A Type System for Borrowing Permissions

Karl Naden **Robert Bocchino**
Jonathan Aldrich **Kevin Bierhoff**

May 2012
CMU-CS-11-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work supported by the National Science Foundation under grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages," grant #CCF-0811592, "Practical Typestate Verification with Assume-Guarantee Reasoning," and grant #1019343 to the Computing Research Association for the CIFellows Project.

Keywords: programming language, permissions, borrowing

Abstract

In object-oriented programming, unique permissions to object references are useful for checking correctness properties such as consistency of typestate and noninterference of concurrency. To be usable, unique permissions must be *borrowed* — for example, one must be able to read a unique reference out of a field, use it for something, and put it back. While one can null out the field and later reassign it, this paradigm is ungainly and requires unnecessary writes, potentially hurting cache performance. Therefore, in practice borrowing must occur in the type system, without requiring memory updates. Previous systems support borrowing with external alias analysis and/or explicit programmer management of *fractional permissions*. While these approaches are powerful, they are also awkward and difficult for programmers to understand. We present an integrated language and type system with unique, immutable, and shared permissions, together with new *local permissions* that say that a reference may not be stored to the heap. Our system also includes *change permissions* such as `unique>>unique` and `unique>>none` that describe how permissions flow in and out of method formal parameters. Together, these features support common patterns of borrowing, including borrowing multiple local permissions from a unique reference and recovering the unique reference when the local permissions go out of scope, without any explicit management of fractions in the source language. All accounting of fractional permissions is done by the type system "under the hood." We present the syntax and static and dynamic semantics of a formal core language and prove soundness results. We also illustrate the utility and practicality of our design by using it to express several realistic examples.

1 Introduction

Permissions are annotations on pointer variables that specify how an object may be aliased, and which aliases may read or write to the object [Boyland et al., 2001]. For example, `unique` [Holt et al., 1988] indicates an unaliased object, `immutable` [Noble et al., 1998] indicates an object that can be aliased but cannot be mutated, and `shared` [Bierhoff and Aldrich, 2007] indicates an object that may be aliased and can be mutated. Permission systems have been proposed to address a diversity of software engineering concerns, including encapsulation [Noble et al., 1998], protocol checking [Fähndrich and DeLine, 2002, Bierhoff and Aldrich, 2007], safe concurrency [Boyapati et al., 2002, Boyland, 2003], security [Bokowski and Vitek, 1999], and memory management [Boyland et al., 2001, Grossman et al., 2002]. Recently, new programming languages have incorporated permissions [Wolff et al., 2011] or related affine types [Tov and Pucella, 2011] as fundamental parts of the type system.

In order to leverage permissions in practice, programmers must be able to manipulate them effectively. One form of manipulation is *permission splitting*: for example, converting a unique permission into multiple shared permissions, or alternatively into multiple immutable permissions. A shared (or immutable) permission can then be split further into more shared (respectively immutable) permissions. A second important form of manipulation is *borrowing*: extracting a permission from a variable field, using it temporarily, and then returning all or part of it to the source. For example, a method may require an immutable permission to the receiver; if we have a unique permission to an object, we'd like to call the method on that object, and provided the method does not allow an alias to the receiver to escape, we'd like to get our unique permission back at the end. Crucially, recovering the permission should not require reassigning the original variable (which may not even be assignable).

Borrowing was originally proposed by [Hogg, 1991]; however, good support for this feature has remained an open and difficult problem. Prior type-based systems [Hogg, 1991, Minsky, 1996, Aldrich et al., 2002] provided a borrowed annotation, but they did not support the immutable references that are an essential part of many recent systems [Boyland, 2003, Bierhoff and Aldrich, 2007, Wolff et al., 2011]. A number of systems supported borrowing via a program analysis [Boyland, 2001, Bierhoff et al., 2009], but the program analysis relies on shape analysis, which is fragile and notoriously difficult for programmers to understand. Boyland proposed fractional permissions [Boyland, 2003] to support splitting and recombining permissions, with borrowing as a special case, but its mathematical fraction abstraction is unnatural for programmers, to such an extent that the automated tools we know of once again hide the fractions behind an (inscrutable) analysis [Bierhoff and Aldrich, 2007, Heule et al., 2011].

A good borrowing facility should have a number of properties. It should support a natural programming style (e.g. avoid awkward constructs like replacing field write with a primitive swap operation [Harms and Weide, 1991], or a requirement to thread a reference explicitly from one call to the next by reassigning the reference each time [Tov and Pucella, 2011]). Reasoning abstractions should likewise be natural (not fractions [Boyland, 2003]). It should support borrowing from unique, immutable, and shared variables and fields. Rules should be local so the programmer can understand them and predict how they operate (vs. a non-local analysis [Boyland, 2001, Bierhoff and Aldrich, 2007] or constraint-based inference).

The contribution of this paper is the first borrowing approach that meets all of the above properties. We provide a type system for a Java-like language with permissions that are tracked through local, predictable rules. Our technical approach includes a number of innovations, including change permissions that show the incoming and outgoing permissions to a method parameter, local permissions that modify shared or immutable permissions to denote that they cannot escape, an expressive rule for handling borrowing across conditional branches, and a way to safely restore permissions to the variable or field from which they were borrowed. We formalize our type system, prove it sound, and demonstrate it via a series of examples which can be checked by our prototype implementation.[1]

The work described here was partially presented at POPL 2012 [Naden et al., 2012]. The differences between the published system and the one detailed here are given in section 3.6.

We describe the features of the language in the next section. Section 3 formalizes our type system and gives soundness results. We cover additional related work in section 4, and section 5 concludes.

2 Language Features

In this section we informally explain the features of our language; the next section gives a more formal treatment. Our language is based on Plaid [Wolff et al., 2011]. For our purposes, Plaid is similar to Java, except:

- Instead of classes, Plaid uses states. This is because Plaid has a first-class notion of *typestate*, which can be used to model object state and transitions between states. We don't use typestate in this work, but we adhere to the Plaid syntax.
- Plaid has a `match` construct that evaluates an argument and then uses its type to pick which of several cases to execute.
- Plaid has no `null` value in the surface syntax or programmer-visible semantics. Instead, every object field and variable is initialized to a non-null object.
- Plaid has a first-class notion of permissions. Supporting typestate is one important application of permissions, including the novel borrowing mechanisms discussed in this work.

In the rest of this section we first give an overview of the permissions in our language. Then we explain the mechanisms for creating aliases of variables and fields with consistent permissions. Then we explain the mechanism of change permissions, which allows modular checking of permission flow in and out of methods. Finally, we explain local permissions, which provide a way to split a unique permission into several permissions and later recombine them to unique, without explicit fractions.

[1] available at http://code.google.com/p/plaid-lang/

2.1 Access Permissions

Permissions are a well-known way of controlling aliasing in applications such as typestate [Bierhoff and Aldrich, 2007] and concurrency control [Boyland, 2003]. Our permission system is adapted from the *access permissions* system of [Bierhoff and Aldrich, 2007]. An access permission is a tag on an object reference that says how the reference may be used to access the fields of the object it refers to and how aliases of that reference may be created. In this work, we use the following permissions:

- A `unique` permission to object reference o says that this is the only usable copy of o: if any alias of o exists, then it has permission `none`. The reference may be used to read and write fields of the object O that it points to.
- A `none` permission says that the reference may not be used to read or write the object it points to.
- An `immutable` permission says that the reference o may be used only for reading, and not writing, the fields of the object O that it points to. Further, all other usable (non-`none`) references to O are guaranteed to be `immutable`, so no aliased reference can be used to write O either.
- A `shared` permission says that the reference o may be used for reading and writing, and there is no restriction on aliases of o (so `shared` is like an ordinary reference in Java).
- A `local immutable` or `local shared` permission is like an `immutable` (respectively `shared`) permission, except it can only be passed around in local variables, and can't be assigned to the heap. Local permissions are new with this work and are explained further in Section 2.4.

If any alias of a unique reference is created, then the `unique` permission must be consumed (but it can be transferred to the alias). A unique permission is therefore like a linear resource [Girard, 1987]. By contrast, `immutable` and `shared` references may be freely replicated.

While other access permissions are possible, these permissions suffice to illustrate the concepts in this paper. Further, these permissions can express a wide range of computation patterns. Other permissions could be added to the system without difficulty.

2.2 Aliasing Variables and Fields

In contrast to a language like Java with unrestricted aliasing, a language with access permissions must maintain careful control over how aliases are created. We now discuss how our language manages permissions for aliases of variables and object fields.

Aliasing Variables: In our language the following rules govern aliasing of local variables:

1. Our local variables are single-assignment (i.e., they are assigned to only in their declaration) and are declared with explicit permissions, so the needed permission is always available on the left-hand side of a variable assignment.

(a) Borrowing Unique from a Variable

```
1 unique x = new S1; // x:unique
2 {
3   unique y = x;    // x:none,y:unique
4 }                  // x:unique
```

(b) Borrowing Unique from One of Two Variables

```
1 unique x = new S1; // x:unique
2 unique y = new S2; // x:unique,y:unique
3 {
4   unique z = match(...) {
5     S1=>x; S2=>y;  // x:none,y:none,z:unique
6   }
7 }                  // x:unique,y:unique
```

(c) Taking Unique from a Variable

```
1 unique x = new S2; // x:unique
2 unique y = new S1; // x:unique,y:unique
3 {
4   unique z = x;    // x:none,y:unique,z:unique
5   y.f1 = z;        // x:none,y:unique,z:none
6 }                  // x:none,y:unique
```

Figure 1: Examples of variable aliasing.

2. In typing and assignment, we compute all variables and fields such that the value returned by the right-hand side of the assignment may be obtained by reading the variable or field. Because we can't statically resolve which `match` case will be taken, there may be more than one.

3. We maintain a typing environment, which we call a *context*, with the permission associated with each variable. For each possible source variable, we make sure there is enough permission in the context to extract the needed permission. We use permission splitting rules, given formally in the next section, to compute the permission remaining in the source variables after the extraction, and we update the context accordingly.

4. At the end of the scope of the assignee variable, we restore whatever permission is left in that variable to the source variables. For this operation we use permission joining rules, which are the reverse of the splitting rules.

Figure 1 shows examples, where the definitions of states `S1` and `S2` are as follows:

```
state S1 { unique S1 f1 = new S2; }
state S2 case of S1 { unique S1 f2 = new S3; }
state S3 case of S2 {}
```

Here `case of` is like `extends` in Java and denotes subclassing. In Figure 1(a), variable `x` is declared `unique` and initialized with a fresh object in line 1. In line 3, variable `y` is created and takes the unique permission out of `x`, leaving `none` in `x`. When `y` goes out of scope in line 4, its `unique` permission flows back into `x`. Figure 1(b) is similar, except that in line 5, we don't know which variable (`x` or `y`) will be read from at runtime, so we take permissions out of both, record both as source variables, and restore permissions to both at the end of `z`'s scope in line 7. The `match` statement in lines 4–5 says to evaluate the selector expression (represented as ellipses here) to an object reference *o*, then evaluate the whole expression to `x` if the type of *o* matches `S1`, to `y` if the type of *o* matches `S2`, and to halt if there is no match. While this example is simple, in general typing `match` in our language is subtle and requires *merging* the contexts generated by the different match cases; the details are given in Section 3.2.4. Figure 1(c) shows an example where the permission read out of `x` into `z` is stored into the heap, so it can't be returned to `x` at the end of `z`'s scope.

(a) Borrowing Unique from a Field

```
1 unique x = new S1; // x:unique
2 {
3   unique y = x.f1; // x:(unique,f1:none),y:unique
4 }                  // x:unique
```

(b) Borrowing Unique from a Variable or Field

```
1 unique x = new S1;    // x:unique
2 unique y = new S1;    // x:unique,y:unique
3 {
4   unique z = match(...) {
5     S1=>x; S2=>y.f1;
6     // x:none,y:(unique,f1:none),z:unique
7   }
8 }                     // x:unique,y:unique
```

(c) Taking Unique from a Field

```
1 unique x = new S1; // x:unique
2 unique y = new S1; // x:unique,y:unique
3 y.f1 = x.f1;       // x:(unique,f1:none),y:unique
```

Figure 2: Examples of field aliasing.

Aliasing Fields: The rules for aliasing of fields are similar to those for aliasing of local variables, with two exceptions. First, pulling a permission out of a field can cause the residual permission to violate the statically declared field permission. For example, pulling `unique` out of a field declared `unique` causes the field to have permission `none`. In this case, we say the field is *unpacked* [DeLine and Fähndrich, 2004]. If an object has any unpacked fields, then we say the object is unpacked. We must account for the actual permissions when accessing the fields of an unpacked object. Second, since fields can be assigned to, the reference in the field at the point of permission restore may not be the same reference that the permission came from. In this case, we must be careful not to restore permissions to the wrong reference, which would violate soundness.
Unpacking fields: To address the first issue, we store the current permission of each unpacked field of each object in the context. Figure 2 shows examples, where `S1` and `S2` are defined as before. In Figure 2(a), line 3 shows the context after taking a unique permission out of `x.f1`. The notation `x:(unique,f1:none)` means that `x` has unique permission and points to an unpacked object with `none` permission for field `f1`. Figure 2(b) shows how to use the same mechanism from Figure 1 to pull a permission out of either a variable or a field. Figure 2(c) shows an example of taking a unique permission from a field and assigning it to another field.

To ensure soundness, we place three restrictions on field unpacking. First, an object can be unpacked via a variable with unique permission, but not immutable or shared permission. This is so outstanding aliases to the object don't become inconsistent.[2] Our language also allows taking an immutable permission out of a unique field $v.f$ given immutable permission to v. In this case our type system does not report v as unpacked in the context; this is sound because once an object is seen with immutable permission it can never become unique again.

Second, once an object o stored in variable v is unpacked, permission to o may not be assigned to another variable or the heap until v is packed again; otherwise we would not be able to track the unpacked state of o through v's information in the context. For example, the following code is not allowed, because `x` is unpacked when it is assigned to `z`:

```
1 unique x = new S1; // x:unique
2 unique y = x.f1;   // x:(unique,f1:none),y:unique
3 unique z = x;      // Disallowed because x is not packed
```

Third, a variable v must be packed at the point where it goes out of scope; in particular method formal parameters must be packed at the end of a method body. That is because the permission stored in v could be flowing back to a different (packed) variable that gave its permission to v when v was declared.

In practice, the programmer can comply with the latter two requriements by carefully managing the scopes of variables that read permissions from unique fields and/or writing fresh objects into the fields to pack the fields at appropriate points. While in some cases these writes may not be strictly necessary, the requirements do not seem to be onerous in the examples we have studied. These restrictions could be relaxed at the cost of additional language complexity (for example, stating in method signatures which fields of an incoming parameter must be packed).

[2]Local permissions, discussed in Section 2.4, provide an additional way to unpack objects.

Consistent permission restore: As an example of the second issue identified above (erroneous permission restore), consider the following code:

```
1 unique x = new S2; // x:unique
2 {
3   unique y = x.f1; // x:(unique,f1:none),y:unique
4   x.f1 = new S2;   // x:unique,y:unique
5   x.f2 = x.f1;     // x:(unique,f1:none),y:unique
6 }                  // x:(unique,f1:none)
```

Restoring a permission to `x.f1` at the end of `y`'s scope in line 6 would create an alias between `x.f1` and `x.f2`, both with unique permission. The problem is that the reference in field `x.f1` in line 6 is not the same reference to which permission was taken in line 3, so restoring to it would be wrong.

To prevent this from happening, the static typing rules maintain an identifier that is updated every time a field goes from packed to unpacked. The permission restore occurs only if the identifier at the point of the field access matches the identifier at the point of restoration. For example, in the code shown above, at line 3 where `x.f1` is unpacked and its permission read into `y`, an identifier i is associated with `x.f1` and with `y`. Then in line 5, when `x.f1` is unpacked again, `x.f1` gets a fresh identifier i'. In line 6, when `y` goes out of scope, its identifier i does not match the identifier i' of the source location `x.f1`, so permission is not restored from `y` into the location. Thus the identifier mechanism approximates object identity at runtime; the approximation is conservative because each assignment is assumed to assign a different object reference, even if the same one is actually assigned twice.

2.3 Modular Checking with Change Permissions

To support modular checking of permission flow across method scopes, we introduce a language feature called a *change permission*. Change permissions are inspired by, and similar to, change types in Plaid [Wolff et al., 2011]. However, whereas a change type in Plaid says that an object may transition from one state to another (to support typestate), in our language change permissions specify only that a reference permission changes from a stronger to a weaker permission; the object type always persists. Further, while previous systems can distinguish borrowed and consumed permissions at method boundaries [Boyland, 2001, DeLine and Fähndrich, 2001], change permissions are more flexible, because they can record a change to any weaker permission (not just `none`).

Syntactically, a change permission looks like $\pi{>>}\pi'$, where π and π' are permissions. Change permissions appear on method formal parameters, including the implicit parameter `this`. For example, a formal parameter can be declared $\pi{>>}\pi'\ s\ x$, where s is a state name. This declaration says the caller must ensure that on entry to the method permission π is available for the reference o stored in x, while the callee must ensure that on exit from the method permission π' is available for o. The bare permission π can also function as a change permission; it is shorthand for $\pi{>>}\pi$.

Change permissions naturally support both borrowing and non-borrowing uses of unique permissions. For example, a change permission `unique>>unique` says that the permission in the parameter value must be unique on entry to the method, and the unique permission will be restored at the end of the method; whereas `unique>>none` says that a unique permission is taken and not returned (for example, because it is stored on the heap).

```
1  state Cell {}
2  state Cons case of Cell {
3    immutable Data data;
4    unique Cell next;
5  }
6  state List {
7    unique Cell head = new Cell;
8    void prepend(immutable Data elt) unique {
9      unique Cons newHead = new Cons with {
10       this.data = elt;
11       this.next = this.head; // this:(unique,head:none)
12     }
13     this.head = newHead;     // this:unique
14   }
15 }
16
17 unique List list = new List; // list:unique
18 list.prepend(new Data);      // list:unique
```

Figure 3: List prepend example.

Figure 3 shows an example using `unique>>unique` (written in shorthand form as `unique`) to update a list with unique links. Line 1 defines a `Cell` state representing an empty list cell. Lines 2–5 define a `Cons` cell which is a substate of `Cell`; it has a `data` field with `immutable` permission and a `next` field with `unique` permission. Lines 6 and following define the actual list. It has a unique `Cell` for its head and a method `prepend` that (1) accepts an `immutable` permission in `elt` and a `unique` permission in `this` (`unique` in line 8 represents the change permission associated with `this`); (2) leaves a `unique` permission in `this` on exit; and (3) returns nothing. The comments in lines 11 and 13 show what happens when checking the method body. In line 11, `this` is unpacked when the unique permission is taken out of `this.head` and assigned into `newHead`. In line 14, `this` is packed back up when `newHead` is assigned into `this.head`.

Lines 17–18 show how things look from the point of view of a caller of `prepend`. In line 17, `list` gets a fresh reference with unique permission. This unique permission is passed into `prepend` in line 18. However, because of the change permission `unique>>unique`, the permission is restored on return from the method, and `list` still has unique permission at the end of line 18. Note that we could also have restored the permission by returning a reference from

prepend and assigning it back into list but this is awkward and would require assignment into local variables.

```
1  state Data {
2    immutable Data publish() unique>>immutable {
3      // Add timestamp
4      this;
5    }
6    ...
7  }
8
9  unique List list = new List;
10 unique Data data = new Data; // data:unique
11 data.publish();              // data:immutable
12 list.prepend(data);          // data:immutable
```

Figure 4: Publication example.

Figure 4 shows another example, this time using a change permission unique>>immutable. In this example, the Data class has a publish method that (1) requires a unique permission to this; (2) uses the unique permission to write a time stamp to the object; then (3) "freezes" the object in an immutable state so it can only be read and never written by the rest of the program. Lines 9–12 show how this state might be used. Line 10 creates a fresh Data object with unique permission. Line 11 calls publish on Data, changing its permission to immutable. Line 12 puts the immutable permission into a list. Notice that since publish also returns an immutable permission to this, we could also have written list.prepend(data.publish()); but using the change permission on variable data makes clear in the client code that the same object is going into publish and into prepend.

2.4 Local Permissions

An important pattern for access permissions is to divide a unique permission up into several weaker permissions, use the weaker permissions for a while, and then put all the permissions back together to reform the original unique. Doing this requires careful accounting to ensure there are no outstanding permissions to the reference (other than none) at the point where the unique is recreated. One way to do this accounting is to use fractions [Boyland, 2003]. However, while powerful, fractions are also difficult to use and suffer from modularity problems [Heule et al., 2011]. Instead, we observe the following:

- The complexity of fraction-based solutions arises in large part because they allow permissions to be stored into the heap, then taken out of the heap and recombined into unique.
- A common use of permissions split off from unique is to pass them into methods, where they are used in local variables and then returned, i.e., they never go into the heap.

Motivated by these observations, we introduce a new kind of permission called a *local permission*. A local permission is designated with the keyword `local`, which may modify a `shared` or `immutable` permission. A local permission annotates a local variable; it says that any aliases of the variable created during its lifetime exist only in local variables and are never stored to the heap. (Local permissions are not necessary for `unique` or `none`, because the permission itself already contains all the information about aliasing to the heap: a `unique` local variable says there is no alias on the heap or anywhere else, and a `none` local variable says it doesn't matter.) Since local permissions exist only in local variables, when all the variables that borrowed local permissions go out of scope, we can reform the original unique permission.

Borrowing Local Permissions from Variables: A local permission may be borrowed from a variable with `unique` permission, leaving a special *borrow permission* in the context. The borrow permission is used internally for accounting purposes, but never appears in the programmer-visible language syntax. For example, borrowing `local immutable` from a `unique` variable leaves `borrow(unique,immutable,1)` in the context for that variable. This entry says we are borrowing `local immutable` permissions from a `unique` permission, and one alias is outstanding. Borrowing again from the same variable increments the counter, while joining decrements the counter. Thus the counter tracks the number of outstanding local aliases to the original `unique` permission; when the counter is 1, joining the last local permission recreates `unique`. Note that we do this counting only when creating `local` permissions, because in this language `immutable` or `shared` permissions are never joined to `unique`.

Figure 5(a) shows an example. The counter for `x` becomes 1 in line 4 when `y` borrows from it, and 2 in line 7 when `z` borrows from it. When `z` goes out of scope, the counter goes back down to 1, and when `y` goes out of scope, `x` becomes `unique` again. When the counter exceeds one (as in line 7), we don't rejoin to `unique` yet because there are still aliases outstanding. Notice how the counters in the borrow permissions effectively account for fractions of permissions, by counting outstanding aliases. However, these fractions are hidden in the typing and never seen or manipulated by the programmer.

We must carefully account for local permissions that might escape the current scope; otherwise we could not soundly reason that all local permissions are out of scope when the local variables holding them are out of scope. Figure 5(b) shows how we do this. In line 3, `y` takes a local immutable permission from `x`, and line 5 attempts to return the local immutable permission to the caller, while retaining the unique permission in `x` (as shown in the signature in line 1 — remember that `unique x` is shorthand for `unique>>unique x`). Of course this should not be allowed. This example doesn't type check, because when `y` goes out of scope at the end of the method, two borrow permissions would have to be joined, and our typing rules don't allow this. Only a local permission can be joined with a borrow permission. More generally, all aliases borrowed from a local permission must be rejoined before the local permission can be rejoined to unique. However, if the method signature in line 1 said `unique>>none S x`, then the example would type check, because now `x` wouldn't have to be rejoined to `unique` to satisfy the parameter's change permission.

Borrowing Local Permissions from Fields: Our language also allows borrowing local permissions from `unique` fields. This is particularly useful when the permission to the object is local.

(a) Rejoining Local Permissions to Unique

```
1  unique x = new S; // x:unique
2  {
3    local immutable y = x;
4    // x:borrow(unique,immutable,1),y:local immutable
5    {
6      local immutable z = x;
7      /* x:borrow(unique,immutable,2),
8           y:local immutable,z:local immutable */
9    }
10   // x:borrow(unique,immutable,1),y:local immutable
11 }
12 // x:unique
```

(b) This Example Does Not Type Check

```
1  local immutable S escape(unique S x) none {
2    // x:unique
3    local immutable y = x;
4    // x:borrow(unique,immutable,1),y:local immutable
5    y; // Local immutable permission taken here
6    /* x:borrow(unique,immutable,1),
7         y:borrow(local immutable,immutable,1) */
8  }
```

Figure 5: Borrowing local permissions from variables.

Otherwise, we would need unique permission to the object even to read a unique field of the object, and it is well known that this requirement is very restrictive (essentially, any access to a linear reference must be through a chain of linear references) [Fähndrich and DeLine, 2002].

Figure 6 shows how this works for a simple example that computes the size of a list with unique references, requiring read-only access to the list. In line 15, `size` requires `local immutable` permission to the list (line 11), so local immutable permission is borrowed from `list` as discussed above, then put back to reform `unique` at the end of the method call. Inside the `size` method of `List`, in line 9, `this` has local immutable permission at the start of the method. At the call of `this.head.size()`, local immutable permission is borrowed from `this.head`. At that point, the context entry for `this` is

```
this:(local immutable,head:borrow(unique,immutable,1))
```

saying that `this` has local immutable permission and points to an unpacked object with one local immutable reference borrowed from its `head` field. On return from `this.head.size()`, the

```
1 state Cell { int size() none {0;} }
2 state Cons case of Cell {
3   immutable Data data;
4   unique Cell next;
5   int size() local immutable { 1 + this.next.size(); }
6 }
7 state List {
8   unique Cell head = new Cell;
9   int size() local immutable { this.head.size(); }
10  ...
11 }
12
13 unique List list = new List; // list:unique
14 list.prepend(new Data);      // list:unique
15 int size = list.size();      // list:unique
```

Figure 6: List size example.

permissions are put back together to repack the object. The same thing happens in `Cell.size` (line 5). Notice that the ability to borrow local permissions is very useful here: without it, we would either have to permanently convert the `unique` to `immutable`, thereby destroying the unique permission to the list to compute its size, or we would have to require unique permission to the list for the size computation, which is too restrictive.

One subtlety of this mechanism is that it allows multiple local variables that point to the same object to have different information about whether the object is packed. For example, consider an object O with a unique field f and two aliases of O, x and y, where y was created as an alias of x by pulling a `local` σ from the `unique` permission in x. This leaves x with the permission `local` σ as well, which allows us to pull a `local` σ from $x.f$, unpacking x. Since the type system does not explicitly track what variables are aliases y remains packed. This is fine because the splitting rules prevent anything other than a `local` σ from being pulled out through $y.f$. Furthermore, by the time y leaves scope and x becomes `unique` again, all `local` aliases to the field f of O must have been returned leaving it `unique` as well. Thus, it will be safe to pull a `unique` permission from $x.f$ as allowed by the static rules. At runtime, we will know that x and y are in fact aliases and do the proper accounting for the permission in field f even when permission are pulled from the field through different aliases. More details on this mechanism are given in Section 3.

3 Formal System

Our formal system is built off a standard nominally-typed object system. We extend reference types with an *access permissions* (section 3.2.1).

3.1 Syntax

3.1.1 States

States are like classes in standard languages and contain a set of methods and fields.

State	S	::=	`state` s `case of` $s\,\{F^*\ M^*\}$
Field	F	::=	$\pi\ s\ f = e$
Method	M	::=	$\pi\ s\ \ m$`(`π`>>`$\pi\ \ s\ \ x$`)`$\ \ \pi$`>>`$\pi\ \{\ \ e\ \}$

A state is declared using the `state` keyword with a name s which introduces a nominal type. Each state is a `case of` another state (possibly itself), from which it inherits method and field declarations. We assume, but do not check, that there are no cycles in the inheritance hierarchy as this is orthogonal from the primary thrust of this work. We also do not model overriding or overloading of declared members and so implicitly require that all declarations have distinct names from the other declarations in the state as well as any declarations inherited from the `case of` state. No members are inherited when the declared state is a `case of` itself. Each field f is annotated with a permission and a state characterizing the kind of references that can be stored in the field. We require that an initialization expression e is provided. Because permissions to a given reference may change over the execution of an expression, method specifications include a summary of how the permissions to each of its inputs change as a result of the execution of method body. This is indicated by a change type $\pi \gg \pi$ which gives the starting and ending permission. Each method declaration includes the state and permission of the returned object, the method name m, the change type and state of the method parameter, the change type of the receiver (the state of the receiver is the state in which the method is declared), and finally a method body e.

3.1.2 Expressions

Our system includes the following expressions:

Expression	e	::=	`new` s | `let` $\pi\ \ x$`=`e `in` e | `match(`e`){(`s`=>`e`;)`$^+$`}` |
			v | v`.`f | v`.`f`=`e | e`.`m`(`e`)` | `{(`e`;)`$^+$`}` |
Value	v	::=	x | `this`

Expressions are standard for an object system, including object creation (`new`), field reads, assignments, variable references, and a series of expressions to be executed in sequence. We also allow new variables to be introduced with `let`. Included in this expression form is the initial permission given to the variable. Finally, a `match` expression determines which branch to execute based on the state of the matched expression.

3.1.3 Programs

A program is made up of a collection of state declarations and an expression to execute:

Program	P	::=	$S^*\ e$

3.2 Static Semantics

Our type system is designed to track the flow of permissions through references in the program over the course of execution.

