the progress theorem, which states that a well typed store and expression step to a new store and expression, or the original expression is a value.

Theorem (Progress)
If $\Sigma; \cdot; P; S \vdash e{:}\tau \Longrightarrow l_e$ and $\Sigma; S \vdash \mu \ \ \mathtt{ok} \Longrightarrow l_s$ then either

(i) $\mu, e \longrightarrow \mu', e'$ for some μ' and some e', or

(ii) e is a value.

To prove this, we need several lemmas. The first of these is the Canonical Forms Lemma which states that values of a given type have a spcific form.

Lemma (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 $L\mathbf{t}.O \leftarrow R$, it has the form ℓ_L.
4. More specifically, if a value has the type ¡obj $\mathbf{t}.O \leftarrow R$, it has the form $\ell_{¡\mathsf{obj}\cdot}$.
5. If a value has the type $\langle\rangle$, it has the form $\mathtt{null}$.

Proof of Canonical Forms
By case analysis on the typing rules. The forms a value can have are $\Lambda\rho_1.\cdots\Lambda\rho_n.\sigma x{:}\tau.e$, $\Lambda\rho_1.\cdots\Lambda\rho_n.¡\sigma x{:}\tau.e$, ℓ_L and $\mathtt{null}$.

1. The only rule which gives a value a type of $\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P} \tau_2$ is T-NLinMethwhich gives this type to a value of the form $\Lambda\rho_1.\cdots\Lambda\rho_n.\sigma x{:}\tau.e$.
2. The only rule which gives a value a type of $\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P}\!\!\circ\ \tau_2$ is T-LinMeth which gives this type to a value of the form $\Lambda\rho_1.\cdots\Lambda\rho_n.¡\sigma x{:}\tau.e$.
3. The rules which give a value a type of $L\mathbf{t}.O \leftarrow R$ are T-NLinLoc, T-LinLoc, T-BorLoc and T-ChLin. These rules give these types to values of the form ℓ_L.
4. The rule which gives a value a type of ¡obj $\mathbf{t}.O \leftarrow R$ is T-NLinLoc. This rule gives this type to values of the form $\ell_{¡\mathsf{obj}\cdot}$.
5. The only rule which gives a value a type of $\langle\rangle$ are T-Null which gives this value to $\mathtt{null}$.

We also need the `metht` Subterm Lemma which says that if we look up the type of a method in the type of an object, the returned type is a subterm of the object type.

Lemma (`metht` Subterm)
If $\mathtt{mtype}(\tau, m) = \tau'$, then $\tau' \in \tau$.
Proof of `metht` Subterm
The proof is by induction on the derivation of $\mathtt{mtype}(\tau, m) = \tau'$.

Proof of Progress

The proof is by induction on the typing derivations.

Case: T-VAR

Impossible as terms are closed.

Case: T-NLINMETH

$\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma x{:}\tau.e$ is a value.

Case: T-LINMETH

$\Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma x{:}\tau.e$ is a value.

Case: T-NLINLOC, T-LINLOC, T-BORLOC

ℓ_L is a value.

Case: T-NULL

`null` is a value.

Case: T-KILL

Impossible as terms are closed.

Case: T-COPY

Impossible as terms are closed.

Case: T-UPD

By the premise of the typing rule, e has type ¡obj $\mathbf{t}.O \leftarrow R$. If e is not a value, μ, e steps to some μ', e' by IH, so $\mu, e \leftarrow\!\!+ m = \sigma$ steps to $\mu', e' \leftarrow\!\!+ m = \sigma$ by C-UPD. If e is a value, it is of the form ℓ_L by canonical forms on its type which is of the form $L\mathbf{t}.O \leftarrow R$. By case analysis on the typing rules and `lmtype` rules, $\Sigma(\ell) =$ ¡obj $\mathbf{t}.O' \leftarrow R'$ with $m \in R$. By inversion on T-STORE, $\Sigma;\cdot;S;\mathsf{Dom}(S) \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_o$. By inversion on T-STORE, $\mu(\ell)$ exists and so is of the form $loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle$ with $m \in \mathsf{Dom}(\langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle)$. Therefore, there exists μ' such that $\mu, \ell_L \leftarrow\!\!+ m = \sigma$ steps to μ', ℓ_L by E-UPD.

Case: T-ADD

By the premise of the typing rule, e has type ¡obj $\mathbf{t}.O \leftarrow R$. If e is not a value, μ, e steps to some μ', e' by IH, so $\mu, e \leftarrow\!\!+ m = \sigma$ steps to $\mu', e' \leftarrow\!\!+ m = \sigma$ by C-UPD. If e is a value, it is of the form ℓ_L by canonical forms on its type which is of the form $L\mathbf{t}.O \leftarrow R$. By case analysis on the typing rules and `lmtype` rules, $\Sigma(\ell) =$ ¡obj $\mathbf{t}.O' \leftarrow R'$ with $m \notin R$. By inversion on T-STORE, $\Sigma;\cdot;S;\mathsf{Dom}(S) \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_o$. By inversion on T-STORE, $\mu(\ell)$ exists and so is of the form $loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle$ with $m \notin \mathsf{Dom}(\langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle)$. Therefore, there exists μ' such that $\mu, \ell_L \leftarrow\!\!+ m = \sigma$ steps to μ', ℓ_L by E-UPD.

Case: T-NLININV

By the premise of the typing rule, e has type obj $\mathbf{t}.O \leftarrow R$. If e is not a value, μ, e steps to some μ', e' by IH, so $\mu, e.m[\varrho_1, \cdots, \varrho_n]$ steps to $\mu', e'.m[\varrho_1, \cdots, \varrho_n]$ by C-NLININV. If e is a value, it is of the form ℓ_L by Canonical Forms on its type, $L\mathbf{t}.O \leftarrow R$. By case analysis on the typing rule, $\mathtt{mtype}(L\mathbf{t}.O \leftarrow R, m) = \tau$. By case analysis on the typing rules and the `metht` Subterm Lemma, $\Sigma(\ell) =$ ¡obj $\mathbf{t}.O' \leftarrow R'$ with $m \in R$. By inversion on T-STORE, $\Sigma;\cdot;S;\mathsf{Dom}(S) \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_o$. By inversion on T-STORE, $\mu(\ell)$ exists and so is of the form $loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle$ with $m \in \mathsf{Dom}(\langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle)$. Therefore, there exists μ' such that this expression steps by E-NLININV.

Case: T-LININV

By the premise of the typing rule, e has type ¡obj $\mathbf{t}.O \leftarrow R$. If e is not a value, μ, e steps to some μ', e' by IH, so $\mu, e.m[\varrho_1, \cdots, \varrho_n]$ steps to $\mu', e'.m[\varrho_1, \cdots, \varrho_n]$ by C-LININV. If e is a value, it is of the form ℓ_L by Canonical Forms on its type, $L\mathbf{t}.O \leftarrow R$. By case analysis on the typing rule, $\mathtt{lmtype}(L\mathbf{t}.O \leftarrow R, m) = \tau$. By case analysis on the typing rules and `lmtype` rules, $\Sigma(\ell) =$ ¡obj $\mathbf{t}.O' \leftarrow R'$ with $m \in R$. By inversion on T-STORE, $\Sigma;\cdot;S;\mathsf{Dom}(S) \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_o$. By inversion on T-STORE, $\mu(\ell)$ exists and so is of the form $loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle$ with $m \in \mathsf{Dom}(\langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle)$. Therefore, there exists μ' such that this expression steps by E-LININV.

Case: T-CHLIN

By the premise of the typing rule, e has type of the form ¡obj $\mathbf{t}.O \leftarrow R$. If e is not a value, μ, e steps to some μ', e' by IH, so $\mu, !e$ steps to $\mu'!e'$ by C-CHLIN. If e is a value, it is of the form $\ell_{¡\mathsf{obj}}$ by Canonical Forms on its type, ¡obj $\mathbf{t}.O \leftarrow R$, so it steps by C-CHLIN.

Case: T-NEW

$\mu, \langle\rangle$ steps to some μ', ℓ_L by E-NEW

Case: T-DEL

By the premise of the typing rule, e_1 and e_2 have types of the form $L\mathbf{t}.O \leftarrow R$. If either is a value, it has the form ℓ_L by Canonical Forms. If e_1 is not a value, μ, e_1 steps to some μ', e_1' by IH, so $\mu, e_1 \leftarrow e_2$ steps to $\mu', e_1' \leftarrow e_2$

by C-DEL$_1$. If e_2 is not a value and e_1 is, μ, e_2 steps to some μ', e_2' by IH, so $\mu, e_1 \leftarrow e_2$ steps to $\mu', e_1 \leftarrow e_2'$ by C-DEL$_2$. If both are values, by Canonical Forms both are of the form ℓ_L. By the premise of the typing rule, ℓ_{L_2} has type ¡obj $\mathbf{t}.O \leftarrow R$ By case analysis on the typing rules, $\Sigma(\ell) =$ ¡obj $\mathbf{t}.O \leftarrow R$. By inversion on T-STORE, $\Sigma; \cdot; S; \texttt{Dom}(S) \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_o$. By inversion on T-STORE, $\mu(\ell)$ exists and so is of the form $loc \leftarrow \langle \cdots \rangle$. Therefore, $\mu, v_1 \leftarrow v_2$ steps to some μ', v_2 by E-DEL.

Case: T-LET!$_1$

By the premise of the typing rule, e_1 has type $L\mathbf{t}.O \leftarrow R$. If e_1 is not a value, μ, e_1 steps to some μ', e_1' by IH, so μ, `let!` $(\rho)\, x_1 = e_1\, x_2 = e_2$ `in` e_3 `end` steps to μ, `let!` $(\rho)\, x_1 = e_1'\, x_2 = e_2$ `in` e_3 `end` by C-LET!$_1$. If e_1 is a value, this expression steps by E-LET!$_1$.

Case: T-LET!$_2$

By the premise of the typing rule, e_2 has type of the form $L\mathbf{t}.O \leftarrow R$. If e_2 is not a value, μ, e_2 steps to some μ', e_2' by IH, so μ, `let!` $(\rho)\, x_1 = v_1\, x_2 = e_2$ `in` e_3 `end` steps to μ, `let!` $(\rho)\, x_1 = v_1\, x_2 = e_2'$ `in` e_3 `end` by C-LET!$_2$. If e_1 is a value, this expression steps by E-LET!$_2$.

Preservation

Preservation says that if a closed expression and heap are well typed, the regions of the expression are a subset of the region context used to type the expression and there are no duplicates in the list of linear locations on the heap, the expression and the map of currently borrowed locations, and the expression and heap step to a new expression and heap, then there exists a new store typing and map of borrowed expressions under which all these properties.

For the proof, we need to define two functions that tell what regions are free in a type and in an expression. We free regions in types as follows.

$$
\begin{aligned}
\texttt{tregions}(o\mathbf{t}) &= \{\} \\
\texttt{tregions}(\varrho\mathbf{t}) &= \{\varrho\} \\
\texttt{tregions}(\langle\rangle) &= \{\} \\
\texttt{tregions}(\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P} \tau_2) &= P \cup \texttt{tregions}(\tau_1) \cup \texttt{tregions}(\tau_2) - \{\rho_1, \cdots, \rho_n\} \\
\texttt{tregions}(\forall\rho_1.\cdots\forall\rho_n.\tau_1 \overset{P}{\multimap} \tau_2) &= P \cup \texttt{tregions}(\tau_1) \cup \texttt{tregions}(\tau_2) - \{\rho_1, \cdots, \rho_n\} \\
\texttt{tregions}(\top) &= \{\}
\end{aligned}
$$

We define free regions in expressions as follows.

$$
\begin{aligned}
\texttt{eregions}(x) &= \{\} \\
\texttt{eregions}(\texttt{null}) &= \{\} \\
\texttt{eregions}(\ell_o) &= \{\} \\
\texttt{eregions}(\ell_\varrho) &= \{\varrho\} \\
\texttt{eregions}(\langle\rangle) &= \{\} \\
\texttt{eregions}(\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma(x{:}\tau).e) &= \texttt{eregions}(e) - \{\rho_1, \cdots, \rho_n\} \\
\texttt{eregions}(\Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma(x{:}\tau).e) &= \texttt{eregions}(e) - \{\rho_1, \cdots, \rho_n\} \\
\texttt{eregions}(e.m[\varrho_1, \cdots, \varrho_n]) &= \texttt{eregions}(e) \cup \{\varrho_1, \cdots, \varrho_n\} \\
\texttt{eregions}(e \leftarrow\!\!+ m = \sigma) &= \texttt{eregions}(e) \cup \texttt{eregions}(\sigma) \\
\texttt{eregions}(e_1 \leftarrow e_2) &= \texttt{eregions}(e_1) \cup \texttt{eregions}(e_2) \\
\texttt{eregions}(\texttt{let!}\ (\varrho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}) &= \texttt{eregions}(e_1) \cup \texttt{eregions}(e_2) \cup \texttt{eregions}(e_3) - \{\varrho\} \\
\texttt{eregions}(!e) &= \texttt{eregions}(e)
\end{aligned}
$$

Theorem (Preservation)

If $\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow l_s$, $\Sigma; \cdot; P; S \vdash e{:}\tau \Longrightarrow l_e$, $\texttt{eregions}(e) \in P$, there are are no duplicates in $l_e, l_s, \texttt{Range}(S)$, and $\mu, e \longrightarrow \mu', e'$, then for some $\Sigma' \geq_\ell \Sigma$, $\Sigma'; S' \vdash \mu'\ \ \texttt{ok} \Longrightarrow l_s'$, $\Sigma'; \cdot; P; S' \vdash e'{:}\tau' \Longrightarrow l_e'$, $\texttt{eregions}(e') \subseteq P$, $\tau' \leq \tau$ and that there are no duplicates in $l_s', l_e', \texttt{Range}(S)$.

For the proof of Preservation, we need several lemmas. The first of these is the Substitution Lemma. This says that if we substitute an expression with a type which is a subtype of a bound variable in for that variable in a second

expression, the resulting expression has a type which is a subtype of the original second expression.

Lemma (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 $\tau_2 \leq \tau_1$ then $\Sigma; A; P; S \vdash [v/x]e{:}\tau_2' \Longrightarrow l$, $\tau_2' \leq \tau_1'$ and $l \subseteq l_e, l_x$.

The proof of Substitution requires several lemmas.

The first of these is the Region Weakening Lemma, which says that if an expression can be typed under a given region context, it can be typed under an expanded region context.

Lemma (Region Weakening)
If $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l_o$ then $\Sigma; A; P, \varrho; S \vdash e{:}\tau \Longrightarrow l_o$.

Proof of Region Weakening
By induction on the derivation of $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l_o$.

Finally, we have the Weakening Lemma, which states that if an expression can be typed under a given type context, it can be typed under the same context expanded with a nonlinear binding.

