A Type System for Borrowing Permissions

Karl Naden Robert Bocchino
Jonathan Aldrich
Carnegie Mellon University, Pittsburgh, PA, USA
{kbn,rbocchin,jonathan.aldrich}@cs.cmu.edu

Kevin Bierhoff
Two Sigma Investments, New York, NY, USA
kevin.bierhoff@cs.cmu.edu

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 state soundness results. We also illustrate the utility and practicality of our design by using it to express several realistic examples.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Borrowing; D.3.3 [*Programming Languages*]: Language Constructs and Features —Permissions

General Terms Design, Languages, Theory, Verification

Keywords Types, Permissions, Borrowing, Uniqueness, Immutability

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 [9]. For example, `unique` [20] indicates an unaliased object, `immutable` [24] indicates an object that can be aliased but cannot be mutated, and `shared` [3] 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 [24], protocol checking [3, 13], safe concurrency [6, 8], security [5], and memory management [9, 15]. Recently, new programming languages have incorporated permissions [28] or related affine types [26] 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 [19]; however, good support for this feature has remained an open and difficult problem. Prior type-based systems [1, 19, 22] provided a borrowed annotation, but they did not support the immutable references that are an essential part of many recent systems [3, 8, 28]. A number of systems supported borrowing via a program analysis [4, 7], but the program analysis relies on shape analysis, which is fragile and notoriously difficult for programmers to understand. Boyland proposed fractional permissions [8] 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 [3, 17].

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 [16], or a requirement to thread a reference explicitly from one call to the next by reassigning the reference each time [26]). Reasoning abstractions should likewise be natural (not fractions [8]). 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 [3, 7] 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

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

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]

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 [28]. 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.

2.1 Access Permissions

Permissions are a well-known way of controlling aliasing in applications such as typestate [3] and concurrency control [8]. Our permission system is adapted from the *access permissions* system of [3]. 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 [14]. 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.

(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.

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.
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 com-

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

pute 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. 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* [12]. 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).

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

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

`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 [28]. 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 [7, 11], 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 [8]. However, while powerful, fractions are also difficult to use and suffer from modularity problems [17]. 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 des-

ignated 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`.

(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.