3.2.1 Permissions

The set of possible permissions in our system includes:

$$\begin{array}{rrcl} \textbf{Permission} & \pi & ::= & \texttt{unique} \mid \texttt{local}\ \sigma \mid \sigma \mid \texttt{none} \mid \mathsf{borrow}(\pi, \sigma, n) \\ \textbf{Symmetric Permission} & \sigma & ::= & \texttt{immutable} \mid \texttt{shared} \end{array}$$

A permission serves two purposes. First it restricts what can be done to the referenced object through this pointer and second it provides guarantees about how other aliases can impact the object.

- `none`: A `none` permission provides no abilities and no guarantees. While a reference has `none` permission, it is unusable.
- `unique`: A `unique` permissions is a linear permission that gives the reference full access to update and modify the object. Furthermore, it guarantees that no other references in the system can update or access the object. In other words, all other aliases of the object must have permission `none`.
- σ: Symmetric permissions σ are non-linear in the sense that they can be freely duplicated. An `immutable` permission provides only read access but guarantees that all other aliases of the object have at most read-only access as well. A `shared` permission provides modifying access to the object, but does not guarantee that other aliases of the object might update it.
- `local`: A `local` σ permission provides the same abilities and guarantees as the underlying σ permission with the additional restriction that any aliases created from this reference be temporary. In practice this means that a `local` reference cannot be assigned into a field. This means we can view a `local` permission as a temporarily non-linear permission.
- `borrow`(π, σ, n): A `borrow` permission also represents a temporarily non-linear permission but records the tracking information that may allow the linear permission to be regained. It includes the original permission π, the underlying non-linear permission σ, and the number of outstanding `local` aliases that have been created from this reference. The `borrow` permission provides the same abilities and guarantees as the underlying σ permission and the same extra restriction as a `local` permission. In addition, once all of the outstanding `local` aliases have been returned, the original permission can be regained. `borrow` permissions are used for internal tracking only and never appear in the source (in fact such a program would not typecheck as per lemma 3.8 below).

Permission Splitting: When a new alias is created from an existing reference and given permission π the permission to the existing reference must be updated to maintain the invariants guaranteed by the permissions. We capture this with the splitting judgment.

$$\boxed{\pi \Rrightarrow \pi \otimes \pi}$$

This judgment says that given the first permission, we can pull the second permission out of it and leave the third.

$\boxed{\pi \Rrightarrow \pi \otimes \pi}$

P-SPLIT-UNIQUE
$$\frac{}{\texttt{unique} \Rrightarrow \texttt{unique} \otimes \texttt{none}}$$

P-SPLIT-UNIQUE-SYMMETRIC
$$\frac{}{\texttt{unique} \Rrightarrow \sigma \otimes \sigma}$$

P-SPLIT-SYMMETRIC
$$\frac{}{\sigma \Rrightarrow \sigma \otimes \sigma}$$

P-SPLIT-UNIQUE-LOCAL
$$\frac{}{\texttt{unique} \Rrightarrow \texttt{local } \sigma \otimes \texttt{borrow}(\texttt{unique}, \sigma, 1)}$$

P-SPLIT-LOCAL
$$\frac{}{\texttt{local } \sigma \Rrightarrow \texttt{local } \sigma \otimes \texttt{borrow}(\texttt{local } \sigma, \sigma, 0)}$$

P-SPLIT-LOCAL-INCREMENT
$$\frac{n \in \mathbb{N}}{\texttt{borrow}(\pi, \sigma, n) \Rrightarrow \texttt{local } \sigma \otimes \texttt{borrow}(\pi, \sigma, n+1)}$$

P-SPLIT-BORROW-SYMMETRIC
$$\frac{n \in \mathbb{N}}{\texttt{borrow}(\texttt{unique}, \sigma, n) \Rrightarrow \sigma \otimes \sigma}$$

P-SPLIT-SYMMETRIC-LOCAL
$$\frac{}{\sigma \Rrightarrow \texttt{local } \sigma \otimes \sigma}$$

P-SPLIT-NONE
$$\frac{}{\pi \Rrightarrow \texttt{none} \otimes \pi}$$

Pulling a non-`none` permission from a `unique` forces us to drop the existing `unique` permission as expected for a linear permission. We can transfer the `unique` permission, leaving `none` (P-SPLIT-UNIQUE) Alternatively, we can make the permission permanently non-linear by pulling a symmetric permission (P-SPLI-UNIQUE-SYMMETRIC), or temporarily non-linear by pulling a `local` σ permission (P-SPLIT-UNIQUE-LOCAL). The latter case leaves a `borrow` permission to keep track of the outstanding `local` permissions and allow the potential to regain the linear permission at a later point. In contrast, non-linear permissions σ can be freely duplicated either as symmetric or `local` permissions (P-SPLIT-SYMMETRIC, P-SPLIT-SYMMETRIC-LOCAL). Only further `local` σ permissions can be split from an existing `local` σ permission in which case the residual permission is a `borrow` permission (P-SPLIT-LOCAL). $\texttt{borrow}(\pi, \sigma, n)$ permissions can split into further `local` σ where the residue includes an incremented count (P-SPLIT-LOCAL-INCREMENT), or in the case that π is `unique` a full σ permission can be split, which consumes the `borrow` permission and reduces it to a σ as well (P-SPLIT-BORROW-SYMMETRIC).

We can prove that multiple permission splits commute and also characterize the conditions under which the original permission and the residue are the same.

Lemma 3.1 (Permission Pull is Commutative)**.** *If $\pi \Rrightarrow \pi_1 \otimes \pi'$ and $\pi' \Rrightarrow \pi_2 \otimes \pi''$, then $\pi \Rrightarrow \pi_2 \otimes \hat{\pi}$ and $\hat{\pi} \Rrightarrow \pi_1 \otimes \pi''$.*

Proof. By straightforward exhaustive case analysis on π, π_1, and π_2. □

Lemma 3.2 (Permission Pull Maintain Residue)**.** *If $\pi \Rrightarrow \pi' \otimes \pi$, then $\pi = \sigma$ and either $\pi' = \sigma$ or $\pi' =$ `local` σ.*

Proof. By straightforward analysis on the splitting rules. □

Field Splitting: Aliases can also be created from fields and so we must be able to split permissions from fields as well. In order to preserve the permission invariants, we must take the permission of the enclosing object into account. In all cases expect pulling a σ from the `unique` field of an object with permission σ, we rely on the permission splitting judgment but restrict what can be pulled. The one exception (P-SPLIT-FIELD-SYMMETRIC) allows us to treat the linear permission locked inside the field of a non-linear object as if it is also non-linear.

$$\boxed{\pi\,.\,\pi \Rrightarrow \pi \otimes \pi}$$

$$\text{P-SPLIT-FIELD-UNIQUE}\ \frac{\pi \Rrightarrow \pi' \otimes \pi''}{\texttt{unique}\,.\,\pi \Rrightarrow \pi' \otimes \pi''}$$

$$\text{P-SPLIT-FIELD-PRESERVE}\ \frac{\pi \neq \texttt{none} \qquad \pi' \Rrightarrow \pi'' \otimes \pi'}{\pi\,.\,\pi' \Rrightarrow \pi'' \otimes \pi'}$$

$$\text{P-SPLIT-FIELD-SYMMETRIC}\ \frac{}{\sigma\,.\,\texttt{unique} \Rrightarrow \sigma \otimes \texttt{unique}}$$

$$\text{P-SPLIT-FIELD-LOCAL}\ \frac{\pi \Rrightarrow \texttt{local}\ \sigma \otimes \pi'}{\texttt{local}\ \sigma\,.\,\pi \Rrightarrow \texttt{local}\ \sigma \otimes \pi'}$$

$$\text{P-SPLIT-FIELD-BORROW}\ \frac{\pi' \Rrightarrow \texttt{local}\ \sigma \otimes \pi''}{\texttt{borrow}(\pi, \sigma, n)\,.\,\pi' \Rrightarrow \texttt{local}\ \sigma \otimes \pi''}$$

We verify the close relationship between normal splitting and field splitting by proving that the latter implies the former.

Lemma 3.3 (Field Split Implies Split)**.** *If* $\pi_1\,.\,\pi_2 \Rrightarrow \pi \otimes \pi'$*, then* $\pi_2 \Rrightarrow \pi \otimes \pi''$.

Proof. Follows directly from the premises of all rules except P-SPLIT-FIELD-SYMMETRIC. In that case, we have `unique` $\Rrightarrow \sigma \otimes \sigma$ by P-SPLIT-UNIQUE-SYMMETRIC. □

Permission Joining: When references go out of scope, they may return their permission to the location it came from at which point the system must join the returned permission to the permission present in the destination. In most cases we can simply reverse the splitting judgment. However, when the permission that is being returned is a symmetric permission, we can only get a symmetric permission back (P-JOIN-SYMMETRIC). We do not restrict what permissions can be returned to fields and so are handled with the same judgment.

$$\boxed{\pi \otimes \pi \Rrightarrow \pi}$$

$$\text{P-JOIN-NOT-SYMMETRIC}\ \frac{\pi \neq \sigma \qquad \pi'' \Rrightarrow \pi \otimes \pi'}{\pi \otimes \pi' \Rrightarrow \pi''}$$

$$\text{P-JOIN-SYMMETRIC}\ \frac{\pi' \in \{\sigma, \texttt{none}\}}{\sigma \otimes \pi' \Rrightarrow \sigma}$$

Permission Downgrade: While typing, many permissions can be pulled or even dropped from a single location, but only in certain circumstances so as to ensure that our invariants are maintained as described below (section 3.4). We capture this with the downgrade judgment. Splitting implies downgrading (P-DOWN-SPLIT). All permissions except `local` permissions and their `borrow` counterparts can be downgraded to `none` (P-DOWN-NONE). `unique` can be downgraded to any

permission (P-DOWN-UNIQUE) and its `borrow` form can be downgraded to borrowed `local` with a higher count (P-DOWN-BORROW-UNIQUE). Finally `local` permissions can be downgraded to `borrow` permissions (P-DOWN-LOCAL-BORROW) and the count of a `borrow` permission can be increased (P-DOWN-BORROW).

$\boxed{\pi \downarrow \pi}$

$$\text{P-DOWN-SPLIT}\ \frac{\pi_1 \Rrightarrow \hat{\pi} \otimes \pi_2}{\pi_1 \downarrow \pi_2} \qquad \text{P-DOWN-NONE}\ \frac{\pi \neq \texttt{local}\ \sigma \qquad \pi \neq \texttt{borrow}(\texttt{local}\ \sigma, \sigma, n)}{\pi \downarrow \text{none}} \qquad \text{P-DOWN-UNIQUE}\ \frac{}{\texttt{unique} \downarrow \pi}$$

$$\text{P-DOWN-BORROW-UNIQUE}\ \frac{n, n' \in \mathbb{N} \qquad n' \geq n}{\text{borrow}(\texttt{unique}, \sigma, n) \downarrow \text{borrow}(\texttt{local}\ \sigma, \sigma, n')} \qquad \text{P-DOWN-LOCAL-BORROW}\ \frac{n \in \mathbb{N}}{\texttt{local}\ \sigma \downarrow \text{borrow}(\texttt{local}\ \sigma, \sigma, n)}$$

$$\text{P-DOWN-BORROW}\ \frac{n, n' \in \mathbb{N} \qquad n' > n}{\text{borrow}(\pi, \sigma, n) \downarrow \text{borrow}(\pi, \sigma, n')}$$

With these definitions, we can state and prove several important lemmas. First, we know that joining preserves a downgrade relationship.

Lemma 3.4 (Join Same to Unequal Permissions)**.** *If* $\pi \otimes \pi_1 \Rrightarrow \pi_1'$ *and* $\pi \otimes \pi_2 \Rrightarrow \pi_2'$ *for* $\pi_1 \downarrow \pi_2$*, then* $\pi_1' \downarrow \pi_2'$*.*

Proof. By straightforward case analysis on the possible permission combinations. □

Second, slitting an returning must either result in the original permission or σ. If the returned permission is downgraded first, then the resulting permission is a downgrade from the original permission.

Lemma 3.5 (Split and Return)**.** *If* $\pi_1 \Rrightarrow \pi \otimes \pi_2$ *and* $\pi \otimes \pi_2 \Rrightarrow \pi_3$*, then either* $\pi_1 = \pi_3$ *or* $\pi = \sigma$*,* $\pi_3 = \sigma$*, and* $\pi_1 \downarrow \pi_3$*.*

Proof. This is straightforward since the restore rules are reversal of the splitting rules with the restriction that anytime a σ permissions can be returned, the result must be σ. □

Lemma 3.6 (Split, Downgrade, and Return)**.** *Given* $\pi_1 \Rrightarrow \pi \otimes \pi_2$ *where* $\pi \downarrow \hat{\pi}$ *and* $\pi_2 \downarrow \pi_2'$*. If* $\hat{\pi} \otimes \pi_2' \Rrightarrow \pi_3$*, then* $\pi_1 \downarrow \pi_3$*.*

Proof. This is similarly straightforward. The fact that the return succeeds restricts what permissions π_2' and $\hat{\pi}$ can be and leads to this result. □

Finally, we can prove several properties about downgrading and splitting.

Lemma 3.7 (Downgrading and Splitting)**.** *Given* $\pi_1 \downarrow \pi_2$*.*

1. *If* $\pi_2 \Rrightarrow \pi \otimes \pi_2'$*, then* $\pi_1 \downarrow \pi_2'$*. Furthermore,* $\pi_1 \Rrightarrow \pi \otimes \pi_1'$ *with* $\pi_1' \downarrow \pi_2'$*.*
2. *If* $\pi \otimes \pi_1 \Rrightarrow \pi_1'$*, then* $\pi_1' \downarrow \pi_2$*.*
3. *If* $\pi_2 \downarrow \pi_3$*, then* $\pi_1 \downarrow \pi_3$*.*

Proof. By straightforward exhaustive case analysis on the permissions and associated rules. □

3.2.2 Permission tracking

In order to track the flow of permissions through a program, we introduce two structures which provide the means for recording where permissions are located.

Linear context:
The linear context maps a value v to permissions π.

Linear Context $\Delta \quad ::= \quad \Delta, v : (\pi\ s; \Pi) \mid \varnothing$

In addition to the permission of the value, we also track its state s which records the fields and methods available in the referenced object, and any of the fields which are currently *unpacked*, Π. When a permission is split off of a field such that the permission of the reference in the field no longer fulfills the permission declared for the field in the state, we consider the field to be unpacked. The unpacked fields are of a reference are tracked in Π:

Unpacked Environment $\Pi \quad ::= \quad \Pi, f : (\pi, i) \mid \varnothing$

Each unpacked field f is mapped to its current permission π and to a *field id* i. Because we allow assignment to fields, there is no guarantee the field has not been updated between the time a permission is pulled and it is returned. As detailed below (see **Return Rules** in section 3.2.4), the field id provides a lightweight mechanism to prevent the permission from being returned in this case as returning a permission to a different object would be unsound.

Source Locations:
In order to give the returned reference a permission, that permission must have been taken from somewhere. A *source location* records a place that a permission has been pulled from:

Source Location $\ell \quad ::= \quad v \mid (v\,.\,f, i)$

The location is either a value v, or a field $v\,.\,f$. A field source location includes an field id i used to ensure that permissions pulled from one reference are not returned to a field containing a different reference.

Because of `nonDet`, the specified permission may come from one of several locations at runtime. So, we consider lists of source locations ℓ^*, called a *source location list* (SLL hereafter).

Well-formed SLL: SLLs refer to locations in a given context where a permission may later be returned. Hence we make sure that only locations actually in Δ appear in its associated ℓ^*.

$$\boxed{\Delta \vdash \ell^* \ \mathbf{ok}}$$

$$\frac{}{\Delta \vdash \varnothing \ \mathbf{ok}}\ \text{SLL-EMPTY}$$

$$\frac{v : (\pi\ s; \Pi) \in \Delta \qquad \Delta \vdash \ell^* \ \mathbf{ok}}{\Delta \vdash \ell^*, v \ \mathbf{ok}}\ \text{SLL-VAL-IN-SCOPE}$$

$$\frac{v : (\pi\ s; \Pi) \in \Delta \qquad \Delta \vdash \ell^* \ \mathbf{ok}}{\Delta \vdash (\ell^*, v\,.\,f, i) \ \mathbf{ok}}\ \text{SLL-FIELD-IN-SCOPE}$$

3.2.3 Useful sets, predicates, and functions

We define several notational abbreviations for properties used throughout the typing judgments below.

State properties:

The sets $\mathsf{fields}(P, s)$ and $\mathsf{methods}(P, s)$ respectively contain all of the methods and fields defined in the state s declared in program P, including definitions from superstates.

$$\mathsf{fields}(P, s) = \{F \mid \texttt{state}\ s\ \texttt{case of}\ s'\ \{\ F^*\ M^*\ \} \in P \wedge (F \in F^* \vee F \in \mathsf{fields}(P, s'))\}$$
$$\mathsf{methods}(P, s) = \{M \mid \texttt{state}\ s\ \texttt{case of}\ s'\ \{\ F^*\ M^*\ \} \in P \wedge (M \in M^* \vee M \in \mathsf{methods}(P, s'))\}$$

Depending on the context and information needed, we will sometimes omit pieces of the field definition when classifying a field declaration as in the set of fields of a state (i.e $f \in \mathsf{fields}(P, s)$).

We also have a function $\mathsf{field\text{-}type}(P, s, f)$ that returns the declared permission and state of field f from state s declared in P.

$$\mathsf{field\text{-}type}(P, s, f) = \pi\ s' \iff \pi\ s'\ f = e \in \mathsf{fields}(P, s)$$

Domains:

We use the notation dom to denote the set of keys in a map, including a context Δ, a heap H, a unpacked environment Π, or a field map Ξ (see section 3.3.1).

Context properties:

Given a mapping $v : (\pi\ s; \Pi) \in \Delta$ there are several important properties of the various components. First, a field from state s is packed if it does not appear in Π.

$$\mathsf{packed}(f, \Pi) \iff f \notin \mathsf{dom}(\Pi)$$

An unpacked field may have a different permission than it is declared to have. The $\mathsf{cur\text{-}field\text{-}perm}$ of a field f takes the permission from the unpacked environment Π if it is mapped there or from the definition of state s in P if it is not.

$$\mathsf{cur\text{-}field\text{-}perm}(P; s; f; \Pi) = \pi \iff f : (\pi, i) \in \Pi \vee (f \notin \Pi \wedge \pi\ s'\ f = e \in \mathsf{fields}(P, s, f))$$

Not all permissions allow assignment to the fields of the object reference associated with it. A permission π is assignable if it is `unique` or some form of `shared`.

$$\mathsf{assignable}(\pi) \iff \left(\begin{array}{c}\pi = \texttt{unique} \vee \exists \pi_o, n.\pi = \texttt{borrow}(\pi_o, \texttt{shared}, n) \vee \\ \pi = \texttt{shared} \vee \pi = \texttt{local shared}\end{array}\right)$$

3.2.4 Typing

Substates:

We use the nominal substate relation in the process of typechecking. The relationship is the reflexive and transitive closure of the `case of` relations declared for each state.

$\boxed{P \vdash s \preceq s'}$

$$\textsc{Substate}\ \frac{\texttt{state}\ s\ \texttt{case of}\ s'\ \{\ F^*\ \} \in P}{P \vdash s \preceq s'} \qquad \textsc{Subtype-Reflexive}\ \frac{}{P \vdash s \preceq s} \qquad \textsc{Substate-Transitive}\ \frac{P \vdash s \preceq s' \quad P \vdash s' \preceq s''}{P \vdash s \preceq s''}$$

Top-Level checks:

To check programs, we check that each state is well-formed and that the program expression can be given a `none` permission in the empty context. For a state to be valid, we require that the state it is a `case of` is in the program and that each declared field and method is valid. A field cannot be declared with a `local` σ or `none`. Otherwise we require that the initializing expression e can be given the declared permission in the empty context. The resulting state of the expression must be a substate of the declared state of the field. For a method to be valid, we must be able to type the method body to the specified return permission in the context with the receiver `this` and the parameter x mapped to their initial permission and state and packed. The resulting permission for each must be the specified output of the change permission.

$\boxed{\vdash P}$

$$\textsc{T-Program}\ \frac{\forall S \in S^*.(P \vdash S) \quad P; \varnothing; \texttt{none} \vdash e : s \dashv \varnothing; \varnothing}{\vdash S^*\ e}$$

$\boxed{P \vdash S}$

$$\textsc{T-State}\ \frac{\texttt{state}\ s'\ \texttt{case of}\ s''\ \{\ F'^*\ M'^*\ \} \in P \quad \forall F \in F^*.(P \vdash F) \quad \forall M \in M^*.(P, s \vdash M)}{P \vdash \texttt{state}\ s\ \texttt{case of}\ s'\ \{\ F^*\ M^*\ \}}$$

$\boxed{P \vdash F}$

$$\textsc{T-Field}\ \frac{\texttt{state}\ s\ \texttt{case of}\ \hat{s}\ \{\ F^*\ M^*\ \} \in P \quad \pi \neq \texttt{local}\ \sigma \quad \pi \neq \texttt{none} \quad P; \varnothing; \pi \vdash e : s' \dashv \varnothing; \varnothing \quad P \vdash s' \preceq s}{P \vdash \pi\ s\ f\texttt{=}e\texttt{;}}$$

$\boxed{P, s \vdash M}$

$$\textsc{T-Method}\ \frac{P \vdash s \quad P \vdash s_2}{P; \texttt{this} : (\pi_1\ s_1; \varnothing), x : (\pi_2\ s_2; \varnothing); \pi \vdash e : s', \texttt{this} : (\pi_1'\ s_1; \varnothing), x : (\pi_2'\ s_2; \varnothing); \ell^*}{P, s_1 \vdash \pi\ s\ m\,(\pi_2 \texttt{>>} \pi_2'\ s_2\ x)\ \pi_1 \texttt{>>} \pi_1'\ \{\ e\ \}}$$

Return Rules:

During typing we periodically return a permission to each of the locations in a SLL. The return judgment $P : \Delta; \pi \vdash \ell^* \dashv \Delta$ outputs a context that is the result of returning permission π to the locations indicated by ℓ^* in the input context. The rules consider each location in the SLL separately.

$$\boxed{\Delta; \pi \vdash \ell^* \dashv \Delta}$$

RESTORE-EMPTY
$$\frac{}{P; \Delta; \pi \vdash \varnothing \dashv \Delta}$$

RESTORE-VALUE
$$\frac{\Delta = \hat{\Delta}, v : (\pi'\ s; \Pi) \qquad \pi \otimes \pi' \Rightarrow \pi'' \qquad P; \hat{\Delta}, v : (\pi''\ s; \Pi); \pi \vdash \ell^* \dashv \Delta'}{P; \Delta; \pi \vdash v, \ell^* \dashv \Delta'}$$

RESTORE-FIELD-PACKED
$$\frac{v : (\pi'\ s; \Pi) \in \Delta \qquad \mathsf{packed}(f, \Pi) \qquad P; \Delta; \pi \vdash \ell^* \dashv \Delta'}{P; \Delta; \pi \vdash (v\,.\,f, i), \ell^* \dashv \Delta'}$$

RESTORE-FIELD-STALE
$$\frac{v : (\pi'\ s; \Pi, f : (\pi'', j)) \in \Delta \qquad i \neq j \qquad P; \Delta; \pi \vdash \ell^* \dashv \Delta'}{P; \Delta; \pi \vdash (v\,.\,f, i), \ell^* \dashv \Delta'}$$

RESTORE-FIELD-UNPACKED
$$\frac{\Delta = \Delta', v : (\pi\ s; \Pi, f : (\pi_2, i)) \qquad \mathsf{field\text{-}type}(P, s, f) = \pi'\ s' \qquad \pi_1 \otimes \pi_2 \Rightarrow \pi_3 \qquad \pi_3 \neq \pi' \qquad P; \Delta', v : (\pi\ s; \Pi, f : (\pi_3, i)); \pi_1 \vdash \ell^* \dashv \Delta''}{P; \Delta; \pi_1 \vdash (v\,.\,f, i), \ell^* \dashv \Delta''}$$

RESTORE-FIELD-PACK
$$\frac{\Delta = \Delta', v : (\pi\ s; \Pi, f : (\pi_2, i)) \qquad \mathsf{field\text{-}type}(P, s, f) = \pi'\ s' \qquad \pi_1 \otimes \pi_2 \Rightarrow \pi' \qquad P; \Delta', v : (\pi\ s; \Pi); \pi_1 \vdash \ell^* \dashv \Delta''}{P; \Delta; \pi_1 \vdash (v\,.\,f, i), \ell^* \dashv \Delta''}$$

Returning to an empty list or a value is straightforward and uses the permission join judgment for the later case. Fields are more complicated because of reassignment. In the case that the field is already packed, it already has its maximal permission, so there is nothing to do RESTORE-FIELD-PACKED. Similarly, if the field ids in the source location and unpacked environment do not match, this means that the field may not contain the same object as when the permission was pulled and so we conservatively do not return the permission RESTORE-FIELD-STALE. In the last two cases, the field is unpacked with the same field id, so we use the permission join judgment to return the permission, packing the field if the resulting permission is the same as the declared permission of the field RESTORE-FIELD-PACK, otherwise leaving it unpacked RESTORE-FIELD-UNPACKED.

Typing Expressions:

Typing an expression in our system guarantees that a new alias can be created for the returned reference with the specified permission. To type an expression e, we need a program P, an input context Δ, and a needed permission π. We output the state of the reference, an updated context Δ' and a SLL ℓ^* representing all the locations that the needed permission may have been taken from in Δ'.

$$\boxed{P; \Delta; \pi \vdash e : s \dashv \Delta'; \ell^*}$$

We have two classes of typing rules: syntactic rules and equalizing rules.

Syntactic rules:

The syntactic rules describe how to type each of the expression forms.