Lemma (Weakening)
If $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$ and τ is nonlinear then $\Sigma; A, x{:}\tau; P; S \vdash e{:}\tau \Longrightarrow l$.

Proof of Weakening
This is a direct consequence of T-KILL

Lemma (Linear Substitution)
If $\Sigma; A, x{:}\tau_1; P; S \vdash e{:}\tau_1' \Longrightarrow l_e$, $\Sigma; \cdot; P'; S \vdash v{:}\tau_l \Longrightarrow l_x$ and $O \leftarrow R \leq \tau_1$, where τ_l is linear, then $\Sigma; A; P, P'; S \vdash e{:}\tau_1' \Longrightarrow l$ where $l \subseteq l_e, l_x$.

Proof of Linear Substitution
This is by induction on the typing rules.

Proof of Substitution
The lemma follows directly from the following stronger statement.

If $\Sigma; A, x{:}\tau_1, \cdots, x{:}\tau_1; P; S \vdash e{:}\tau_1' \Longrightarrow l_e$ where there is at most one x if τ is linear, $\Sigma; \cdot; P'; S \vdash v{:}\tau_2 \Longrightarrow l_x$ and $\tau_2 \leq \tau_1$ then $\Sigma; A; P, P'; S \vdash [v/x]e{:}\tau_2' \Longrightarrow l$, $\tau_2' \leq \tau_1'$ and $l \subseteq l_e, l_x$.

This is proven by case analysis on the derivation of $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$.

Case: T-NLINMETH

$\Sigma; A, x{:}\tau, \cdots, x{:}\tau; P; S \vdash \Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma y{:}\tau_1.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_2 \Longrightarrow \{\}$	Assumption
$x \neq y$	α-varying
$\Sigma; \cdot; P'; S \vdash v{:}\tau_s \Longrightarrow l_x$	Assumption
$\tau_s \leq \tau$	Assumption
$\Sigma; A, x{:}\tau, \cdots, x{:}\tau, y{:}\tau_1; P''; S \vdash e{:}\tau_2 \Longrightarrow \{\}$	Case Premise
$\texttt{tregions}(\tau_2) \subseteq P''$	Region Type
$y \notin \texttt{Dom}(A, x{:}\tau, \cdots, x{:}\tau)$	Case Premise
$A, x{:}\tau, \cdots, x{:}\tau$ `nonlinear`	Case Premise
$P'' \subseteq P, \rho_1, \cdots, \rho_n$	Case Premise
$\Sigma; A, y{:}\tau_1; P', P''; S \vdash [v/x]e{:}\tau_{2_s} \Longrightarrow l'$	IH
$\Sigma; A, y{:}\tau_1; P''; S \vdash [v/x]e{:}\tau_{2_s} \Longrightarrow l'$	Region Type
$\tau_{2_s} \leq \tau_2$	IH
$l' \subseteq l_x$	IH
$y \notin \texttt{Dom}(A)$	$y \notin \texttt{Dom}(A, x{:}\tau, \cdots, x{:}\tau)$
A `nonlinear`	$A, x{:}\tau, \cdots, x{:}\tau$ `nonlinear`
$\Sigma; A; P; S \vdash \Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma y{:}\tau_1.[v/x]e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \Longrightarrow l'$	T-NLINMETH
$\Sigma; A; P; S \vdash [v/x]\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma y{:}\tau_1.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \Longrightarrow l'$	Definition of Substitution

$\Sigma; A; P, P'; S \vdash [v/x]\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma y{:}\tau_1.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \Longrightarrow l'$ Region Weakening
$\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \leq \tau_1 \xrightarrow{P''} \tau_2$ Subtyping Rules

Case: T-LINMETH

$\Sigma; A, x{:}\tau, \cdots, x{:}\tau; P; S \vdash \Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma y{:}\tau_1.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_2 \Longrightarrow l_e$ Assumption
$x \neq y$ α-varying
$\Sigma; \cdot; P'; S \vdash v{:}\tau_s \Longrightarrow l_x$ Assumption
$\tau_s \leq \tau$ Assumption
$\Sigma; A, x{:}\tau, \cdots, x{:}\tau, y{:}\tau_1; P''; S \vdash e{:}\tau_2 \Longrightarrow l_e$ Case Premise
$y \notin \mathtt{Dom}(A, x{:}\tau, \cdots, x{:}\tau)$ Case Premise
$P' \subseteq P, \rho_1, \cdots, \rho_n$ Case Premise
$\mathtt{tregions}(\tau_2) \subseteq P''$ Region Type
$\Sigma; A, y{:}\tau_1; P', P''; S \vdash [v/x]e{:}\tau_{2_s} \Longrightarrow l'$ IH
$\Sigma; A, y{:}\tau_1; P''; S \vdash [v/x]e{:}\tau_{2_s} \Longrightarrow l'$ Region Type
$\tau_{2_s} \leq \tau_2$ IH
$l' \subseteq l_e, l_x$ IH
$y \notin \mathtt{Dom}(A)$ $y \notin \mathtt{Dom}(A, x{:}\tau, \cdots, x{:}\tau)$
$\Sigma; A; P; S \vdash \Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma y{:}\tau_1.[v/x]e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \Longrightarrow l'$ T-NLINMETH
$\Sigma; A; P; S \vdash [v/x]\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma y{:}\tau_1.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \Longrightarrow l'$ Definition of Substitution
$\Sigma; A; P, P'; S \vdash [v/x]\Lambda\rho_1.\cdots\Lambda\rho_n.\varsigma y{:}\tau_1.e{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \Longrightarrow l'$ Region Weakening
$\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P''} \tau_{2_s} \leq \tau_1 \xrightarrow{P''} \tau_2$ Subtyping Rules

Case: 1 of T-VAR

$\Sigma; A, x{:}\tau, \cdots, x{:}\tau; P; S \vdash x{:}\tau \Longrightarrow \{\}$ Assumption
$\Sigma; \cdot; P'; S \vdash v{:}\tau_s \Longrightarrow l_x$ Assumption
$\Sigma; \cdot; P'; S \vdash [v/x]x{:}\tau_s \Longrightarrow l_x$ Definition of Substitution
$\Sigma; A; P'; S \vdash [v/x]x{:}\tau_s \Longrightarrow l_x$ Weakening
$\Sigma; A; P, P'; S \vdash [v/x]x{:}\tau_s \Longrightarrow l_x$ Region Weakening
$\tau_s \leq \tau$ Assumption
$l_x \subseteq l_x$ Set Theory

Case: 2 of T-VAR

$\Sigma; A, x{:}\tau, \cdots, x{:}\tau; P; S \vdash y{:}\tau \Longrightarrow \{\}$ Assumption
$x \neq y$ Assumption
$\Sigma; A, x{:}\tau; P'; S \vdash [v/x]y{:}\tau \Longrightarrow \{\}$ Definition of Substitution
$\Sigma; A, x{:}\tau; P, P'; S \vdash [v/x]y{:}\tau \Longrightarrow \{\}$ Region Weakening
$\tau \leq \tau$ T-REFL
$\{\} \subseteq l_x$ Set Theory

Case: T-LET!$_1$

$\Sigma; A, x{:}\tau; P; S \vdash \mathtt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \mathtt{in}\ e_3\ \mathtt{end}{:}\tau_3 \Longrightarrow l_e$ Assumption
$\Sigma; \cdot; P'; S \vdash v{:}\tau_s \Longrightarrow l_x$ Assumption
$\tau_s \leq \tau$ Assumption
$x_1 \notin \mathtt{Dom}(A)$ Case Premise
$x_2 \notin \mathtt{Dom}(A)$ Case Premise
$\rho \notin \tau_2$ Case Premise
$\rho \notin P'$ Case Premise

$x_1 \neq x_2$	Case Premise
$\Sigma; A_1, x : \tau; P'; S \vdash e_1{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l_1$	Case Premise
$\Sigma; A_1; P; S \vdash [v/x]e_1{:}L\mathbf{t}.O_s \leftarrow R_s \Longrightarrow l'$	IH
$L\mathbf{t}.O_s \leftarrow R_s \leq L\mathbf{t}.O \leftarrow R$	IH
$l' \subseteq l_1, l_x \; \Sigma; A_2, x_1{:}\rho\mathbf{t}.O \leftarrow R; P, \rho; S \vdash e_2{:}\tau_2 \Longrightarrow l_2$	Case Premise
$\Sigma; A_2, x_1{:}\rho\mathbf{t}.O \leftarrow R; P, \rho, P'; S \vdash [v/x]e_2{:}\tau_2' \Longrightarrow l_2'$	IH
$\tau_2' \leq \tau_2$	IH
$\Sigma; A_2, x_1{:}\rho\mathbf{t}.O \leftarrow R; P, \rho, P'; S \vdash [v/x]e_2{:}\tau_2'' \Longrightarrow l_2'$	Variable Subtyping
$\tau_2'' \leq \tau_2'$	Variable Subtyping
$\tau_2'' \leq \tau_2$	T-SubTrans
$l_2' \subseteq l_x, l_2$	IH
$\Sigma; A_3, x_1{:}L\rho.tO \leftarrow R, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$	Case Premise
$\Sigma; A_3, x_1{:}L\rho.tO \leftarrow R, x_2{:}\tau_2; P, P'; S \vdash [v/x]e_3{:}\tau_3' \Longrightarrow l_3'$	IH
$\tau_3' \leq \tau_3$	IH
$\Sigma; A_3, x_1{:}L\rho.tO_s \leftarrow R_s, x_2{:}\tau_2''; P, P'; S \vdash [v/x]e_3{:}\tau_3'' \Longrightarrow l_3'$	Variable Subtyping
$\tau_3'' \leq \tau_3'$	Variable Subtyping
$\tau_3'' \leq \tau_3$	T-SubTrans
$l_3' \subseteq l_x, l_2$	IH
$\Sigma; A; P, P'; S \vdash \texttt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ [v/x]e_3\ \texttt{end}{:}\tau_{3''} \Longrightarrow l_1', l_2', l_3'$	T-Let!$_1$
Subcase: v is $\ell_{\text{¡obj}}$ or ¡$\varsigma(x{:}\tau).e$	
$\tau_s =$ ¡obj $\mathbf{t}.O_{ls} \leftarrow R_{ls}$	Case Analysis
$\tau =$ ¡obj $\mathbf{t}.O_l \leftarrow R_l$	Subtyping rules
At most one x in $A, x{:}\tau, \cdots, x{:}\tau$	Premise
No duplicates in l_1', l_2', l_3'	Linear Substitution
Subcase: v is $\tau_v \overset{P}{\multimap} \tau_v'$	
$\tau_s = \tau_{vs} \overset{P}{\multimap} \tau_{vs}'$	Case Analysis
$\tau = \tau_v \overset{P}{\multimap} \tau_v'$	Subtyping rules
At most one x in $A, x{:}\tau, \cdots, x{:}\tau$	Premise
No duplicates in l_1', l_2', l_3'	Linear Substitution
Subcase: Otherwise	
$\Sigma; \cdot; P'; S \vdash v{:}\tau_s \Longrightarrow \{\}$	Case Analysis
$l_1', l_2', l_3' \subseteq l_1, l_2, l_3$	Set Theory

The case for T-Let!$_2$ is similar.

We also have the Region Substitution Lemma, which states that if a region is substituted in for a region variable in an epxression, the resulting expression then has the type of the original expression with the same substitution performed on the type.

Lemma (Region Substitution)

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

For this we need the Region Subtyping Lemma which says that subtyping relations are preserved under the same region substitution on both types.

Lemma (Region Subtyping)

$\tau \leq \tau'$ if and only if $[\varrho/\rho]\tau \leq [\varrho/\rho]\tau'$.

Proof of Region Substitution

The proof is by induction on the typing judgement. The difficult case, that of borrowed location typing, follows.

$\Sigma; A; P; S \vdash \ell_{\varrho'}{:}\varrho'\mathbf{t}.O' \leftarrow R' \Longrightarrow \{\}$	Assumption

$\Sigma; A; P; S \vdash \ell_o{:}o\mathbf{t}.O \leftarrow R \Longrightarrow l$	Premise
$o\mathbf{t}.O \leftarrow R \leq o\mathbf{t}.O' \leftarrow R'$	Premise
$\texttt{eregions}(\varrho'\mathbf{t}.O' \leftarrow R') \subseteq P$	Premise
$\Sigma; [\varrho/\rho]A; [\varrho/\rho]P; S \vdash [\varrho/\rho]\ell_o{:}[\varrho/\rho]o\mathbf{t}.O \leftarrow R \Longrightarrow l$	IH
$[\varrho/\rho]o\mathbf{t}.O \leftarrow R \leq [\varrho/\rho]o\mathbf{t}.O' \leftarrow R'$	Region Subtyping
$[\varrho/\rho]\texttt{eregions}(\varrho'\mathbf{t}.O' \leftarrow R') \subseteq [\varrho/\rho]P$	Set Theory
$\Sigma; [\varrho/\rho]A; [\varrho/\rho]P; S \vdash [\varrho/\rho]\ell_{\varrho'}{:}[\varrho/\rho]\varrho'\mathbf{t}.O' \leftarrow R' \Longrightarrow \{\}$	T-BORLOC

We also have the Store Weakening Lemma, which says if we change a store typing Σ to Σ' such that $\Sigma' \geq_\ell \Sigma$, any expressions typed under the first expression can still be typed under the second.

Lemma (Store Weakening)
If $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l_e$, $\Sigma; S \vdash \mu\ \texttt{ok} \Longrightarrow l_s$, $\Sigma' \geq_\ell \Sigma$ no duplicates in $l_e, l_s, \texttt{Range}(S)$ and $\ell \notin l_e$, then $\Sigma'; A; P; S \vdash e{:}\tau \Longrightarrow l$

To prove this, we need several lemmmas.

The Store Contraction Lemma says that if an expression is well typed and does not mention a given location, that location can be removed from the store typing and the expression will remain well typed.

Lemma (Store Contraction)
If $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$ and $\ell_L \notin e$, then $[/\ell \mapsto s]\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$.

Proof of Store Contraction
The proof is by induction on the typing judgement.

The Linear Location Store Lemma says that if a heap is well typed and the heap typing judgement does not return a given location in the list of linear locations in the heap, a reference to that location is not on the heap.

Lemma (Linear Location Store)
If $\Sigma; A; P; S \vdash \ell_L{:}\tau \Longrightarrow l_e$, $\Sigma; S \vdash \mu\ \texttt{ok} \Longrightarrow l_s$ and $\ell \notin l_s$ then $\ell_L \notin \texttt{Range}(\Sigma)$

Proof of Linear Location Store
The proof is by case analysis on T-STORE and T-ODESCR and induction on the typing rules.

The Borrowed Location Store Lemma says that if a region is not in the set of currently borrowed regions, no reference to it is in any typable method in the heap.