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.

```
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.

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. 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) [13].

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 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 Language

In this section we formalize the ideas developed in the previous section. We give a syntax, static semantics, and dynamic semantics for a core language. Then we state the key soundness results, which are proved in our companion technical report [23].

3.1 Syntax

Figure 7 gives the syntax for the core language. A program consists of a number of state declarations S and an expression e to evaluate. A state consists of a state name s, a parent state s' (possibly itself), and field and method declarations. Fields F and methods M are declared in the usual way, except that field types specify permissions π, and method parameters specify change permissions π>>π'. Fields have initializer expressions. The change permission appearing in the method declaration before the method body is the change permission associated with the implicit parameter `this`.

The rest of the language features are standard for an expression-based object-oriented language. The `match` expression evaluates e to an object reference and compares its runtime state s' to the states s named in the cases. It executes the first case for which s' is a subtype of s, with a runtime error if there is no match. $v.f$=e updates the object in the field $v.f$ to the value of e, which it also returns as the result of the expression. `new` s creates an object of state s and initializes its fields by running the initializer expressions given in the state definition. The sequence expression evaluates each of its subexpressions in order and returns the result of the last one as the value of the whole expression.

Programs	P	$S^*\ e$
States	S	`state` s `case of` s `{` F^* M^* `}`
Fields	F	T f`=`e`;`
Methods	M	T m`(`π`>>`π s x`)` π`>>`π `{` e `}`
Permissions	π	`unique` | `none` | σ | `local` σ
	σ	`immutable` | `shared`
Types	T	π s
Expressions	e	`let` π x`=`e `in` e | e`.`m`(`e`)` | v | v`.`f | v`.`f`=`e | `match(`e`){(`s`=>`e`;)`$^+$`}` | `new` s | `{(`e`;)`$^+$`}`
Variables	v	`this` | x

Figure 7. Core language syntax. s, f, m, and x are identifiers.

3.2 Static Semantics

We introduce our static semantics by formalizing the notions of states and permissions. We then define the ways that permissions are manipulated and show how the flow of permissions is integrated into the type system.

States: States take the place of standard types in our system. Figure 8 gives the judgments and rules relating to states. $P \vdash s$ says a state is valid if it is declared in P. $P \vdash s \leq s'$ says that s is a substate of s'. Substate relations are defined by the `case of` relation from the state definitions, reflexivity, and transitivity.

$\boxed{P \vdash s}$

TYPE-STATE
$$\frac{\texttt{state}\ s\ \texttt{case of}\ s'\ \{\ F^*\ M^*\ \} \in P}{P \vdash s}$$

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

SUBSTATE
$$\frac{\texttt{state}\ s\ \texttt{case of}\ s'\ \{\ F^*\ M^*\ \} \in P}{P \vdash s \leq s'}$$

SUBTYPE-REFLEXIVE
$$P \vdash s \leq s$$

SUBSTATE-TRANSITIVE
$$\frac{P \vdash s \leq s' \qquad P \vdash s' \leq s''}{P \vdash s \leq s''}$$

Figure 8. Valid states and substates.

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

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

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

SPLIT-LOCAL-INCREMENT
$$\mathsf{borrow}(\pi, \sigma, n) \Rrightarrow \texttt{local}\ \sigma \otimes \mathsf{borrow}(\pi, \sigma, n+1)$$

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

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

SPLIT-SYMMETRIC
$$\sigma \Rrightarrow \sigma \otimes \sigma$$

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

SPLIT-ALL
$$\pi \Rrightarrow \pi \otimes \texttt{none}$$

Figure 9. Splitting variable permissions.

Permissions: To the set of source permissions we add an additional form needed to track `local` permissions so they can be joined back to `unique`:

$$\pi ::= \ldots \mid \mathsf{borrow}(\pi, \sigma, n),$$

where n is a natural number. The borrow permission appears only in the context for a local variable or field that has had local permissions split from it. Inside borrow, π represents the original permission before the first split (`unique` or `local` σ), σ represents the kind of local permission borrowed (`immutable` or `shared`), and n counts the number of borrowings.

Splitting variable permissions: The judgment $\pi \Rrightarrow \pi' \otimes \pi''$ says that if variable v has permission π, then π' can be taken out of v leaving π'' in v. Figure 9 gives the rules for this judgment. Notice that SPLIT-UNIQUE-SYMMETRIC permanently splits `unique` into symmetric permissions, making it impossible to ever regain the `unique` permission. In contrast, the SPLIT-LOCAL-* rules pull a `local` permission and leave behind a borrow permission which will be turned back into the original permission when all the `local` permissions are returned. In these rules, the count is set to reflect the net gain in `local` permissions: splitting a `local` from a `unique` generates a new `local`, so the count starts at 1 (SPLIT-UNIQUE-LOCAL); pulling from an existing `local` replaces the original `local` with a borrow, resulting in no net gain in `local` permissions, so the count starts at 0 (SPLIT-LOCAL); taking a `local` from an existing borrow creates a new `local` and thus increments the count (SPLIT-LOCAL-INCREMENT). Also, SPLIT-SYMMETRIC-LOCAL says we may pull `local` σ out of σ. However, we can't get a non-`local` permission out of a `local`, so (in conjunction with the FIELD typing rule) we can't store local permissions to the heap.