$\boxed{P;\Delta;\pi \vdash e : s \dashv \Delta;\ell^*}$

T-NEW

$$\frac{\texttt{state}\ s\ \texttt{case of}\ s'\ \{\ F^*\ \ M^*\ \}\in P \qquad \texttt{unique} \Rrightarrow \pi \otimes \pi'}{P;\Delta;\pi \vdash \texttt{new}\ s : s \dashv \Delta;\varnothing}$$

T-VARIABLE

$$\frac{\pi \Rrightarrow \pi' \otimes \pi''}{P;\Delta, x : (\pi\ s;\varnothing);\pi' \vdash x : s \dashv \Delta, x : (\pi''\ s;\varnothing);x}$$

T-LET

$$\frac{P;\Delta;\pi_1 \vdash e_1 : s_1 \dashv \Delta_1;\ell_1^* \qquad P;\Delta_1, x : (\pi_1\ s_1;\varnothing);\pi \vdash e : s \dashv \Delta_2, x : (\pi_2\ s_1;\varnothing);\ell^*,\hat{\ell}^* \qquad \Delta_2 \vdash \ell^*\ \mathbf{ok} \qquad P;\Delta_2;\pi_2 \vdash \ell_1^* \dashv \Delta'}{P;\Delta;\pi \vdash \texttt{let}\ \ \pi_1\ \ x = e_1\ \ \texttt{in}\ \ e : s \dashv \Delta';\ell^*}$$

T-FIELD-ACCESS-PACKED-PACKED

$$\frac{v : (\pi'\ s;\Pi) \in \Delta \qquad \mathsf{packed}(f,\Pi) \qquad \mathsf{field\text{-}type}(P,s,f) = \pi''\ s' \qquad \pi'\,.\,\pi'' \Rrightarrow \pi \otimes \pi''}{P;\Delta;\pi \vdash v\,.\,f : s' \dashv \Delta;\varnothing}$$

T-FIELD-ACCESS-PACKED-UNPACKED

$$\frac{\Delta = \Delta', v : (\pi_1\ s;\Pi) \qquad \mathsf{packed}(f,\Pi) \qquad \mathsf{field\text{-}type}(P,s,f) = \pi_2\ s' \qquad \pi_1\,.\,\pi_2 \Rrightarrow \pi_3 \otimes \pi_4 \qquad \pi_2 \neq \pi_4 \qquad \mathsf{fresh}(i) \qquad \Delta'' = \Delta', v : (\pi_1\ s;\Pi, f : (\pi_4, i))}{P;\Delta;\pi_3 \vdash v\,.\,f : s' \dashv \Delta'';(v\,.\,f,i)}$$

T-FIELD-ACCESS-UNPACKED-MAINTAIN

$$\frac{\Delta = \Delta', v : (\pi_1\ s;\Pi, f : (\pi_2, i)) \qquad \mathsf{field\text{-}type}(P,s,f) = \pi_3\ s' \qquad \pi_1\,.\,\pi_2 \Rrightarrow \pi_4 \otimes \pi_2}{P;\Delta;\pi_4 \vdash v\,.\,f : s' \dashv \Delta;\varnothing}$$

T-FIELD-ACCESS-UNPACKED-UPDATE

$$\frac{\Delta = \Delta', v : (\pi_1\ s;\Pi, f : (\pi_2, i)) \qquad \mathsf{field\text{-}type}(P,s,f) = \pi_3\ s' \qquad \pi_1\,.\,\pi_2 \Rrightarrow \pi_4 \otimes \pi_5 \qquad \pi_2 \neq \pi_5 \qquad \Delta'' = \Delta', v : (\pi_1\ s;\Pi, f : (\pi_5, i))}{P;\Delta;\pi_4 \vdash v\,.\,f : s' \dashv \Delta'';(v\,.\,f,i)}$$

T-FIELD-ASSIGN-PACKED

$$\frac{v : (\pi_1\ s_1;\Pi) \in \Delta \qquad \mathsf{field\text{-}type}(P,s_1,f) = \pi_2\ s_2 \qquad P;\Delta;\pi_2 \vdash e : s_3 \dashv \Delta';\ell^* \qquad v : (\pi_3\ s_1;\Pi') \in \Delta' \qquad \mathsf{assignable}(\pi_3) \qquad \mathsf{packed}(f,\Pi') \qquad P \vdash s_3 \preceq s_2 \qquad \pi_2 \Rrightarrow \pi \otimes \pi_2}{P;\Delta;\pi \vdash v\,.\,f{=}e : s_3 \dashv \Delta';\varnothing}$$

T-FIELD-ASSIGN-UNPACKED

$$\frac{v : (\pi_1\ s_1;\Pi) \in \Delta \qquad \mathsf{field\text{-}type}(P,s_1,f) = \pi_2\ s_2 \qquad P;\Delta;\pi_2 \vdash e : s_3 \dashv \Delta_1;\ell^* \qquad \Delta_1 = \Delta_2, v : (\pi_3\ s_1;\Pi', f : (\pi_4, i)) \qquad \mathsf{assignable}(\pi_3) \qquad P \vdash s_3 \preceq s_2 \qquad \pi_2 \Rrightarrow \pi \otimes \pi_2 \qquad \Delta_3 = \Delta_2, v : (\pi_3\ s_1;\Pi')}{P;\Delta;\pi \vdash v\,.\,f{=}e : s_3 \dashv \Delta_3;\varnothing}$$

T-INVOKE

$$\frac{P;\Delta;\pi_1 \vdash e_1 : s_1' \dashv \Delta_1;\ell_1^* \qquad \pi\ s\ \ m\,(\pi_2{>>}\pi_2'\ \ s_2\ \ x)\ \ \pi_1{>>}\pi_1'\ \ \{\ \ e\ \ \} \in \mathsf{methods}(P,s_1) \qquad P;\Delta_1;\pi_2 \vdash e_2 : s_2' \dashv \Delta_2;\ell_2^* \qquad P;\Delta_2;\pi_2' \vdash \ell_2^* \dashv \Delta_3 \qquad P;\Delta_3;\pi_1' \vdash \ell_1^* \dashv \Delta'}{P;\Delta;\pi \vdash e_1\,.\,m(e_2) : s \dashv \Delta';\varnothing}$$

T-SEQUENCE-SINGLE

$$\frac{P;\Delta;\pi \vdash e : s \dashv \Delta';\ell^*}{P;\Delta;\pi \vdash \{e;\} : s \dashv \Delta';\ell^*}$$

T-SEQUENCE-MULTIPLE

$$\frac{P;\Delta;\texttt{none} \vdash e : s \dashv \Delta';\hat{\ell}^* \qquad P;\Delta';\pi \vdash \{(e';)^+\} : s' \dashv \Delta'';\ell^*}{P;\Delta;\pi \vdash \{e;\ \ (e';)^+\ \} : s' \dashv \Delta'';\ell^*}$$

T-MATCH-SINGLE

$$\frac{P;\Delta;\texttt{none} \vdash e : s \dashv \Delta';\hat{\ell}^* \qquad \exists s_l.P \vdash s_l = \mathsf{lub}(s,s_c) \qquad P;\Delta';\pi \vdash e' : s';\Delta'';\ell^*}{P;\Delta;\pi \vdash \texttt{match}\,(e)\,\{s_c{=>}e';\} : s' \dashv \Delta'';\ell^*}$$

T-MATCH-MULTIPLE

$$\frac{P;\Delta;\pi \vdash \texttt{match}\,(e)\,\{(s_c{=>}e_1;)^+\} : s \dashv \Delta_1;\ell^* \qquad P;\Delta;\pi \vdash \texttt{match}\,(e)\,\{s_c'{=>}e_2;\} : s;\Delta_2;\ell^* \qquad \Delta_1 \overset{\text{ID}}{\approx} \Delta_2 \dashv \Delta' \qquad \ell_3^* = \ell_1^* \cup \ell_2^*}{P;\Delta;\pi \vdash \texttt{match}\,(e)\,\{(s_c{=>}e_1;)^+\ \ s_c'{=>}e_2;\} : s;\Delta';\ell^*}$$

- T-NEW: Since a `new` expression creates a new object, it can be given any permission that can be split from `unique` and gives an empty SLL since the object was created anew. The restriction on the permission ensures that objects cannot start with a `borrow` permission.

- T-VARIABLE: When typing a variable x we simply use the permission splitting judgment, update the outgoing context to include the residual permission from the split for x and give x as the SLL. To ensure that the invariants on the state of the returned reference hold, we require that the variable be packed.

- T-LET: When we encounter a `let` expression, we first pull the required permission for x from the bound expression. We then type the body in the resulting context with x added to the context. Whatever permission remains to x after typing the body is then returned to the SLL generated by typing the bound expression. We give the context resulting from the return as output along with the portion of the SLL generated by typing the body that is well-formed in the output context.

- T-FIELD-ACCESS-*: These rules each deal with a different case of field read. In each, the needed permission is pulled from the current type of the field f, given either by the permission for f in the unpacked environment, or the declared permission for the field if f is packed. If the field remains packed or the permission otherwise is not changed by the pull, we make no updates and do not include the field in the SLL. If the permission is updated, then the unpacked environment is updated with the new permission. f is added to the environment with a fresh field id if it is not present already. The field with its current field ID from the unpacked environment is given as the SLL.

- T-FIELD-ASSIGN-*: Assignment to a field always packs up the field. The assignment is only possible if for the permission to the object π_3 after typing the value expression we have $\mathsf{assignable}(\pi_3)$. We type the target expression with the declared permission of the field and ensure that its state is a substate of the required state for the field. The reference that is returned is an alias to the object assigned into the field, so we ensure that we can pull the needed permission from the permission to the field without reducing it. Since our system does not track permission returns through fields, the SLL is empty.

- T-INVOKE: Method calls require that we find the method declaration in the type of the receiver (non-deterministically, or through a pre-pass). We then chain the typing of the receiver and parameter expressions using the input permissions for the method as the needed permission. We then return the output permission specified in the change permission for each input reference to the SLL generated by the typing of the respective expressions. We require that the needed permission is the permission specified for the return.

- T-SEQUENCE-*: Sequence expressions are typed by typing each expressions in the sequence in order using `none` for the needed permission of all (T-SEQUENCE-MULTIPLE) but the last expression in the sequence from which is pulled the permission needed from the whole expression (T-SEQUENCE-SINGLE).

- T-MATCH-*: The final syntactic form is `match` which requires more explanation. Since we do not know at compile time which of the branches will be executed, we must be conservative and output and come up with a context that captures the updates made by either branch. The primary tool we use to do this are the equalizing typing rules, described below, which serve two purposes. First, they allow modifications of the outgoing context that could occur as part of typing an expression to occur without the expression. Second, they allow the resulting permission of a `unique` permission that ended as `immutable` in one branch and `shared` in the other to be harmonized at `none`. However, the output contexts could still be different in terms of the field ids appearing. We conservatively use the id-equalizing judgment to harmonize these ids.

$$\boxed{\Delta \overset{\text{ID}}{\approx} \Delta \dashv \Delta}$$

ID-EQUIV-EMPTY

$$\frac{}{\varnothing \overset{\text{ID}}{\approx} \varnothing \dashv \varnothing}$$

ID-EQUIV-PACKED

$$\frac{\Delta_1 \overset{\text{ID}}{\approx} \Delta_2 \dashv \Delta'}{\Delta_1, v : (\pi\ s; \varnothing) \overset{\text{ID}}{\approx} \Delta_2, v : (\pi\ s; \varnothing) \dashv \Delta', v : (\pi\ s; \varnothing)}$$

ID-EQUIV-UNPACKED-EQUAL

$$\frac{\Delta_1 = \Delta_1', v : (\pi\ s; \Pi_1, f : (\pi', i)) \quad \Delta_2 = \Delta_2', v : (\pi\ s; \Pi_2, f : (\pi', i)) \quad \Delta_1', v : (\pi\ s; \Pi_1) \overset{\text{ID}}{\approx} \Delta_2', v : (\pi\ s; \Pi_2) \dashv \Delta', v : (\pi\ s; \Pi')}{\Delta_1 \overset{\text{ID}}{\approx} \Delta_2 \dashv \Delta', v : (\pi\ s; \Pi', f : (\pi', i))}$$

ID-EQUIV-UNPACKED-DIFFERENT

$$\frac{\Delta_1 = \Delta_1', v : (\pi\ s; \Pi_1, f : (\pi', i)) \quad \Delta_2 = \Delta_2', v : (\pi\ s; \Pi_2, f : (\pi', j)) \quad \Delta_1, v : (\pi\ s; \Pi_1) \overset{\text{ID}}{\approx} \Delta_2, v : (\pi\ s; \Pi_2) \dashv \Delta', v : (\pi\ s; \Pi') \quad i \neq j \quad k \text{ fresh}}{\Delta_1 \overset{\text{ID}}{\approx} \Delta_2 \dashv \Delta', v : (\pi\ s; \Pi', f : (\pi', k))}$$

If the field ids for a given unpacked field f are the same between contexts, it is maintained, otherwise each is changed to a newly generated fresh id. One could imagine a more permissive, but still safe variant, but we will use this conservative approximation to keep the rules relatively simple.

With this, typing a `match` with a single case (T-MATCH-SINGLE) requires that the matched expression can be typed to the `none` permission and that the case expression can be typed to the needed permission in the resulting context. We also require that the least upper bound of the states of the matched expression and the particular case state exists. This is strictly a well-formedness requirement that prevents cases that could never apply from typechecking. To typecheck a `match` with multiple cases, we type each case individually to the same context and then harmonize the field ids and join the SLLs (T-MATCH-MULTIPLE).

Equalizing rules:

Equalizing rules simulating typing an expression without the expression, allowing us to non-deterministically find a typing that outputs the same context (modulo field ids) for two different expressions.

T-EQ-PULL-VAL

$$\frac{P; \Delta; \pi \vdash e : s \dashv \Delta'; \ell^* \qquad v \notin \ell^* \qquad P; \Delta'; \pi \vdash v : s' \dashv \Delta''; v}{P; \Delta; \pi \vdash e : s \dashv \Delta''; \ell^*, v}$$

T-EQ-PULL-FIELD

$$\frac{P; \Delta; \pi \vdash e : s \dashv \Delta'; \ell^* \qquad \nexists\, i.(v\,.\,f, i) \in \ell^* \qquad P; \Delta'; \pi \vdash v\,.\,f : s' \dashv \Delta''; (v\,.\,f, j)}{P; \Delta; \pi \vdash e : s \dashv \Delta''; \ell^*, (v\,.\,f, j)}$$

T-EQ-DROPPERM-VAL

$$\frac{P; \Delta; \pi \vdash e : s \dashv \Delta', v : (\pi'\ s'; \varnothing); \ell^* \qquad \pi' \downarrow \pi'' \qquad \pi' \neq \pi''}{P; \Delta'; \pi \vdash e \dashv \Delta', v : (\pi''\ s'; \varnothing)\ell^*}$$

T-EQ-DROPPERM-FIELD-PACKED

$$\frac{P; \Delta; \pi \vdash e : s \dashv \Delta', v : (\pi_1\ s_1; \Pi); \ell^* \qquad \mathsf{packed}(f, \Pi) \qquad \mathsf{field\text{-}type}(P, s_1, f) = \pi_2\ s_2 \qquad \pi_2 \downarrow \pi_2' \qquad \pi_2 \neq \pi_2' \qquad \mathsf{fresh}(i)}{P; \Delta; \pi \vdash e : s \dashv \Delta', v : (\pi_1\ s_1; \Pi, f : (\pi_2', i)); \ell^*}$$

T-EQ-DROPPERM-FIELD-UNPACKED

$$\frac{P; \Delta; \pi \vdash e : s \dashv \Delta', v : (\pi_1\ s'; \Pi, f : (\pi_2, i)); \ell^* \qquad \pi_2 \downarrow \pi_2' \qquad \pi_2 \neq \pi_2'}{P; \Delta; \pi \vdash e : s \dashv \Delta', v : (\pi'\ s'; \Pi, f : (\pi_2', i)); \ell^*}$$

T-EQ-DROPRET

$$\frac{\Delta; \pi \vdash e : s \dashv \Delta'; \ell, \ell^*}{\Delta; \pi \vdash e : s \dashv \Delta'; \ell^*}$$

T-EQ-SUPERSTATE

$$\frac{\Delta; \pi \vdash e : s' \dashv \Delta'; \ell^* \qquad P \vdash s' \preceq s}{\Delta; \pi \vdash e : s \dashv \Delta'; \ell^*}$$

We can add a location to the SLL if it is not already there and typing the expression representing that location adds it to the SLL (T-EQ-PULL-VAL,T-PULL-FIELD). Recall that fields are not added to the SLL if their permission is not changed. Otherwise, we can simply downgrade the permission of any value or field as if we pulled permissions from it (T-EQ-DROPPERM-*). T-EQ-DROPRET allows a location to be removed from the SLL and T-EQ-SUPERSTATE allows us to lose information about the state of resulting reference.

Properties of expression typing:

There are several important properties of typing of expressions that we note and prove now. In general, the inputs and outputs of the typing judgment follow a well-defined pattern.

Lemma 3.8 (Properties of Typing)**.** *Given* $P; \Delta; \pi \vdash e : s \dashv \Delta'; \ell^*$*, then*

1. $x : (\pi_1\ s_1; \Pi_1) \in \Delta$ *if and only if* $x : (\pi_2\ s_2; \Pi_2) \in \Delta'$ *and* $s_1 = s_2$.
2. $\Delta \vdash \ell^*$ **ok**.
3. *There exists no* π_o*,* σ*, and* $n \in \mathbb{N}$ *such that* $\pi = \texttt{borrow}(\pi_o, \sigma, n)$.

Proof. By straightforward induction on the typing rules. □

With a little bit more information about the inputs or the rules used, we can get further guarantees. If an equalizing rule was the last rule used, we have very specific information about how the output changed as a result of applying that rule.

Lemma 3.9 (Equalizing Effects)**.** *Given* $P; \Delta; \pi \vdash e : s \dashv \Delta'; \ell^*$ *where the last rule used in the typing derivation is one of* T-Eq-* *the premise of which gives* $P; \Delta; \pi \vdash e : s' \dashv \hat{\Delta}; \hat{\ell}^*$*. Then,* $v : (\pi_1\ s_1; \Pi_1) \in \hat{\Delta}$ *if and only if* $v : (\pi_2\ s_1; \Pi_2) \in \Delta'$ *and for each such* v*,*

1. $\pi_1 \downarrow \pi_2$*, and*
2. *For all* $f \in$ *fields*(P, s_1)*,* *cur-field-perm*$(P; s_1; f; \Pi_1) \downarrow$ *cur-field-perm*$(P; s_1; f; \Pi_2)$

Proof. By straightforward case analysis on the equalizing rules. □

When a variable starts with a `none` permission, we know it ends with it, too.

Lemma 3.10 (Start `none`, End `none`)**.** *If* $P; \Delta, x : (\texttt{none}\ s_1; \varnothing); \pi \vdash e : s \dashv \Delta', x : (\pi'\ s_1; \Pi); \ell^*$*, then* $\pi' = \texttt{none}$*,* $\Pi = \varnothing$*, and if* $x \in \ell^*$ *then* $\pi = \texttt{none}$*.*

Proof. By straightforward induction on the typing rules. □

This fact allows us to characterize how permissions in the context can change as a result of typing, in particular if their location appears in the SLL.

Lemma 3.11 (Typing Permission Change for Variables)**.** *Given any of the following sets of* ℓ*,* π_1*, and* π_2

1. $P; \Delta, x : (\pi_1\ s_1; \Pi_1); \pi \vdash e : s \dashv \Delta', x : (\pi_2\ s_1; \Pi_2); \ell^*$ *with* $\ell = x$*.*
2. $P; \Delta, x : (\pi'\ s'; \Pi_1, f : (\pi_1, i)); \pi \vdash e : s \dashv \Delta', x : (\pi'\ s'; \Pi_2, f : (\pi_2, i)); \ell^*$ *with* $\ell = f : (\pi_2, i)$*.*
3. $P; \Delta, x : (\pi'\ s'; \Pi_1); \pi \vdash e : s \dashv \Delta', x : (\pi''\ s'; \Pi_2, f : (\pi_2, i)); \ell^*$ *where* *packed*(f, Π_1) *or* $f : (\pi_2, j) \in \Pi_1$ *for* $i \neq j$ *with* $\ell = f : (\pi_2, i)$*.*

A. The following implications regarding the starting and ending permission hold:

(a) $\pi_1 = \sigma$ *implies* $\pi_2 \in \{\sigma, \texttt{none}\}$*.*

(b) $\pi_1 = \texttt{local}\ \sigma$ *implies there exists* $n \in \mathbb{N}$ *such that* $\pi_2 \in \{\texttt{local}\ \sigma, \texttt{borrow}(\texttt{local}\ \sigma, \sigma, n)\}$*.*

(c) $\pi_1 = \texttt{borrow}(\pi_o, \sigma, n)$ *implies either*

i. $\pi_2 = \texttt{borrow}(\pi_o, \sigma, n')$ *where* $n' \geq n$*, or*

ii. $\pi_o = \texttt{unique}$ *and* $\pi_2 \in \{\sigma, \texttt{none}\}$*.*

B. If, furthermore, $\ell \in \ell^*$*, then*

(a) $\pi = \texttt{unique}$ *implies* $\pi_1 = \texttt{unique}$ *and* $\pi_2 = \texttt{none}$*.*

(b) $\pi = \sigma$ *implies exists* $n \in \mathbb{N}$ *such that* $\pi_1 \in \{\texttt{unique}, \texttt{borrow}(\texttt{unique}, \sigma, n), \sigma\}$ *and* $\pi_2 \in \{\sigma, \texttt{none}\}$*.*

(c) $\pi = \texttt{local}\ \sigma$ *implies exists* $n \in \mathbb{N}$ *such that* $\pi_1 \in \{\texttt{unique}, \texttt{local}\ \sigma, \texttt{borrow}(\hat{\pi}, \sigma, n), \sigma\}$ *and*

i. $\pi_1 = \sigma$ *implies* $\pi_2 \in \{\sigma, \texttt{none}\}$.

ii. $\pi_1 \in \{\texttt{unique}, \texttt{local}\ \sigma\}$ *implies* $\pi_2 = \texttt{borrow}(\pi_1, \sigma, \hat{n})$ *for some* $\hat{n} \in \mathbb{N}$.

iii. $\pi_1 = \texttt{borrow}(\hat{\pi}, \sigma, n)$ *implies* $\texttt{borrow}(\hat{\pi}, \sigma, n')$ *where* $n' > n$.

Proof. By straightforward induction on the typing derivation and exhaustive analysis of the permissions involved. The variable case is the basic case. The first field case follows from the variable analysis since having the same field ID in the incoming and outgoing contexts means that the field cannot have been reassigned like variables. The last case follows since either through starting packed or through getting reassigned, all permission changes reflected in the outgoing context must have happened after that point. □

Corollary 3.12 (Typing Downgrades Variables)**.** *If* $P; \Delta, x : (\pi_1\ s_1; \Pi_1); \pi \vdash e : s \dashv \Delta', x : (\pi_2\ s_1; \Pi_2); \ell^*$*, then* $\pi_1 \downarrow \pi_2$.

3.3 Dynamic Semantics

We model execution of programs in our system using a standard heap-based semantics with an extra level of indirection to explicitly track permissions to aliases.

3.3.1 Heap

Our heap keeps explicit track of aliases to objects so that we can track the permissions of the different aliases and ensure they are compatible. Thus, our heap contains both objects O and object references o:

$$\begin{array}{rrcl} \textbf{Heap} & H & ::= & H, O \mapsto s\{\Xi\} \mid H, o \mapsto O \mid \mathsf{null} \\ \textbf{Field Map} & \Xi & ::= & \Xi, f \mapsto o \mid \varnothing \end{array}$$

Allocated objects O appear in the heap mapped to a state s and a field map Ξ which maps fields f to object references. Each object reference o is mapped to an object O. The heap also always includes the distinguished object reference null which is used for uninitialized fields.

3.3.2 Intermediate Expressions

We extend the expression language with to include partially executed forms and add object references into the class of values:

$$\begin{array}{rrcl} \textbf{Expressions} & e & ::= & \ldots \mid \texttt{alias}(o) \mid \texttt{bind}\ o\ \texttt{in}\ e \mid \texttt{alias}(o).f \mid \texttt{alias}(o).f{=}e \\ \textbf{Values} & v & ::= & \ldots \mid o \end{array}$$

First, we have a `bind` form indicating that o replaced the bound variable in the body e and that after e the permissions in o should be returned to their source location. The form $\texttt{alias}(o)$ gives computational meaning to alias creation. Field references can appear with $\texttt{alias}(o)$ as the target expression.

3.3.3 Evaluation Contexts

We use standard evaluation contexts to specify in what subexpressions reduction can occur:

$$\begin{array}{lll} \textbf{Eval Ctx} & \mathrm{E} & ::= \quad \square \mid \texttt{let}\ \pi\ x{=}E\ \texttt{in}\ e \mid \texttt{bind}\ o\ \texttt{in}\ E \mid \{E;\ (e;)^*\} \mid \\ & & \quad\ \texttt{alias}(o).f{=}E \mid E.m(e) \mid o.m(E) \mid \\ & & \quad\ \texttt{match}\ (E)\ \{\texttt{(case}\ s_i \Rightarrow e_i)^+\ \} \end{array}$$

3.3.4 Reduction Rules

Expressions are executed in the context of a heap and produce an updated heap and expression.

- E-CONGRUENCE: a step is taken in a subexpression as defined by the evaluation context
- E-NEW: to execute a `new` expression, we create a new object O and object reference o. The fields of O are initially all mapped to null, but the resulting expression includes a sequence that assigns each field to its initialization expression. Thus, each field will be non-null before it is ever accessed in a well-typed program as the field initializer must typecheck without access to the object.
- E-LET: `let` expressions that have an object reference o as the bound expression reduce to a `bind` expression with the bound variable x replaced with $\texttt{alias}(o)$ in the body e. Using the `alias` form instead of the raw o ensures that any uses of x in e will create new aliases of o and allow us to track the permissions separate from the original alias.
- E-BIND: Here we simply remove the `bind` once the body is reduced to an object reference.
- E-MATCH: To reduce `match`, we can replace it with the expression case corresponding the the first case state which the matched object is in a substate of.
- E-ALIAS: An $\texttt{alias}(o)$ expression reduces to a fresh object reference o' that is mapped to the same object O in H as the original object reference o.
- E-FIELD: Like an alias, executing a field creates a new object reference that aliases the object mapped to by the field f of the object of o.
- E-ASSIGN: Assignment updates the field mapping of f to the provided object reference o' and then generates a new alias of the same object as the result of the reduction.
- E-INVOKE: A method call gets turned into a `bind` expression binding both the receiver and parameter object reference with the body the method body with `this` and x replaced by `alias` expressions of the respective object references.
- E-SEQUENCE-*: Reducing a sequence proceeds by removing each expression in turn as it is reduced to an object reference until it can return the result of the final expression.

$$\boxed{P \vdash H; e \to H; e}$$

E-CONGRUENCE
$$\frac{P \vdash H; e \to H'; e'}{P \vdash H; E[e] \to H'; E[e']}$$

E-NEW
$$\frac{\mathsf{fields}(P, s) = \{(\pi_i\ s_i\ \ f_i{=}e_i)_{i\in[1,n]}\} \qquad o, O \notin \mathsf{dom}(H) \qquad H' = H, o \mapsto O, O \mapsto s\{(f_i \mapsto \mathbf{null})_{i\in[1,n]}\}}{P \vdash H; \texttt{new}\ \ s \to H'; \{(\texttt{alias}\,(o)\ .\, f_i = e_i\,;)_{i\in[1,n]}\ \ o;\}}$$

E-LET
$$\frac{}{P \vdash H; \texttt{let}\ \ \pi_1\ \ x = o\ \ \texttt{in}\ \ e \to H; \texttt{bind}\ \ o\ \ \texttt{in}\ \ e[x \leftarrow \texttt{alias}(o)]}$$

E-BIND
$$\frac{}{P \vdash H; \texttt{bind}\ \ o\ \ \texttt{in}\ \ o' \to H; o'}$$

E-MATCH
$$\frac{H = H', o \mapsto O \qquad O \mapsto s\{\Xi\} \in H' \qquad P \vdash s \preceq s_j \qquad \neg\exists k \in [1, j-1].(P \vdash s \preceq s_k)}{P \vdash H; \texttt{match}\ \ (o)\ \ \{(s_i \Rightarrow e_i)_{i\in[1,n]}\} \to H; e_j}$$

E-ALIAS
$$\frac{o' \notin \mathsf{dom}(H) \qquad H = H', o \mapsto O}{P \vdash H; \texttt{alias}(o) \to H, o' \mapsto O; o'}$$

E-FIELD
$$\frac{H = H', o \mapsto O, O \mapsto s\{\Xi, f \mapsto o'\}, o' \mapsto O' \qquad o'' \notin \mathsf{dom}(H) \qquad H'' = H, o'' \mapsto O'}{P \vdash H; \texttt{alias}\,(o)\ .\, f \to H''; o''}$$

E-ASSIGN
$$\frac{H; \texttt{alias}\,(o') \to H'; o'' \qquad H' = H'', o \mapsto O, O \mapsto s\{\Xi, f \mapsto o'''\} \qquad H''' = H'', o \mapsto O, O \mapsto s\{\Xi, f \mapsto o''\}}{P \vdash H; \texttt{alias}\,(o)\ .\, f{=}o' \to H'''; o'}$$

E-INVOKE
$$\frac{o \mapsto O, O \mapsto s\{\Xi\} \in H \qquad \pi_r\ s_r\ \ m\,(\pi_2 {>>} \pi_2'\ \ s_2\ \ x)\ \ \pi_1 {>>} \pi_1'\ \ \{\ \ e\ \ \} \in \mathsf{methods}(P, s) \qquad e' = \texttt{bind}\ \ o\ \ \texttt{in}\ \ \texttt{bind}\ \ o'\ \ \texttt{in}\ \ e[\texttt{this} \leftarrow \texttt{alias}(o), x \leftarrow \texttt{alias}(o')]}{P \vdash H; o.m(o') \to H; e'}$$

E-SEQUENCE-SINGLE
$$\frac{}{P \vdash H; \{o;\} \to H; o}$$

E-SEQUENCE-MULTIPLE
$$\frac{}{P \vdash H; \{o;\ \ (e;)^+\} \to H; \{(e;)^+\}}$$

3.4 Safety

We will prove the safety of our system using standard progress and preservation proofs. Before we state and prove the theorems, we introduce typing rules for intermediate expressions and state lemmas for key properties used in the proof.

3.4.1 Intermediate Typing Rules

In order to provide correct source locations for object references, we extend the context to *return specifications* of the form $o \Rightarrow \ell$, mapping object reference o to location ℓ where o will return its permissions after its scope.

Linear Context $\Delta \quad ::= \quad \ldots \mid \Delta, o \Rightarrow \ell$

$$\textsc{T-ObjRef1}\quad \frac{\Delta = \Delta', o : (\pi\ s; \varnothing), o \Rightarrow \ell \qquad \texttt{unique} \Rrightarrow \pi \otimes \pi'}{P; \Delta; \pi \vdash o : s \dashv \Delta'; \ell}$$

$$\textsc{T-ObjRef2}\quad \frac{\nexists\, \ell.o \Rightarrow \ell \in \Delta \qquad \texttt{unique} \Rrightarrow \pi \otimes \pi'}{P; \Delta, o : (\pi\ s; \varnothing); \pi \vdash o : s \dashv \Delta; \varnothing}$$

$$\textsc{T-Bind1}\quad \frac{o : (\pi'\ s'; \Pi) \in \Delta \qquad P; \Delta; \pi \vdash e : s \dashv \Delta_1, o : (\pi''\ s'; \varnothing), o \Rightarrow \ell; \ell^*, \hat{\ell}^* \qquad \Delta_1 \vdash \ell^*\ \mathbf{ok} \qquad \Delta_1; \pi'' \vdash \ell \dashv \Delta'}{P; \Delta; \pi \vdash \texttt{bind}\ o\ \texttt{in}\ e : s \dashv \Delta'; \ell^*}$$

$$\textsc{T-Bind2}\quad \frac{(\pi'\ s'; \Pi) \in \Delta \qquad \nexists\, \ell.o \Rightarrow \ell \in \Delta \qquad \Delta; \pi \vdash e : s \dashv \Delta', o : (\pi''\ s'; \varnothing); \ell^*, \hat{\ell}^* \qquad \Delta' \vdash \ell^*\ \mathbf{ok}}{\Delta; \pi \vdash \texttt{bind}\ o\ \texttt{in}\ e : s \dashv \Delta'; \ell^*}$$

$$\textsc{T-Alias}\quad \frac{\pi \Rrightarrow \pi' \otimes \pi''}{\Delta, o : (\pi\ s; \varnothing); \pi' \vdash \texttt{alias}(o) : s \dashv \Delta, o : (\pi''\ s; \varnothing); o}$$

$$\textsc{T-Alias-Field-Access}\quad \frac{x \notin \mathsf{dom}(\Delta) \qquad P; \Delta, x : (\pi\ s, \Pi); \pi' \vdash x\,.\,f : s' \dashv \Delta, x : (\pi\ s, \Pi'); \ell^*}{P; \Delta, o : (\pi\ s, \Pi); \pi' \vdash \texttt{alias}\,(o)\ .\ f : s' \dashv \Delta, o : (\pi\ s, \Pi'); \ell^*[x \leftarrow o]}$$

$$\textsc{T-Alias-Field-Assign}\quad \frac{x \notin \mathsf{dom}(\Delta) \qquad P; \Delta, x : (\pi\ s, \Pi); \pi' \vdash x\,.\,f{=}e : s' \dashv \Delta', x : (\pi'\ s, \Pi'); \varnothing}{P; \Delta, o : (\pi\ s, \Pi); \pi' \vdash \texttt{alias}\,(o)\ .\ f{=}e : s' \dashv \Delta', o : (\pi'\ s, \Pi'); \varnothing}$$

- (T-OBJREF1/2): When we type an object reference o, we require that it have exactly the permission we are looking for, which itself must be able to be pulled from `unique` (making sure object references cannot be typed to a `borrow` permission), that it be packed, and we do not include it in the outgoing context. The intuition for this is that the object reference was created explicitly for one of four purposes: to be substituted for a let bound variable (E-LET), to be assigned into a field (E-ASSIGN), to be used in a method invocation (E-INVOKE), or discarded in a sequence (E-SEQUENCE-MULTIPLE). Removing it from the context ensures that it does not appear anywhere else in the expression which might impact its ability to perform these functions (see lemma 3.17 which relies on this property). We also discard any return specification pertaining to the bound variable. The SLL consists of the specified return location for o in the context if it exists and is empty otherwise.