Lemma (Borrowed Location Store)
If $\Sigma; S \vdash \mu\ \texttt{ok} \Longrightarrow l_s$ and $r \notin \texttt{Dom}(S)$, then for all σ in all s in $\texttt{Dom}(\mu)$ such that $\Sigma; A; P'; S \vdash \sigma_i{:}\tau_i \Longrightarrow l_i$ if $P' \subseteq \texttt{Dom}(S)$, $\ell_r \notin \texttt{Range}(\Sigma)$.

Proof of Borrowed Location Store
The proof is by inversion on T-STORE and induction on the typing rules.

Proof of Store Weakening
The proof is by case analysis on the derivation of $Ps' \geq_\ell \Sigma$. In each case, it proceeds by straighforward induction on the derivation of $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l_e$.

Case: S-Grow
By induction on the typing rules.

Case: S-LObj
By induction on the typing rules.

Case: S-ChLin
By induction on the typing rules.

There is also the mbody Type Lemma, which states that if an object is well typed and its type contains the type of a method, that method can be typed with a subtype of this type.

Lemma (mbody Type)

If $\texttt{mbody}(\mu, \ell_L, m) = \Lambda\rho_1.\cdots\Lambda\rho_n.\sigma$, $\texttt{mtype}(L\mathbf{t}.[\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$, $\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow l_s$ and $\Sigma; \cdot; P; S \vdash \ell_L{:}L\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R \Longrightarrow l_l$, then $\Sigma; \cdot; P; S \vdash \sigma{:}\tau \Longrightarrow l_e$ and $\tau \leq \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$

Proof of mbody Type

By induction on the derivation of $\texttt{mtype}(\tau, m) = \tau'$.

Case: T-MethT$_1$

$\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow l_s$	Assumption
$\forall \ell' \in \texttt{Dom}(\mu).\Sigma; \cdot; \texttt{Dom}(S); S \vdash \mu(\ell'){:}\Sigma(\ell') \Longrightarrow l_{\ell'}$	Inversion
$\texttt{Dom}(\mu) = \texttt{Dom}(\Sigma)$	Inversion
$\texttt{mtype}(L\mathbf{t}.[\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$	Assumption
$\texttt{lmtype}(L\mathbf{t}.[\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$	Premise
$m{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2 \in [\mathbf{t}.O' \leftarrow R'/\mathbf{t}]R$	Inversion
$\texttt{mbody}(\mu, \ell_L, m) = \sigma$	Assumption
If $m \in \texttt{Dom}(\mu(\ell))$ then $m = \sigma \in \mu(\ell)$	Case Analysis
Subcase: T-LinLoc	
$\Sigma; \cdot; P; S \vdash \ell_L{:}$¡obj $\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R \Longrightarrow l_e$	Assumption
$\Sigma(\ell) =$ ¡obj $\mathbf{t}.O_s \leftarrow R_s$	Subcase Premise
¡obj $\mathbf{t}.O_s \leftarrow R_s \leq$ ¡obj $\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R$	Subcase Premise
$m{:}\forall\rho_1.\cdots\forall\rho_n.\tau_{1_s} \xrightarrow{P'} / \overset{P'}{\multimap} \tau_{2_s} \in [\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}]R_s$	Subtyping Rules
$\texttt{tregions}($¡obj $\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R) \subseteq P$	Inversion
$P' \subseteq P$	Definition of Substitution, $\texttt{tregions}(\tau)$
$\forall\rho_1.\cdots\forall\rho_n.\tau_{1_s} \xrightarrow{P'} / \overset{P'}{\multimap} \tau_{2_s} \leq \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$	Subtyping Rules
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle{:}$ ¡obj $\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O_s \leftarrow R_s \Longrightarrow l_{loc}, l_1, \cdots, l_n$	Inversion on T-STORE
$m = \sigma \in \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n\rangle$	Modus Ponens
$\forall i \in 1..n \vert P'' \subseteq \texttt{Dom}(S).\Sigma; \cdot; P''; S \vdash \sigma_i{:}\forall\rho_1.\cdots\forall\rho_{n_i}.\tau_i \overset{P'',\rho_1,\cdots,\rho_n}{\longrightarrow} / \overset{P'',\rho_1,\cdots,\rho_n}{\multimap} \tau_i' \Longrightarrow l_i$	Inversion on T-ODESCR
$P \subseteq \texttt{Dom}(S)$	Region Contexts
$P' \subseteq \texttt{Dom}(S)$	Set Theory
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash \sigma{:}\forall\rho_1.\cdots\forall\rho_n.\tau_{1_s} \xrightarrow{P'} / \overset{P'}{\multimap} \tau_{2_s} \Longrightarrow l_i$	Instantiation
$[¡]\varsigma(x{:}\tau_{1_s}).e = \sigma$	Definition
Subsubcase: T-LinMeth	
$\Sigma; x{:}\tau_{1_s}; P'; S \vdash e{:}\tau_{2_s} \Longrightarrow l_i$	Inversion
$P' \subseteq \texttt{Dom}(S), \rho_1, \cdots, \rho_n$	Inversion
$\rho_1, \cdots, \rho_n \notin \texttt{Dom}(S)$	Inversion
$x \notin \cdot$	Inversion
$P' \subseteq P, \rho_1, \cdots, \rho_n$	Set Theory
$\rho_1, \cdots, \rho_n \notin P$	Set Theory
$\Sigma; \cdot; P; S \vdash \sigma{:}\forall\rho_1.\cdots\forall\rho_n.\tau_{1_s} \xrightarrow{P'} / \overset{P'}{\multimap} \tau_{2_s} \Longrightarrow l_i$	T-LINMETH
Subsubcase: T-NLinMeth	
$\Sigma; x{:}\tau_{1_s}; P'; S \vdash e{:}\tau_{2_s} \Longrightarrow l_i$	Inversion
$P' \subseteq \texttt{Dom}(S), \rho_1, \cdots, \rho_n$	Inversion
$\rho_1, \cdots, \rho_n \notin \texttt{Dom}(S)$	Inversion
$x \notin \cdot$	Inversion

$\cdot$ `nonlinear`	Inversion
$P' \subseteq P, \rho_1, \cdots, \rho_n$	Set Theory
$\rho_1, \cdots, \rho_n \notin P$	Set Theory
$\Sigma; \cdot; P; S \vdash \sigma{:}\forall\rho_1.\cdots\forall\rho_n.\tau_{1_s} \xrightarrow{P'} / \overset{P'}{\multimap} \tau_{2_s} \Longrightarrow l_i$	T-NLinMeth

Subcase: T-NLinLoc

Symmetric.

Subcase: T-BorLoc

Either, $o =$ obj or $o =$ ¡obj, in which case, symmetric to the respective case above.

Case: T-MethT$_2$

$\texttt{mtype}(L\mathbf{t}.[\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$	Assumption
$\texttt{lmtype}(L\mathbf{t}.[\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R, m) \neq$	Premise
$m{:}\forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2 \notin R$	Inversion
$\texttt{mtype}([\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O, m) = \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$	Inversion
$\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow \ell_s$	Assumption
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_o$	Inversion
$m{:}\tau \notin [\mathbf{t}.O' \leftarrow R'/\mathbf{t}]R$	Inversion
SubCase: T-LinLoc or T-NLinLoc	
$\Sigma; \cdot; P; S \vdash \ell_L{:}L\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R \Longrightarrow l_e$	Assumption
$\Sigma(\ell) = L\mathbf{t}.L_o\mathbf{t_o}.O_o \leftarrow R_o \leftarrow R_s$	Subcase Premise
$L\mathbf{t}.L_o\mathbf{t_o}.O_o \leftarrow R_o \leftarrow R_s \leq L\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R$	Subcase Premise
$\texttt{tregions}(L\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O \leftarrow R) \subseteq P$	Subcase Premise
$R_s \leq [\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}R$	Subtyping Rules
$m{:}\tau' \notin R_s$	Subtyping Rules
$m = \sigma \notin \texttt{Range}(\mu(\ell))$	Inversion on T-ODescr
$L_o\mathbf{t_o}.O_o \leftarrow R_o \leq [\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]O$	Subtyping rules
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc{:}[\mathbf{t_o}/\mathbf{t_o}.O_o \leftarrow R_o][\mathbf{t}.O' \leftarrow R'/\mathbf{t}]L_o\mathbf{t_o}.O_o \leftarrow R_o \Longrightarrow l'_o$	Inversion on T-ODescr
$\texttt{mtype}([L\mathbf{t}.\mathbf{t}.O' \leftarrow R'/\mathbf{t}]L_o\mathbf{t_o}.O_o \leftarrow R_o, m) = \forall\rho_1.\cdots\forall\rho_n.\tau'_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau'_2$	`mtype` Subtyping
$\forall\rho_1.\cdots\forall\rho_n.\tau'_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau'_2 \leq \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$	`mtype` Subtyping
$\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, \cdots, m_n = \sigma_n\rangle$	Case Analysis
$\texttt{mbody}(\mu, loc, m) = \sigma$	Case Analysis
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash \sigma{:}\forall\rho_1.\cdots\forall\rho_n.\tau'_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau''_2 \Longrightarrow l_o$	IH
$\forall\rho_1.\cdots\forall\rho_n.\tau''_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau''_{2_s} \leq \forall\rho_1.\cdots\forall\rho_n.\tau'_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau'_2$	IH
$\forall\rho_1.\cdots\forall\rho_n.\tau''_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau''_{2_s} \leq \forall\rho_1.\cdots\forall\rho_n.\tau_1 \xrightarrow{P'} / \overset{P'}{\multimap} \tau_2$	T-SubTrans

The Store Change Lemma says that if we change a linear object on the heap and update its type in the store typing, the store remains well typed.

Lemma (Store Change)

If $\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow l_s$, $\Sigma; A; P; S \vdash \ell_L{:}$¡obj $\mathbf{t}.O \leftarrow R \Longrightarrow l_e$, no duplicates in $l_s, l_e, \texttt{Range}(S)$, $\mu(\ell) = s$, $\Sigma; \cdot; \texttt{Dom}(S); S \vdash s{:}\tau \Longrightarrow l_o$ and $\Sigma; \cdot; \texttt{Dom}(S); S \vdash s'{:}\tau' \Longrightarrow l'_o$, then $[\ell \mapsto \tau']\Sigma; S \vdash [\ell \mapsto s']\mu\ \ \texttt{ok} \Longrightarrow l_s - l_o, l'_o$

Proof of Store Change

$\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow l_s$	Assumption
$\Sigma; A; P; S \vdash \ell_L{:}$¡obj $\mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Assumption
No duplicates in $l_s, l_e, \texttt{Range}(S)$	Assumption
$\Sigma; \cdot; P; S \vdash s{:}\tau \Longrightarrow l_o$	Assumption

$\Sigma; \cdot; P; S \vdash s'{:}\tau' \Longrightarrow l'_o$ Assumption

$\Sigma; A; P; S \vdash \ell_L{:}\text{¡obj } \mathbf{t}.O \leftarrow R \Longrightarrow \{\ell\}$ Case Analysis

$\ell \notin l_s$ No duplicates in $l_s, l_e, \texttt{Range}(S)$

$\ell \notin \texttt{Range}(S)$ No duplicates in $l_s, l_e, \texttt{Range}(S)$

$\forall \ell' \in \texttt{Dom}(\mu).\Sigma; \cdot; \texttt{Dom}(S); S \vdash \mu(\ell'){:}\Sigma(\ell') \Longrightarrow l_{\ell'}$ Inversion

$\texttt{Dom}(\Sigma) = \texttt{Dom}(\mu)$ Inversion

$\forall loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle \in \texttt{Range}(\mu).\forall i \in 1..n | P \subseteq \texttt{Dom}(S).$
$\quad \Sigma; A; P; S \vdash \sigma_i{:}\forall \rho_1. \cdots \forall \rho_n.\tau_i \xrightarrow{P} / \overset{P}{\multimap} \tau'_i \Longrightarrow l_i$ Inversion

$\ell_{\text{¡}obj} \notin \texttt{Range}(\Sigma)$ Linear Location Store

$\forall loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle \in \texttt{Range}(\mu).\forall i \in 1..n | P \subseteq \texttt{Dom}(S).\ell_{\text{¡}obj} \notin \sigma_i$ Set Theory

$\forall loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle \in \texttt{Range}(\mu).\forall i \in 1..n | P\tau_i \subseteq \texttt{Dom}(S).\ell_\varrho \notin \sigma_i$ Borrowed Location Store

$\forall \ell' \in \texttt{Dom}([/\ell \mapsto s]\mu).\Sigma; \cdot; \texttt{Dom}(S); S \vdash [/\ell \mapsto s]\mu(\ell'){:}[/\ell \mapsto \tau]\Sigma(\ell') \Longrightarrow l_{\ell'}$ Store Contraction

$\texttt{Dom}([/\ell \mapsto \tau']\Sigma) = \texttt{Dom}([/\ell \mapsto s']\mu)$ Definition of Substitution

$[/\ell \mapsto \tau']\Sigma; S \vdash [/\ell \mapsto s']\mu \ \texttt{ok} \Longrightarrow l_s - l_o, l'_o$ T-STORE

We also have the Subterm Location Lemma. This states that if a subexpression containing a set of linear locations disjoint from those in the rest of the program is evaluated, the linear locations in it are still disjoint from those of the rest of the program.

Lemma (Subterm Location)
If no duplicates in $l_e, l_s, \texttt{Range}(S)$, $l \subseteq l_e$, $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l_e$, $\Sigma; A; P; S \vdash e_s{:}\tau_s \Longrightarrow l$, $\Sigma; S \vdash \mu \ \texttt{ok} \Longrightarrow l_s$, $\mu, e_s \longrightarrow \mu', e'_s$, $\Sigma'; A; P; S' \vdash e'_s{:}\tau_s \Longrightarrow l'$, and $\Sigma'; S' \vdash \mu' \ \texttt{ok} \Longrightarrow l'_s$ then no duplicates in $(l_e - l), l', l_s, \texttt{Range}(S')$.

Proof of Subterm Location
The proof is by induction on $\mu, e \longrightarrow \mu', e'$.

The Region Type Lemma and the Region Expression Lemma state that the regions found in a well typed expression and its type are a subset of those in scope.

Lemma (Region Type)
$\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$ iff $\texttt{tregions}(\tau) \subseteq P$

Proof of Region Type
By induction on the typing rules.

Lemma (Region Expression)
$\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$ iff $\texttt{eregions}(e) \subseteq P$

Proof of Region Expression
By induction on the typing rules.

The List Equality Lemma says that if an expression is typed with two different types, the list of linear locations produced by both judgements is the same.

Lemma (List Equality)
If $\Sigma; A; P; S \vdash e{:}\tau \Longrightarrow l$ and $\Sigma'; A'; P'; S' \vdash e{:}\tau' \Longrightarrow l'$, then $l = l'$.

Proof of List Equality
By induction on the typing rules.

The Region Contexts Lemma says that if we can type an expression under a given region context and a given map from regions to locations that are borrowed at them, the context is a subset of the domain of the map.