- (T-BIND1/2): We type a `bind` similar to a `let` except that the expression is already bound. Thus all we do is type the body to the needed expression and then return the permission remaining in the bound object reference o to its return location if specified. Since the life-span of o is over, we can safely remove it along with its return specification from the context for typing future expressions.

- (T-ALIAS): An alias is typed just like a variable, splitting the permission from the specified object reference and using it as the return specification.

- (T-ALIAS-FIELD-*): Typing the field of an alias proceeds just like the field of a variable. We reuse the variable typing rules by choosing a fresh variable to substitute in temporarily.

A note on the decision to remove object references and return specifications from the context following the typing of an object reference or exiting a `bind` scope. The problem with leaving

them in, even with a `none` permission, is that it would allow expressions that later restore permission to the object reference such that it could be used again. Removing the object reference from the context means that our proofs do not have to consider this case.

We also have two extra equalizing rules. The rules T-EQ-FRESH-FIELD-ID* are used in the proof of safety as `match` statements are reduced away and these rules are pushed through the typing.

T-EQ-FRESH-FIELD-ID

$$\frac{\mathsf{fresh}(j) \quad P;\Delta;\pi \vdash e : s \dashv \Delta', v : (\pi'\ s'; \Pi, f : (\pi'', i)); \ell^*}{P;\Delta;\pi \vdash e : s \dashv \Delta', v : (\pi'\ s'; \Pi, f : (\pi'', j)); \ell^*}$$

T-EQ-FRESH-FIELD-ID-PRESERVERET

$$\frac{\mathsf{fresh}(j) \quad P;\Delta;\pi \vdash e : s \dashv \Delta', v : (\pi'\ s'; \Pi, f : (\pi'', i)); \ell^*, (v\,.\,f, i)}{P;\Delta;\pi \vdash e : s \dashv \Delta', v : (\pi'\ s'; \Pi, f : (\pi'', j)); \ell^*, (v\,.\,f, j)}$$

3.4.2 Properties of typing intermediate expressions

We introduce several lemmas that will be useful in proving preservation. First, we characterize what additions can be made to the context without impacting the typing derivation.

Lemma 3.13 (Typing in an Extended Context)**.** *Given $P;\Delta;\pi \vdash e : s \dashv \Delta'; \ell^*$ where Δ contains only mappings of the form $x : (\pi'\ s'; \Pi')$ (variables only), then for all $\hat{\Delta}$, $P;\Delta,\hat{\Delta};\pi \vdash e : s \dashv \Delta', \hat{\Delta}; \ell^*$.*

Proof. By straightforward induction on the typing derivation. Since the original context contains no object references, we know that none of the intermediate typing rules may apply. All other typing rules act only on variables in the context, so weakening the context with new information will not impact the typing or the new mappings. □

Lemma 3.14 (Context Weakening)**.** *If $P;\Delta;\pi \vdash e : s \dashv \Delta'; \ell^*$, then $P;\Delta, v : (\pi_1\ s_1; \Pi);\pi \vdash e : s \dashv \Delta', v : (\pi_1\ s_1; \Pi); \ell^*$.*

Proof. By straightforward induction on the typing derivation. Since v was not there originally, we cannot even return to it. □

Other changes to the context impact the derivation in a very predictable way.

Lemma 3.15 (Typing with Substate)**.** *If $P;\Delta, v : (\pi_1\ s_1; \Pi_1);\pi \vdash e : s \dashv \Delta'; \ell^*$ and $P \vdash s_1' \preceq s_1$, then $P;\Delta, v : (\pi_1\ s_1'; \Pi_1);\pi \vdash e : s' \dashv \Delta'; \ell^*$ and $P \vdash s' \preceq s$.*

Proof. By straightforward induction on the typing derivation. The T-OBJREF1/2, T-VARIABLE, and T-ALIAS cases are those in which the state may become more specific. Everything else can use the induction hypothesis. □

Lemma 3.16 (Typing With Stronger Permission)**.** *Given $P;\Delta, v : (\pi_1\ s_1; \Pi_1);\pi \vdash e : s \dashv \Delta', v : (\pi_2\ s_1; \Pi_2); \ell^*$ where $\pi_1 \downarrow \pi_2$. If $\hat{\pi}_1 \downarrow \pi_1$, then $P;\Delta, v : (\hat{\pi}_1\ s_1; \Pi_1);\pi \vdash e : s \dashv \Delta', v : (\hat{\pi}_2\ s_1; \Pi_2); \ell^*$ where $\hat{\pi}_2 \downarrow \pi_2$.*

Proof. By straightforward induction on the typing derivation. □

The reduction of `let` bindings substitutes `alias` expressions for variables in the body. We prove as a lemma that given a typing derivation before this step, one exists afterwards as well in the appropriately modified context.

Lemma 3.17 (Alias Substitution)**.** *Given* $P; \Delta, x : (\pi_1\ s_1; \varnothing); \pi \vdash e : s \dashv \Delta', x : (\pi_1'\ s_1; \varnothing); \ell^*$. *If* $o \notin \mathit{dom}(\Delta)$*, then for* $\hat{\Delta} = \varnothing$ *or* $\hat{\Delta} = o \Rightarrow \ell$*, we have* $P; \Delta, o : (\pi_1\ s_1; \varnothing), \hat{\Delta}; \pi \vdash e[x \leftarrow \texttt{alias}(o)] : s \dashv \Delta', o : (\pi_1'\ s_1; \varnothing), \hat{\Delta}; \ell^*[x \leftarrow o]$.

Proof. By structural induction on the typing rules. The only interesting rule is T-VARIABLE since that is where the substitution happens in both the expression and the SLL, but it is straightforward since we instead use the rule T-ALIAS which mirrors T-VARIABLE. □

We can also prove several lemmas regarding the outputs of a typing derivation. In particular, bindings cannot be added to the context, SLLs are well-formed, and return specifications cannot be self-referential.

Lemma 3.18 (Mappings Not Added)**.** *Given* $P; \Delta; \pi \vdash e : s \dashv \Delta'; \ell^*$.

- *If* $o \Rightarrow o' \in \Delta'$*, then* $o \Rightarrow o' \in \Delta$.
- *If* $v \in \mathit{dom}(\Delta')$*, then* $v \in \mathit{dom}(\Delta)$.

Proof. By straightforward induction on the typing derivation as no rules add return specifications, object references, or variables. □

Lemma 3.19 (Source Location Lists Are Well-formed)**.** *If* $P; \Delta; \pi \vdash e : s \dashv \Delta'; \ell^*$ *and* $\vdash \Delta$ **ok***, then* $\Delta \vdash \ell^*$ **ok**.

Proof. By straightforward induction on the typing derivation, using lemma 3.20 that an object reference cannot return to itself for the cases T-OBJREF1/2. □

Lemma 3.20 (Objects Cannot Return to Themselves)**.** *Given* $o = \ell \in \Delta$ *with* $\ell = (o' \,.\, f, o'')$ *or* $\ell = o'$*. If* $P; \Delta; \pi \vdash \texttt{bind o in } e : s \dashv \Delta'; \ell^*$ *then* $o \neq o'$.

Proof. By straightforward induction on the typing derivation. The last non-equalizing rule used must have been T-BIND1. By inversion on this rule, we know that the return succeeded in a context in which o did not appear. Therefore, o' must appear in the context and be distinct from o. □

While the rule to type an object reference o is straightforward, the presence of equalizing rules in our system means that a typing derivation for o might be more complicated than the single rule. However, there are still strict limits on the changes from the input to the output as characterized by the following lemma.

Lemma 3.21 (Effects of Object Reference Typing)**.** *If* $P; \Delta, v : (\pi_1\ s_1; \Pi_1); \pi \vdash o : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$ *then* $v \neq o$ *and* $\pi_1 \downarrow \pi_1'$*. Furthermore,*

1. *If* $\pi_1 \neq \pi_1'$*, then* $\Pi_1 = \varnothing$.

2. *If* $\Pi_1' = \varnothing$*, then* $\Pi_1 = \varnothing$ *and there exists no* f, i *such that* $(v \,.\, f, i) \in \ell^*$.

3. *If* $f : (\pi_f', i) \in \Pi_1'$*, then either* **packed**(f, Π_1) *or* $f : (\pi_f, j) \in \Pi_1$ *where* $\pi_f \downarrow \pi_f'$ *where either* $j = i$ *or* **fresh**(j).

4. *If* $v \in \ell^*$*, then* $\pi_1 \Rrightarrow \pi \otimes \hat{\pi}_1$ *where* $\hat{\pi}_1 \downarrow \pi_1'$.

5. *If* $(v \,.\, f, i) \in \ell^*$*, then letting* **cur-field-perm**$(P; s_1; f; \Pi_1) = \pi_f$ *and* **cur-field-perm**$(P; s_1; f; \Pi_1') = \pi_f'$*, then* $\pi_f \Rrightarrow \pi \otimes \hat{\pi}_f$ *where* $\hat{\pi}_f \downarrow \pi_f'$.

Proof. By straightforward induction on the typing derivation which can only include equalizing rules and T-OBJREF1/2. v cannot be o since o will not appear in the outgoing context. Only equalizing rules can update the permission of v since it remains in the context after typing, therefore it can only be reduced. For the first additional requirement, since the equalizing rules do not allow permissions to be pulled once the reference is unpacked, we must have v at least started unpacked if its permission was changed. The second follows from the fact that fields cannot be packed by equalizing rules, so permissions must decrease and only one field id can appear in both the input and output context. Third, since equalizing rules can not pack fields, then if v ends packed, then it also starts packed. Rules that add fields of v to the SLL must unpack v, so cannot apply. We are allowed to update the field ID of an unpacked field, so that can chance. For the forth, we know that T-EQ-PULL-VAL must be applies on v at some point. Regardless of the order or pulls or drops from v by lemma 3.7 we know that π can be pulled from the initial permission of v. For the fifth, we use the same ideas as the third. □

At certain points during the execution of a program, permissions may be returned to a location in the context. This occurs when `bind` expressions are reduced away, so we can consider only the case when we are typing an object reference o. The following lemma shows that if we can return a permission to a value v in the context before and after typing o, then there exists a typing derivation with the updated input and output permissions for v.

Lemma 3.22 (Object Reference Typing Change)**.** *Given* $P; \Delta, v : (\pi_1\ s_1; \Pi_1); \pi \vdash o : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$. *If* $\hat{\pi} \otimes \pi_1 \Rrightarrow \hat{\pi}_1$ *and* $\hat{\pi} \otimes \pi_1' \Rrightarrow \hat{\pi}_1'$*, then* $P; \Delta, v : (\hat{\pi}_1\ s_1; \Pi_1); \pi \vdash o : s \dashv \Delta', v : (\hat{\pi}_1'\ s_1; \Pi_1'); \ell^*$

Proof. We know that $v \neq o$, so thus, only equalizing rules apply to v in the derivation. We can remove all rules pulling permissions from v from the derivation to get $P; \Delta, v : (\pi_1\ s_1; \Pi_1); \pi \vdash o : s \dashv \Delta', v : (\pi_1\ s_1; \Pi_1'); \hat{\ell}^*$ We can then update the permission for π_1 to $\hat{\pi}_1$ which will give a derivation of $P; \Delta, v : (\hat{\pi}_1\ s_1; \Pi_1); \pi \vdash o : s \dashv \Delta', v : (\hat{\pi}_1\ s_1; \Pi_1'); \hat{\ell}^*$. If $\hat{\pi}_1 = \hat{\pi}_1'$, then we are done. Otherwise, it must be the case that $\pi_1 \neq \pi_1'$. By lemma 3.21 we know that $\pi_1 \downarrow \pi_1'$ and also $\Pi_1 = \varnothing$. Applying lemma 3.4 to conclude that $\hat{\pi}_1 \neq \hat{\pi}_1'$. Now, we know that we can add instances of equalizing rules to pull permissions from v before the equalizing rules that unpack v. If $v \in \ell^*$, then we use lemma 3.21 clause 4 to find that $\hat{\pi}_1 \Rrightarrow \pi \otimes \tilde{\pi}_1$ and $\tilde{\pi}_1 \downarrow \hat{\pi}_1'$. In this case we add T-EQ-PULL-VAL and if $\tilde{\pi}_1 \neq \hat{\pi}_1'$ T-EQ-DROPPERM-VAL. Otherwise, we just use T-EQ-DROPPERM-VAL and we are done. □

Removing a `bind` expression also removes the point from the typing where the bound object reference o' and its return specification are returned. The following lemma proves that at this point, typing the body o can proceed without o' in the context.

Lemma 3.23 (Removing Object References)**.** *If* $P; \Delta, \hat{\Delta}, o' : (\pi_1\ s_1; \varnothing); \pi \vdash o : s \dashv \Delta', \hat{\Delta}, o' : (\pi_1'\ s_1; \varnothing); \ell^*, \hat{\ell^*}$ *where* $o' \notin \ell^*$*,* $\hat{\ell^*} = \varnothing$ *or* $\hat{\ell^*} = o'$*, exists no* ℓ_1 *such that* $o' \Rightarrow \ell_1 \in \Delta$ *and* $\hat{\Delta} = o' \Rightarrow \hat{\ell}$ *or* $\hat{\Delta} = \varnothing$*, then* $P; \Delta; \pi \vdash o : s \dashv \Delta'; \ell^*$*.*

Proof. By induction on the derivation of the typing which can only include equalizing rules and T-OBJREF1/2, case analyzing on the last rule used.

1. T-OBJREF1/2: Since $e = o$, typing does not depend on o', so removing it from the context has no impact on the typing other than removing o' from the output context as well. Furthermore, $\hat{\ell^*} = \varnothing$, so nothing more is needed for that. The rule also does not use $o' \Rightarrow \ell$ since $o \neq o'$ so if $\hat{\Delta} = o' \Rightarrow \ell$, then it can be removed too.

2. T-EQ-*: In each case we can apply the induction hypothesis on the premise $P; \Delta, o' : (\pi_1\ s_1; \varnothing); \pi \vdash o : s \dashv \Delta'', o' : (\pi_1''\ s_1; \varnothing); \ell_1^*, \hat{\ell_1^*}$ with $o' \notin \ell_1^*$ and $\hat{\ell_1^*} = \varnothing$ or $\hat{\ell_1^*} = o'$, to get $P; \Delta; \pi \vdash o : s \dashv \Delta''; \ell_1^*$. $\hat{\Delta}$ can always be removed because equalizing rules never use return specifications. Now, we have different cases for each specific rule.

   case: T-EQ-PULL-VAL
   Then $\ell^* = \ell_1^*, o''$. If $o'' \neq o'$, then the rule's premises do not require o', so it applies with o' removed from the context. If $o'' = o'$, then we can remove this rule since it does not impact ℓ^* or the context other than o'.

   case: T-EQ-PULL-FIELD
   Then $\ell^* = \ell_1^*, (o''.f, \hat{o})$. If $o'' \neq o'$, then the rule's premises do not require o', so it applies with o' removed from the context. o'' cannot be o' since we know that o' is packed in the final context and applying this rule would force it to be unpacked.

   case: T-EQ-DROPPERM-VAL
   There there exists some $v : (\pi_2\ s_2; \varnothing)$ in the input context where $v : (\pi_2'\ s_2; \varnothing)$ is in the output context and $\pi_2 \downarrow \pi_2'$. If $v \neq o'$, the the rule applies with o' removed from the context. Otherwise, we can simply remove the rule since the rule does not impact any part of the output except the mapping of o' in Δ. $\ell^* = \ell_1^*$ so there is nothing more to prove.

   case: T-EQ-DROPPERM-FIELD-*
   There there exists some $v : (\pi_2\ s_2; \Pi_2)$ in the input context where $v : (\pi_2\ s_2; \Pi_2')$ is in the output context and $\Pi_2 \neq \Pi_2'$. If $v \neq o'$, the the rule applies with o' removed from the context. v cannot be o' since either rule would leave o' unpacked which we know not to be the case. $\ell^* = \ell_1^*$ so there is nothing more to prove.

case: T-EQ-FRESH-FIELD-ID*
Since o' is packed, then these rules cannot apply to it, so we are done.

case: T-EQ-DROPRET
Then $\ell^* = \hat{\ell}^*, \ell$ for some ℓ. We know from 3.21 that $\ell \neq (o' \,.\, f, i)$ for any f, i since o' is packed. If $\ell = o'$, then $\hat{\ell}^*_1 = o'$, so we do not need this rule anymore. If neither is the case then it does not depend on o' in the context.

case: T-EQ-SUPERSTATE
This rule does not need any information from the context.

□

Finally, we can characterize the typing of object references appearing inside an evaluation context.

Lemma 3.24 (Object Reference in an Evaluation Context)**.** *Given* $\Delta = \hat{\Delta}, \Delta_m$ *where* $o : (\pi\ s; \Pi) \in \hat{\Delta}$. *If* $P; \Delta; \pi' \vdash E[o] : s' \dashv \Delta'; \ell^*$, *then* $P; \hat{\Delta}; \pi \vdash o : s'' \dashv \hat{\Delta}'; \hat{\ell}^*$. *In other words,*

1. $\texttt{unique} \Rrightarrow \pi \otimes \hat{\pi}$,
2. $\Pi = \varnothing$, *and*
3. $o \notin \mathsf{dom}(\Delta')$.

Proof. By straightforward induction on the structure of $E[o]$. At each point, typing the hole is the first this that happens, which will either by o or a recursive definition $E'[o]$. The only exception is the form $o' \,.\, m(E)$ in which case the only effect on the context is to remove o'. As this can be done an arbitrary number of times before we type o, we allow the context to shrink in this way. When we do eventually type o, we can get the extra information by inversion on the typing rule. □

3.4.3 Environment Invariants

Safety in this system hinges on never begin able to gain a linear `unique` permission to an object for which there there exist other aliases with non-`none` permissions. The key goal of our system is to track returns under the covers and allow `unique` permissions to be safely regained. Our invariants are designed to ensure that each step preserves this property and they focus heavily on ensuring that if the typing tells us that a return succeeds, then it succeeds in our tracking and preserves the correct accessibility. This is especially tricky for fields because unlike variables, the permission of the field in the context is decoupled from the permission of the field representative.

For our purposes, we consider an environment to include a program P, a context Δ, a needed permission π, a heap H, and an expression e. We define judgments that define the consistency of different parts of this environment.

Useful sets and predicates:

We define here several useful notations used to summarize concepts in the rules below.

Permission sets: We have two important classes of permissions that we will refer to below. The first is the set $LINEAR$, which contains all permissions that through any series of joins may end up at `unique`. Thus, it includes `unique` itself, along with any `borrow` permissions that have `unique` as the original permission. We define $LINEAR$ itself as the union of the set restricted to `borrow` permissions of a single σ:

$$LINEAR(\sigma) = \texttt{unique} \cup \{\texttt{borrow}(\texttt{unique}, \sigma, n) \mid n \geq 1\}$$

$$LINEAR = LINEAR(\texttt{shared}) \cup LINEAR(\texttt{immutable})$$

The second is the set $LOCAL(\sigma)$, which includes `local` σ and all `borrow` permissions with `local` σ as the original permission.

$$LOCAL(\sigma) = \texttt{local}\ \sigma \cup \{\texttt{borrow}(\texttt{local}\ \sigma, \sigma, n) \mid n \geq 0\}$$

Object reference permission class: We also define a predicate characterizing the permission of an object reference in a context as local. $\mathsf{local}(\Delta; o; \sigma)$ says that o is mapped to a permission from the set $LOCAL(\sigma)$ in Δ.

$$\mathsf{local}(\Delta; o; \sigma) \iff o : (\pi\ s; \Pi) \in \Delta \wedge \pi \in LOCAL(\sigma)$$

Heap and context returns: A return specification $o \Rightarrow \ell$ provides both the location in the context and in the heap where permissions are returned. In the case that ℓ is an object reference, those locations coincide. However for fields $\ell = (o'\,.\,f, o'')$, $o'\,.\,f$ is the location in Δ and o'' is the location in H. In this case we also need to check H to verify that o and o' actually point to the same location as a field assignment may have invalidated the heap return. The predicate $\mathsf{heapRet}(\Delta; H; o; o')$ to captures this

$$\mathsf{heapRet}(\Delta; H; o; o') \iff o \Rightarrow o' \in \Delta \vee (o \Rightarrow (o''\,.\,f, o') \in \Delta \wedge o \mapsto O, o' \mapsto O \in H)\,.$$

Similarly, an object reference o returns to the context when its return specification points to another object reference, or a field location that still exists in the context.

$$\mathsf{ctxRet}(\Delta; o; o') \iff o \Rightarrow o' \in \Delta \vee (o \Rightarrow (o''\,.\,f, o') \in \Delta \wedge o'' : (\pi\ s; \Pi, f : (\pi_f, o')))$$

Restore borrow: The count n of a permission $\texttt{borrow}(\pi_o, \sigma, n)$ records how many outstanding splits of a `local` σ there are from this permission. However, for other bookkeeping reasons, the number is interpreted differently depending on the original permission π_o. We include a predicate $\mathsf{restoreOrig}$ that abstracts away this difference.

$$\text{restoreOrig}(\texttt{borrow}(\pi_o, \sigma, n), n') \iff (\pi_o = \texttt{unique} \wedge n' = n) \vee (\pi_o = \texttt{local}\ \ \sigma \wedge n' = n{+}1)$$

Valid object permissions:

There are two requirements for the set of permissions to a single object to be valid. First, the list of permissions must be consistent. Second, the return structure amongst the aliases must be well-formed.

Consistent permission lists: Given a context Δ and a heap H, we can collect all of the permissions to aliases of a given object O into a single list $[\pi^*]$. We simply recurse over the heap finding all object reference o which point to O and record the permission of o from Δ.

$$\textsc{Perms-No-Match}\ \frac{o \mapsto O \notin H}{\text{perms}(\Delta, H, O) = []} \qquad \textsc{Perms-Match}\ \frac{H = o \mapsto O, H' \qquad o : \pi \in \Delta \qquad \text{perms}(\Delta, H', O) = [\hat{\pi}^*]}{\text{perms}(\Delta, H, O) = [\pi] +\!\!+ [\hat{\pi}^*]}$$

Now we can define what a consistent permission list, $[\pi^*]$ **consistent** looks like. We have three cases.

1. (C-UNIQUE): if there is a `unique` permission in the list, then all other permissions must be `none`.
2. (C-BORROWUNIQUE): If there is a $\texttt{borrow}(\texttt{unique}, \sigma, n)$ permission in the list, then there is the possibility that the linear `unique` permission may be regained at a later point. We honor this by ensuring that all non-linear permission are temporary and have the same underlying symmetric permission. Finally, there must be exactly the right number of `local` permissions needed to restore all `borrow` permissions such that only the `unique` remains.
3. (C-GENERAL): If no $\texttt{borrow}(\texttt{unique}, \sigma, n)$ is in the list, then there is no chance that the `unique` permission can be regained, so we allow any permissions so long as all non-linear permissions have the same underlying symmetric permission.

$$\boxed{[\pi^*]\ \textbf{consistent}} \qquad \textsc{C-UNIQUE}\ \frac{}{[(\texttt{none})^*] +\!\!+ [\texttt{unique}]\ \textbf{consistent}}$$

$$\textsc{C-BORROWUNIQUE}\ \frac{L = [(\texttt{local}\ \sigma)^*] \qquad B = [(\text{borrow}(\texttt{local}\ \sigma, \sigma, n_i))_{i \in [1,m]}] \qquad m \geq 0 \qquad n' + \sum_{i=1}^{m} n_i = |L|}{[(\texttt{none})^*] +\!\!+ L +\!\!+ B +\!\!+ [\text{borrow}(\texttt{unique}, \sigma, n')]\ \textbf{consistent}}$$

$$\textsc{C-GENERAL}\ \frac{L = [(\texttt{local}\ \sigma)^*] \qquad B = [(\text{borrow}(\texttt{local}\ \sigma, \sigma, n_i))^*]}{[(\texttt{none})^*] +\!\!+ L +\!\!+ [(\sigma)^*] +\!\!+ B\ \textbf{consistent}}$$

Downgrade, Splitting, and Joining preserve permission list consistency.

Lemma 3.25 (Preservation of Permission List Consistency)**.** *Given* $[\pi^*]{+}{+}[\pi_1]{+}{+}[\pi_2]$ **consistent**.

1. *(Pull) If* $\pi_2 \Rrightarrow \pi \otimes \pi_2'$*, then* $[\pi^*]{+}{+}[\pi_1]{+}{+}[\pi_2']{+}{+}[\pi]$ **consistent**.
2. *(Join) If* $\pi_1 \downarrow \pi_1'$ *and* $\pi_1' \otimes \pi_2 \Rrightarrow \pi$*, then* $[\pi^*]{+}{+}[\pi]{+}{+}[\texttt{none}]$ **consistent**.
3. *(Downgrade) If* $\pi_2 \downarrow \pi_2'$ *and* $\texttt{unique} \Rrightarrow \pi_2' \otimes \hat{\pi}$*, then* $[\pi^*]{+}{+}[\pi_1]{+}{+}[\pi_2']$ **consistent**

Proof. By straightforward analysis on the permission splitting and joining rules. The downgrade in the second clause is allowed because if a `unique` or `local` σ is downgraded to a `borrow` permission, then the return will fail. In all other cases the downgrade is not problematic. The requirement that the newly downgraded permission can be pulled from `unique` ensures that `borrow` permission cannot be created that would invalidate the `local` permission accounting. □

Valid return structure: Simply having a consistent list of permissions using rule C-BORROWUNIQUE does not ensure that permissions can be returned correctly to allow the `unique` permission to be regained. This requirement gives rise to three additional restrictions on every object reference o with permission π which maps to object O in the heap. First, any object reference o' that o returns to must also be mapped to O in the heap. Also, if another object reference o' with permission π in the set $LOCAL(\sigma)$ returns to o in the heap, then π cannot be `unique`. Finally, if $\pi = \texttt{borrow}(\pi_o, \sigma, n)$, then the number of object references with permissions from $LOCAL(\sigma)$ that return to o in the heap must be exactly the number of joins of `local` σ permissions to π needed to restore the original permission.

OBJ-PERMS-VALID

$$\frac{\begin{array}{c}\mathsf{perms}(\Delta; H; O)\ \mathbf{consistent} \\ \forall o \mapsto O \in H.o : (\pi\ s; \Pi) \in \Delta \wedge \left(\begin{array}{l} (\mathsf{heapRet}(\Delta; H; o; o') \implies o' \mapsto O \in H) \wedge \\ (\mathsf{heapRet}(\Delta; H; o'; o) \wedge \mathsf{local}(\Delta, o', \sigma) \implies (\pi \neq \texttt{unique}) \wedge \\ (\pi = \texttt{borrow}(\pi_o, \sigma, n) \implies \mathsf{restoreOrig}(\pi, |\{o' | \mathsf{heapRet}(\Delta; H; o'; o) \wedge \mathsf{local}(\Delta, o', \sigma)\}|)) \end{array} \right)\end{array}}{\Delta; H \vdash O\ \mathbf{validPerms}}$$

The **validPerms** property is independent of other invariants, and so we can prove that it is preserved by the important manipulations of permissions in the system, including splitting, joining, and downgrading.

Lemma 3.26 (Aliasing Preserves **validPerms**)**.** *Given* $\Delta; H \vdash O$ **validPerms** *with* $o \mapsto O \in H$ *and* $\Delta = \Delta'', o : (\pi\ s; \varnothing)$*. If* $\pi \Rrightarrow \pi' \otimes \pi''$*,* $\mathsf{fresh}(o')$*,* $\Delta' = \Delta'', o : (\pi''\ s; \varnothing), o' : (\pi'\ s; \varnothing), o' \Rrightarrow \ell$*, and* $H' = H, o' \mapsto O$ *where* $\mathsf{heapRet}(\Delta'; H'; o'; o)$*, then* $\Delta'; H' \vdash O$ **validPerms**.

Proof. We updated o and added o' to the list of aliases of O. By lemma 3.25, we know that permission list consistency is maintained since the updated permission are defined by a pull operation on an existing permission to O.

We check that the three requirements hold for added object reference o'. The first holds since the we specified o' to return o which mapped to the object we chose to map o' to. The second holds trivially since o' is fresh which means that there cannot be any object references which return to it. The third is also trivially true since the permission π' of o' was successfully pulled from another permission which means that it cannot be a `borrow` permission.

Finally, we check that the three requirements hold for updated object reference o. The first holds by the assumption since we did not update any existing return specifications or object reference mappings. The second holds by assumption if $\pi' \neq \texttt{local}\ \sigma$ since the pull either does not create or change a `borrow` permission or it removes the `borrow` using rule P-SPLIT-BORROW-SYMMETRIC replacing it with a σ which would mean the implication still holds. In the case that $\pi' = \texttt{local}\ \sigma$ then since o' returns to o and has a $\texttt{local}\ \sigma$ permission the premise is true, but by the definition of permission split π'' is of the form $\texttt{borrow}(\pi_o, \sigma, n)$ which means that the conclusion is also true. The final requirement is also true by assumption in the case that $\pi' \neq \texttt{local}\ \sigma$ for the same reason, and if $\pi' = \texttt{local}\ \sigma$ is true because adding o' increased the number of object references returning to o with a local permission and pulling the $\texttt{local}\ \sigma$ from π must result in π'' with a borrow count increased by 1, or a new `borrow`, either of which will preserve restoreOrig. □

Lemma 3.27 (Return Preserves **validPerms**). *Given* $\Delta; H \vdash O$ **validPerms** *with* heapRet$(\Delta; H; o_1; o_2)$ *and* $o_1 \mapsto O, o_2 \mapsto O \in H$, $\Delta = \Delta', o_1 : (\pi_1\ s_1; \Pi_1), o_2 : (\pi_2\ s_2; \Pi_2)$. *If* $\pi_1 \downarrow \pi_1'$ *and* $\pi_1' \otimes \pi_2 \Rrightarrow \hat{\pi}_2$, *then we have* $\Delta', o_1 : (\texttt{none}\ s_1; \Pi_1), o_2 : (\hat{\pi}_2\ s_2; \Pi_2); H \vdash O$ **validPerms**.

Proof. By lemma 3.25, we know that permission list for O with π_1 and π_2 is consistent. Therefore, downgrading π_1 to π_1' and returning it results in a consistent permission list for the output. Since we updated the permission of o_1 and o_2, we need to check that the other three requirements are preserved. In both cases, the first is trivially maintained since no updates are made to return specifications.

For o_1, the second and third requirements trivially because the conclusion of the second is true and the premise of the third is false.

For o_2, we need to check that the updated permission $\hat{\pi}_2$ still satisfies the requirements. If $\pi_1' = \texttt{none}$, then no change occurs and so we are done. Otherwise, we continue by case analysis on π_2:

case: $\pi_2 = \texttt{borrow}(\pi_o, \sigma, n)$:
π_1' must have been $\texttt{local}\ \sigma$ for the return to have succeeded, which implies that $\pi_1 = \texttt{local}\ \sigma$ as well. By the assumption of validPerms, we know that there are exactly the right number of reference that return to o_2 and have local permissions to restore the original permission. o_1 must be one of these. Therefore, after the join, n has been reduced by 1 and o_1 no long has a local permission, so the last implication is maintained, or $\hat{\pi}_2$ is no longer a `borrow`, in which case it is trivially true. In this later case though, there actually cannot be any more such o_i since we started with exactly the right number, so the premise of the second implication is false. Otherwise, the premise is still true, as is the conclusion.

case: $\pi_2 = \sigma$

After the return, we know that $\hat{\pi}_2 = \sigma$ as well, so no changes are made that can have an impact on the implications for this object reference, so the assumption suffices.

case: otherwise
The returned permission cannot have been local, so there has not been any chances in local permissions in the system meaning the implications cannot have changed. □

Lemma 3.28 (Downgrade Preserves **validPerms**). *Given* $\Delta, o : (\pi\ s; \Pi); H \vdash O$ **validPerms** *with* $o \mapsto O \in H$ *and* $\pi \downarrow \pi'$ *with* $\pi \neq \pi'$. *If* $\texttt{unique} \Rrightarrow \pi' \otimes \hat{\pi}$, *then* $\Delta, o : (\pi'\ s; \Pi); H \vdash O$ **validPerms**.

Proof. By lemma 3.25, we know that permission list consistency is maintained. Since `borrow` permission cannot be pulled and π' must be distinct from π, we know that π cannot be in $LOCAL(\sigma)$ or $\texttt{borrow}(\texttt{unique}, \sigma, n)$ for any σ and n. Therefore, we have not broken the requirements of any object references that return to o. Furthermore, since we know that the new permission for o is not `unique` or `borrow`, its requirements are trivially maintained as well because the conclusion of the second is true and the premise of the third is false. □

Valid object:

A valid return structure is only part of what makes an Object well-formed in an environment. In particular, we also need to check that the fields of the object have valid representatives. Each object O in H is mapped to a state $\hat{s}$ and a field map Ξ. We require that for every field $\pi\ s\ f = e_f$ from fields$(P, \hat{s})$ f is mapped to a representative object reference o_f. This object reference can either be null or mapped to permission π' and state s' in Δ. In the later case it fulfills the following requirements:

1. The declared permission π can be downgraded to the representative permission π'
2. The declared state s must be a superstate of the representative state s'.
3. Either any object references that return to this field in the context also return to this field representative in the heap, or the current permission of the field representative π' is in $LINEAR$ and the set of object references that return to this field in the heap and are local (by **validPerms** must be exactly the number needed to regain `unique`) is a subset of those object references that return to the field in the context and are local.
4. If the field is declared to be `unique` and there exists an alias of O that has a permission from the set $LINEAR$, then π' must also come from $LINEAR$.

The first two requirements are straightforward. The third requirement is needed to help keep fields and their representatives in sync. Either returns happen in both places or we allow some extra local permissions to be outstanding in the context (this accounts for reassignment of `unique` fields in objects with a permission from $LOCAL(\texttt{shared})$). The fourth requirement stems from the

fact that a `unique` permission can be taken from a `unique` field of a `unique` reference. Thus, if an alias of O has a `unique` permission or could have one later in the execution, then the field representative must also be able to have a `unique` permission to make the permission available when it could be pulled. Other invariants make this more precise.

We capture these requirements formally with the following judgment:

$$\boxed{P; \Delta; H \vdash O \ \mathbf{ok}}$$

OBJ-ID-OK

$$\frac{\begin{array}{c}\Delta; H \vdash O \ \mathbf{validPerms} \qquad O \mapsto \hat{s}\{\Xi\} \in H \\ \forall \pi\ s\ f \in \mathsf{fields}(P, \hat{s}).f \mapsto \mathsf{null} \in \Xi \vee \left(\begin{array}{l} f \mapsto o \in \Xi \wedge o : (\pi'\ s'; \Pi) \in \Delta \wedge \pi \downarrow \pi' \wedge P \vdash s' \preceq s \wedge \\ \left(\begin{array}{c} (\forall o'.\mathsf{ctxRet}(\Delta; o'; o) \implies \mathsf{heapRet}(\Delta; H; o'; o)) \vee \\ \left(\pi' \in LINEAR \wedge \left(\begin{array}{c} \{o' | \mathsf{heapRet}(\Delta; H; o', o) \wedge \mathsf{local}(\Delta, o', \sigma)\} \subseteq \\ \{o' | \mathsf{ctxRet}(\Delta; H; o', o) \wedge \mathsf{local}(\Delta, o', \sigma)\} \end{array} \right) \right) \end{array} \right) \wedge \\ (\pi = \texttt{unique} \wedge \exists \hat{\pi} \in \mathsf{perms}(\Delta; H; O).\hat{\pi} = \texttt{borrow}(\texttt{unique}, \sigma, n) \implies \pi' \in LINEAR) \end{array} \right)}{P; \Delta; H \vdash O \ \mathbf{ok}}$$

Field fulfillment:

One requirement for a valid object reference is that the permission of each field must be fulfilled by the permission of the representative object reference. A permission π_1 *fulfills* a field permission $\pi_2 \,.\, \pi_3$ if any permission that can be split from $\pi_2 \,.\, \pi_3$ can also be pulled from π_1. In order to ensure that a `unique` access can be regained if needed, we require that if the object permission and its field permission are in the set $LINEAR$, then so is the fulfilling permission.

$$\boxed{\pi \rhd \pi \,.\, \pi}$$

FIELD-FULFILL

$$\frac{\begin{array}{c}\forall \pi \in \{\hat{\pi} | \pi_2 \,.\, \pi_3 \Rrightarrow \hat{\pi} \otimes \hat{\pi}'\}.\pi_1 \Rrightarrow \pi \otimes \pi' \\ \pi_2 \in LINEAR \wedge \pi_3 \in LINEAR \implies \pi_1 \in LINEAR \end{array}}{\pi_1 \rhd \pi_2 \,.\, \pi_3}$$

We can identify several specific cases where we know that field fulfillment holds:

Lemma 3.29 (Fulfillment of `none`)**.** *For all π and π', we have $\pi \rhd \texttt{none} \,.\, \pi'$.*

Proof. Since we cannot pull any permissions from fields of an object with a `none` permission, any permission can approximate any of its fields and $\texttt{none} \notin LINEAR$ so the implication holds trivially. □

Lemma 3.30 (Same Permission Fulfills)**.** *For all π and π' we have $\pi \rhd \pi' \,.\, \pi$.*

Proof. By lemma 3.3, we know that if a permission can be pulled from a field, it can be pulled from the raw permission in the field. The implication for $LINEAR$ holds since the field and its rep are the same. □

Lemma 3.31 (`unique` Permission Fulfills)**.** *For all π and π' we have $\texttt{unique} \rhd \pi' \,.\, \pi$.*

Proof. Any permission that can be split can be split from `unique` and `unique` $\in LINEAR$. □

Furthermore, the common manipulations of permissions that we care about also maintain field fulfillment.

Lemma 3.32 (Object Permission Downgrade Preserves Fulfillment)**.** *If* $\pi \rhd \pi_1 \,.\, \pi_2$ *and* $\pi_1 \downarrow \pi_1'$*, then* $\pi \rhd \pi_1' \,.\, \pi_2$.

Proof. By straightforward case analysis on the possible combinations. Reducing the permission to the owning object only reduces the set of permission that could be pulled from the field and therefore fulfillment is maintained. π does not change and π_1' cannot become $LINEAR$, so the implication is also preserved based on the assumption. □

Lemma 3.33 (Field Permission Return Preserves Fulfillment)**.** *If* $\pi \rhd \pi_1 \,.\, \pi_2$ *and* $\hat{\pi} \otimes \pi \Rrightarrow \pi'$*, then* $\pi' \rhd \pi_1 \,.\, \pi_2$.

Proof. Follows from the fact that returning gives a stronger permission from which can be pulled anything that could be pulled from the weaker permission. If $\pi \in LINEAR$, then the return cannot take it out of the set $LINEAR$ so the implication is maintained. □

Lemma 3.34 (Pull Preserves Field Fulfillment)**.** *Given* $\pi_3 \rhd \pi_1 \,.\, \pi_2$*. If* $\pi_1 \,.\, \pi_2 \Rrightarrow \pi \otimes \pi_2'$ *and* $\pi_3 \Rrightarrow \pi \otimes \pi_3'$*, then* $\pi_3' \rhd \pi_1 \,.\, \pi_2'$*. Furthermore, if* $[\pi^*]{+}{+}[\pi_1]$ **consistent***, then for all* $\hat{\pi} \in \pi^*$ *If* $\pi_3 \rhd \hat{\pi} \,.\, \hat{\pi}_2$ *then* $\pi_3' \rhd \hat{\pi} \,.\, \hat{\pi}_2$.

Proof. By straightforward exhaustive case analysis on the permissions. For the condition on fields of other consistent permissions, we only need to consider `unique` permissions because they are the only fields that can really get unpacked. It follows from the fact that if there is a single `unique` permission to the object, in which case fulfillment of the fields through the other `none` references is trivial via lemma 3.29. Otherwise, the only permissions that can be pulled from a `unique` field are σ or `local` σ. In either case, pulling one from the field does not prevent more from being pulled, so field fulfillment through all other aliases is maintained. We also need to worry about the $LINEAR$ implication. However, if π_3 starts $LINEAR$ then it can only become non-linear in the following situations

1. $\pi_1 =$ `unique` and $\pi =$ `unique` or $\pi = \sigma$: In this case π_2' cannot be linear either so the implication is maintained.

2. $\pi_1 = \sigma$ and $\pi = \sigma$: But then $\pi_1 \notin LINEAR$, so the implication holds trivially.

In each case there cannot be any other permission in the list that is in $LINEAR$, so the requirement for the other object references holds as well. □

Lemma 3.35 (Alias Fulfills)**.** *Given* $\pi_1 \rhd \pi_2 \,.\, \pi_3$*. If* $\pi_2 \Rrightarrow \pi \otimes \pi_2'$*, then* $\pi_1 \rhd \pi \,.\, \pi_3$.

Proof. By exhaustive case analysis on the permissions. Many of the cases are taken care of by lemma 3.32. Splitting off a `local` σ permission can occur from a permission that does not downgrade to a `local` σ. However, in all cases the permissions that can be pulled from the fields do not increase. The last trivial case is $\pi =$ `none` which comes from lemma 3.29. π can be in $LINEAR$ only if π_2 is, so that means that the implication must be preserved as well. □

Valid object reference:

We also have well-formedness conditions on object references appearing in Δ. Given $o : (\pi\ s; \Pi) \in \Delta$, we require that o is mapped to object O in the heap, and O itself has type s' and field map Ξ. Based on this we have several requirements:

1. The state s from the context should be a superstate of the state s' from the heap.
2. If o has permission `none`, then it is packed ($\Pi = \varnothing$).
3. If field f is unpacked in Π with object reference o' as a field id, then f is mapped to o' in Ξ.
4. For each field $\pi_1\ s_1\ f = e_f$ declared in s, f is mapped to an object reference o_f in the field map Ξ of O. Furthermore, either o_f is null and f is unpacked in Π to permission `none`, or the following are true given the current permission π_1' of the field through the type of o in Δ:
   (a) The declared permission of the field π_1 downgrades to π_1'.
   (b) The permission of o_f fulfills the field permission $\pi\,.\,\pi_1'$.
   (c) If the current permission in the field is $\texttt{borrow}(\pi_o, \sigma, n)$, then the context contains exactly the right number of object references o' that return to this field and are local in terms of their permission.

$$\boxed{P; \Delta; H \vdash o\ \mathbf{ok}}$$

OBJ-REF-OK

$$\frac{\begin{array}{c} o : (\pi\ s; \Pi) \in \Delta \qquad o \mapsto O, O \mapsto s'\{\Xi\} \in H \qquad P \vdash s' \preceq s \\ \pi = \texttt{none} \implies \Pi = \varnothing \qquad f : (\pi_f, o') \in \Pi \implies f \mapsto o' \in \Xi \\ \forall \pi_1\ s_1\ f \in \mathsf{fields}(P, s).\ \begin{pmatrix} f \mapsto \mathsf{null} \in \Xi \wedge \\ f : (\texttt{none}, i) \in \Pi \end{pmatrix} \vee \left(\begin{array}{c} \mathsf{cur\text{-}field\text{-}perm}(P; s; f; \Pi) = \pi_1' \wedge \pi_1 \downarrow \pi_1' \wedge \\ f \mapsto o_1 \in \Xi \wedge o_1 : (\pi_1''\ s_1'; \Pi') \in \Delta \wedge \pi_1'' \rhd \pi\,.\,\pi_1' \wedge \\ \left(\begin{array}{c} \pi_1' = \texttt{borrow}(\pi_o, \sigma, n) \implies \\ \mathsf{restoreOrig}(\pi_1', |\{o' | o' \Rightarrow (o\,.\,f, o_1) \in \Delta \wedge \mathsf{local}(\Delta, o', \sigma)\}|) \end{array} \right) \end{array} \right) \end{array}}{P; \Delta; H \vdash o\ \mathbf{ok}}$$

The second requirement ensures that fields cannot be unpacked if the reference is unusable. The third ensures that the unpacked state matches the reality in the heap which is needed to make sure that the field representative tracks the permission in the field appropriately. The final requirement makes sure that the permission of the field recorded in the type is valid and that the field representative accurately represents the field from the stand point that if the type system indicates that a permission can be pulled from or returned to the field, then the same is true of the underlying representation.

Distinct fields:

In the judgments above, we have mixed object references that represent fields and those that do not. However, when stating and proving preservation, we will need to keep these sets separate so as to ensure that updates to the permission of a local variable do not also update the permission of a

field. In other words, we need to enforce our model that each alias in the program, including local variables and field references, is represented by a separate object reference. The typing judgments that remove object references from the context after they appear in the expression enforce this property for local variables. The **distinctFields** judgment provides this guarantee. it requires that given a context Δ and an object reference o appearing in a field map Ξ in heap H

1. o must be null or mapped to a permission and packed in Δ, and
2. o must be null or distinct from all other object reference appearing in a field map in H.

$\boxed{\Delta \vdash H \ \mathbf{distinctFields}}$

$$\textsc{Distinct-Fields}\ \frac{\begin{array}{c} O \mapsto s\{\Xi, f \mapsto o\} \in H \implies (o = \mathsf{null} \vee o : (\pi\ s; \varnothing) \in \Delta) \\ \big(O \mapsto s\{\Xi, f \mapsto o\} \in H \wedge O' \mapsto s'\{\Xi', f' \mapsto o\} \in H\big) \implies \big(o = \mathsf{null} \vee (O = O' \wedge f = f')\big) \end{array}}{\Delta \vdash H\ \mathbf{distinctFields}}$$

Environment well-formed:

For an environment consisting of program P, context Δ, needed permission π, heap H, and expression e, we consider it to be well-formed when the following hold.

1. P can be typed.
2. The set of non-null object references mapped in Δ an H must be the same.
3. The context can be split in two pieces, Δ_f which proves that H has **distinctFields**, and Δ_e in which e can be typed with needed permission π.
4. All non-null object references (those in the context Δ) are well-formed.
5. All objects are well-formed.

$$\textsc{Env-ok}\ \frac{\begin{array}{cc} \vdash P & \{o \mid o \in \mathsf{dom}(H) \wedge o \neq \mathsf{null}\} = \{o \mid o \in \mathsf{dom}(\Delta)\} \\ \Delta = \Delta_e, \Delta_f & P; \Delta_e; \pi \vdash e : s \dashv \Delta'; \ell^* \qquad \Delta_f \vdash H\ \mathbf{distinctFields} \\ \forall o \in \mathsf{dom}(\Delta).\ (P; \Delta; H \vdash o\ \mathbf{ok}) & \forall O \in \mathsf{dom}(H).\ (P; \Delta; H \vdash O\ \mathbf{ok}) \end{array}}{P; \Delta; \pi \vdash H; e : s \dashv \Delta'; \ell^*}$$

3.4.4 Lemmas

There are some manipulations used in the proof of preservation that we use over and over again. First we create aliases which should be well-formed object references.

Lemma 3.36 (Alias Generation)**.** *Given*

1. $o \mapsto O \in H$
2. $\Delta = \Delta', o : (\pi\ s; \varnothing)$,
3. $P; \Delta; H \vdash o\ \mathbf{ok}$,
4. $o' \notin \mathsf{dom}(\Delta)$, *and*
5. $P \vdash s \preceq s'$.

If $\pi \Rrightarrow \pi' \otimes \hat{\pi}$, *then letting* $\hat{\Delta} = \Delta', o : (\hat{\pi}\ s; \varnothing), o' : (\pi'\ s'; \varnothing)$ *and* $\hat{H} = H, o' \mapsto O$, *we have* $P; \hat{\Delta}; H \vdash o'\ \mathbf{ok}$.

Proof. By 4 and 3, we know that the actual type of object O is a substate of s which is itself a substate of the s' the state of o'. By transitivity of substates this invariant holds for o'. o' starts packed and so the implications regarding the permission of o' and the unpacked state and field mappings in the heap are both trivially true. For the fields, we know that since o is packed that none of the fields of O can be mapped to null. Thus since both o and o' are packed, the only thing we have to check is that given that a field f with declared permissions π_f is fulfilled by field representative permission π'_f when the object permission is π ($\pi'_f \rhd \pi \,.\, \pi_f$), then $\pi'_f \rhd \pi' \,.\, \pi_f$. This follows from lemma 3.35. □

Second, we return permissions to the context. These returns always happen after arbitrary equalizing rules which must come after the return once the execution step for the return has been taken.

Lemma 3.37 (Move Return Forward)**.** *Given* $[\pi^*]{+}{+}[\pi_1]{+}{+}[\pi_2]$ **consistent** *where* $\pi_1 = \texttt{local}\ \sigma$ *implies* $\pi_2 \neq \texttt{unique}$. *If* $\pi_1 \downarrow \pi'_1$, $\pi_2 \downarrow \pi'_2$, *and* $\pi'_1 \otimes \pi'_2 \Rrightarrow \hat{\pi}'$, *then* $\pi'_1 \otimes \pi_2 \Rrightarrow \hat{\pi}$ *where* $\hat{\pi} \downarrow \hat{\pi}'$.

Proof. We case analyze on π'_1.

1. $\pi'_1 = \texttt{none}$: This case is trivial since returning none is a no-op and we know by assumption $\pi_2 \downarrow \pi'_2$.
2. $\pi'_1 = \texttt{unique}$: By the definition of downgrade, we know that π_1 is also unique. By consistency, $\pi_2 = \texttt{none}$ and again by the definition of downgrade, $\pi'_2 = \texttt{none}$. Thus $\hat{\pi} = \hat{\pi}' = \texttt{unique}$ and we are done since downgrade is reflexive.

3. $\pi_1' = \sigma$: By the definition of permission joining, we know that returning a σ always results in a σ, but is only possible when returning to σ or $\texttt{none}$. We know that π_2' is in that set. By the definition of downgrade, $\pi_1 \in \{\texttt{unique}, \sigma, \texttt{borrow}(\texttt{unique}, \sigma, n)\}$ and $\pi_2 \in \{\texttt{unique}, \sigma, \texttt{none}\}$. Regardless of which permission π_1 is by consistency the only valid choices for π_2 are $\texttt{none}$ or σ (cannot be σ in the case of $\texttt{borrow}$), so the return must succeed and we are done.

4. $\pi_1' = \texttt{local}\ \sigma$: By the definition of downgrade, we must have $\pi_1' = \texttt{local}\ \sigma$ as well. Thus, by our assumption, $\pi_2 \neq \texttt{unique}$. Since the return of a local permission succeeds, we know that $\pi_2 = \texttt{borrow}(\pi_o, \sigma, n)$ or $\pi_2 = \sigma$. In the first case, by the definition of downgrading $\pi_2' = \texttt{borrow}(\pi_o', \sigma, n')$ where $n' \geq n$. Returning a $\texttt{local}\ \sigma$ to each succeeds and also preserves the downgrading relationship. In the second case, by the definition of downgrading plus the fact that the return of a $\texttt{local}$ permission succeeded, we know that $\pi_2' = \sigma$ as well. Thus, $\hat{\pi} = \sigma = \hat{\pi}'$.

□

Moving Equalizing Rules

These lemmas deal with moving equalizing rules when the object reference they surround is reduced away.

Lemma 3.38 (Move Downgrade Value)**.** *Given* $P; \Delta, v : (\pi_1\ s_1; \Pi_1); \pi \vdash e : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$. *If* $\hat{\pi}_1 \downarrow \pi_1$*, then* $P; \Delta, v : (\hat{\pi_1}\ s_1; \Pi_1); \pi \vdash e : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$.

Proof. By straightforward induction on the typing derivation. In short, there is always a place to insert a equalizing rule to drop the permission before a return occurs (which could cause problems). □

Lemma 3.39 (Move Downgrade Field)**.** *Given* $P; \Delta, v : (\pi_1\ s_1; \Pi_1, f : (\pi_2; i)); \pi \vdash e : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$. *If* $\hat{\pi}_2 \downarrow \pi_2$*, then* $P; \Delta, v : (\pi_1\ s_1; \hat{\Pi}_1); \pi \vdash e : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$ *where* *cur-field-perm*$(P; s_1; f; \hat{\Pi}_1) = \hat{\pi}_2$.

Proof. By straightforward induction on the typing derivation. It is exactly the same as for values in the case that $f : (\pi_2', i) \in \Pi_1'$. Otherwise, we know that f must have been reassigned and therefore adding permissions at the start has no impact on either typing or the result. □

Lemma 3.40 (Move Pull Value)**.** *Given* $\pi_1 \Rrightarrow \pi \otimes \pi_1'$ *and* $\pi \downarrow \hat{\pi}$*. If* $P; \Delta, v : (\pi_1'\ s_1; \Pi_1); \pi_e \vdash e : s \dashv \Delta', v : (\pi_1''\ s_1; \Pi_1'); \ell^*$ *and* $P; \Delta'', v : (\pi_1''\ s_1; \Pi_1'); \hat{\pi} \vdash \hat{\ell}^*, v \dashv \Delta''', v : (\hat{\pi}_1\ s_1; \Pi_1')$*, then* $P; \Delta, v : (\pi_1\ s_1; \Pi_1); \pi_e \vdash e : s \dashv \Delta', v : (\hat{\pi}_1\ s_1; \Pi_1'); \ell^*$.

Proof. By straightforward induction on the typing derivation. The variable and alias cases follow from the commutativity of permission pulls (lemma 3.1) and the effect of splitting and pulling (lemma 3.5) which is used to prove that an instance of T-EQ-DROPPERM-VAL can be added to the existing derivation to ensure that we can make up for the extra permission to v that we started with. The rest of the cases are by the induction hypothesis because we can always revert the first outgoing context to the old output and use the assumption for the rest. □

Lemma 3.41 (Move Pull Field)**.** *Given* $\hat{\pi}_2 \Rrightarrow \pi \otimes \pi_2$ *where* $\pi_2 \neq \pi_2'$ *and* $\pi \downarrow \hat{\pi}$*. If* $P; \Delta, v : (\pi_1\ s_1; \Pi_1, f : (\pi_2, i)); \pi_e \vdash e : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$ *and* $P; \Delta'', v : (\pi_1'\ s_1; \Pi_1'); \hat{\pi} \vdash \hat{\ell}^*, (v\,.\,f, i) \dashv \Delta''', v : (\pi_1''\ s_1; \Pi_1'')$*, then* $P; \Delta, v : (\pi_1\ s_1; \hat{\Pi}_1); \pi_e \vdash e : s \dashv \Delta', v : (\pi_1''\ s_1; \Pi_1''); \ell^*$ *where*
cur-field-perm$(P; s_1; f; \hat{\Pi}_1) = \hat{\pi}_2$*.*

Proof. By straightforward induction on the typing derivation. Follows the same argument as the corresponding lemma for values in the case that $f : (\pi_2', i) \in \Pi_1'$ because the return will occur to the field. Otherwise, the return will be a no-op by either rule RESTORE-FIELD-STALE or RESTORE-FIELD-PACKED. Therefore, adding permission to the field at the start of the typing has no impact. □

Lemma 3.42 (Move Field Havok)**.** *If* $P; \Delta, o : (\pi_1\ s_1; \Pi_1, f : (\pi_2, i)); \pi \vdash e : s \dashv \Delta', v : (\pi_1'\ s_1; \Pi_1'); \ell^*$*, then* $P; \Delta, o : (\pi_1\ s_1; \Pi_1, f : (\pi_2, j)); \pi \vdash e : s \dashv \Delta', o : (\pi_1'\ s_1; \Pi_1'); \ell^*$

Proof. As with moving a downgrade, the important thing is to insert the field ID havok before any returns occur. However, we also need to ensure that we preserve the relationship between any returns in ℓ^* and the context. We need to worry about this because a field read could be the first thing that occurs. In this case, we can use the T-EQ-FRESH-FIELD-PRESERVERET rule to ensure that the return is still valid. □

The only case we have not yet handled is when we reduce the permission of an object reference that is then directly typed in the body. In this case, we can safely downgrade the permission of the object reference in the context without breaking the necessary invariants.

Lemma 3.43 (Remove Equalizing Rule By Updating Context)**.** *Given*

1. $P; \Delta, o : (\pi\ s; \Pi); \pi_1 \vdash o' : s_1 \dashv \Delta_1, o : (\pi'\ s; \Pi'); \ell_1^*$ *where* $\pi \neq \pi'$ *and* $\pi \downarrow \pi'$*,*
2. $P; \Delta_1, o : (\pi'\ s; \Pi'); \pi_2 \vdash E[o] : s_2 \dashv \Delta_2; \ell_2^*$*,*
3. $P; \Delta_2'; \pi_3 \vdash \ell_1^* \dashv \Delta_3$*, and*
4. $o \mapsto O \in H$*.*

If $P; \Delta, o : (\pi\ s; \Pi) \vdash o\ \mathbf{ok}$ *and* $P; \Delta, o : (\pi\ s; \Pi) \vdash O\ \mathbf{ok}$*, then*

I. $P; \Delta, o : (\pi'\ s; \Pi); H \vdash o\ \mathbf{ok}$*,*

II. $P; \Delta, o : (\pi'\ s; \Pi); H \vdash O\ \mathbf{ok}$*, and*

III. $P; \Delta, o : (\pi'\ s; \Pi); \pi_1 \vdash o' : s_1 \dashv \Delta_1, o : (\pi'\ s; \Pi'); \ell_1^*$*.*

Proof. By lemma 3.21, we know that $\Pi = \varnothing$. Furthermore, by lemma 3.24, we know that $\Pi' = \varnothing$ and $o \notin \mathsf{dom}(\Delta_2)$. Therefore, since the return succeeded, $o \notin \ell_1^*$ and also there are no elements of ℓ_1^* of the form $(o\,.\,f, o'')$. Therefore, we can construction a derivation of $P; \Delta, o : (\pi'\ s; \Pi); \pi_1 \vdash o' : s_1 \dashv \Delta_1, o : (\pi'\ s; \Pi'); \ell_1^*$ by eliminating any equalizing rules that update o.