Lemma (Region Contexts)
If $\Sigma; A; P; S \vdash \ell_L{:}\tau \Longrightarrow l_e$ then $P \subseteq \texttt{Dom}(S)$
Proof of Region Contexts
By induction on the typing rules.

The Variable Subtyping Lemma says that if an expression is typed at a type with a given variable bound in it, the expression is typed at a subtype of its original type if the variable is bound at a subtype of its original type.

Lemma (Variable Subtyping)
If $\Sigma; A, x\tau_1; P; S \vdash e{:}\tau_2 \Longrightarrow l$ and $\tau_1' \leq \tau_1$, then $\Sigma; A, x\tau_1'; P; S \vdash e{:}\tau_2' \Longrightarrow l$ and $\tau_2' \leq \tau_2$.
Proof of Variable Subtyping
The proof is by induction on the typing judgements.

The `lmtype` Subtyping Lemma and the `mtype` Subtyping Lemma say that if an object type contains a method type, a subtype of the object type contains a subtype of the method type.

Lemma (`lmtype` Subtyping)
If $\texttt{lmtype}(\tau_1, m) = \tau_2$ and $\tau_1' \leq \tau_1$, then $\texttt{lmtype}(\tau_1', m) = \tau_2'$ and $\tau_2' \leq \tau_2$.
Proof of Proof of `lmtype` Subtyping
By case analysis on derivation of $\texttt{lmtype}(\tau_1, m) = \tau_2$.

Lemma (`mtype` Subtyping)
If $\texttt{mtype}(\tau_1, m) = \tau_2$ and $\tau_1' \leq \tau_1$, then $\texttt{mtype}(\tau_1', m) = \tau_2'$ and $\tau_2' \leq \tau_2$.
Proof of `mtype` Subtyping
By induction on the derivation of $\texttt{mtype}(\tau_1, m) = \tau_2$.

The Linear Location Change Lemma says that if typing an expression does not yield a given linear location in the list of contained linear locations, changing the type of this location will not change the type of the expression.

Lemma (Linear Location Change)
If $\Sigma; A; P; S \vdash \ell_L{:}¡\text{obj } \mathtt{t}.O \leftarrow R \Longrightarrow l$, $\Sigma; A'; P; S \vdash e{:}\tau \Longrightarrow l'$ and $\ell \notin l'$, then $[\ell \mapsto \tau']\Sigma; A'; P; S \vdash e{:}\tau \Longrightarrow l'$
Proof of Linear Location Change
By induction on the typing rules.

The Region Contraction Lemma says that if the type of an expression does not contain a given region, the region is not needed in the region context to type the expression.

Lemma (Region Contraction)
If $\Sigma; A; P, r; S \vdash v{:}\tau \Longrightarrow l$ and $r \notin \texttt{tregions}(\tau)$ then $\texttt{eregions}(e) \subseteq P$.
Proof of Region Contraction
By induction on the typing rules.

The Region and Substitution Lemma says that substituting one expression typed under a given region context into another typed under the same region context produces an expression which can be typed under the same context.

Lemma (Region and Substitution)
If $\texttt{eregions}(e) \subseteq P$ and $\texttt{eregions}(e') \subseteq P$ then $\texttt{eregions}([e'/x][\varrho/\rho]e) \subseteq [\varrho/\rho]P$.
Proof of Region and Substitution
This follows from the definiton of $\texttt{eregions}(e)$.

The Folding Subtyping Lemma says that subtyping relations are preserved under folding or unfolding both sides.

Lemma (Folding Subtyping)
If $L\mathbf{t}.O_1 \leftarrow R_1 \leq L\mathbf{t}.O_2 \leftarrow R_2$, $\tau_1 = L\mathbf{t}.[\mathbf{t}/\tau_1]O_1 \leftarrow R_1$ and $\tau_2 = L\mathbf{t}.[\mathbf{t}/\tau]O_2 \leftarrow R_2$, then $\tau_1 \leq \tau_2$.
Proof of Folding Subtyping
By induction on the typing rules.

Finally, the Folding Subtyping Lemma says that the same region context can be used to type an expression after folding or unfolding.

Lemma (Folding Region)
$\texttt{tregions}(L\mathbf{t}.O \leftarrow R) \subseteq P$ if and only if $\texttt{tregions}(L\mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}]O \leftarrow R) \subseteq P$.
Proof of Folding Region
The proof is by induction on the typing rules.

Proof of Preservation
The proof is by induction on the derivation of $\mu, e \longrightarrow \mu', e'$.
Case: C-INV

$\mu, e \longrightarrow \mu', e'$	Case Premise
$\Sigma; S \vdash \mu$ $\texttt{ok} \Longrightarrow l_s$	Assumption
$\texttt{eregions}(e.m[\varrho_1, \cdots, \varrho_n]) \subseteq P$	Assumption
subcase T-NLININV	
$\Sigma; \cdot; P; S \vdash e.m[\varrho_1, \cdots, \varrho_n]{:}\tau' \Longrightarrow l_e$	Assumption
No duplicates in l_e,l_s, $\texttt{Range}(S)$	Assumption
$\Sigma; \cdot; P; S \vdash e{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l$	Subcase Premise
$\texttt{mtype}([\mathbf{t}O \leftarrow R/\mathbf{t}]L\mathbf{t}.O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.L\mathbf{t}'.O' \leftarrow R' \xrightarrow{P'} \tau'$	Subcase Premise
$L\mathbf{t}.O \leftarrow R \leq [\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O' \leftarrow R'/\rho_1, \cdots, \rho_n, \mathbf{t}']L\mathbf{t}'.O' \leftarrow R'$	Subcase Premise
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P' \subseteq P$	Subcase Premise
$\Sigma'; \cdot; P; S' \vdash e'{:}L\mathbf{t}.O_s \leftarrow R_s \Longrightarrow l'_e$	IH
$L\mathbf{t}.O_s \leftarrow R_s \leq L\mathbf{t}.O \leftarrow R$	IH
$\texttt{mtype}([\mathbf{t}O_s \leftarrow R_s/\mathbf{t}]L\mathbf{t}.O_s \leftarrow R_s, m) = \forall\rho_1.\cdots\forall\rho_n.L\mathbf{t}'.O'_s \leftarrow R'_s \xrightarrow{P'} \tau'_s$	`mtype` Subtyping
$\forall\rho_1.\cdots\forall\rho_n.L\mathbf{t}'.O'_s \leftarrow R'_s \xrightarrow{P'} \tau'_s \leq \forall\rho_1.\cdots\forall\rho_n.L\mathbf{t}'.O' \leftarrow R' \xrightarrow{P'} \tau'$	`mtype` Subtyping
$\tau'_s \leq \tau'$	T-SUBNLINMETH
$L\mathbf{t}'.O' \leftarrow R' \leq L\mathbf{t}'.O'_s \leftarrow R'_s$	T-SUBNLINMETH
$[\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O' \leftarrow R'/\rho_1, \cdots, \rho_n, \mathbf{t}']L\mathbf{t}'.O' \leftarrow R' \leq$ $[\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O'_s \leftarrow R'_s/\rho_1, \cdots, \rho_n, \mathbf{t}']L\mathbf{t}'.O'_s \leftarrow R'_s$	Region Subtypingand Folding Subtyping
$L\mathbf{t}.O \leftarrow R \leq [\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O'_s \leftarrow R'_s/\rho_1, \cdots, \rho_n, \mathbf{t}']L\mathbf{t}'.O'_s \leftarrow R'_s$	Region Subtypingand Folding Subtyping
$\Sigma' \geq_\ell \Sigma$	IH
$\Sigma'; S' \vdash \mu'$ $\texttt{ok} \Longrightarrow l'_s$	IH
No duplicates in l'_e,$l'_s\texttt{Range}(S')$	IH
$\texttt{eregions}(e'.m[\varrho_1, \cdots, \varrho_n]) \subseteq P$	IH
$\Sigma'; \cdot; P; S' \vdash e'.m[\varrho_1, \cdots, \varrho_n]{:}\tau'_s \Longrightarrow l_e$	T-NLININV
subcase T-LININV	
$\Sigma; \cdot; P; S \vdash e.m[\varrho_1, \cdots, \varrho_n]{:}\tau' \Longrightarrow l_e$	Assumption
No duplicates in l_e,l_s, $\texttt{Range}(S)$	Assumption
$\Sigma; \cdot; P; S \vdash e{:}¡\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Subcase Premise
$\texttt{lmtype}([\mathbf{t}O \leftarrow R/\mathbf{t}]¡\text{obj } \mathbf{t}.O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.¡\text{obj } \mathbf{t}'.O' \leftarrow R' \xrightarrow{P'}\!\circ\ \tau_2$	Subcase Premise
$¡\text{obj } \mathbf{t}.O'' \leftarrow R'' = [\mathbf{t}O \leftarrow R/\mathbf{t}]¡\text{obj } \mathbf{t}.O \leftarrow R$	Subcase Premise
$\tau_f = [\mathbf{t}/\tau_f]¡\text{obj } \mathbf{t}.O'' \leftarrow [/m{:}\tau]R''$	Subcase Premise
$\tau_f \leq [\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O' \leftarrow R'/\rho_1, \cdots, \rho_n, \mathbf{t}']¡\text{obj } \mathbf{t}'.O' \leftarrow R'$	Subcase Premise
$[\varrho_1, \cdots, \varrho_n/\rho_n, \cdots \rho_n]P' \subseteq P$	Subcase Premise

$\Sigma' \geq_\ell \Sigma$ IH

$\Sigma'; \cdot; P; S' \vdash e'{:}\text{¡obj } \mathbf{t}.O_s \leftarrow R_s \Longrightarrow l'_e$ IH

$\text{¡obj } \mathbf{t}.O_s \leftarrow R_s \leq \text{¡obj } \mathbf{t}.O \leftarrow R$ IH

$\texttt{lmtype}([\mathbf{t}O_s \leftarrow R_s/\mathbf{t}]\text{¡obj } \mathbf{t}.O_s \leftarrow R_s, m) =$

$\quad \forall \rho_1. \cdots \forall \rho_n.\text{¡obj } \mathbf{t}'.O'_s \leftarrow R'_s \xrightarrow{P'} \tau'_s$ `lmtype` Subtyping

$\forall \rho_1. \cdots \forall \rho_n.\text{¡obj } \mathbf{t}'.O'_s \leftarrow R'_s \xrightarrow{P'} \tau'_s \leq \forall \rho_1. \cdots \forall \rho_n.\text{¡obj } \mathbf{t}'.O' \leftarrow R' \xrightarrow{P'} \tau'$ `lmtype` Subtyping

$\tau'_s \leq \tau'$ T-SubLinMeth

$\text{¡obj } \mathbf{t}'.O_s \leftarrow R_s \leq \text{¡obj } \mathbf{t}'.O'_s \leftarrow R'_s$ T-SubLinMeth

$[\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O' \leftarrow R'/\rho_1, \cdots, \rho_n, \mathbf{t}']\text{¡obj } \mathbf{t}'.O' \leftarrow R' \leq$

$\quad [\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O'_s \leftarrow R'_s/\rho_1, \cdots, \rho_n, \mathbf{t}']\text{¡obj } \mathbf{t}'.O'_s \leftarrow R'_s$ Region Subtypingand Folding Subtyping

$\tau_f \leq [\varrho_1, \cdots, \varrho_n, \mathbf{t}'.O'_s \leftarrow R'_s/\rho_1, \cdots, \rho_n, \mathbf{t}']\text{¡obj } \mathbf{t}'.O'_s \leftarrow R'_s$ Region Subtypingand Folding Subtyping

$\Sigma'; S' \vdash \mu'\ \texttt{ok} \Longrightarrow l'_s$ IH

No duplicates in l'_e,l'_s, $\texttt{Range}(S')$ IH

$\texttt{eregions}(e'.m[\varrho_1, \cdots, \varrho_n]) \subseteq P$ IH

$\Sigma'; \cdot; P; S \vdash e'.m[\varrho_1, \cdots, \varrho_n]{:}\tau'_s \Longrightarrow l'_e$ T-LinInv

Case: E-NLinInv

$\texttt{mbody}(\mu, \ell_L, m) = \Lambda\rho_1. \cdots \Lambda\rho_n.\varsigma x{:}\tau'.e$ Case Premise

$\Sigma; S \vdash \mu\ \texttt{ok} \Longrightarrow l_s$ Assumption

$\texttt{eregions}(\ell_L.m[\varrho_1, \cdots, \varrho_n]) \subseteq P$ Assumption

$\texttt{eregions}(\ell_L) \subseteq P$ Definition of $\texttt{eregions}(e)$

subcase T-NLinInv

$\Sigma; \cdot; P; S \vdash \ell_L.m[\varrho_1, \cdots, \varrho_n]{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau \Longrightarrow l_e$ Assumption

No duplicates in l_e,l_s, $\texttt{Range}(S)$ Assumption

$\Sigma; \cdot; P; S \vdash \ell_L{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l$ Subcase Premise

$\texttt{mtype}([\mathbf{t}.O \leftarrow R/\mathbf{t}]L\mathbf{t}.O \leftarrow R, m) = \forall \rho_1. \cdots \forall \rho_n L\mathbf{t}'.O' \leftarrow R' \xrightarrow{P'} \tau$ Subcase Premise

$L\mathbf{t}.O \leftarrow R \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}'.O' \leftarrow R'$ Subcase Premise

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P' \subseteq P$ Subcase Premise

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}.O \leftarrow R \leq$

$\quad [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}'.O' \leftarrow R'$ Region Subtyping

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}.O \leftarrow R \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}'.O' \leftarrow R'$ Definition of Substitution

$\Sigma \geq_\ell \Sigma$ S-Refl

$\Sigma; \cdot; P; S \vdash \ell_L{:}L\mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O \leftarrow R/\mathbf{t}]O \leftarrow R \Longrightarrow l$ Definition of Substitution

$\Sigma; \cdot; P; S \vdash \Lambda\rho_1. \cdots \Lambda\rho_n.\varsigma x{:}\tau_0.e{:}\forall \rho_1. \cdots \forall \rho_n L\mathbf{t}'.O'' \leftarrow R'' \xrightarrow{P'} \tau' \Longrightarrow l_e$ `mbody` Type

$\forall \rho_1. \cdots \forall \rho_n L\mathbf{t}'.O'' \leftarrow R'' \xrightarrow{P'} \tau' \leq \forall \rho_1. \cdots \forall \rho_n L\mathbf{t}'.O' \leftarrow R' \xrightarrow{P'} \tau$ `mbody` Type

$\Sigma; \cdot; P; S \vdash \Lambda\rho_1. \cdots \Lambda\rho_n.\varsigma x{:}\tau_0.e{:}\forall \rho_1. \cdots \forall \rho_n L\mathbf{t}'.O'' \leftarrow R'' \xrightarrow{P'} \tau' \Longrightarrow \{\}$ Case Analysis on Typing Rules