To prove that $P; \Delta, o : (\pi'\ s; \Pi); H \vdash o\ \mathbf{ok}$, we only need to verify that all the fields are fulfilled. This is given by lemma 3.32, which tells us that downgrading the permission to the object does not impact field fulfillment. Since the object reference was already packed, we do not need to check that requirement. All other requirements are not impacted by changing the permission to o.

For $P; \Delta, o : (\pi'\ s; \Pi); H \vdash O\ \mathbf{ok}$, we need only to verify that $\Delta, o : (\pi'\ s; \Pi); H \vdash O\ \mathbf{validPerms}$ since the other requirements only pertain to field representatives. Lemma 3.24 tells us that $\texttt{unique} \Rrightarrow \pi' \otimes \hat{\pi}$ for some $\hat{\pi}$ so we can use lemma 3.28 to conclude that **validPerms** is maintained. □

Lemma 3.44 (Move Equalizing Rule for value not in output Context)**.** *Given*

1. $P; \Delta, o : (\pi\ s; \Pi); \pi_1 \vdash o' : s_1 \dashv \Delta_1, o : (\pi'\ s; \Pi'); \ell_1^*$ *where* $\pi \neq \pi'$, $\pi \downarrow \pi'$, *and for all* $f : (\pi_f, i_f) \in \Pi'$ *we have, letting* $\mathsf{cur\text{-}field\text{-}perm}(P; s; f; \Pi) = \pi'_f$, $\pi'_f \downarrow \pi_f$ *and* $\pi'_f \neq \pi_f$.
2. $P; \Delta_1, o : (\pi'\ s; \Pi'); \pi_2 \vdash e : s_2 \dashv \Delta_2; \ell_2^*$ *where* $o \notin \mathsf{dom}(\Delta_2)$,
3. $\nexists\ E$ *such that* $e = E[o]$
4. $P; \Delta'_2; \pi_3 \vdash \ell_1^* \dashv \Delta_3$, *and*
5. $o \mapsto O \in H$.

Then, exists E', e' *such that* $e = E'[e']$ *where* $P; \Delta, o : (\pi'\ s; \Pi'); \hat{\pi} \vdash e' : \hat{s} \dashv \hat{\Delta}, o : (\pi''\ s; \Pi'); \hat{\ell}^*$ *and* $\pi' \downarrow \pi''$. *Furthermore, there exists a derivation* $P; \Delta_1, o : (\pi\ s; \Pi); \pi_2 \vdash e : s_2 \dashv \Delta_2; \ell_2^*$.

Proof. By lemma 3.21, we know that $\Pi = \varnothing$. Since $o \notin \mathsf{dom}(\Delta_2)$ and the return succeeded, $o \notin \ell_1^*$ and also there are no elements of ℓ_1^* of the form $(o\,.\,f, o'')$. Therefore, we can summarize the effects of all equalizing rules that impact o in the derivation of $P; \Delta, o : (\pi\ s; \Pi); \pi_1 \vdash o' : s_1 \dashv \Delta_1, o : (\pi'\ s; \Pi'); \ell_1^*$ in terms of $\pi \downarrow \pi'$ where $\pi \neq \pi'$ and similar field pulls. Thus, if we can find a E' and e' as above, we can add an instances of T-EQ-DROPPERM-VAL followed by T-EQ-DROPPERM-FIELD to construct the new typing. Even if the permission has been downgraded, this is fine since downgrades can be moved after pulls by lemma 3.7.

We find this E' and e' by induction on the length of the derivation of 2. Since there exists no E such that $e = E[o]$, we will eventually find a sub-derivation where o is in the output context. If the permission is greater, then it must have been returned to in an inner scope, so we can keep recursing until inside the `bind` that performs the return. In this way, we will eventually find the E' and e' needed. □

Lemma 3.45 (Move Equalizing Rule)**.** *Given* Δ *and* $\hat{\Delta}$ *with* $\Delta \subseteq \hat{\Delta}$ *where*

1. $P; \Delta; \pi_1 \vdash o : s_1 \dashv \Delta_1; \ell_1^*$,
2. $P; \Delta_1, o : (\pi_1\ s_1; \varnothing); \pi \vdash e : s \dashv \Delta_2, o : (\pi'_1\ s_1; \varnothing); \ell_2^*$,
3. $P; \Delta_2; \pi'_1 \vdash \ell_1^* \dashv \Delta_3$,
4. $\forall o \in \mathsf{dom}(\hat{\Delta}).\ \big(P; \hat{\Delta}; H \vdash o\ \mathbf{ok}\big)$, *and*

5. $\forall O \in \mathsf{dom}(H).\, \big(P; \hat{\Delta}; H \vdash O\ \mathbf{ok}\big)$.

where the length of the derivation of 1. *is greater than* 1 *rule. Then there exists* $\Delta' \subseteq \Delta$*, such that there are derivations of*

I. $P; \Delta'; \pi_1 \vdash o : s_1' \dashv \Delta_1'; \hat{\ell}_1^*$

II. $P; \Delta_1', o : (\pi_1\ s_1'; \varnothing); \pi \vdash e : s \dashv \Delta_2', o : (\pi_1'\ s_1'; \varnothing); \ell_2^*$,

III. $P; \Delta_2'; \pi_1' \vdash \hat{\ell}_1^* \dashv \Delta_3$,

IV. $\forall o \in \mathsf{dom}(\hat{\Delta}).\, \big(P; \hat{\Delta}; H \vdash o\ \mathbf{ok}\big)$*, and*

V. $\forall O \in \mathsf{dom}(H).\, \big(P; \hat{\Delta}; H \vdash O\ \mathbf{ok}\big)$.

such that the derivation of I. *is strictly shorter than that of* 1.

Proof. Since the derivation of 1. is greater than 1 rule, we know that the last rule used must be an equalizing rule. We case analyze on this equalizing rule. In all cases except the last two, we do not change the context at all so D and E imply IV and V.

case: T-EQ-SUPERSTATE
By lemma 3.15, we know that we can substitute a substate in for the state of a variable in the context and get a typing derivation that is the same except for the output state of that value and the output state of the derivation, which may be a substate of the original. Since the value in our case is the one going out of scope, we can ignore it. If the updated derivation results in a substate, we can add instances of T-EQ-SUPERSTATE to the derivation to restore it to the original output.

case: T-EQ-FRESH-FIELD-ID
We cannot have the field impact o since o is packed. Then we can use lemma 3.42 to show what is needed.

case: T-EQ-FRESH-FIELD-ID-PRESERVERET
We cannot have updated a field of o since o is packed. Thus we can get rid of this rule because it is essentially a no-op.

case: T-EQ-DROPRET
This means that we have a derivation of $P; \Delta; \pi_1 \vdash o : s_1 \dashv \Delta_1; \ell_1^*, \ell$ for some ℓ. If there exists a first use of a rule of the form T-EQ-PULL-* that adds ℓ to the SLL backwards up the derivation tree, then we have two cases. First, if it is not the exactly previous rule, then we can reorder the equalizing rules, moving this use of T-EQ-DROPRET backwards until it is right before the paired T-EQ-PULL-* possibly with some instances T-EQ-FRESH-FIELD-ID-PRESERVERET in between. We then restart the case analysis on the newly exposed last rule. In the second case, we can either get rid of this series of rules, or turn them into a use of T-EQ-DROPPERM-*. If T-EQ-PULL-* leaves the permission of ℓ the same as before the pull, then removing each rule results in the same output. If T-EQ-PULL-* reduces the permission, then we can replace it with

T-EQ-DROPPERM-* since pulling implies downgrading by rule P-DOWN-SPLIT and remove the instance of T-EQ-DROPRET. In either case, we have reduced the length of the derivation but left the outputs the same.

If there exists no such rule of the form T-EQ-PULL-*, then we know that it must be the case that $\Delta = \Delta', o \Rightarrow \ell$. Δ' meets our criteria and then we can remove this use of T-EQ-DROPRET to make the derivation strictly shorter but with the same outputs.

case: T-EQ-PULL-VAL
By lemma 3.21, we know that this cannot apply to o, so the output for o is the same. The premise of the lemma gives that $v : (\pi_v\ s_v; \Pi_v) \in \Delta$ and $\pi_v \Rrightarrow \pi_1 \otimes \pi'_v$. The fact that the return succeeds tells us that $v \in \mathsf{dom}(\Delta_2)$, so lemma 3.40 tells us that we can adapt the derivation of the typing of e to incorporate the downgrade into it.

case: T-EQ-PULL-FIELD
Analogous to the previous case using lemma 3.41.

case: T-EQ-DROPPERM-VAL
By lemma 3.21, we know that this cannot apply to o, so the output for o is the same. The premise of the lemma gives that $v : (\pi_v\ s_v; \Pi_v) \in \Delta$ and $\pi_v \downarrow \pi'_v$. We have three cases: First, if $v \in \mathsf{dom}(\Delta_2)$, then we can can apply lemma 3.38 to find a derivation that removes the need for this pull and gives the same output permission for v without the return. Otherwise $v \notin \mathsf{dom}(\Delta_2)$ and $v = o'$ since variables cannot be removed from the context except within scopes where they were added. Our second case is if If $e = E[o']$ for some evaluation context E. then we can use 3.43 to remove this and all instances of equalizing rules that update o'. This shortens the derivation and also preserves well-formedness as needed. In the third case, we cannot find E such that $e = E[o']$, and so we instead use 3.44 without changing the context.

case: T-EQ-DROPPERM-FIELD-*
Analogous to the previous case using lemma 3.39 in the first case. The second case cannot occur because in that case o' could not be unpacked as this rule would make it. The third case is handled as in the last by 3.44. □

Corollary 3.46 (Equivalent Let)**.** *Given* $P; \Delta; \pi \vdash H; \texttt{let}\ \pi_1\ x\texttt{=}o\ \texttt{in}\ e : s \texttt{–} \Delta'; \ell^*$. *There exists* $\hat{\Delta}$ *and choices of* $\hat{\Delta}_e$ *and* $\hat{\Delta}_f$ *such that* $\Delta = \hat{\Delta}_e, \hat{\Delta}_f$, $P; \hat{\Delta}; \pi \vdash H; \texttt{let}\ \pi_1\ x\texttt{=}o\ \texttt{in}\ e : s \texttt{–} \Delta'; \ell^*$, *and the derivation of* $P; \hat{\Delta}_e; \pi_1 \vdash o : s_1 \dashv \Delta_1; \ell_1^*$ *is only a single rule long.*

Proof. By repeated application of lemma 3.45. Each application maintains all of the invariants as well as shortens the derivation of typing the let bound object reference o. □

Corollary 3.47 (Equivalent Sequence and Match)**.** *Given* $P; \Delta; \pi \vdash H; e : s - \Delta'; \ell^*$ *where* $e = \{o; e^+\}$ *or* $e = \texttt{match}\ (o)\ \{\texttt{case}\ s_i \Rightarrow e_i\}$ *There exists* $\hat{\Delta}$ *and choices of* $\hat{\Delta}_e$ *and* $\hat{\Delta}_f$ *such that* $\Delta = \hat{\Delta}_e, \hat{\Delta}_f$, $P; \hat{\Delta}; \pi \vdash H; e : s - \Delta'; \ell^*$, *and the derivation of* $P; \hat{\Delta}_e; \pi_1 \vdash o : s_1 \dashv \Delta_1; \ell_1^*$ *is only a single rule long.*

Proof. By inversion on either typing rule, we know $P; \Delta; \texttt{none} \vdash o : s_1 \dashv \Delta_1; \ell_1^*$ and $P; \Delta_1; \pi \vdash e' : s \dashv \Delta'; \ell^*$ where e' is the "rest" of the expression: the other expressions in the sequence

or the case. Since returning `none` is a no-op, we also know that $P; \Delta'; \texttt{none} \vdash \ell_1^* \dashv \Delta'$. By lemma 3.14 we can also add $o : (\texttt{none}\ s_1; \varnothing)$ to get $P; \Delta_1, o : (\texttt{none}\ s_1; \varnothing); \pi \vdash e' : s \dashv \Delta', o : (\texttt{none}\ s_1; \varnothing); \ell^*$ since we know that o cannot be mapped in Δ_1. Thus, we can invoke lemma 3.45 repeatedly. Each application maintains all of the invariants as well as shortens the derivation of typing the object reference o that comes first in the sequence. □

Corollary 3.48 (Equivalent Method Receiver)**.** *Given $P; \Delta; \pi \vdash H; o\,.\,m(e) : s - \Delta'; \varnothing$. There exists $\hat{\Delta}$ and choices of $\hat{\Delta}_e$ and $\hat{\Delta}_f$ such that $\Delta = \hat{\Delta}_e, \hat{\Delta}_f$ such that $P; \hat{\Delta}; \pi \vdash H; o\,.\,m(e) : s - \Delta'; \varnothing$ and the derivation of $P; \hat{\Delta}; \pi_1 \vdash o : s_1 \dashv \Delta_1; \ell_1^*$ is only a single rule long.*

Proof. By inversion on the typing rule, we know

1. $P; \Delta; \pi_1 \vdash o : s_1 \dashv \Delta_1; \ell_1^*$
2. $P; \Delta_1; \pi_2 \vdash e_1 : s_2 \dashv \Delta_2; \ell_2^*$
3. $P; \Delta_2; \pi_2' \vdash \ell_2^* \dashv \Delta_3$
4. $P; \Delta_3; \pi_1' \vdash \ell_1^* \dashv \Delta'$

Since we are guaranteed by the typing of the method and corollary 3.12 that $\pi_1 \downarrow \pi_1'$, and the fact that o cannot appear in Δ_1 that we can find a derivation of $P; \Delta_1, o : (\pi_1\ s_1; \varnothing); \pi_2 \vdash e_1 : s_2 \dashv \Delta_2, o : (\pi_1'\ s_1; \varnothing); \ell_2^*$. o also cannot appear in ℓ_2^*, so $P; \Delta_2, o : (\pi_1'\ s_1; \varnothing); \pi_2' \vdash \ell_2^* \dashv \Delta_3, o : (\pi_1'\ s_1; \varnothing)$. The use of lemma 3.45 can handle the extra return, so we use is to achieve the intended goal. □

3.4.5 Progress

The progress theorem for our system is straightforward.

Theorem 3.49 (Progress)**.** *If $P; \Delta; \pi \vdash H; e : s \dashv \Delta', \ell^*$, then either $e = o$ or $P \vdash H; e \to H'; e'$ or execution is stuck at a `match` for which no cases apply.*

Proof. By straightforward induction on the typing rules. □

3.4.6 Preservation

For preservation, we prove that a well-formed environment is preserved by taking a step.

Theorem 3.50 (Preservation)**.** *If $P; \Delta; \pi \vdash H; e : s \dashv \Delta'; \ell^*$ and $P \vdash H; e \to H'; e'$, then there exists $\hat{\Delta}$ such that $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\hat{\Delta})$ and $P; \hat{\Delta}; \pi \vdash H'; e' : s \dashv \Delta'; \ell^*$*

Proof. We start with the following information given from $P; \Delta; \pi \vdash H; e : s \dashv \Delta'; \ell^*$:

A) $\vdash P$

B) $\{o \mid o \in \mathsf{dom}(H) \wedge o \neq \mathsf{null}\} = \{o \mid o \in \mathsf{dom}(\Delta)\}$

C) $\Delta = \Delta_e, \Delta_f$

D) $P; \Delta_e; \pi \vdash e : s \dashv \Delta'; \ell^*$

E) $\Delta_f \vdash H$ **distinct fields**

F) $\forall o \in \mathsf{dom}(\Delta).\, (P; \Delta; H \vdash o\ \mathbf{ok})$

G) $\forall O \in \mathsf{dom}(H).\, (P; \Delta; H \vdash O\ \mathbf{ok})$

We continue by structural induction on the derivation of the typing from D, case analyzing on the last rule used, proving for each case that the above properties are preserved. A is always preserved since P never changes.

case T-NEW:

1. This rule only applies if $e = \texttt{new}\ \ s$, so we know by inversion on the typing rule that $\Delta' = \Delta_e$ and $\ell^* = \varnothing$.

2. We know that reduction rule E-NEW must apply, so by inversion on that rule we know

   (a) $\texttt{state}\ \ s\ \ \texttt{case of}\ \ s'\ \{\ \ (\pi_i\ \ s_i\ \ f_i = e_i)_{i \in [1,n]}\ \ M^*\ \ \} \in P$

   (b) $H' = H, O \mapsto s\{(f_i \mapsto \mathsf{null})_{i \in [1,n]}\}, o \mapsto O$ where $o, O \notin \mathsf{dom}(H)$

   (c) $e' = \{(\texttt{alias}(o)\,.\,f_i = e_i;)_{i \in [1,n]}\ o;\}$.

3. We have two cases for the updated context.

   (a) If n from 2a is 0 (there are no fields), then let $\hat{\Delta}_e = \Delta_e, o : (\pi\ s; \varnothing)$.

   (b) If n from 2a is at least 1, then let $\hat{\Delta}_e = \Delta_e, o : (\texttt{unique}\ s; \Pi)$ where for all $i \in [1, n]$ we have $f_i : (\texttt{none}, j_i) \in \Pi$ with the j_i fresh.

   In both cases, let $\hat{\Delta}_f = \Delta_f$ and $\hat{\Delta} = \hat{\Delta}_e, \hat{\Delta}_f$. Since we added a mapping for o regardless, the domain of $\hat{\Delta}$ is strictly larger than that of Δ as needed. In each case, we have added o to both $\hat{\Delta}$ and H', so B is maintained. F is also maintained since we did not update Δ_f and the only field mappings we added for O were null which are handled as a special case.

4. From A along with 2a we know that for all $i \in [1, n]$ $P \vdash \pi_i\ s_i\ f_i = e_i$. Thus $P; \varnothing; \pi_i \vdash e_i : s_i \dashv \varnothing; \ell_i^*$. Since each field can be typed in the empty context by lemma 3.13 we know $P; \hat{\Delta}_e; \pi_i \vdash e_i : s_i \dashv \hat{\Delta}_e; \ell_i^*$. Furthermore, by the definitions of T-ALIAS-ASSIGN and T-ASSIGN-UNPACKED (since each field starts unpacked), we know that $P; \Delta_e, o : (\pi\ s; \Pi_i, f_i : (\texttt{none}, j_i)); \pi_i \vdash \texttt{alias}(o)\,.\,f_i{=}e_i : s_i \dashv \Delta_e, o : (\pi\ s; \Pi_i); \varnothing$ packing field f_i.

5. By the typing rule T-OBJREF2, we have $P; \Delta_e, o : (\pi\ s; \varnothing); \pi \vdash o : s \dashv \Delta_e; \varnothing$.

6. In the case that state s has no fields, we know $e' = \{o;\}$, so by T-OBJREF2, typing requirement D is proven. Otherwise, we start with o having a `unique` permission and must downgrade it to π. This is possible since `unique` can downgrade to any other permission. Thus, we add an instance of rule T-EQ-DROPPERM-VAL to the derivation of the type of the

last assignment. Now, using the definition of the typing for sequences along with 4, we know that because each field gets packed in turn, $P; \hat{\Delta}_e; \pi \vdash \{(\texttt{alias}(o)\,.\,f_i{=}e_i : s_i)_{i\in[1,n]}\} : s_n \dashv \Delta_e, o : (\pi\ s; \varnothing); \varnothing$ leaving o packed. Then we can use 5 to get $P; \hat{\Delta}_e; \pi \vdash e' : s \dashv \Delta_e; \varnothing$ when o is added to the sequence. Therefore, the typing requirement D is satisfied.

7. To prove that F is maintained, we only need to consider o because the rest of the heap is not changed. For o, we have that o has the same state s in the context and heap. For the fields of s, each field is mapped to null in the heap, but they are also all unpacked to `none` in Π.

8. To prove that G is maintained, we only need to consider O because the rest of the heap is not changed. O is well-formed because it has only one object reference o and a list of a single, non-`borrow` permission is always consistent. Furthermore, all of the fields of O map to null.

case T-LET:

This rule only applies if $e = \texttt{let}\ \ \pi_1\ \ x{=}e_1\ \ \texttt{in}\ \ e_2$, so by inversion of the typing rule from F, we know

I) $P; \Delta_e; \pi_1 \vdash e_1 : s_1 \dashv \Delta_1; \ell_1^*$

II) $P; \Delta_1, x : (\pi_1\ s_1; \varnothing); \pi \vdash e_2 : s \dashv \Delta_2, x : (\pi_1'\ s_1; \varnothing); \ell^*, \hat{\ell^*}$

III) $\Delta_2 \vdash \ell^*$ **ok**

IV) $P; \Delta_2; \pi_1' \vdash \ell_1^* \dashv \Delta'$

We have two cases for the reduction rule:

- E-CONGRUENCE:

  1. By inversion on the reduction rule, we know that $P \vdash H; e_1 \to H'; e_1'$. Furthermore, we know by I along with our original assumption that $P; \Delta; \pi_1 \vdash H; e_1 : s_1 \dashv \Delta_1; \ell_1^*$ since all the other conditions do not involve the expression. The induction hypothesis guarantees that there exists $\hat{\Delta}$ such that $P; \hat{\Delta}; \pi_1 \vdash H'; e_1' : s_1 \dashv \Delta_1; \ell_1^*$ where $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\hat{\Delta})$.
  2. Inversion on 1 tells us that there exists $\hat{\Delta}_e$ such that $P; \hat{\Delta}_e; \pi_1 \vdash e_1' : s_1 \dashv \Delta_1; \ell_1^*$
  3. 2 with II, III, and IV prove that $P; \hat{\Delta}_e; \pi \vdash \texttt{let}\ \ \pi_1\ \ x{=}e_1'\ \ \texttt{in}\ \ e_2 : s \dashv \Delta'; \ell^*$.
  4. The induction hypothesis from 1 with 3 gives us $P; \hat{\Delta}; \pi \vdash H'; \texttt{let}\ \ \pi_1\ \ x{=}e_1'\ \ \texttt{in}\ \ e_2 : s \dashv \Delta'; \ell^*$ where $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\hat{\Delta})$.

- E-LET:

  1. By inversion on the reduction rule, we know
     (a) $e_1 = o$

(b) $e' = \texttt{bind } o \texttt{ in } e_2[x \leftarrow \texttt{alias}(o)]$.

2. By lemma 3.46, we know that we can update the context structure of the derivation to move all of the equalizing rules out of the typing of o. We assume this has occurred and use the same context variables.
3. With 1a, 2, and I we know that
   (a) $\Delta_e = \Delta'_e, \Delta''_e, o : (\pi_1\ s_1; \varnothing)$ with $o \Rightarrow \hat{\ell} \in \Delta'_e$ for no ℓ and $\Delta''_e = \varnothing$ or $\Delta''_e = o \Rightarrow \ell$,
   (b) $\Delta_1 = \Delta'_e$, and
   (c) If $o \Rightarrow \ell \in \Delta_e$, then $\ell^*_1 = \ell$, otherwise $\ell^*_1 = \varnothing$.
4. II with lemma 3.17 implies $P; \Delta'_e, \Delta''_e, o : (\pi_1\ s_1; \varnothing); \pi \vdash e_2[x \leftarrow \texttt{alias}(o)] : s \dashv \Delta_2, \Delta''_e, o : (\pi'_1\ s_1; \varnothing); (\ell^*, \hat{\ell}^*)[x \leftarrow o]$
5. By III and the meaning of a well-formed SLL, we know that x cannot have appeared in ℓ^* since it is not mapped in Δ_2. Thus, $\ell^*[x \leftarrow o] = \ell^*$. Thus, we still have $\Delta_2 \vdash \ell^*$ **ok** since Δ_2 also does not map o to a type.
6. Let $\hat{\Delta} = \Delta$ and choose $\hat{\Delta}_e = \Delta_e$. We can use 4 and 5 with either rule T-BIND1 or T-BIND2. T-BIND1 will be used in the case that $o \Rightarrow \ell \in \Delta_e$, in which case by 2c, we have $\ell^*_1 = \ell$, so we have from IV that $P; \Delta_2; \pi'_1 \vdash \ell \dashv \Delta'$ which means that all the premises are fulfilled. In case $\nexists\ \ell.o \Rightarrow \ell \in \Delta_e$, then by 3c we have $\ell^*_1 = \varnothing$. By rule RESTORE-EMPTY $\Delta' = \Delta_2$, so T-BIND2 applies. In either case, we have $P; \Delta_e; \pi \vdash e' : s \dashv \Delta'; \ell^*$, so D is maintained.
7. For the rest of the requirements, they are also true by our assumption since neither Δ nor H changed.

case T-BIND1/2: These rules only apply if $e = \texttt{bind } o \texttt{ in } e_1$, so by inversion on the typing rule from F, we know

I) $\Delta_e = \Delta'_e, \hat{\Delta}_e, o : (\pi_1\ s_1; \Pi)$ where $o \Rightarrow \ell \notin \Delta'_e$, and $\hat{\Delta}_e = \varnothing$ or $\hat{\Delta}_e = o \Rightarrow \ell$.

II) $P; \Delta_e; \pi \vdash e_1 : s \dashv \Delta_1, \Delta'_1, o : (\pi'_1\ s_1; \varnothing); \ell^*, \hat{\ell}^*$ with $\Delta'_1 = o \Rightarrow \ell$ if T-BIND1 used and $\Delta'_1 = \varnothing$ otherwise.

III) $\Delta_1 \vdash \ell^*$ **ok**

IV) $P; \Delta_1; \pi'_1 \vdash \ell \dashv \Delta'$ only if T-BIND1 used, otherwise $\Delta' = \Delta_1$.

We have two possibilities for reduction rules:

- E-CONGRUENCE:
  1. By inversion on the reduction rule, we know that $P \vdash H; e_1 \to H'; e'_1$. Furthermore, we know by II along with our original assumption that $P; \Delta; \pi \vdash H; e_1 : s_1 \dashv \Delta_1, o : (\pi'_1\ s_1; \varnothing); \ell^*, \hat{\ell}^*$ since all the other conditions do not involve the expression. The induction hypothesis guarantees that there exists $\hat{\Delta}$ such that

(a) $P; \hat{\Delta}; \pi_1 \vdash H'; e_1' : s \dashv \Delta_1, \hat{\Delta}_1, o : (\pi_1' \ s_1; \varnothing); \ell^*, \hat{\ell}^*$

(b) $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\hat{\Delta})$

2. 1a shows that there exists a $\hat{\Delta}_e$ such that $P; \hat{\Delta}_e; \pi_1 \vdash e_1' : s \dashv \Delta_1, \hat{\Delta}_1, o : (\pi_1' \ s_1; \varnothing); \ell^*, \hat{\ell}^*$. Lemma 3.18 tells us that $o : (\hat{\pi} \ s_1; \Pi) \in \hat{\Delta}_e$.
3. Lemma 3.18 tells us that if it was the case by II that $o \Rightarrow \ell \in \hat{\Delta}_1$, then $o \Rightarrow \ell \in \Delta_e$ and also it must then be the case that $o \Rightarrow \ell \in \hat{\Delta}_e$. Therefore, $o \Rightarrow \ell \in \Delta_e$ if and only if $o \Rightarrow \ell \in \hat{\Delta}_e$.
4. 2 and 3 along with III and IV prove that T-BIND1/2 still apply and provide the same outputs.