$\tau' \leq \tau$ Subtyping rules

$L\mathbf{t}'.O' \leftarrow R' \leq L\mathbf{t}'.O'' \leftarrow R''$ Subtyping Rules

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}'.O' \leftarrow R' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}'.O'' \leftarrow R''$ Region Subtyping

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}.O \leftarrow R \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}'.O'' \leftarrow R''$ Subtyping Rules

$L\mathbf{t}.O \leftarrow R \leq L\mathbf{t}'.O'' \leftarrow R''$ Region Subtyping

$\Sigma; x{:}L\mathbf{t}'.O'' \leftarrow R''; P'; S \vdash e{:}\tau' \Longrightarrow \{\}$ Inversion

$\texttt{eregions}(e) \subseteq P'$ Region Expression

$\texttt{eregions}([\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\ell_L/x]e) \subseteq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P'$ Region and Substitution

$\texttt{eregions}([\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\ell_L/x]e) \subseteq P$ Set Theory

$\Sigma; x{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]L\mathbf{t}'.O'' \leftarrow R''; [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P'; S \vdash$

$\quad [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow \{\}$ Region Substitution

$\rho_1, \cdots, \rho_n \notin P$ Inversion

$\rho_1, \cdots, \rho_n \notin L\mathbf{t}'.O'' \leftarrow R''$ Region Type

$\Sigma; x{:}L\mathbf{t}'.O'' \leftarrow R''; [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P'; S \vdash$
$\quad [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow \{\}$ — Definition of Substitution
$\Sigma; \cdot; P; S \vdash [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow \{\}$ — Region Weakening
$\Sigma; \cdot; P; S \vdash [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\ell_L/x]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow l'$ — Substitution
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'$ — Substitution
$l' \subseteq l_e$ — Substitution
No duplicates in l' — Set Theory
No duplicates in $l', l_s, \texttt{Range}(S)$ — Set Theory
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau$ — Region Subtyping
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau$ — Subtyping Rules

Case: E-LININV

$\mu(\ell) = loc \leftarrow \langle \cdots, m = \Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma x{:}\tau_0.e, \cdots \rangle$ — Case Premise
$\mu' = [\ell \mapsto \langle \cdots \rangle]\mu$ — Case Premise
$\Sigma; S \vdash \mu \ \ \texttt{ok} \Longrightarrow l_s$ — Assumption
$\texttt{eregions}(\ell_L.m[\varrho_1, \cdots, \varrho_n]) \subseteq P$ — Assumption
$\texttt{eregions}(\ell_L) \subseteq P$ — Definition of $\texttt{eregions}(e)$
subcase T-LININV
$\Sigma; \cdot; P; S \vdash \ell_L.m[\varrho_1, \cdots, \varrho_n]{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau \Longrightarrow l_e$ — Assumption
No duplicates in l_e,$l_s, \texttt{Range}(S)$ — Assumption
$\Sigma; \cdot; P; S \vdash \ell_L{:}¡\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$ — Subcase Premise
$\Sigma; \cdot; \cdot; S \vdash \ell_L{:}¡\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow \{\ell\}$ — Case Analysis on Typing Rules

$\texttt{lmtype}(¡\text{obj } \mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}].O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.¡\text{obj } \mathbf{t}'.O' \leftarrow R' \overset{P'}{\multimap} \tau$ — Subcase Premise
$[¡]\text{obj } \mathbf{t}.O_u \leftarrow R_u = [¡]\text{obj } \mathbf{t}.[\mathbf{t}.O \leftarrow R/\tau]O \leftarrow R$ — Subcase Premise
$\tau_f = [¡]\text{obj } \mathbf{t}.[\mathbf{t}/\tau_f]O_u \leftarrow [/m{:}\tau_1]R_u$ — Subcase Premise
$\tau_f \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]¡\text{obj } \mathbf{t}'.O' \leftarrow R'$ — Subcase Premise
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P' \subseteq P$ — Subcase Premise
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau_f \leq$
$\quad [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]¡\text{obj } \mathbf{t}'.O' \leftarrow R'$ — Region Subtyping
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau_f \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]¡\text{obj } \mathbf{t}'.O' \leftarrow R'$ — Definition of Substitution
$\Sigma(\ell) = ¡\text{obj } \mathbf{t}.O_s \leftarrow R_s$ — Inversion
$¡\text{obj } \mathbf{t}.O_s \leftarrow R_s \leq ¡\text{obj } \mathbf{t}.O \leftarrow R$ — Inversion
$\texttt{tregions}(¡\text{obj } \mathbf{t}.O \leftarrow R) \subseteq P$ — Inversion
$\tau_s = [¡]\text{obj } \mathbf{t}.[\mathbf{t}/\tau_s]([\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}]O_s) \leftarrow [/m{:}\tau_1][\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}]R_s$ — Definition
$\tau_s \leq \tau_f$ — Folding Subtyping
$[\ell \mapsto \tau_s]\Sigma \geq_\ell \Sigma$ — S-LOBJ

$\texttt{mtype}(¡\text{obj } \mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}].O \leftarrow R, m) = \forall\rho_1.\cdots\forall\rho_n.¡\text{obj } \mathbf{t}'.O' \leftarrow R' \overset{P'}{\multimap} \tau$ — T-METHT$_1$
$\Sigma; \cdot; \cdot; S \vdash \ell_L{:}¡\text{obj } \mathbf{t}.[\mathbf{t}/\mathbf{t}.O \leftarrow R][\mathbf{t}.O \leftarrow R/\mathbf{t}]O \leftarrow R \Longrightarrow \{\ell\}$ — Definition of Substitution

$\Sigma; \cdot; P; S \vdash \Lambda\rho_1.\cdots\Lambda\rho_n.¡\varsigma x{:}\tau_0.e{:}\forall\rho_1.\cdots\forall\rho_n¡\text{obj } \mathbf{t}'.O'' \leftarrow R'' \overset{P'}{\multimap} \tau' \Longrightarrow l$ — `mbody` Type
$\forall\rho_1.\cdots\forall\rho_n¡\text{obj } \mathbf{t}'.O'' \leftarrow R'' \overset{P'}{\multimap} \tau' \leq \forall\rho_1.\cdots\forall\rho_n¡\text{obj } \mathbf{t}'.O' \leftarrow R' \overset{P'}{\multimap} \tau$ — `mbody` Type
$\tau' \leq \tau$ — Subtyping rules
$¡\text{obj } \mathbf{t}'.O' \leftarrow R' \leq ¡\text{obj } \mathbf{t}'.O'' \leftarrow R''$ — Subtyping Rules
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]¡\text{obj } \mathbf{t}'.O' \leftarrow R' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]¡\text{obj } \mathbf{t}'.O'' \leftarrow R''$ — Region Subtyping
$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau_f \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]¡\text{obj } \mathbf{t}'.O'' \leftarrow R''$ — Subtyping Rules
$\tau_f \leq ¡\text{obj } \mathbf{t}'.O'' \leftarrow R''$ — Region Subtyping
$\Sigma; x{:}¡\text{obj } \mathbf{t}'.O'' \leftarrow R''; P'; S \vdash e{:}\tau' \Longrightarrow l$ — Inversion
$\texttt{eregions}(e) \subseteq P'$ — Region Expression
$\texttt{eregions}([\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\ell_L/x]e) \subseteq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P'$ — Region and Substitution
$\texttt{eregions}([\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\ell_L/x]e) \subseteq P$ — Set Theory
$\texttt{tregions}([¡]\text{obj } \mathbf{t}.O_u \leftarrow R_u) \subseteq P$ — Folding Region
$\texttt{tregions}([¡]\text{obj } \mathbf{t}.O_u \leftarrow [/m : \tau_1]R_u) \subseteq P$ — Definition of $\texttt{tregions}(\tau)$

$\texttt{tregions}(\tau_f) \subseteq P$ — Folding Region

$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc \leftarrow \langle \cdots, m = \Lambda\rho_1. \cdots \Lambda\rho_n \text{¡}\varsigma x{:}\tau_0.e, \cdots \rangle:$
$\quad \text{¡obj } \mathbf{t}.O_s \leftarrow R_s \Longrightarrow l_{loc}, l_1, \cdots, l, \cdots, l_n$ — Inversion on T-STORE

$\forall i \in 1..n | P' \in \texttt{Dom}(S). \Sigma; \cdot; \texttt{Dom}(S); S \vdash \sigma_i{:}\forall\rho_1. \cdots \forall\rho_n\tau_{o_i} \xrightarrow{P'} / \overset{P'}{\multimap} \tau'_{o_i} \Longrightarrow l_i$ — Inversion on T-ODESCR

$P \subseteq \texttt{Dom}(S)$ — Region Contexts

$\Sigma; \cdot; \texttt{Dom}(S); S \vdash \Lambda\rho_1. \cdots \Lambda\rho_n \text{¡}\varsigma x{:}\tau_0.e{:}\forall\rho_1. \cdots \forall\rho_n\tau_{o_i} \xrightarrow{P'} / \overset{P'}{\multimap} \tau'_{o_i} \Longrightarrow l_i$ — Instantion

$l_i = l$ — List Equality

$[\ell \mapsto \tau_s]\Sigma; \cdot; P; S \vdash \ell_L{:}\tau_f \Longrightarrow \ell$ — T-LINLOC

$\Sigma; x{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\text{¡obj } \mathbf{t}'.O'' \leftarrow R''; [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P'; S \vdash$
$\quad [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow l$ — Region Substitution

$[\ell \mapsto \tau_s]\Sigma; x{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\text{¡obj } \mathbf{t}'.O'' \leftarrow R''; [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P'; S \vdash$
$\quad [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow l$ — Linear Location Change

$\rho_1, \cdots, \rho_n \notin P$ — Inversion

$\rho_1, \cdots, \rho_n \notin \text{¡obj } \mathbf{t}'.O'' \leftarrow R''$ — Region Type

$[\ell \mapsto \tau_s]\Sigma; x{:}\text{¡obj } \mathbf{t}'.O'' \leftarrow R''; [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]P'; S \vdash$
$\quad [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow l$ — Definition of Substitution

$[\ell \mapsto \tau_s]\Sigma; \cdot; P; S \vdash [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow l_e$ — Region Weakening

$[\ell \mapsto \tau_s]\Sigma; \cdot; P; S \vdash [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n][\ell_L/x]e{:}[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \Longrightarrow l'$ — Substitution

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'$ — Substitution

$l' \subseteq l_e, l$ — Substitution

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau$ — Region Subtyping

$[\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau'' \leq [\varrho_1, \cdots, \varrho_n/\rho_1, \cdots, \rho_n]\tau$ — Subtyping Rules

$\Sigma; \cdot; \texttt{Dom}(S); S \vdash \Lambda\rho_1. \cdots \Lambda\rho_n \text{¡}\varsigma x{:}\tau_0.e{:}\forall\rho_1. \cdots \forall\rho_n\tau_{o_i} \xrightarrow{P'} / \overset{P'}{\multimap} \tau'_{o_i} \Longrightarrow l$ — Equality

$[\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}]R_s = m_1{:}\tau_{o1}, \cdots, m{:}\tau_o, \cdots, m_n{:}\tau_{on}$ — Inversion on T-ODESCR

$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc{:}O \Longrightarrow l_{loc}$ — Inversion on T-ODESCR

$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc \leftarrow [m = \sigma/]\langle \cdots \rangle{:}\tau_s \Longrightarrow l_{loc}, l_1, \cdots, l_n$ — T-ODESCR

$\ell \notin l_s$ — No Duplicates in $l_s, l_e, \texttt{Range}(S)$

$[\ell \mapsto \tau_s]\Sigma; S \vdash \langle \cdots \rangle \ \texttt{ok} \Longrightarrow l_s - l$ — Store Change

No duplicates in $l', l_s - l, \texttt{Range}(S)$ — Set Theory

Case: C-UPD

$\mu, e \longrightarrow, \mu' e'$ — Case Premise

$\Sigma; S \vdash \mu \ \texttt{ok} \Longrightarrow l_s$ — Assumption

$\texttt{eregions}(e \leftarrow\!\!+ m = \sigma) \subseteq P$ — Assumption

$\texttt{eregions}(e) \subseteq P$ — Definition of $\texttt{eregions}(e)$

subcase T-UPD

$\Sigma; \cdot; P; S \vdash e \leftarrow\!\!+ m = \sigma{:}\tau'' \Longrightarrow l_e, l'_e$ — Assumption

No duplicates in $l_e, l'_e, l_s, \texttt{Range}(S)$ — Assumption

$\Sigma; \cdot; P; S \vdash \sigma{:}\tau \Longrightarrow l_e$ — Subcase Premise

$\Sigma; \cdot; P; S \vdash e{:}\text{¡obj } \mathbf{t}.O \leftarrow R \Longrightarrow l'_e$ — Subcase Premise

$\texttt{lmtype}(\text{¡obj } \mathbf{t}.O \leftarrow R, m) = \tau'$ — Subcase Premise

$\text{¡obj } \mathbf{t}.R' \leftarrow O' = [\mathbf{t}.O \leftarrow R/\mathbf{t}]\text{¡obj } \mathbf{t}.R' \leftarrow O'$ — Subcase Premise

$\tau'' = [\mathbf{t}/tau'']\text{¡obj } \mathbf{t}.R' \leftarrow O'$ — Subcase Premise

$\Sigma' \geq_\ell \Sigma$ — IH

$\Sigma'; S' \vdash \mu' \ \texttt{ok} \Longrightarrow l'_s$ — IH

$\Sigma'; \cdot; P; S' \vdash e'{:}\text{¡obj } \mathbf{t}.O_s \leftarrow R_s \Longrightarrow l''_e$ — IH

$L\mathbf{t}.O_s \leftarrow R_s \leq L\mathbf{t}.O \leftarrow R$ — IH

$\texttt{eregions}(e') \subseteq P$ — IH

$\texttt{eregions}(e \leftarrow\!\!+ m = \sigma) \subseteq P$ — Definition of $\texttt{eregions}(e)$

$\texttt{mtype}(O_s \leftarrow R_s, m) = \tau'_s$ — $\texttt{lmtype}$ Subtyping

$\tau'_s \leq \tau'$ — `lmtype` Subtyping
$R' \leq [m{:}\tau'_s/m{:}\tau]R_s$ — T-SUBROW$_2$
$L\mathbf{t}.O \leftarrow R' \leq L\mathbf{t}.O_s \leftarrow [m{:}\tau'_s/m{:}\tau]R_s$ — T-SUBLOC
No duplicates in $l_e, l''_e, l'_s, \texttt{Range}(S')$ — Subterm Location
¡obj $\mathbf{t}.R'_s \leftarrow O'_s = [\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}]$¡obj $\mathbf{t}.R'_s \leftarrow O'_s$ — Definition
¡obj $\mathbf{t}.R'_s \leftarrow O'_s \leq$ ¡obj $\mathbf{t}.R' \leftarrow O'$ — Folding Subtyping
$\tau''_s = [\mathbf{t}/tau'']$¡obj $\mathbf{t}.R' \leftarrow O'$ — Definition
$\tau''_s \leq \tau''$ — Folding Subtypingand Subtyping Rules
$\Sigma'; \cdot; P; S \vdash e' \leftarrow\!\!+ m = \sigma{:}\tau''_s \Longrightarrow l_e, l''_e$ — T-UPD
subcase T-ADD
$\Sigma; \cdot; P; S \vdash e \leftarrow\!\!+ m = \sigma{:}\tau' \Longrightarrow l_e, l'_e$ — Assumption
No duplicates in $l_e, l'_e, l_s, \texttt{Range}(S)$ — Assumption
$\Sigma; \cdot; P; S \vdash \sigma{:}\tau \Longrightarrow l_e$ — Subcase Premise
$\Sigma; \cdot; P; S \vdash e$:¡obj $\mathbf{t}.O \leftarrow R \Longrightarrow l'_e$ — Subcase Premise
$\texttt{lmtype}($¡obj $\mathbf{t}.O \leftarrow R, m) \neq$ — Subcase Premise
¡obj $\mathbf{t}.O' \leftarrow R' =$ ¡obj $\mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}](O \leftarrow R)$ — Subcase Premise
$\tau' =$ ¡obj $\mathbf{t}.[\mathbf{t}/\tau'](O' \leftarrow R', m{:}\tau)$ — Subcase Premise
$\Sigma' \geq_\ell \Sigma$ — IH
$\Sigma'; S' \vdash \mu'$ ok $\Longrightarrow l'_s$ — IH
$\Sigma'; \cdot; P; S' \vdash e'$:¡obj $\mathbf{t}.O_s \leftarrow R_s \Longrightarrow l''_e$ — IH
$\Sigma'; \cdot; P; S \vdash \sigma{:}\tau \Longrightarrow l_e$ — Store Weakening
$\texttt{eregions}(e') \subseteq P$ — IH
$\texttt{eregions}(e \leftarrow\!\!+ m = \sigma) \subseteq P$ — Definition of $\texttt{eregions}(e)$
$\texttt{mtype}(O_s \leftarrow R_s, m) \neq$ — `lmtype` Subtyping
$R_s, m{:}\tau \leq R, m{:}\tau$ — T-SUBROW$_2$
$L\mathbf{t}.O \leftarrow R, m{:}\tau \leq L\mathbf{t}.O_s \leftarrow R_s, m{:}\tau$ — T-SUBLOC
No duplicates in $l_e, l''_e, l'_s\texttt{Range}(S')$ — Subterm Location
¡obj $\mathbf{t}.O'_s \leftarrow R'_s =$ ¡obj $\mathbf{t}.[\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}](O_s \leftarrow R_s)$ — Definition
¡obj $\mathbf{t}.O'_s \leftarrow R'_s \leq$ ¡obj $\mathbf{t}.O' \leftarrow R'$ — Folding Subtyping
$\tau'_s =$ ¡obj $\mathbf{t}.[\mathbf{t}/\tau'_s](O'_s \leftarrow R'_s, m{:}\tau)$ — Subcase Premise
$\tau'_s \leq \tau'$ — Folding Subtypingand Subtyping Rules
$\Sigma'; \cdot; P; S \vdash e' \leftarrow\!\!+ m = \sigma$:¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau'_s \Longrightarrow l_e, l'_e$ — T-ADD

Case: E-UPD

$\Sigma; S \vdash \mu$ ok $\Longrightarrow l_s$ — Assumption
$\texttt{eregions}(\ell_L \leftarrow\!\!+ m = \sigma) \subseteq P$ — Assumption
$\texttt{eregions}(\ell_L) \subseteq P$ — Definition of $\texttt{eregions}()$
$\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma_a, \cdots, m_n = \sigma_n \rangle$ — Case Premise
$\mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma_b, \cdots, m_n = \sigma_n \rangle]\mu$ — Case Premise
subcase T-UPD
$\Sigma; \cdot; P; S \vdash \ell_L \leftarrow\!\!+ m = \sigma_b{:}\tau \Longrightarrow l_e, l'_e$ — Assumption
No duplicates in $l_e, l'_e, l_s, \texttt{Range}(S)$ — Assumption
$\Sigma; \cdot; P; S \vdash \sigma_b{:}\tau_b \Longrightarrow l_e$ — Subcase Premise
$\Sigma; \cdot; P; S \vdash \ell_L$:¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau_a \Longrightarrow l'_e$ — Subcase Premise
¡obj $\mathbf{t}.O_u \leftarrow R_u, m{:}\tau_{a_u} =$ ¡obj $\mathbf{t}.[\mathbf{t}.O \leftarrow R, m{:}\tau_a/\mathbf{t}]O \leftarrow R, m{:}\tau_a$ — Subcase Premise
$\tau =$ ¡obj $\mathbf{t}.[\mathbf{t}/\tau]O_u \leftarrow R_u, m{:}\tau_b$ — Subcase Premise
$\Sigma; \cdot; P; S \vdash \ell_L$:¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau_a \Longrightarrow \{\ell\}$ — Case Analysis on Typing Rules
$\texttt{tregions}($¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau_a) \subseteq P$ — Inversion
$\Sigma(\ell) =$ ¡obj $\mathbf{t}.O' \leftarrow R', m{:}\tau'_a$ — Inversion
$O' \leftarrow R', m{:}\tau'_a \leq O \leftarrow R, m{:}\tau_a$ — Inversion
$R', m{:}\tau'_a \leq R, m{:}\tau_a$ — Inversion on T-SUBROW
$O' \leq O$ — Inversion on T-SUBROW

$\texttt{tregions}(O \leftarrow R, m:\tau_a) \subseteq P$ Inversion
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc \leftarrow \langle m_1 = \sigma'_1, \cdots, m = \sigma_a, \cdots, m_n = \sigma_n \rangle$:
¡obj $\mathbf{t}.O' \leftarrow R' \Longrightarrow l, l_{oc}\, l_1, \cdots, l_a, \cdots, l_n$ Inversion on T-STORE
$[\mathbf{t}.O' \leftarrow R', m{:}\tau_a/\mathbf{t}]R' = m_1{:}\tau_1, \cdots, m_n{:}\tau_n$ Inversion on T-ODESCR
$\forall i \in 1..n | P \subseteq \texttt{Dom}(S).\Sigma; A; P; S \vdash \sigma_i{:}\forall \rho_1. \cdots \forall \rho_n.\tau_i \xrightarrow{P} / \multimap^{P} \tau'_i \Longrightarrow l_i$ Inversion on T-ODESCR
$P \subseteq \texttt{Dom}(S)$ Region Contexts
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash \sigma_b{:}\tau_b \Longrightarrow l_e$ Region Weakening
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc{:}O' \Longrightarrow l_{loc}$ Inversion on T-ODESCR
$\tau_s = $ ¡obj $\mathbf{t}.[\mathbf{t}/\tau_s]O' \leftarrow [\mathbf{t}.O' \leftarrow R_s, m{:}\tau_b/\mathbf{t}]R_s, m:\tau_b$ Definition
$[\ell_o \mapsto \tau']\Sigma \geq_\ell \Sigma$ S-LOBJ
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma_b, \cdots, m_n = \sigma_n \rangle$:
$\tau_s \Longrightarrow l_{loc}, l_1, \cdots, l_b, \cdots, l_n, l_e$ T-ODESCR
$[\ell \mapsto \tau_s]\Sigma; S \vdash [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m = \sigma_b, \cdots, m_n = \sigma_n, m = \sigma \rangle]\mu\ \ \texttt{ok} \Longrightarrow$
$l_s, l_e - l_a, l_b$ Store Change
$\texttt{tregions}(\tau_b) \subseteq P$ Region Type
$\texttt{tregions}($¡obj $\mathbf{t}.O \leftarrow R) \subseteq P$ Definition of $\texttt{tregions}(\tau)$
$\texttt{tregions}($¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau_b) \subseteq P$ Definition of $\texttt{tregions}(\tau)$
$\texttt{tregions}(\tau) \subseteq P$ Definition of $\texttt{tregions}(\tau)$
$\tau_b \leq \tau_b$ T-SUBREFL
$R, m{:}\ \tau_b \leq R', m{:}\tau_b$ T-SubRow$_2$
¡obj $\mathbf{t}.O' \leftarrow R', m{:}\tau_b \leq$ ¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau_b$ T-SUBLOC
$\tau_s \leq \tau$ Folding Subtyping
$[\ell \mapsto$ ¡obj $\mathbf{t}.O' \leftarrow R', m{:}\tau_b]\Sigma; \cdot; P; S \vdash \ell_L{:}$¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau_b \Longrightarrow l'_e$ T-LinLoc
No duplicates in $l'_e, l_s, l_e - l_a, l_b, \texttt{Range}(S)$ Lists
$\tau \leq \tau$ T-SUBREFL

Case: E-ADD

$\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle$ Case Premise
$\forall i. m \neq m_i$ Case Premise
$\mu' = [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n, m = \sigma \rangle]\mu$ Case Premise
$\Sigma; S \vdash \mu\ \ \texttt{ok} \Longrightarrow l_s$ Assumption
$\texttt{eregions}(\ell_L \leftarrow\!\!+ m = \sigma) \subseteq P$ Assumption
$\texttt{eregions}(\ell_L) \subseteq P$ Definition of $\texttt{eregions}()$
subcase T-ADD
$\Sigma; \cdot; P; S \vdash \ell_L \leftarrow\!\!+ m = \sigma{:}$¡obj $\mathbf{t}.O \leftarrow R, m{:}\tau_m \Longrightarrow l_e, l'_e$ Assumption
No duplicates in $l_e, l'_e, l_s, \texttt{Range}(S)$ Assumption
$P \subset \texttt{Dom}(S)$ Region Contexts
$\Sigma; \cdot; P; S \vdash \sigma{:}\tau_m \Longrightarrow l_e$ Subcase Premise
$\Sigma; \cdot; P; S \vdash \ell_L{:}$¡obj $\mathbf{t}.O \leftarrow R \Longrightarrow l'_e$ Subcase Premise
¡obj $\mathbf{t}.O_u \leftarrow R_u =$ ¡obj $\mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}]O \leftarrow R$ Subcase Premise
$\tau =$ ¡obj $\mathbf{t}.[\mathbf{t}/\tau]O_u \leftarrow R_u$ Subcase Premise
$\Sigma; \cdot; P; S \vdash \ell_L{:}$¡obj $\mathbf{t}.O \leftarrow R \Longrightarrow \ell$ Case Analysis on Typing Rules
$\texttt{lmtype}($¡obj $\mathbf{t}.O \leftarrow R, m) \neq$ Subcase Premise
$\texttt{tregions}($¡obj $\mathbf{t}.O \leftarrow R) \subseteq P$ Inversion
$\Sigma(\ell) =$ ¡obj $\mathbf{t}.O' \leftarrow R'$ Inversion
$O' \leftarrow R' \leq O \leftarrow R$ Inversion
$R' \leq R$ Inversion on T-SUBROW
$O' \leq O$ Inversion on T-SUBROW
$\texttt{tregions}(O \leftarrow R) \subseteq P$ Inversion
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots m_n = \sigma_n \rangle$:¡obj $\mathbf{t}.O' \leftarrow R' \Longrightarrow l, l_{oc}\, l_1, \cdots, l_n$ Inversion on T-STORE
$[\mathbf{t}.O' \leftarrow R'/\mathbf{t}]R' = m_1{:}\tau_1, \cdots, m_n{:}\tau_n$ Inversion on T-ODESCR

$\forall i \in 1..n | P' \subseteq \texttt{Dom}(S).\Sigma; A; P; S \vdash \sigma_i{:}\forall \rho_1.\cdots\forall\rho_n.\tau_i \xrightarrow{P'} / \overset{P'}{\multimap} \tau_i' \Longrightarrow l_i$ Inversion on T-ODESCR
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc{:}O' \Longrightarrow l_{loc}$ Inversion on T-ODESCR
$P \subseteq \texttt{Dom}(S)$ Region Contexts
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash \sigma{:}\tau_m \Longrightarrow l_e$ Region Weakening
$\tau_s = \text{¡obj } \mathbf{t}.[\mathbf{t}/\tau_s]O' \leftarrow [\mathbf{t}.O' \leftarrow R', m{:}\tau_m/\mathbf{t}]R', m : \tau_m$ Definition
$[\ell \mapsto \tau']\Sigma \geq_\ell \Sigma$ S-LOBJ
$\Sigma; \cdot; \texttt{Dom}(S); S \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots m_n = \sigma_n, m = \sigma\rangle{:}\tau_s \Longrightarrow l_{loc}, l_1, \cdots, l_n, l_e$ T-ODESCR
$\ell \notin l_s$ No duplicates in $l_e, l_e', l_s, \texttt{Range}(S)$
$[\ell \mapsto \tau_s]\Sigma; S \vdash [\ell \mapsto loc \leftarrow \langle m_1 = \sigma_1, \cdots m_n = \sigma_n, m = \sigma\rangle]\mu \ \texttt{ok} \Longrightarrow l_s, l_e$ Store Change
$\texttt{tregions}(\tau) \subseteq P$ Region Type
$\texttt{tregions}(\text{¡obj } \mathbf{t}.O \leftarrow R, m{:}\tau) \subseteq P$ Definition of $\texttt{tregions}(\tau)$
$\texttt{tregions}(\tau) \subseteq P$ Definition of $\texttt{tregions}(\tau)$
$\tau \leq \tau$ T-SUBREFL
$R, m{:}\ \tau_m \leq R', m{:}\tau_m$ T-SubRow$_2$
$\text{¡obj } \mathbf{t}.O' \leftarrow R', m{:}\tau \leq \text{¡obj } \mathbf{t}.O \leftarrow R, m{:}\tau$ T-SUBLOC
$\tau_s \leq \tau$ Folding Subtyping
$[\ell \mapsto \text{¡obj } \mathbf{t}.O' \leftarrow R', m{:}\tau_m]\Sigma; \cdot; P; S \vdash \ell_L{:}\text{¡obj } \mathbf{t}.O \leftarrow R, m{:}\tau_m \Longrightarrow l_e'$ T-LinLoc
No duplicates in $l_e', l_s, l_e, \texttt{Range}(S)$ Lists
$\tau \leq \tau$ T-SUBREFL