- E-BIND:

1. By inversion on the reduction rule, we know that
   (a) $e_1 = o'$
   (b) $e' = o'$
   (c) $H' = H$
2. I and II along with 3.21 give $\pi_1 \downarrow \pi_1'$ and $\Pi_1 = \varnothing$.
3. $o \Rightarrow \ell \in \Delta_1$ only if $o \Rightarrow \ell \in \Delta_e$ by lemma 3.18.
4. By lemma 3.23 with I and II, we know that $P; \Delta_e' \vdash o' : s \dashv \Delta_1; \ell^*, \hat{\ell}^*$.
5. To construct $\hat{\Delta} = \hat{\Delta}_e, \hat{\Delta}_f$, we case analyze on where o returns to:

   case: T-BIND2 - permissions not returned
   From 2 and 3, we know that $\Delta_e = \Delta_e', o : (\pi_1 \ s_1; \varnothing)$. Let $\hat{\Delta}_e = \Delta_e'$ and $\hat{\Delta}_f = \Delta_f, o : (\pi_1 \ s_1; \varnothing), \hat{\Delta}_1$. 4 gives us that D is maintained. E is maintained because adding elements to Δ_f does not add requirements, only adding fields to H. F and G are maintained since the heap and context do not change.

   case: T-BIND1 and $o \Rightarrow o'' \in \Delta$
   IV gives us $P; \Delta_1; \pi_1' \vdash o'' \dashv \Delta'$. By inversion on this rule, we know that $\Delta_1 = \Delta_1', o'' : (\pi_2 \ s_2; \Pi_2)$ and $\pi_1' \otimes \pi_2 \Rrightarrow \pi_2'$ and $o'' : (\pi_2' \ s_2; \Pi_2) \in \Delta'$. We know by lemma 3.18 that $\Delta_e' = \Delta_e'', o'' : (\hat{\pi}_2 \ s_2; \hat{\Pi}_2)$. By lemma 3.21, we know that $\hat{\pi}_2 \downarrow \pi_2$. Since $o \Rightarrow o'' \in \Delta$, we know that $\mathsf{heapRet}(\Delta; H; o; o'')$. We must have $o \mapsto O \in H$ since $P; \Delta; H \vdash o \ \mathbf{ok}$. It must also be the case that $P; \Delta; H \vdash O \ \mathbf{ok}$, which means that $\Delta; H \vdash O \ \mathbf{validPerms}$. This requires that $\pi_1 = \texttt{local} \ \sigma$ implies $\pi_2 \neq \texttt{unique}$. Also, since we know $\mathsf{heapRet}(\Delta; H; o; o'')$ we have $o' \mapsto O \in H$. Therefore, π_1 and $\hat{\pi}_2$ are both in the permission list for O which must be consistent. We can now apply lemma 3.37 to conclude that $\pi_1' \otimes \hat{\pi}_2 \Rrightarrow \hat{\pi}_2'$ where $\hat{\pi}_2' \downarrow \hat{\pi}_2$. This gives us enough to apply lemma 3.22 for o'' to conclude with 4 that $P; \Delta_e'', o'' : (\hat{\pi}_2' \ s_2; \hat{\Pi}_2) \vdash o' : s \dashv \Delta_1', o'' : (\pi_2' \ s_2; \Pi_2); \ell^*, \hat{\ell}^*$. Furthermore, we know that we must in fact have $\Delta_1', o'' : (\hat{\pi}_2' \ s_2; \Pi_2) = \Delta'$.

Thus, we let $\hat{\Delta}_e = \Delta'_e, o'' : (\hat{\pi}'_2\ s_2; \hat{\Pi}_2)$, $\hat{\Delta}_f = \Delta_f, o : (\pi_1\ s_1; \varnothing), \hat{\Delta}_1$, and $\hat{\Delta} = \hat{\Delta}_e, \hat{\Delta}_f$. This maintains D as shown and E as in the previous argument. We need to check that $P; \hat{\Delta}; H \vdash O$ **ok** since we changed the permissions. By lemma 3.27, we know that $\hat{\Delta}; H \vdash O$ **validPerms**. We need nothing else because we did not change any field representatives. Next we need to check that $P; \hat{\Delta}; H \vdash o$ **ok** since we updated its permission to `none`. We know that all fields are still fulfilled by lemma 3.29. The only other requirement now that its permission is `none` is that o must be packed, which we are given by 2. Finally, we need to check that $P; \hat{\Delta}; H \vdash o''$ **ok**. For this, we need to check that field fulfillment is maintained. We case analyze on the returned permission π'_1. We don't need to check $\pi'_1 =$ `none` because no changes are made.

(a) $\pi'_1 =$ `unique`: This implies that $\pi_1 =$ `unique` as well. Since we assumed that $P; \Delta; H \vdash o$ **ok**, we know that all of the fields of o were fulfilled. o was packed and o'' is as well since it must have had a `none` permission in Δ (by permission list consistency) and $P; \Delta; H \vdash o''$ **ok**. Thus, o'' looks exactly like o did in Δ and so fields must still be fulfilled for o'' as they were for o.

(b) $\pi'_1 = \sigma$: Since the return succeeded, then $\pi_2 =$ `none` or $\pi_2 = \sigma$ and in either case $\hat{\pi}'_2 = \sigma$. By the definition of downgrade π_1 must have been either `unique`, $\texttt{borrow}(\texttt{unique}, \sigma, n)$, or σ. Since all of these downgrade to σ, we know by lemma 3.32 that since the fields of o were fulfilled when o had permission π_1 that they are also fulfilled with permission σ and we are done.

(c) $\pi'_1 =$ `local` σ: We know from downgrades that $\pi_1 =$ `local` σ as well. Furthermore, since the return succeeded, we know that $\pi_2 = \sigma$ or $\pi_2 = \texttt{borrow}(\pi_o, \sigma, n)$. In the former case, there is nothing to prove since the permission is not updated. In the later case, we case analyze on $\hat{\pi}'_2$.

- $\hat{\pi}'_2 = \texttt{borrow}(\pi_o, \sigma, n-1)$: Then there is nothing to prove since decrementing the count of the borrow does not allow access of more permissions from fields, so we have it by the assumption.
- $\hat{\pi}'_2 = \pi_o$: If $\pi_o =$ `local` σ, then we have nothing to prove because the same permissions can be pulled from fields with a `borrow` or a local permission. If $\pi_o =$ `unique`, then we need to be sure that for any `unique` field f declared in state s_2, we have that they are either represented by a `unique` permission, or they are unpacked. Since the permission list of O is consistent and $\hat{\pi}'_2 =$ `unique` all other aliases of O must be packed and so f is only unpacked if o'' is unpacked. We also know that since $\pi_2 \in LINEAR$ that f must also be represented by a permission $\pi_f \in LINEAR$. Furthermore, $P; \Delta; H \vdash O$ **ok** tells us via **validPerms** that if $\pi_f = \texttt{borrow}(\texttt{unique}, \sigma, n)$ then there are exactly the right number of local permissions waiting to be returned to π_f. Finally, the number of local permissions waiting to be returned to the field in the context must be a superset of these permissions, implying that if π_f is a `borrow` permission, then f must be unpacked in o''. So $\pi_f =$ `unique` if and only if $f \notin \mathsf{dom}(\hat{\Pi}_2)$. All other field permissions don't need to be checked for fulfillment since we can always pull σ from σ.

case: T-BIND1 and $o \Rightarrow (o''.f, o''') \in \Delta$
We further case analyze on the typing rule used for $P; \Delta_1; \pi_1' \vdash (o''.f, o''') \dashv \Delta'$ given by IV:

(a) RESTORE-FIELD-PACKED: In this case, no update to the context is made. Therefore, we are free to ignore the return From 2 and 3, we know that $\Delta_e = \Delta_e', o : (\pi_1\ s_1; \varnothing), o \Rightarrow (o''.f, o''')$. Let $\hat{\Delta}_e = \Delta_e'$ and $\hat{\Delta}_f = \Delta_f, o : (\pi_1\ s_1; \varnothing), o \Rightarrow (o''.f, o''')$. 4 gives us that D is maintained. E is maintained because adding elements to Δ_f does not add requirements, only adding fields to H does that. F and G are maintained since the heap and context do not change.

(b) RESTORE-FIELD-STALE: As in the previous case no permissions are actually returned to the context, so we can ignore the return in the heap too.

(c) RESTORE-FIELD-UNPACKED: By inversion on this rule, we know that $\Delta_1 = \Delta_1', o'' : (\pi_2\ s_2; \Pi_2, f : (\pi_3, o'''))$ with $\pi_1' \otimes \pi_3 \Rrightarrow \pi_3'$ where π_3' is not the declared permission of field f in state s_2 and $o'' : (\pi_2\ s_2; \Pi_2, f : (\pi_3', o''')) \in \Delta'$. From lemma 3.18 we can conclude that o'' was not added during the typing and so $\Delta_e' = \Delta_e'', o'' : (\hat{\pi}_2\ s_2; \hat{\Pi}_2)$. Furthermore, we know from lemma 3.21 that either $\mathsf{packed}(f, \Pi_2)$ or $\hat{\Pi}_2 = \hat{\Pi}_2', f : (\hat{\pi}_3, o''')$. In the former case, we treat the return as with the previous return rules because the return doesn't really need to occur. We might be concerned that doing so would break our ability to regain a `unique` permission to a field. However, this is only the case if the field representative has a $LINEAR$ permission in which case the well-formedness conditions on objects means that the set of object references that return to the field in the heap is a subset of those that return to the field in the context. Since o' does not return to the field in the context, this means it cannot return to the field in the heap either. In the later case, we are guaranteed by lemma 3.21 that $\hat{\pi}_2 = \pi_2$. We are also guaranteed by the well-formedness of the object reference o'' that $o'' \mapsto O'', O'' \mapsto s''\{\Xi'', f \mapsto o'''\} \in H$. Therefore, we have $\Delta_f = \Delta_f', o''' : (\pi_4\ s_4; \varnothing)$. Now we consider two further cases:

case $\mathsf{heapRet}(\Delta; H; o', o''')$:
Since this is true, we know $o''' \mapsto O \in H$, the same object that o points to. We need to update the permissions of both the field in the context and possibly the permission representative in the heap. By lemma 3.33, since we don't change the field permission of any other references, fulfillment is maintained, so it suffices to check that fulfillment for o'' is maintained.
We know that the permission list of O is consistent, including π_1 and π_4. This restricts the possible values of π_4 which in turn restricts what the unpacked permission of the field $\hat{\pi}_3$ in Δ can be. We need to prove the following for each possible combination: $\pi_1' \otimes \hat{\pi}_3 \Rrightarrow \hat{\pi}_3'$ where $\hat{\pi}_3' \downarrow \pi_3'$, and $\pi_1' \otimes \pi_4 \Rrightarrow \pi_4'$ with $\pi_4' \rhd \pi_2.\hat{\pi}_e'$, or $\pi_4 \rhd \pi_2.\hat{\pi}_e'$. In doing this, we will let $\hat{\Delta}_e = \Delta_e', o'' : (\pi_2\ s_2; \hat{\Pi}_2', f : (\hat{\pi}_3', o'''))$, $\hat{\Delta}_f = \Delta_f', o''' : (\hat{\pi}_4\ s_4; \varnothing), o : (\mathtt{none}\ s_1; \varnothing), \hat{\Delta}_e$, where $\hat{\pi}_4$ is π_4 if no join is needed and π_4' otherwise, and $\hat{\Delta} = \hat{\Delta}_e, \hat{\Delta}_f$. Since $\hat{\pi}_3' \downarrow \pi_3'$, this context will fulfill the

typing for condition D and also preserve E since we don't add new heap representatives in H.
We case analyze on π_1.

i. $\pi_1 = \texttt{unique}$: Then $\pi_4 = \texttt{none}$ by permission list consistency and $\hat{\pi}_3 = \texttt{none}$ as well by field fulfillment. This also implies that $\pi_3 = \texttt{none}$ and we know that we can return π_1' to π_3. Therefore, the returns work, $\hat{\pi}_3' = \pi_3'$ so the downgrade follows by reflexivity, and field fulfillment follows by lemma 3.30. Since the return only makes π_4' be non-`borrow` permissions, we don't need to prove anything more for O well-formed and the field fulfillment argument gives that all the o are well formed too, so we are done.

ii. $\pi_1 = \sigma$: Then $\pi_1' = \texttt{none}$ for which we do not need to make any changes, or $\pi_1' = \sigma$. In the latter case, π_3 is either σ or `none`. Since the field is unpacked in Δ before the downgrade and after the return, we know that the permission of the field must be `unique` and $\hat{\pi}_3$ must also be σ or `none` (or $\texttt{borrow}(\texttt{unique}, \sigma, n)$. In this side case, the return fails, but would result in a weaker permission anyway, so we do not make any updates.), or the field must be σ and $\hat{\pi}_3 = \texttt{none}$. In the first case, π_4 or π_4' must be at least σ, so we can easily preserve field fulfillment. The downgrade requirement holds because the permission is σ at the start and end. In the later case the return must be `none` to keep the field unpacked which is the trivial case. Again since we don't introduce any new `borrow` permissions, we do not need to prove anything extra for O well-formed and so we are done.

iii. $\pi_1 = \texttt{local } \sigma$: In this case π_1' must also be `local` σ. π_3 must then be a `borrow` permission or σ. Therefore, we know from validPerms that π_4 cannot be `unique`. Since f was originally unpacked, we know that $\hat{\pi}_3$ must come from this same set. It follows that $\texttt{local } \sigma \otimes \hat{\pi}_3 \Rrightarrow \hat{\pi}_3'$ and furthermore it is easy to show that this return leaves $\hat{\pi}_3' \downarrow \pi_3'$. Since π_4 cannot be `unique`, it is also either `borrow` or σ which means that $\texttt{local } \sigma \otimes \pi_4 \Rightarrow \pi_4'$. In the case that $\pi_4 = \sigma$, $\pi_4' = \sigma$ as well and since the field remains unpacked after the return, fulfillment will remain as does the well-formedness of O. If $\pi_4' = \texttt{borrow}(\pi_o, \sigma, n)$, then we have further invariants to check. For **validPerms**, this is given by lemma 3.27. For O well-formed, we have decreased the number of local object references that return in both the heap and context to this field by one and otherwise unchanged the set of object reference that heap and context return to o''', so it must still hold (since the representative is still in $LINEAR$ as well).

case otherwise:
Since we know that $\mathsf{ctxRet}(\Delta; o; o''')$, this means that by O well-formed, we must have $\pi_4 \in LINEAR$. We take $\hat{\Delta}_e = \Delta_e', o'' : (\pi_2\ s_2; \hat{\Pi}_2', f : (\pi_3', o'''), \hat{\Delta}_f = \Delta_f, o : (\texttt{none}\ s_1; \varnothing), \hat{\Delta}_e$, and $\hat{\Delta} = \hat{\Delta}_e, \hat{\Delta}_f$. Since we have updated the permission of the field in the context, we need to check that the field is still fulfilled. We case

analyze on π_4. If $\pi_4 = \texttt{borrow}(\texttt{unique}, \sigma, n)$ for some σ, then we know by O well formed that there are the right number of local object references which heap return to it to restore it to `unique`. Therefore, $\pi_2 = \texttt{unique}$ and $\pi_3' = \texttt{borrow}(\texttt{unique}, \sigma, n')$ or $\pi_2 \in LOCAL(\sigma) \cup LINEAR(\sigma) \cup \sigma$ and some other alias is unpacked. In either case the `borrow` permission in π_4 can fulfill the field f of o''. Also, we reduced the number of object references that return to the field in the context, but since this one did not return to the heap, we maintain the required subset relation for O well-formed. If $\pi_4 = \texttt{unique}$, then by lemma 3.31 we know that any permission in the field is fulfilled.

(d) RESTORE-FIELD-PACK: This case is much that same as the previous case. However, there is one more consideration that we have to deal with. Since the field is packed up, it might be the case that access to a `unique` permission in a field might become available again. The two ways this could happen is if `unique` is returned to the field. This is the simple case since by permission list consistency we know that the field representative must be `none` and so it also becomes `unique`.
The second case where a `local` σ is returned to a field of a `unique` object and gets packed back to `unique` itself. By field fulfillment, we know that the representative permission of the field must have also be in $LINEAR$. Now, it is possible by O well-formed that the return to the field only occurs in the context and not in the heap. However, we know from object reference well-formedness that there are exactly enough local object references to restore the field to packed. These are included in the set of local references that return to the field representative in the context, which is a superset of those that restore to the field representative in the heap. Since this object reference has a `unique` permission, we know that all the other aliases of O have a `none` permission and are therefore packed. Therefore, there can be no other object references that return to the context at this field. Therefore, there either must be no object references that return to the field in the heap, in which case its permission cannot be `borrow` by **validPerms** and so would have to be `unique`, or there must be exactly one, the same that is currently being returned, which means that it gets restored to `unique` as well and we are done.

case T-ALIAS

1. This rule applies only if $e = \texttt{alias}(o)$. By inversion on the typing rule, we know
   (a) $\Delta_e = \Delta_e', o : (\pi_1\ s; \varnothing)$
   (b) $\pi_1 \Rrightarrow \pi \otimes \pi_1'$
   (c) $\Delta' = \Delta'', o : (\pi_1'\ s; \varnothing)$
2. By inversion on the reduction rule, we know
   (a) $o' \notin \mathsf{dom}(H)$

(b) $H = \hat{H}, o \mapsto O$

(c) $e' = o'$

(d) $H' = H, o' \mapsto O$

3. Let $\hat{\Delta}_e = \Delta'_e, o : (\pi'_1\ s; \varnothing), o' : (\pi\ s; \varnothing), o' \Rightarrow o$ and $\hat{\Delta} = \hat{\Delta}_e, \Delta_f$. We have $\mathsf{dom}(\hat{\Delta}) \subseteq \mathsf{dom}(\Delta)$ as needed and we also maintained B by adding o' to both the heap and context. We did not change any field representatives, so E is maintained.

4. We know that o is still well formed since by lemma 3.32, we can downgrade the permission without impacting field fulfillment.

5. We know by lemma 3.36 that o' is also well formed.

6. Finally, by lemma 3.26 we know that **validPerms** for O is maintained, and since we didn't touch any field representatives, that also means that O is well-formed and we are done.

case T-FIELD-ACCESS-PACKED-PACKED

1. This rule applies and e reduces only if $e = \texttt{alias}(o)\,.\,f$. which by inversion on reduction and the typing rule from D implies

   (a) $H = \hat{H}, o \mapsto O, O \mapsto s\{\Xi, f \mapsto \hat{o}\}, \hat{o} \mapsto O'$

   (b) $o' \notin \mathsf{dom}(H)$

   (c) $H' = H, o' \mapsto O'$

   (d) $e' = o'$

   (e) $\Delta_e = \Delta'_e, o : (\pi_1\ s_1; \Pi_1)$ and $\Delta' = \Delta_e$

   (f) $\mathsf{packed}(f, \Pi_1)$

   (g) $\mathsf{field\text{-}type}(P, s_1, f) = \pi_f\ s$

   (h) $\pi_1\,.\,\pi_f \Rrightarrow \pi \otimes \pi_f$

   (i) $\ell^* = \varnothing$

2. We know by assumption that $\Delta_f = \Delta'_f, \hat{o} : (\hat{\pi}\ \hat{s}; \varnothing)$ and furthermore that $\hat{\pi} \rhd \pi_1\,.\,\pi_f$.

3. 2 tells us that since $\pi_1\,.\,\pi_f \Rrightarrow \pi \otimes \pi_f$ by 1h, there exists $\hat{\pi}'$ such that $\hat{\pi} \Rrightarrow \pi \otimes \hat{\pi}'$.

4. Let $\hat{\Delta}_e = \Delta_e, o' : (\pi\ s; \varnothing)$ and $\hat{\Delta}_f = \Delta'_f, \hat{o} : (\hat{\pi}'\ \hat{s}; \varnothing)$. Now if $\hat{\Delta} = \hat{\Delta}_e, \hat{\Delta}_f$, then $\mathsf{dom}(\Delta) \subseteq \mathsf{dom}(\hat{\Delta})$ as required. B is maintained because we added o' to both the heap and the context. E is maintained because we didn't update any field mappings or the contents of Δ_f, only their permissions.

5. We know that $P; \Delta_e, o' : (\pi\ s; \varnothing); \pi \vdash o' : s \dashv \Delta_e; \varnothing$ by the typing rule T-OBJREF2, so D is maintained.

6. Since we updated the permission of a field representative, we need to ensure that field fulfillment is maintained for each existing object reference mapped to O, including o. This follows from lemma 3.34.

7. We need to check that the new object reference o' is well-formed. This follows from lemma 3.36.

8. Finally, we need to check that $P; \hat{\Delta}; H' \vdash O$ **ok**. We know by lemma 3.26 that **validPerms** is maintained. Now, since we have updated the permission of field f, we need to check that we have not broken any invariants on it. By lemmas 3.3 and 3.2, we know that since the permission of the field did not change that $\pi = \texttt{none}$, which is trivial since no changes were made, $\pi = \sigma$, which is also fine because it cannot introduce any `borrow` or local permissions, or $\pi = \texttt{local}\ \sigma$. In this last case, it is possible that the field representative could have a `unique` permission. However, since we must be pulling a $\texttt{local}\ \sigma$ from a field declared as σ, we can actually replace that `unique` permission with a σ. This preserves permission list consistency by lemma 3.25 given our assumption about consistency before the update. This also will not break field fulfillment for any object references since one can only pull a σ from a field declared to be σ and unpacked fields must have less permissions (ie could not have `unique`). We assume this is done in the original environment. Now this case is the same as the last since pulling a local from a σ does not generate a `borrow` permission. Thus, we have proved that G is maintained.

case T-FIELD-ACCESS-PACKED-UNPACKED

1. This rule applies and e reduces only if $e = \texttt{alias}(o)\,.\,f$. which by inversion on reduction and the typing rule from D implies

   (a) $H = \hat{H}, o \mapsto O, O \mapsto s\{\Xi, f \mapsto \hat{o}\}, \hat{o} \mapsto O'$

   (b) $o' \notin \mathsf{dom}(H)$

   (c) $H' = H, o' \mapsto O'$

   (d) $e' = o'$

   (e) $\Delta_e = \Delta'_e, o : (\pi_1\ s_1; \Pi_1)$ and $\Delta' = \Delta_e$

   (f) $\mathsf{packed}(f, \Pi_1)$

   (g) $\mathsf{field\text{-}type}(P, s_1, f) = \pi_f\ s$

   (h) $\pi_1\,.\,\pi_f \Rrightarrow \pi \otimes \pi'_f$ where $\pi_f \neq \pi'_f$

   (i) $\mathsf{fresh}(i)$,

   (j) $\Delta' = \Delta'_e, o : (\pi_1\ s_1; \Pi_1, f : (\pi'_f, i))$

   (k) $\ell^* = (o\,.\,f, i)$

2. This case is similar to T-FIELD-ACCESS-PACKED-PACKED. There are two differences. First, we need to update the type of o and second, we need to add a return specification for o'. We unpack f using the object reference $\hat{o}$ that is the field representative (we can assume that this is the field id that was chosen). So $\hat{\Delta}_e = \Delta'_e, o : (\pi_1\ s_1; \Pi_1, f : (\pi'_f, \hat{o})), o' : (\pi\ s; \varnothing), o' \Rightarrow (o\,.\,f, \hat{o})$. $\hat{\Delta}_f$ is defined again as $\Delta_f = \Delta'_f, \hat{o} : (\hat{\pi}'\ \hat{s}; \varnothing)$.

3. We can use the same arguments as the previous case to see that B, D, and E are maintained. Lemma 3.34 still proves that the existing object references still have their fields fulfilled and lemma 3.36 checks the new object reference o'. In the case that $\pi = \texttt{local}\ \sigma$ and $\pi_f = \texttt{unique}$, we know that the new alias now returns this field in the context and is local. Therefore, $\pi'_f = \texttt{borrow}(\texttt{unique}, \sigma, 1)$. Thus, the last clause of o well-formed holds because we have exactly the right number of local object references returning to $o\,.\,f$ to restore it to unique.

4. We must check that $P; \hat{\Delta}; H' \vdash O$ **ok**. We know by lemma 3.26 that **validPerms** is maintained. Now, since we have updated the permission of field f and added a return to it, we need to check that we have not broken any invariants. We case analyze on the permission π that caused the field to become unpacked. We know that π cannot be none, so we disregard that case:

   (a) $\pi = \texttt{unique}$: Then $\pi_f = \texttt{unique}$ and it must be the case that $\pi_1 = \texttt{unique}$ as well. $\pi'_f = \texttt{none}$, which is not a $LINEAR$, so we need to ensure that any object references that return to the field in the context also return in the heap. However, this is trivial because the field was previously packed on this object reference and all others aliasing O (since they must have permission none which implies packed). Therefore, by the definition of ctxRet there cannot be any such object references in the original context. Following, the new alias returns to both the context and the heap, so the implication holds.

   (b) $\pi = \sigma$: Then $\pi_f = \texttt{unique}$ as well since pulling from σ does not change the permission and so would not unpack. In this case $\pi'_f = \sigma$ which is not $LINEAR$, so we need to show that all object references that return in the context also return in the heap. As in the previous case, since the permission of the object is unique and this field was previously packed, we cannot have any existing object references of this nature. The new alias will return to both and so the implication holds.

   (c) $\pi = \texttt{local}\ \sigma$: It must again be the case that $\pi_f = \texttt{unique}$, so $\pi'_f = \texttt{borrow}(\texttt{unique}, \sigma, 1)$. Now, by assumption, we know that $P; \Delta; H \vdash O$ **ok**. Therefore, $\hat{\pi} \in LINEAR$ if there is an outstanding borrow permission originally unique as the permission to some o'' pointing to O. If this is the case, $\hat{\pi}' = \texttt{borrow}(\texttt{unique}, \sigma, n)$ for some $n \geq 1$. Therefore, the second implication is maintained. The first is also maintained because $\hat{\pi}' \in LINEAR$ and adding $o' \Rightarrow (o\,.\,f, \hat{o})$ and mapping o' to a $\texttt{local}\ \sigma$ permission increases the set of object reference returning to this field in both the heap and context by 1, thus maintaining the inclusion of the heapRet set. Thus, even if f just became unpacked, the inclusion will be maintained. If $\hat{\pi}$ is not necessarily in $LINEAR$, then

in could also be σ since `local` σ can be pulled from σ. In that case, all object references returning to the field in the context must also return in the heap. As before, the new alias preserves this.

That completes the proof that O is well-formed and we are done.

case T-FIELD-ACCESS-UNPACKED-MAINTAIN
This is exactly the same as case T-FIELD-ACCESS-PACKED-PACKED since the set of permissions that can be pulled without changing the original permission is the same.

case T-FIELD-ACCESS-UNPACKED-CHANGE
This case is similar to T-FIELD-ACCESS-PACKED-UNPACKED. The only difference is that the field must start unpacked, which means that π cannot be `unique`. However as before the assumption is strong enough that when we create the new alias which returns to both the heap and the context, it is all maintained.

case T-FIELD-ASSIGN-PACKED
This case applies only if $e = \texttt{alias}(o)$=e_1. Inversion on the typing rule tells us

I) $\Delta_e = \Delta'_e, o : (\pi_1\ s_1; \Pi_1)$

II) $\mathsf{field\text{-}type}(P, s_1, f) = \pi_2\ s_2$

III) $P; \Delta_e; \pi_2 \vdash e_1 : s'_2 \dashv \Delta_2, o : (\pi'_1\ s_1; \Pi'_1); \ell^*$ where $\mathsf{assignable}(\pi'_1)$ and $P \vdash s'_2 \preceq s_2$.

IV) $\pi_2 \Rrightarrow \pi \otimes \pi_2$

There are two possibilities for reduction rules. The first is E-CONGRUENCE which is uninteresting because we can simply use the induction hypothesis on III which is the first action that the rule takes, so everything else comes from the guarantees of the induction hypothesis. The rule E-ASSIGN is more interesting:

1. By inversion on the reduction rule, we know
   (a) $e_1 = o'$
   (b) $o'' \notin \mathsf{dom}(H)$
   (c) $H = \hat{H}, o \mapsto O, O \mapsto s'_1\{\Xi, f \mapsto o_f\}, o' \mapsto O'_f$
   (d) $H' = \hat{H}, o \mapsto O, O \mapsto s'_1\{\Xi, f \mapsto o''\}, o' \mapsto O'_f, o'' \mapsto O'_f$
   (e) $e' = o''$
2. From III, we know that $\Delta_e = \Delta''_e, o : (\pi_1\ s_1; \Pi_1), o' : (\pi_2\ \hat{s}_2; \varnothing)$ and $P; \Delta_e; \pi_2 \vdash o' : s'_2 \dashv \Delta_1, o : (\pi'_1\ s_1; \Pi'_1)$.

3. Let $\hat{\Delta}_e = \Delta''_e, o : (\pi_1\ s_1; \Pi_1), o' : (\pi\ s_2; \varnothing)$ and $\hat{\Delta}_f = \Delta_f, o'' : (\pi_2\ \hat{s}_2; \varnothing)$. Then by using $\hat{\Delta} = \hat{\Delta}_e, \hat{\Delta}_f$, we have expanded the domain of Δ and also included o'' in both the heap and context, so B is maintained.

4. The typing requirement D follows from updating the permission of o' in the context as o' is still removed from the outgoing context.

5. **distinctFields** (E) is maintained because we added o'' which was assigned into the field to $\hat{\Delta}_f$.

6. We need to check that the new object reference o'' is well formed, which comes from lemma 3.36.

7. We need to check that O'_f is still well-formed. However, this comes from showing that **validPerms** still holds with lemma 3.26. No fields were updated.

8. We need to check that O is still well-formed as well as the field fulfillment of object references point to O. We case analyze on the permission list to O assuming that o must have an assignable permission. We only consider fields that are declared `unique` since they are the only fields that can be effectively unpacked.

   (a) $\mathsf{perms}(\Delta; H; O) = [\texttt{none}^*]{+}{+}[\texttt{unique}]$: Then we know that $\pi_1 = \texttt{unqiue}$ and so all other object references have permission `none` by permission list consistency. Therefore, since object references are well-formed, all of them must be packed. This means that replacing the permission in the type of their field is the same as the new representative, which by lemma 3.30 means that field fulfillment is preserved for all object references point to O. Since f is now packed in all object references that point to O the `borrow` requirement holds trivially. For O well-formed, we did not update any of the permissions of object references point to O, so **validPerms** must still hold. If the field f is declared to be `unique` then we might need to worry about the other implications of O well-formedness. Since o' was newly generated, there cannot be any object references that return to it and since all objects pointing to O are packed, there cannot be an object references that return to the field in the context so the return requirement holds.

   (b) $\texttt{borrow}(\texttt{unique}, \texttt{shared}, n) \in \mathsf{perms}(\Delta; H; O)$: We know that since the field is declared to be `unique` and there is a `borrow` originally `unique` permission to O that π_f, the permission to old field representative o_f must also be in $LINEAR$. This implies that the set of object references that return to o_f in the heap are a subset of the object references that return to $(\hat{o}.f, o_f)$ in the context for any $\hat{o}$ that is an alias of O. In this case, we must update all return specifications of this form to $(\hat{o}.f, o'')$. This means that the set of object references that returns in the context to the field remains constant. However, these will not return to the heap because of the mismatch between the new object in the field O'_f which must distinct from the original field object since we have the single `unique` permission to O'_f already. This is fine because the subset

relationship is maintained. In order to preserve the return structure of o_f, we change its permission in $\hat{\Delta}_f$ to `shared`. This is safe since o_f cannot be used either as a field or in the expression any longer. In this way the invariants are preserved.

(c) `shared` $\in \mathsf{perms}(\Delta; H; O)$: It is possible as in the last case for other object references to be unpacked. However π_f does not have to be in $LINEAR$, so we do not have to track as carefully. Thus, we update the return specifications as before, but we do not need to update the permission to π_f.

case T-FIELD-ASSIGN-UNPACKED
This case is exactly the same in its details as the previous case. The only additional step is packing f in o, which is no different than updating the mapping of a packed field.

case T-INVOKE
The invoke case is much like the T-LET case with a few differences. First, we have two possibilities for the congruence rule. The first operates on the receiver expression and follows directly from the induction hypothesis. The second operates on the parameter expression when $e = o.m(e_1)$. For this, we cannot use the induction hypothesis directly because some typing occurs as a part of typing o that could update the context. Thus, we first apply corollary 3.48 to move this typing to e_1. Now, while typing removes o and its return specification from the output context, since these were the only changes by T-OBJREF1/2, we can reconstruct the original context by adding them back to Δ_f. This does not change the typing or **distinctFields** and so we can make use of our assumptions to invoke the induction hypothesis on the reduction of e_1 and we are done.

The final problem is reducing to the nested `bind` expression using E-INVOKE when $e = o.m(o')$. By the typing rules, we know that $o \neq o'$. By the typing of the method body e_2, we know that $P; \texttt{this} : (\pi_1\ s_1; \varnothing), x : (\pi_2\ s_2; \varnothing); \pi \vdash e_2 : s \dashv \texttt{this} : (\pi_1'\ s_1; \varnothing), x : (\pi_2'\ s_2; \varnothing); \ell^*$. Thus, we can apply lemma 3.17 twice to get $P; o : (\pi_1\ s_1; \varnothing), o' : (\pi_2\ s_2; \varnothing); \pi \vdash e_2[\texttt{this} \leftarrow \texttt{alias}(o), x \leftarrow \texttt{alias}(o')] : s \dashv o : (\pi_1'\ s_1; \varnothing), o' : (\pi_2'\ s_2; \varnothing); \ell^*[\texttt{this} \leftarrow o, x \leftarrow x]$. Using corollary 3.48 we can remove the extra typing rules from the receiver and move them to the parameter. Now, by adding o to the typing of o' using lemma 3.14, we can invoke lemma 3.45 in order to move equalizing rules into the body e_2. This gives us the correct Δ and choices of Δ_e and Δ_f in order to type the updated expression `bind` o `in bind` o' `in` e_2 as needed.

case T-SEQUENCE-MULTIPLE
$e = \{e_1; e_2;^+\}$. We have two possibilities for the reduction rule. The congruence rule reduces e_1 and we get what we need directly from the induction hypothesis. Otherwise, E-SEQUENCEMULTIPLE must have been used so $e = \{o; \hat{e};^+\}$ and $e' = \{\hat{e};^+\}$. As with T-LET, we need to move equalizing rules since o is being reduced away. We can use corollary 3.47 to do this. To finish, we choose the same context, but move o and any return specification for it from Δ_e to Δ_f and as with `bind`.

case T-SEQUENCE-SINGLE
This case is simple, using only the straightforward congruence rule for which we can use the induction hypothesis, or E-SEQUENCE-SINGLE for which no changes are needed since the object reference is just pulled out the sequence but left in the expression.

case T-MATCH-SINGLE
We have two cases for reduction. Either the E-CONGRUENCE rules is used in which case we simply use the induction hypothesis, or E-MATCH is used to select the single expression. However, we can apply lemma 3.47 to move all equalizing rules into the `case` body. Then we can type the updated expression, which will be the `case` body, in the same context with o and any return specification for it moved from Δ_e to Δ_f and as with `bind`.

case T-MATCH-MULTIPLE
We have the same cases for reduction. Since T-MATCHSINGLE is ultimately applied to each `case` body, the same reasoning follows as for T-MATCHSINGLE. The only extra step we have to take is to account for the equalizing of field ids. This can be done by adding instances of T-EQ-FRESH-FIELD-ID to the end of the typing.

case T-EQ-*:
All of these cases follow directly from the induction hypothesis on the premise of the rule that types the expression before making the equalizing updates. Since we get the same outputs after the reduction via the induction hypothesis, we can still make the same equalizing updates.

□

3.5 Deterministic Algorithm

The algorithm specified by the typing rules above is non-deterministic because the existence of a resulting context of a `match` expression is asserted, but no algorithm is given to compute it. This is undesirable for an implementation of our system. Therefore, we define deterministic rules for typing `match`. We believe, but do not prove, that these two formulations are equivalent.

3.5.1 Merging permissions:

To develop a deterministic algorithm, we first need a way to merge two permissions. The mergePerms judgment takes two permissions and gives the merged permission as an output.

$$\boxed{\pi = \mathsf{mergePerms}(\pi_1, \pi_2)}$$

We merge two permissions by either taking the permission π_2 that can be obtained by pulling some permission π from π_1 (P-MERGE-PULL), or the borrow permission with a larger count (P-MERGE-BORROW), or `none` (P-MERGE-INCOMPATIBLE).

$$\text{P-MERGE-SYMMETRIC}\ \frac{\pi = \mathsf{merge}(\pi_2, \pi_1)}{\pi = \mathsf{merge}(\pi_1, \pi_2)} \qquad \text{P-MERGE-PULL}\ \frac{\exists \pi . \pi_1 \Rrightarrow \pi \otimes \pi_2}{\pi_2 = \mathsf{mergePerms}(\pi_1, \pi_2)}$$

$$\text{P-MERGE-BORROW}\ \frac{\pi_2 \in \{\pi\} \cup \{\mathsf{borrow}(\pi, \sigma, n') \mid n' \leq n\}}{\mathsf{borrow}(\pi, \sigma, n) = \mathsf{mergePerms}(\mathsf{borrow}(\pi, \sigma, n), \pi_2)}$$

P-MERGE-INCOMPATIBLE

$$\frac{\pi_1 \in \{\sigma_1, \texttt{local}\ \ \sigma_1, \mathsf{borrow}(\pi, \sigma_1, n_1)\} \quad \pi_2 \in \{\sigma_2, \texttt{local}\ \ \sigma_2, \mathsf{borrow}(\pi, \sigma_2, n_2)\} \quad \sigma_1 \neq \sigma_2}{\texttt{none} = \mathsf{mergePerms}(\pi_1, \pi_2)}$$

The definition mergePerms provides several useful properties:

Lemma 3.51 (Merged Permissions)**.** *If $\pi = \mathsf{mergePerms}(\pi_1, \pi_2)$, then*

1. *$\pi_i \downarrow \pi$ for $i = 1, 2$, and*
2. *$\pi \in \{\pi_1, \pi_2, none\}$.*

Proof. By exhaustive case analysis on the rules for mergePerms. □

3.5.2 Merging contexts:

Next, we provide explicit rules for computing the merge of two contexts.

$$\boxed{\Delta = \mathsf{merge}(\Delta_1; \Delta_2)}$$

C-MERGE-EMPTY

$$\frac{}{\varnothing = \mathsf{merge}(\varnothing; \varnothing)}$$

C-MERGE-PACKED

$$\frac{\Delta_3 = \mathsf{merge}(\Delta_1; \Delta_2) \quad \pi = \mathsf{mergePerms}(\pi_1, \pi_2)}{\Delta_3, v : (\pi\ s; \varnothing) = \mathsf{merge}(\Delta_1, v : (\pi_1\ s; \varnothing); \Delta_2, v : (\pi_2\ s; \varnothing))}$$

C-MERGE-UNPACKED1

$$\frac{\Delta_3 = \mathsf{merge}(\Delta_1; \Delta_2) \quad \pi_2 \downarrow \pi_1}{\Delta, v : (\pi_1\ s; \Pi) = \mathsf{merge}(\Delta_1, v : (\pi_1\ s; \Pi); \Delta_2, v : (\pi_2\ s; \varnothing))}$$

C-MERGE-UNPACKED2

$$\frac{\Delta_3 = \mathsf{merge}(\Delta_1; \Delta_2) \quad \pi_1 \downarrow \pi_2}{\Delta, v : (\pi_2\ s; \Pi) = \mathsf{merge}(\Delta_1, v : (\pi_1\ s; \varnothing); \Delta_2, v : (\pi_2\ s; \Pi))}$$

C-MERGE-UNPACKED-BOTH1

$$\frac{\Delta_3, v : (\pi\ s; \Pi) = \mathsf{merge}(\Delta_1, v : (\pi\ s; \Pi_1); \Delta_2, v : (\pi\ s; \Pi_2)) \quad \pi' = \mathsf{mergePerms}(\pi_1, \pi_2)}{\Delta, v : (\pi\ s; \Pi, f : (\pi', i)) = \mathsf{merge}(\Delta_1, v : (\pi\ s; \Pi_1, f(\pi_1, i)); \Delta_2, v : (\pi\ s; \Pi_2, f(\pi_2, i)))}$$

C-MERGE-UNPACKED-BOTH2

$$\frac{\Delta_3, v : (\pi\ s; \Pi) = \mathsf{merge}(\Delta_1, v : (\pi\ s; \Pi_1); \Delta_2, v : (\pi\ s; \Pi_2)) \quad \pi' = \mathsf{mergePerms}(\pi_1, \pi_2) \quad i \neq j \quad \mathsf{fresh}(k)}{\Delta, v : (\pi\ s; \Pi, f : (\pi', k)) = \mathsf{merge}(\Delta_1, v : (\pi\ s; \Pi_1, f(\pi_1, i)); \Delta_2, v : (\pi\ s; \Pi_2, f(\pi_2, j)))}$$

For packed values, we simply take the merged permission. For values in which only one of the two is unpacked, we take the unpacked state as long as the permission of the version that is unpacked downgrades to the other. Otherwise, we would be conceptually pulling permissions from an unpacked value. We take the merged unpacked permission for fields and either the field ID in both contexts if it is the same, or a fresh one if they are not.

3.5.3 Combining source location lists:

We also need a judgment to deterministically reconcile the SLLs that come from each arm of a `nonDet`. This judgment says that given the output context Δ and requested permission π, SLL ℓ_1^* associated with context Δ_1 merges with the SLL from the pair $\Delta 2; \ell_2^*$ to produce ℓ^*. Once we have gone through all of the elements from both SLL, we are done. The COMBINE-SYMMETRIC rule

indicates that, as in merging permissions, order does not matter. When any source location appears in both lists, we include it in the output (COMBINE-BOTH). If a variable appears in only one of the lists, we include it only if the required permission π can be pulled from the variable in the other context without changing the output of mergePerms (COMBINE-VAR-ONE-INCLUDE). Otherwise the variable is not included in the SLL (COMBINE-VAR-ONE-EXCLUDE1, COMBINE-VAR-ONE-EXCLUDE2). The case is similar for fields with the additional case that the field appears in both SLL with different field IDs. In this case, it is not included (COMBINE-VAR-ONE-EXCLUDE3).

$$\boxed{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*) = \ell^*}$$

COMBINE-EMPTY
$$\frac{}{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \varnothing; \Delta_2; \varnothing) = \varnothing}$$

COMBINE-SYMMETRIC
$$\frac{\Delta; \pi \vdash \mathsf{combine}(\Delta_2; \ell_2^*; \Delta_1; \ell_1^*) = \ell^*}{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*) = \ell^*}$$

COMBINE-BOTH
$$\frac{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*) = \ell^*}{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*, \ell; \Delta_2; \ell_2^*, \ell) = \ell^*, \ell}$$

COMBINE-VAR-ONE-INCLUDE
$$\frac{\begin{array}{c}\Delta, x : (\pi'\ s; \Pi'); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2); \ell_2^*) = \ell^* \\ x \notin \ell_1^* \qquad P; \Delta_1, x : (\pi_1\ s; \Pi_1); \pi \vdash x : s \dashv \Delta_1, x : (\pi_1'\ s; \Pi_1); x \qquad \pi' = \mathsf{mergePerms}(\pi_1', \pi_2)\end{array}}{\Delta, x : (\pi'\ s; \Pi'); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2); \ell_2^*, x) = \ell^*, x}$$

COMBINE-VAR-ONE-EXCLUDE1
$$\frac{\begin{array}{c}\Delta, x : (\pi'\ s; \Pi'); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2); \ell_2^*) = \ell^* \\ x \notin \ell_1^* \qquad P; \Delta_1, x : (\pi_1\ s; \Pi_1); \pi \vdash x : s \dashv \Delta_1, x : (\pi_1'\ s; \Pi_1); x \qquad \pi' \neq \mathsf{mergePerms}(\pi_1', \pi_2)\end{array}}{\Delta, x : (\pi'\ s; \Pi'); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2); \ell_2^*, x) = \ell^*}$$

COMBINE-VAR-ONE-EXCLUDE2
$$\frac{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*) = \ell^* \qquad x \notin \ell_1^* \qquad \nexists\, \Delta_1'. P; \Delta_1; \pi \vdash x : s \dashv \Delta_1'; x}{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*, x) = \ell^*}$$

COMBINE-FIELD-ONE-INCLUDE
$$\frac{\begin{array}{c}\Delta, x : (\pi_3\ s; \Pi', f : (\pi_3, i)); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2, f : (\pi_2', i)); \ell_2^*) = \ell^* \\ (x\,.\,f, i) \notin \ell_1^* \qquad P; \Delta_1, x : (\pi_1\ s; \Pi_1); \pi \vdash x\,.\,f : s \dashv \Delta_1, x : (\pi_1\ s; \Pi_1', f : (\pi_1', i)); \ell \qquad \pi_3 = \mathsf{mergePerms}(\pi_1', \pi_2')\end{array}}{\Delta, x : (\pi_3\ s; \Pi', f : (\pi_3, i)); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2, f : (\pi_2', i)); \ell_2^*, (x\,.\,f, i)) = \ell^*, (x\,.\,f, i)}$$

COMBINE-VAR-ONE-EXCLUDE1
$$\frac{\begin{array}{c}\Delta, x : (\pi_3\ s; \Pi', f : (\pi_3, i)); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2, f : (\pi_2', i)); \ell_2^*) = \ell^* \\ (x\,.\,f, i) \notin \ell_1^* \qquad P; \Delta_1, x : (\pi_1\ s; \Pi_1); \pi \vdash x\,.\,f : s \dashv \Delta_1, x : (\pi_1\ s; \Pi_1', f : (\pi_1', i)); \ell \qquad \pi_3 \neq \mathsf{mergePerms}(\pi_1', \pi_2')\end{array}}{\Delta, x : (\pi_3\ s; \Pi', f : (\pi_3, i)); \pi \vdash \mathsf{combine}(\Delta_1, x : (\pi_1\ s; \Pi_1); \ell_1^*; \Delta_2, x : (\pi_2\ s; \Pi_2, f : (\pi_2', i)); \ell_2^*, (x\,.\,f, i)) = \ell^*, (x\,.\,f, i)}$$

COMBINE-FIELD-ONE-EXCLUDE2
$$\frac{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*) = \ell^* \qquad (x\,.\,f, i) \notin \ell_1^* \qquad \nexists\, \Delta_1', \ell. P; \Delta_1, x : (\pi_1\ s; \Pi_1); \pi \vdash x\,.\,f : s \dashv \Delta_1'; \ell}{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*, (x\,.\,f, i)) = \ell^*}$$

COMBINE-VAR-ONE-EXCLUDE3
$$\frac{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*; \Delta_2; \ell_2^*) = \ell^* \qquad i \neq j}{\Delta; \pi \vdash \mathsf{combine}(\Delta_1; \ell_1^*, (x\,.\,f, i); \Delta_2; \ell_2^*, (x\,.\,f, j)) = \ell^*}$$

3.6 Comparison to the Published POPL System

Following the camera-ready deadline for the version of this work published at POPL 2012 [Naden et al., 2012], we found two flaws in the proof claimed in the paper. First, there was a soundness issue with regards to the handling of SLLs generated by match expressions. Second, the invariants

for the proof of soundness were found to be insufficient. As a part of fixing the proof, we also switched the rules for match to what we believe is the non-deterministic equivalent of the published rules. This change substantially simplified the proof without impacting the core ideas presented.

The following section outlines the changes that we made to the system following publication at POPL 2012.

3.6.1 Changes to Permission Operations

New splitting rule

Splitting rule P-SPLIT-BORROW-SYMMETRIC was added to allow a σ permission to be split from a $\texttt{borrow}(\texttt{unique}, \sigma, n)$. This is needed because the POPL system allowed a σ permission to be split from a `unique`, followed by a `local` σ:

$$\texttt{unique} \Rrightarrow \sigma \otimes \sigma \Rrightarrow \texttt{local}\ \sigma \otimes \sigma \otimes \sigma.$$

However, it was not possible to first split a `local` σ and then a σ from a `unique`:

$$\texttt{unique} \Rrightarrow \texttt{local}\ \sigma \otimes \texttt{borrow}(\sigma, \texttt{unique}, 1) \Rrightarrow \sigma \otimes \texttt{local}\ \sigma \otimes \sigma.$$

There are examples using `match` for which progress will fail without allowing both splitting orders. The new rule corrects this.

Joining Characterization

With the original joining rules there existed pairs of permissions for which either rule could apply, though both gave the same result. For each pair of input permissions only one of the two updated rules can apply.

Permission Downgrades

Permission downgrades $\pi \downarrow \pi$, or the *weaker* relationship as it was called in the POPL system, was used in the original proof of safety to relate contexts before and after an evaluation step. In the updated system, downgrading represents the effect of multiple permission splits from a single permission and is used in typing rules as well as the invariants and safety proof. We found that more cases were needed in the updated system.

3.6.2 Changes to Typing Judgments

The biggest change in the system occurred in the typing judgements. These changes resolved a soundness issue and simplified the proof by using non-deterministic rules.

Soundness Issue

Consider the following example. In a local scope the variable t is given a `unique` permission that comes either from a new object or the local variable s from the outer scope. However, in the case that a new object is returned, the permission for s is placed into a field, consuming it

```
state S { }
state C { unique S f = new S; }

unique C c = new C;
unique S s = new S;
{
     unique S t = match (...)
        S=>{
            c.f = s;
            new S;          //s:none, no return
        };
        C=>s;               //s:none, return to s
    }                       //s:none, return to s
}                           //s:unique
unique S u = s;             //unsafe!
```

permanently. After this inner scope, a `unique` alias of s is created. This last alias is unsafe because in the case that the first `case` was executed, s must have a `none` permission.

In order to remove the unsoundness, we defined a judgment for combining source local lists (see section 3.5.3). These rules take the type of the location in the outgoing context from each case into account when deciding whether a particular location should be included in the combined SLL.

Non-deterministic Rules

In addition, we determined that our proof would be simpler and easier to understand if we switched to non-deterministic rules for match typing. The deterministic rules are important and necessary for an implementation of our system, but were awkward in our formalization. Thus, the T-MATCH-MULTIPLE rule was updated to require that each case output the same context and SLL, modulo generated field IDs. We also introduced a set of *equalizing rules* in section 3.2.4. These rules prevent the soundness issue by allowing locations to be dropped from the output SLL at any time (T-EQ-DROPRET), but require the needed permission to be pulled from the location in the output context in order to add a location to the SLL (T-EQ-PULL-*). Thus, in the example above s could not be added to the SLL of the first case since at the end of the case it did not have a `unique` permission available to be split from s.

In section 3.5, we also provide a set of deterministic rules that we use in our implementation of this work. We believe that these rules are equivalent to the non-deterministic formulation, though we have not proved this.

Source Locations

We made three cosmetic changes to source locations and the return rules to make the formalism cleaner.

- Source locations that are not present in the associated output context are removed and the return rules no longer allow for missing locations.

- In order to be consistent with field source locations, we added the rule T-FIELD-ACCESS-UNPACKED-MAINTAIN so that any split that did not change the permission of the field would not add the field to the SLL, regardless of whether the field was packed or unpacked.
- For the intermediate typing rules, we no longer provide an empty return location, so we distinguish rules for object references and `bind` expressions depending on if the there is a return specification for their object reference in the context.

Permission for `new`

To ensure that permission lists for all objects were consistent, we restricted the permissions that a new object could be allocated at to those that could be split from `unique`. In particular, this restriction excludes isolated `borrow` permissions from being created with `new`.

3.6.3 Changes to Reduction

In the POPL system reduction rules removed object references from the heap when they went out of scope. In the updated formalism, these object references are left in the heap. We decided that it was cleaner to handle this issue as a part of the safety proof.

3.6.4 Changes to Safety Statement

The statements of progress in each system are the same. For the statement of preservation, the use of equalizing rules in the new system allowed us to keep the outputs after the reduction the same. This saved us from the need to relate the original outputs to the new outputs and prove that typechecking succeeded in the new outputs. Instead, the only extra condition we need is that the context for typechecking grows monotonically larger to account for newly created references. We feel that this change results in a much simpler statement of preservation.

3.6.5 Changes to Invariants

The invariants included in the judgement $P; \Delta; \pi \vdash H; e : s \dashv \Delta', \ell^\star$ for the system presented here are much more explicit than those in the POPL system. We found that the original invariants were not quite strong enough to guarantee that when a `unique` permission was regained all of the other non-`none` aliases of the object were actually gone. To this end, the OBJECT-REF-OK judgment now includes an extra requirement that if the permission to the field of an object reference $o.f$ is a `borrow` permission, then exactly the right number of object references with a `local` permission will return their permissions to $o.f$ to restore its original permission. The field fulfillment judgment was also strengthened to ensure that if a `unique` permission might at some later point become available through the type of the object and field, then the permission to the representative object reference must also potentially be `unique` at a later point. In particular, it will need to become `unique` prior to the `unique` permission becoming available through the field.

This last point is guaranteed partly by the new judgment OBJ-ID-OK which strengthens the requirement on the permissions to different references of a single object ID O. In the POPL version,

we only required that the permission list generated by the permissions of each object reference o_i that point to O, be consistent. The new judgment also requires that the return structure between the various o_i is also well-formed. This is captured by the rule OBJ-PERMS-VALID as explained above. Secondly, there are additional restrictions placed on the fields of O which correspond to the added restrictions on fields of an object reference.

4 Related Work

Wadler first introduced the concept of temporarily converting a linear (`unique`) reference into a non-linear reference using the `let!` construct [Wadler, 1990]. Other early uniqueness type systems built on his work and added support for borrowing as a special annotation, such as "borrowed" or "lent" [Hogg, 1991, Minsky, 1996, Aldrich et al., 2002]. While convenient, these systems did not support borrowing immutable pointers, and generally provided weak guarantees: multiple borrowed pointers could co-exist and interfere with one another. Boyland devised alias burying to address this issue, using shape analysis to ensure that whenever a unique variable was read, all aliases to it were dead (or "buried") [Boyland, 2001]. While this approach works well in an analysis tool, it is inappropriate for a type system: programmers would have to understand a shape analysis to comprehend and fix a type error message. The authors of Plural [Bierhoff and Aldrich, 2007], which also uses analysis to support borrowing, have observed this to be a problem in practice. In contrast, our system provides a more natural abstraction for reasoning by modeling the flow of permissions through locations in the source.

Boyland proposed *fractional permissions* as a generalization of borrowing that does not require a stack discipline for creating and destroying borrowed aliases [Boyland, 2003]. Although fractions have received a lot of attention in the verification community, we know of no practical tool support that leverages fractions—possibly because programmers find fractions an unintuitive abstraction. Instead, tools like Plural [Bierhoff and Aldrich, 2007], Chalice [Heule et al., 2011], and VeriFast [Jacobs et al., 2011] provide abstractions (including borrowing) that hide fractions from users, but the use of program analysis and theorem provers makes these systems less predictable and more difficult to understand than the type system presented here. Boyland and Retert later developed a type system that allows borrowing unique permissions [Boyland and Retert, 2005]. Like our system, their system tracks permissions taken out of individual fields using a technique they call "carving." However, where they use a substructural logic for tracking permissions, we use a more predictable linear context to type individual variables.

One of the authors previously observed the importance of borrowing for tracking permissions and presented the first fraction-free permission type system we are aware of that supports borrowing unique and immutable permissions [Bierhoff, 2011]. While the technical details are somewhat different, the system presented in [Bierhoff, 2011], similar to this system, avoids fractions by counting split-off permissions in variable types. We propose `local` permissions to distinguish borrowed permissions syntactically, as well as `none` permissions, both of which remain implicit in [Bierhoff, 2011]. Our system additionally supports share permissions, match expressions and sequences, and our system tracks permissions taken out of individual fields. Unlike [Bierhoff, 2011], we provide a dynamic semantics and prove our system sound.

Other programming languages have incorporated the ideas of uniqueness and borrowing into their type systems but use less flexible or more complicated mechanisms. The Clean programming language [Smetsers et al., 1994] is a functional language with support for unique references. However, since the language is functional, there is no concept of returning permissions to a location as in our system.

The Vault programming language [Fähndrich and DeLine, 2002] allows linear (`unique`) references to be split into guarded (`immutable`) types that are valid in the scope of a key. Their use of type-level keys adds notational and algorithmic complexity to the scoping of borrowed permissions that we avoid by using our simpler local permissions. On the other hand, Vault supports adoption whereby a linear permission stored in a field of a non-linear object can be treated linearly. Vault also includes annotations on methods that consume permissions similar to our change permissions. However ours are strictly more flexible because in our richer set of permissions we can specify a partial return of a permission (e.g. `unique>>immutable`).

The Cyclone language [Hicks et al., 2004] has a feature similar to Vault's keys. Cyclone also includes explicit support for borrowing through reference counting that is similar to the mechanism that underlies our `local` permissions. However, unlike our approach, it is exposed to the programmer in the syntax. Also, Cyclone allows borrowing only for permissions stored in local variables; field accesses occur via swap, which is awkward. In contrast, we have designed a field unpacking mechanism that supports direct field access.

Other recent work on the Plaid type system [Wolff et al., 2011] integrates permissions with typestate and uses change types, providing some of the expressiveness of our system. While this other work can express the publication example from Figure 4, it has very limited support for borrowing, and can only change field values with a swap operation, which is unnatural for programmers.

An alternative to borrowing is explicitly "threading" references from one call to another, as supported in Alms [Tov and Pucella, 2011]. In this approach, the permission is tied to the reference; it is given up permanently when the reference is passed to a function, but the function may return the reference again along with a permission. This approach is very clear and explicit, but it is quite awkward and furthermore results in additional writes when the result reference is re-assigned to the reference variable.

Overall, the system presented in this paper is distinguished by supporting natural programming and reasoning abstractions together with a broad set of permissions including immutable, unique, and shared. As a type system defined by local rules, it is easy for programmers to follow, and separating permission flow from references makes it more succinct than systems in which references must be threaded explicitly. We hope it will serve as a robust foundation for making permission-based programming languages such as Plaid practical enough for widespread use.

5 Conclusion and Future Work

We have described a new type system for flexible borrowing of unique, shared, and immutable permissions without explicit fractions. As future work, we would like to integrate our borrowing mechanism with Plaid's typestate features, and gain experience using the type system on larger

codebases.

References

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

Kevin Bierhoff. Automated program verification made SYMPLAR: SYMbolic Permissions for Lightweight Automated Reasoning. In *Onward!*, 2011.

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *OOPSLA*, 2009.

B. Bokowski and J. Vitek. Confined types. In *OOPSLA*, 1999.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, 2003.

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

John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, 2001.

John T. Boyland and William Retert. Connecting Effects and Uniqueness With Adoption. In *POPL*, 2005.

Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, 2004.

Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

J.-Y. Girard. Linear logic. *Theoretical Comp. Sci.*, 50(1):1–102, 1987.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI*, 2002.

D.E. Harms and B.W. Weide. Copying and Swapping: Influences on the design of reusable software components. *Trans. Software Engineering*, 17(5):424–435, May 1991.

Stefan Heule, Rustan Leino, Peter Müller, and Alexander Summers. Fractional permissions without the fractions. In *FTfFP*, 2011.

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, 2004.

John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*, 1991.

Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Language: Design and Definition*. Prentice-Hall, 1988.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, 2011.

Naftaly H. Minsky. Towards alias-free pointers. In *ECOOP*, 1996.

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *POPL*, 2012.

J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*. Springer, 1998.

Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Dagstuhl Seminar on Graph Transformations in Comp. Sci.*, volume 776 of *LNCS*. Springer, 1994.

Jesse A. Tov and Riccardo Pucella. Practical affine types. In *POPL*, 2011.

Philip Wadler. Linear types can change the world! In *Working Conf. on Programming Concepts and Methods*, 1990.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP*, 2011.