Case: C-LET!$_1$

$\mu, e_1 \longrightarrow, \mu' e_1'$ Case Premise
$\Sigma; S \vdash \mu \ \texttt{ok} \Longrightarrow l_s$ Assumption
$\texttt{eregions}(\texttt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}) \subseteq P$ Assumption
$\texttt{eregions}(e_1) \subseteq P$ Definition of $\texttt{eregions}(e)$
$\texttt{eregions}(e_2) \subseteq P$ Definition of $\texttt{eregions}(e)$
$\texttt{eregions}(e_3) \subseteq P$ Definition of $\texttt{eregions}(e)$
subcase T-LET!
$\Sigma; \cdot; P; \texttt{let!}\ (\rho)\, x_1 = e_1\, x_2 = e_2\ \texttt{in}\ S\ \texttt{end} e_3 \vdash \tau_3{:}l_1, l_2, l_3 \Longrightarrow$ Assumption
No duplicates in $l_1, l_2, l_3, l_s, \texttt{Range}(S)$ Assumption
$\Sigma; \cdot; P; S \vdash e_1{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l_1$ Subcase Premise
$\Sigma; x_1{:}P, \rho\mathbf{t}.O \leftarrow R; \rho; S \vdash e_2{:}\tau_2 \Longrightarrow l_2$ Subcase Premise
$\Sigma; x_1{:}L\rho.tO \leftarrow R, x_2{:}\tau_2; P, \rho; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$ Subcase Premise
$\rho \notin \tau_2$ Subcase Premise
$\rho \notin P$ Subcase Premise
$x_1 \notin \texttt{Dom}(\cdot)$ Subcase Premise
$x_2 \notin \texttt{Dom}(\cdot)$ Subcase Premise
$x_1 \neq x_2$ Subcase Premise
$\Sigma' \geq_\ell \Sigma$ IH
$\Sigma'; S' \vdash \mu' \ \texttt{ok} \Longrightarrow l_s'$ IH
$\texttt{eregions}(e_1') \subseteq P$ IH
$\Sigma'; x_1{:}P, \rho\mathbf{t}.O \leftarrow R; \rho; S \vdash e_2{:}\tau_2 \Longrightarrow l_2$ Store Weakening
$\Sigma'; x_1{:}L\rho.tO \leftarrow R, x_2{:}\tau_2; P, \rho; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$ Store Weakening
$\texttt{eregions}(\texttt{let!}\ (\rho)\, x_1 = e_1'\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}) \subseteq P$ Definition of $\texttt{eregions}(e)$
No duplicates in $l_1', l_2, l_3, l_s', \texttt{Range}(S')$ Subterm Location
$\Sigma'; \cdot; P; S' \vdash e_1'{:}L\mathbf{t}.O_s \leftarrow R_s \Longrightarrow l_1'$ IH
$L\mathbf{t}.O_s \leftarrow R_s \leq L\mathbf{t}.O \leftarrow R$ IH
$\Sigma; x_1{:}L\rho.StO_s \leftarrow R_s, x_2{:}\tau_2; P, \rho; \cdot \vdash e_3{:}\tau_{3_s} \Longrightarrow l_3$ Variable Subtyping
$\tau_{3_s} \leq \tau_3$ Variable Subtyping
$\Sigma; \cdot; P; S \vdash \texttt{let!}\ (\rho)\, x_1 = e_1'\, x_2 = e_2\ \texttt{in}\ e_3\ \texttt{end}{:}\tau_3 \Longrightarrow l_1', l_2, l_3$ T-LET!

Case: E-LET!$_1$

r `fresh`	Case Premise
$\Sigma; S \vdash \mu$ `ok` $\Longrightarrow l_s$	Assumption
`eregions(let!` $(\rho)\, x_1 = \ell_L\, x_2 = e_2$ `in` e_3 `end)` $\subseteq P$	Assumption
`eregions`$(v_1) \subseteq P$	Definition of `eregions`(e)
`eregions`$(e_2) \subseteq P$	Definition of `eregions`(e)
`eregions`$(e_3) \subseteq P$	Definition of `eregions`(e)
subcase T-LET!	
$\Sigma; \cdot; P; S \vdash$ `let!` $(\rho)\, x_1 = \ell_L\, x_2 = e_2$ `in` e_3 `end`$:\tau_3 \Longrightarrow l_1, l_2, l_3$	Assumption
No duplicates in $l_1, l_2, l_3, l_s,$ `Range`(S)	Assumption
$\Sigma; \cdot; P; S \vdash \ell_L{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l_1$	Subcase Premise
$\Sigma; x_1{:}\rho\mathbf{t}.O \leftarrow R; P, \rho; S \vdash e_2{:}\tau_2 \Longrightarrow l_2$	Subcase Premise
$\Sigma; x_1{:}L\mathbf{t}.O \leftarrow R, x_2{:}\tau_2; \cdot; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$	Subcase Premise
$x_1 \notin$ `Dom`$(\cdot)$	Subcase Premise
$x_2 \notin$ `Dom`$(\cdot)$	Subcase Premise
$x_1 \neq x_2$	Subcase Premise
$r \notin \tau_2$	r `fresh`
$\rho \notin \cdot$	r `fresh`
$\rho \subseteq \rho$	Definition of $\subseteq$
$L\mathbf{t}.O \leftarrow R \leq L\mathbf{t}.O \leftarrow R$	T-SUBREFL
$\Sigma; \cdot; P; S \vdash \ell_L{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l_1$	Region Weakening
$\Sigma; \cdot; P; S \vdash \ell_\rho{:}\rho\mathbf{t}.O \leftarrow R \Longrightarrow \{\}$	T-BORLOC
$\rho\mathbf{t}.O \leftarrow R \leq \rho\mathbf{t}.O \leftarrow R$	T-SUBREFL
$\Sigma \geq_\ell \Sigma$	S-REFL
$\Sigma; \cdot; P, [r/\rho]r; S \vdash [r/\rho][\ell_\rho/x]e_2{:}\tau_2' \Longrightarrow l'$	Region Substitution and Substitution
$\tau_2' \leq \tau_2$	Substitution
$l' \subseteq l_2$	Substitution
`eregions`$([r/\rho][\ell_r/x]e_2) \subseteq P$	Definition of `eregions`(e)
`eregions(let!` $(\rho)\, x_1 = \ell_L\, x_2 = [r/\rho][\ell_\rho/x]e_2$ `in` e_3 `end`$) \subseteq P$	Definition of `eregions`(e)
$\Sigma; x_1{:}L\mathbf{t}.O \leftarrow R, x_2{:}\tau_2'; \cdot; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$	Variable Subtyping
$\tau_3 \leq \tau_3'$	Variable Subtyping
$\Sigma; \cdot; P; S \vdash$ `let!` $(\rho)\, x_1 = \ell_L\, x_2 = [r/\rho][\ell_\rho/x]e_2$ `in` e_3 `end`$:\tau_3 \Longrightarrow l_1, l', l_3$	T-LET!$_1$
No duplicates in l_1, l', l_3	Set Theory

Case: C-LET!$_2$

$\mu, e_2 \longrightarrow, \mu' e_2'$	Case Premise
$\Sigma; S \vdash \mu$ `ok` $\Longrightarrow l_s$	Assumption
`eregions(let!` $(r)\, x_1 = v_1\, x_2 = e_2$ `in` e_3 `end`$) \subseteq P$	Assumption
`eregions`$(v_1) \subseteq P$	Definition of `eregions`(e)
`eregions`$(e_2) \subseteq P, r$	Definition of `eregions`(e)
`eregions`$(e_3) \subseteq P$	Definition of `eregions`(e)
subcase T-LET!	
$\Sigma; \cdot; P; S \vdash$ `let!` $(r)\, x_1 = v_1\, x_2 = e_2$ `in` e_3 `end`$:\tau_3 \Longrightarrow l_1, l_2, l_3$	Assumption
No duplicates in $l_1, l_2, l_3, l_s,$ `Range`(S)	Assumption
$\Sigma; \cdot; P; S \vdash v_1{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l_1$	Subcase Premise
$\Sigma; x_1{:}r\mathbf{t}.O \leftarrow R; P, r; S \vdash e_2{:}\tau_2 \Longrightarrow l_2$	Subcase Premise
$\Sigma; x_1{:}L\mathbf{r}.tO \leftarrow R, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$	Subcase Premise
$r \notin \tau_2$	Subcase Premise
$r \notin P$	Subcase Premise
$x_1 \notin$ `Dom`$(\cdot)$	Subcase Premise
$x_2 \notin$ `Dom`$(\cdot)$	Subcase Premise

$x_1 \neq x_2$	Subcase Premise
$\Sigma' \geq_\ell \Sigma$	IH
$\Sigma'; S' \vdash \mu'$ $\texttt{ok} \Longrightarrow l'_s$	IH
$\texttt{eregions}(e'_2) \subseteq P, r$	IH
$r \notin \texttt{eregions}(e'_2)$	Definiton of $\texttt{eregions}(e)$
$\texttt{eregions}(e'_2) \subseteq P$	IH
$\texttt{eregions}(\texttt{let!}\ (r)\, x_1 = v_1\, x_2 = e'_2\ \texttt{in}\ e_3\ \texttt{end}) \subseteq P$	Definition of $\texttt{eregions}(e)$
No duplicates in $l_1, l'_2, l_3, l'_s, \texttt{Range}(S')$	Subterm Location
$\Sigma; x_1{:}r\mathbf{t}.O \leftarrow R; P, r; S \vdash e'_2{:}\tau_{2_s} \Longrightarrow l'_2$	IH
$\tau_{2_s} \leq \tau_2$	IH
$\Sigma; x_1{:}L\rho.tO \leftarrow R, x_2{:}\tau_{2_s}; P; S' \vdash e_3{:}\tau_{3_s} \Longrightarrow l_3$	Variable Subtyping
$\tau_{3_s} \leq \tau_3$	Variable Subtyping
$\Sigma'; \cdot; P; S \vdash v_1{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l_1$	Store Weakening
$\Sigma'; x_1{:}L\mathbf{r}.tO \leftarrow R, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$	Store Weakening
$\Sigma; \cdot; P; S \vdash \texttt{let!}\ (r)\, x_1 = v_1\, x_2 = e'_2\ \texttt{in}\ e_3\ \texttt{end}{:}\tau_3 \Longrightarrow l_1, l'_2, l_3$	T-LET!

Case: E-LET!$_2$

$\Sigma; S \vdash \mu$ $\texttt{ok} \Longrightarrow l_s$	Assumption
$\texttt{eregions}(\texttt{let!}\ (r)\, x_1 = v_1\, x_2 = v_2\ \texttt{in}\ e_3\ \texttt{end}) \subseteq P$	Assumption
$\texttt{eregions}(v_1) \subseteq P$	Definition of $\texttt{eregions}(e)$
$\texttt{eregions}(v_2) \subseteq P, r$	Definition of $\texttt{eregions}(e)$
$\texttt{eregions}(e_3) \subseteq P$	Definition of $\texttt{eregions}(e)$
subcase T-LET!	
$\Sigma; \cdot; P; S \vdash \texttt{let!}\ (r)\, x_1 = v_1\, x_2 = v_2\ \texttt{in}\ e_3\ \texttt{end}{:}\tau_3 \Longrightarrow l_1, l_2, l_3$	Assumption
No duplicates in $l_1, l_2, l_3, l_s, \texttt{Range}(S)$	Assumption
$\Sigma; \cdot; P; S \vdash v_1{:}L\mathbf{t}.O \leftarrow R \Longrightarrow l_1$	Subcase Premise
$\Sigma; x_1{:}r\mathbf{t}.O \leftarrow R; P, r; S \vdash v_2{:}\tau_2 \Longrightarrow l_2$	Subcase Premise
$\Sigma; x_1{:}L\mathbf{r}.tO \leftarrow R, x_2{:}\tau_2; P; S \vdash e_3{:}\tau_3 \Longrightarrow l_3$	Subcase Premise
$r \notin \texttt{tregions}(\tau_2)$	Subcase Premise
$\texttt{eregions}(v_2) \subseteq P$	Region Contraction
$L\mathbf{t}.O \leftarrow R \leq L\mathbf{t}.O \leftarrow R$	T-SUBREFL
$\tau_s \leq \tau_s$	T-SUBREFL
$\texttt{eregions}([v_1, v_2/x_1, x_2]e_3) \subseteq P$	Region and Substitution
$\Sigma; \cdot; P; S \vdash [v_1, v_2/x_1, x_2]e_3{:}\tau'_3 \Longrightarrow l'$	Substitution
$\tau'_3 \leq \tau_3$	Substitution
$l' \subseteq l_1, l_2, l_3$	Substitution
No duplicates in l'	Set Theory

Case: C-DEL$_1$

$\mu, e_1 \longrightarrow, \mu' e'_1$	Case Premise
$\Sigma; S \vdash \mu$ $\texttt{ok} \Longrightarrow l_s$	Assumption
$\texttt{eregions}(e_1 \leftarrow e_2) \subseteq P$	Assumption
$\texttt{eregions}(e_1) \subseteq P$	Definition of $\texttt{eregions}(e)$
$\texttt{eregions}(e_2) \subseteq P$	Definition of $\texttt{eregions}(e)$
subcase T-DEL	
$\Sigma; \cdot; P; S \vdash e_1 \leftarrow e_2{:}\tau \Longrightarrow l_e, l'_e$	Assumption
No duplicates in $l_e, l'_e, l_s, \texttt{Range}(S)$	Assumption
No duplicates in $l'_e, l_s, \texttt{Range}(S)$	No duplicates in $l_e, l'_e, l_s, \texttt{Range}(S)$

$\Sigma; S; \cdot; \cdot \vdash e_2{:}\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Subcase Premise
$\Sigma; \cdot; P; S \vdash e_1{:}O'' \Longrightarrow l'_e$	Subcase Premise
¡obj $\mathbf{t}.O' \leftarrow R' = $ ¡obj $\mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}]O \leftarrow R$	Subcase Premise
$\tau = $ ¡obj $\mathbf{t}.[\mathbf{t}/\tau]O'' \leftarrow R$	Subcase Premise
$\Sigma' \geq_\ell \Sigma$	IH
$\Sigma'; S' \vdash \mu'$ ok $\Longrightarrow l'_s$	IH
No duplicates in $l_e, l''_e, l'_s, \texttt{Range}(S')$	Subterm Location
$\Sigma; \cdot; P; S' \vdash e'_1{:}O'_s \Longrightarrow l'_e$	IH
$O''_s \leq O''$	IH
$\texttt{eregions}(e'_1) \subseteq P$	IH
$\tau = $ ¡obj $\mathbf{t}.[\mathbf{t}/\tau]O''_s \leftarrow R$	Subcase Premise
$\tau_s \leq \tau$	Folding Subtyping
$\texttt{eregions}(e'_1 \leftarrow e_2) \subseteq P$	Definition of $\texttt{eregions}(e)$
$L\mathbf{t}.O'_s \leftarrow R \leq L\mathbf{t}.O'' \leftarrow R$	T-SUBLOC
$\Sigma'; S; \cdot; \cdot \vdash e_2{:}\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Store Weakening
$\Sigma; \cdot; P; S \vdash e'_1 \leftarrow e_2{:}\tau_s \Longrightarrow l_e, l''_e$	T-DEL

Case: C-DEL$_2$

$\mu, e_2 \longrightarrow, \mu' e'_2$	Case Premise
$\Sigma; S \vdash \mu$ ok $\Longrightarrow l_s$	Assumption
$\texttt{eregions}(v_1 \leftarrow e_2) \subseteq P$	Assumption
$\texttt{eregions}(v_1) \subseteq P$	Definition of $\texttt{eregions}(e)$
$\texttt{eregions}(e_2) \subseteq P$	Definition of $\texttt{eregions}(e)$
subcase T-DEL	
$\Sigma; \cdot; P; S \vdash v_1 \leftarrow e_2{:}\tau \Longrightarrow l_e, l'_e$	Assumption
No duplicates in $l_e, l'_e, l_s, \texttt{Range}(S)$	Assumption
$\Sigma; \cdot; P; S \vdash e_2{:}\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Subcase Premise
$\Sigma; \cdot; P; S \vdash v_1{:}O'' \Longrightarrow l'_e$	Subcase Premise
¡obj $\mathbf{t}.O' \leftarrow R' = $ ¡obj $\mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}]O \leftarrow R$	Subcase Premise
$\tau = $ ¡obj $\mathbf{t}.[\mathbf{t}/\tau]O'' \leftarrow R$	Subcase Premise
$\Sigma' \geq_\ell \Sigma$	IH
$\Sigma'; S' \vdash \mu'$ ok $\Longrightarrow l'_s$	IH
No duplicates in $l''_e, l'_e, l'_s, \texttt{Range}(S')$	IH
$\Sigma; \cdot; P; S' \vdash e'_2{:}\text{obj } \mathbf{t}.O_s \leftarrow R_s \Longrightarrow l''_e$	IH
obj $\mathbf{t}.O_s \leftarrow R_s \leq$ obj $\mathbf{t}.O \leftarrow R$	IH
$\texttt{eregions}(e'_2) \subseteq P$	IH
¡obj $\mathbf{t}.O'_s \leftarrow R'_s = $ ¡obj $\mathbf{t}.[\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}]O_s \leftarrow R_s$	Subcase Premise
¡obj $\mathbf{t}.O'_s \leftarrow R'_s \leq$ ¡obj $\mathbf{t}.O' \leftarrow R'$	Folding Subtyping
$\tau_s = $ ¡obj $\mathbf{t}.[\mathbf{t}/\tau]O''_s \leftarrow R_s$	Subcase Premise
$\tau_s \leq \tau$	Folding Subtypingand Subtyping Rules
$\texttt{eregions}(v_1 \leftarrow e'_2) \subseteq P$	Definition of $\texttt{eregions}(e)$
$\Sigma'; \cdot; P; S \vdash v_1{:}O'' \Longrightarrow l'_e$	Store Weakening
$\Sigma; \cdot; P; S \vdash v_1 \leftarrow e'_2{:}\tau_s \Longrightarrow l''_e, l'_e$	T-DEL

Case: E-DEL

$\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots m_n = \sigma_n \rangle$	Case Premise
$\mu' = [\ell \mapsto loc' \leftarrow \langle m_1 = \sigma_1, \cdots m_n = \sigma_n \rangle]\mu$	Case Premise

$\Sigma; S \vdash \mu \ \ \mathtt{ok} \Longrightarrow l_s$	Assumption
$\mathtt{eregions}(loc' \leftarrow \ell_L) \subseteq P$	Assumption
$\mathtt{eregions}(\ell_L) \subseteq P$	Definition of $\mathtt{eregions}(e)$
subcase T-DEL	
$\Sigma; \cdot; P; S \vdash loc' \leftarrow \ell_L{:}L\mathbf{t}.O'' \leftarrow R \Longrightarrow l_e, l'_e$	Assumption
No duplicates in $l_e, l'_e, l_s, \mathtt{Range}(S)$	Assumption
$\Sigma; S; \cdot; P \vdash \ell_L{:}¡\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Subcase Premise
$\Sigma; S; \cdot; P \vdash \ell_L{:}¡\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow \ell$	Case Analysis on Typing Rules
$\Sigma; \cdot; P; S \vdash loc'{:}O'' \Longrightarrow l'_e$	Subcase Premise
$¡\text{obj } \mathbf{t}.O' \leftarrow R' = ¡\text{obj } \mathbf{t}.[\mathbf{t}.O \leftarrow R/\mathbf{t}]O \leftarrow R$	Subcase Premise
$\tau = ¡\text{obj } \mathbf{t}.[\mathbf{t}/\tau]O'' \leftarrow R'$	Subcase Premise
$\mathtt{tregions}(O'') \subseteq P$	Inversion
$[\ell \mapsto \tau_s]\Sigma \geq_\ell \Sigma$	S-LOBJ
$\Sigma(\ell) = ¡\text{obj } \mathbf{t}.O_s \leftarrow R_s$	Inversion
$¡\text{obj } \mathbf{t}.O_s \leftarrow R_s \leq ¡\text{obj } \mathbf{t}.O'' \leftarrow R$	Inversion
$R_s \leq R$	Subtyping rules
$\mathtt{tregions}(R) \subseteq P$	Inversion
$\Sigma; \cdot; \mathtt{Dom}(S); S \vdash loc \leftarrow \langle m_1 = \sigma_1, \cdots m_n = \sigma_n\rangle{:}¡\text{obj } \mathbf{t}.O_s \leftarrow R_s \Longrightarrow l, _{loc}\, l_1, \cdots, l_n$	Inversion on T-STORE
$\forall i \in 1..n \vert P \subseteq \mathtt{Dom}(S).\Sigma; \cdot; \mathtt{Dom}(S); S \vdash \sigma_n{:}\forall\rho_1.\cdots\forall\rho_n.\tau_n \xrightarrow{P'} / \overset{P'}{\multimap} \tau'_n \Longrightarrow l_n$	Inversion on T-ODESCR
$[\mathbf{t}.R_s \leftarrow O_s/\mathbf{t}]R_s = m_1{:}\tau_1, \cdots, m_n{:}\tau_n$	Inversion on T-ODESCR
$\Sigma; \cdot; \mathtt{Dom}(S); S \vdash loc{:}O_s \Longrightarrow l_{loc}$	Inversion on T-ODESCR
$\Sigma; \cdot; \mathtt{Dom}(S); S \vdash loc'{:}O'' \Longrightarrow l'_e$	Region Weakening
$\tau_s = ¡\text{obj } \mathbf{t}.[\mathbf{t}/\mathbf{t}.O'' \leftarrow R_s]O'' \leftarrow R_s$	Definition
$\Sigma; \cdot; \mathtt{Dom}(S); S \vdash loc' \leftarrow \langle m_1 = \sigma_1, \cdots m_n = \sigma_n\rangle{:}\tau_s \Longrightarrow l'_e, l_1, \cdots, l_n$	T-ODESCR
$\ell \notin l_s$	No duplicates in $l_e, l'_e, l_s, \mathtt{Range}(S)$
$[\ell \mapsto \tau_s]\Sigma; S \vdash [\ell \mapsto \tau_s]\mu \ \ \mathtt{ok} \Longrightarrow l_s - l_{loc}, l'_e$	Store Change
$\mathtt{tregions}(\tau) \subseteq P$	Definition of $\mathtt{tregions}(\tau)$
$\tau_s \leq \tau$	Folding Subtyping
$[\ell \mapsto ¡\text{obj } \mathbf{t}.O'' \leftarrow R_s]\Sigma; \cdot; P; S \vdash \ell_L{:}\tau \Longrightarrow l_e$	T-LinLoc
No duplicates in $l_e, l_s - l_{loc}, l'_e, \mathtt{Range}(S)$	Lists
$\tau \leq \tau$	T-SUBREFL

Case: E-NEW

$\ell\ \mathtt{fresh}$	Case Premise
$\Sigma; S \vdash \mu \ \ \mathtt{ok} \Longrightarrow l_s$	Assumption
$\mathtt{eregions}(\langle\rangle) \subseteq P$	Assumption
subcase T-NEW	
$\Sigma; \cdot; P; S \vdash \langle\rangle{:}¡\text{obj } \mathbf{t}.\langle\rangle \leftarrow \cdot \Longrightarrow l_e$	Assumption
No duplicates in $l_s, \mathtt{Range}(S)$	Assumption
$[\ell \mapsto ¡\text{obj } \mathbf{t}.\langle\rangle \leftarrow \cdot]\Sigma; S \vdash [\ell \mapsto \langle\rangle \leftarrow \langle\rangle]\mu \ \ \mathtt{ok} \Longrightarrow l_s$	T-STORE
$\mathtt{tregions}(¡\text{obj } \mathbf{t}.\langle\rangle \leftarrow \cdot) \subseteq P$	Definition of $\mathtt{eregions}(\tau)$
$¡\text{obj } \mathbf{t}.\langle\rangle \leftarrow \cdot \leq ¡\text{obj } \mathbf{t}.\langle\rangle \leftarrow \cdot$	T-SUBREFL
$[\ell \mapsto ¡\text{obj } \mathbf{t}.\langle\rangle \leftarrow \cdot/]\Sigma; \cdot; P; S \vdash \ell{:}¡\text{obj } \mathbf{t}.\langle\rangle \leftarrow \cdot \Longrightarrow l_e$	T-LINLOC
No duplicates in $l_s, \mathtt{Range}(S), \ell$	$\ell\ \mathtt{fresh}$

Case: C-CHLIN

$\mu, e \longrightarrow, \mu' e'$	Case Premise

$\Sigma; S \vdash \mu \ \ \mathtt{ok} \Longrightarrow l_s$	Assumption
$\mathtt{eregions}(!e) \subseteq P$	Assumption
subcase T-CHLIN	
$\Sigma; \cdot; P; S \vdash !e{:}\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Assumption
No duplicates in $l_s, \mathtt{Range}(S)$	Assumption
$\Sigma; \cdot; P; S \vdash e{:}¡\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Subcase Premise
$\Sigma' \geq_\ell \Sigma$	IH
$\Sigma'; S' \vdash \mu' \ \ \mathtt{ok} \Longrightarrow l'_s$	IH
No duplicates in $l'_e, l'_s, \mathtt{Range}(S')$	IH
$\Sigma; \cdot; P; S' \vdash e'{:}¡\text{obj } \mathbf{t}.O_s \leftarrow R_s \Longrightarrow l'_e$	IH
¡obj $\mathbf{t}.O_s \leftarrow R_s \leq$ ¡obj $\mathbf{t}.O \leftarrow R$	IH
$\mathtt{eregions}(e') \subseteq P$	IH
$O_s \leq O$	Subtyping Rules
$R_s \leq R$	Subtyping Rules
$\Sigma; \cdot; P; S' \vdash !e'{:}\text{obj } \mathbf{t}.O_s \leftarrow R_s \Longrightarrow l'_e$	T-CHLIN
obj $\mathbf{t}.O_s \leftarrow R_s \leq$ obj $\mathbf{t}.O \leftarrow R$	Subtyping Rules
$\mathtt{eregions}(!e') \subseteq P$	Definition of $\mathtt{eregions}(e)$

Case: E-CHLIN

$\Sigma; S \vdash \mu \ \ \mathtt{ok} \Longrightarrow l_s$	Assumption
subcase T-CHLIN	
$\Sigma; \cdot; P; S \vdash !\ell_{¡\text{obj}}{:}\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Assumption
No duplicates in $l_e, l_s, \mathtt{Range}(S)$	Assumption
$\Sigma; \cdot; P; S \vdash \ell_{¡\text{obj}}{:}¡\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow l_e$	Subcase Premise
$\Sigma(\ell) =$ ¡obj $\mathbf{t}.O_s \leftarrow R_s$	Inversion
¡obj $\mathbf{t}.O_s \leftarrow R_s \leq$ ¡obj $\mathbf{t}.O \leftarrow R$	Inversion
$\mathtt{tregions}($¡obj $\mathbf{t}.O \leftarrow R) \subseteq P$	Inversion
$O_s \leq O$	Subtyping Rules
$R_s \leq R$	Subtyping Rules
obj $\mathbf{t}.O_s \leftarrow R_s \leq$ obj $\mathbf{t}.O \leftarrow R$	Subtyping Rules
$[\ell \mapsto$ obj $\mathbf{t}.O_s \leftarrow R_s]\Sigma(\ell) =$ obj $\mathbf{t}.O_s \leftarrow R_s$	Definition of Substution
$\mathtt{tregions}($obj $\mathbf{t}.O \leftarrow R) \subseteq P$	Definition of $\mathtt{tregions}(\tau)$
$[\ell \mapsto$ obj $\mathbf{t}.O_s \leftarrow R_s]\Sigma \geq_\ell \Sigma$	S-CHLIN
$[\ell \mapsto$ obj $\mathbf{t}.O_s \leftarrow R_s]\Sigma; \cdot; P; S \vdash \ell_{\text{obj}}{:}\text{obj } \mathbf{t}.O \leftarrow R \Longrightarrow \{\}$	T-NLINLOC
obj $\mathbf{t}.O \leftarrow R \leq$ obj $\mathbf{t}.O \leftarrow R$	T-SUBREFL
No duplicates in $l_s, \mathtt{Range}(S)$	Set Theory
$\mathtt{eregions}(\ell_{\text{obj}}) \subset P$	Definition of $\mathtt{eregions}(e)$
$\forall \ell \in \mathtt{Dom}(\mu).\Sigma; \cdot; \mathtt{Dom}(S); S \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_\ell$	Inversion on T-STORE
$\mathtt{Dom}(\mu) = \mathtt{Dom}(\Sigma)$	Inversion on T-STORE
$\Sigma; \cdot; \mathtt{Dom}(S); S \vdash \mu(\ell){:}\Sigma(\ell) \Longrightarrow l_\ell$	Instantiation
$\mu(\ell) = loc \leftarrow \langle m_1 = \sigma_1, \cdots, m_n = \sigma_n \rangle$	$\mathtt{Dom}(\mu) = \mathtt{Dom}(\Sigma)$
$\forall i \in 1..n.\Sigma; A; \mathtt{Dom}(S); S \vdash \sigma_i{:}\tau_i \xrightarrow{P} / \xrightarrow{P}\!\!\circ\ \tau'_i \Longrightarrow l_i$ if $\mathtt{Dom}(S) \subseteq \mathtt{Dom}(S)$	Inversion
$[\mathbf{t}.O_s \leftarrow R_s/\mathbf{t}]R_s = m_1{:}\tau_1, \cdots, m_n{:}\tau_n$	Inversion
$\Sigma; A; P; S \vdash loc{:}O_s \Longrightarrow l_{loc}$	Inversion
$[\ell \mapsto$ obj $\mathbf{t}.O_s \leftarrow R_s]\Sigma; \cdot; \mathtt{Dom}(S); S \vdash \mu(\ell){:}[\ell \mapsto$ obj $\mathbf{t}.O_s \leftarrow R_s]\Sigma(\ell) \Longrightarrow l_\ell$	T-ODESCR
$\Sigma; S \vdash \mu \ \ \mathtt{ok} \Longrightarrow l_s$	Store Change