Splitting field permissions: The judgment $\pi_1.\pi_2 \Rrightarrow \pi_3 \otimes \pi_4$ says that if variable v has permission π_1, and field $v.f$ has permission π_2, then permission π_3 can be taken out of f, leaving π_4 behind.

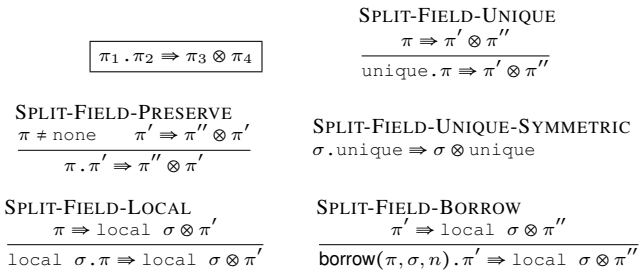

Figure 10. Splitting field permissions.

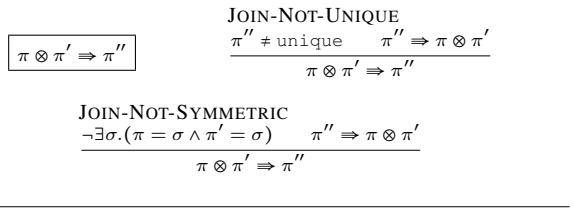

Figure 11. Joining permissions.

Figure 10 gives the rules. If the variable permission is `unique`, we can split permissions from the field as if it were a variable (SPLIT-FIELD-UNIQUE), possibly unpacking the field. Otherwise, we allow splitting in three cases. First, if the variable permission is not `none`, then we can do any splitting allowed for variables that preserves the original field permission (SPLIT-FIELD-PRESERVE). Second, if the variable permission is σ, then we can take permission σ out of a `unique` field $v \,.\, f$ without unpacking the object (SPLIT-FIELD-UNIQUE-SYMMETRIC). This is sound because all other references to the object must have permission σ for the remainder of program execution, and so all other accesses will consistently see the field with permission σ. Third, if the variable permission is `local` or borrow, then we can take a matching `local` permission out of a field with `unique` or borrow permission (SPLIT-FIELD-LOCAL, SPLIT-FIELD-BORROW). This is sound because all `local` permissions to the object and to the field must go out of scope before a `unique` permission to the object can be regained. In the mean time, all references to the object and to the field have compatible `local` or borrow permissions.

Joining permissions: The judgment $\pi \otimes \pi' \Rrightarrow \pi''$ says that permissions π and π' may be joined to form permission π''. The joining rules, given in Figure 11, are very simple: we just reverse all the variable splitting rules, except for SPLIT-UNIQUE-SYMMETRIC, which can't be undone.

Linear Context: To track permission flow, our typing rules use a *linear context*. It is similar to a standard typing environment, except that it maps variables to types that include a permission that may change over the course of typing. The linear context Δ is defined as follows, where i is chosen from an arbitrary set of identifiers:

$$\Delta ::= \varnothing \mid v : (T; \Pi), \Delta \qquad \Pi ::= \varnothing \mid f : (\pi, i), \Pi$$

For each variable v in scope, Δ stores a type $T = \pi\ s$ and a set Π containing the *unpacked state* $f : (\pi', i)$ of the fields f in the state s. A field $f : (\pi', i) \in \Pi$ records two pieces of information. First, π' indicates the current permission in the field. f will only appear in the unpacked state of v if π' is distinct from the declared permission of the field f in the state of v. Second, the object identifier i is used to prevent the restoration of permissions from one object to another object. For any $v : (T; \Pi) \in \Delta$ and field f declared in the state

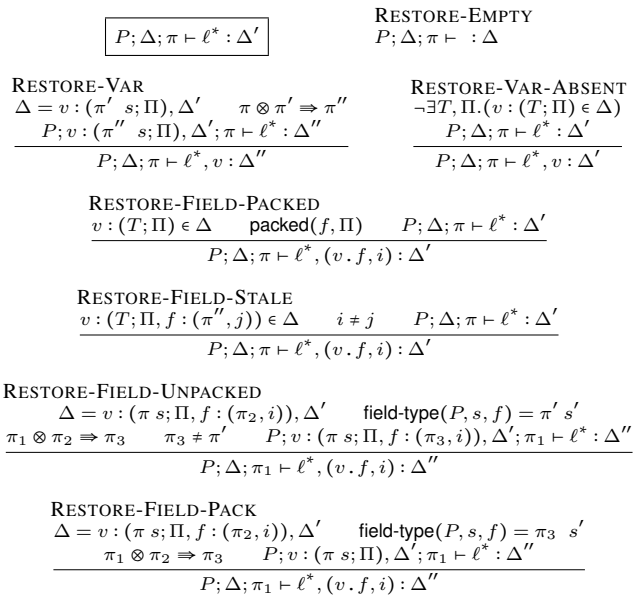

Figure 12. Restoring permissions. The predicate $\mathsf{packed}(f, \Pi)$ is true if no binding for f appears in Π. $\mathsf{field\text{-}type}(P, s, f) = T$ says that field f is defined in state s with type T in the program P.

of v, if there exists $f : (\pi, i) \in \Pi$, then we say "the field f of v is unpacked"; otherwise we say "the field f of v is packed." By extension, if $\Pi = \varnothing$, then we say the object pointed to by v (or just v) is packed; otherwise it is unpacked.

Restoring Permissions: At certain points in the typing, permissions flow back into the linear context, such as when a let-bound variable goes out of scope. We restore permissions to a *source location list* consisting of zero or more elements ℓ:

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

Each element ℓ stores a location to restore to (either a variable v or a field $v \,.\, f$). For fields $v \,.\, f$, the identifiers i ensure that we restore permissions only to fields that have not been assigned to since the permission was taken out. We use a list because `match` expressions may report different source locations for each `case`. However, we do not allow duplicates in the list.

The judgment for restoring permissions is $P; \Delta; \pi \vdash \ell^* : \Delta'$. It says that if we start with context Δ and join permission π with the permission in each location in ℓ^*, then we get a new context Δ'. Figure 12 gives the rules for this judgment. If the next element in the source location list is a variable appearing in the context, then we restore the permission to it (RESTORE-VAR). Because we do not check that all of the locations in a source location list remain valid when passed outside of a let scope, it is possible that the variable may not be in the context to return to, in which case we do nothing (RESTORE-VAR-ABSENT).

When restoring to a field, there are several cases to consider. If the field is already packed, this means that pulling the permission did not unpack the field (left it unchanged) or the field was reassigned since the permission was pulled, so we do nothing (RESTORE-FIELD-PACKED). If the field is unpacked, and the identifiers in the context and the source location don't match, then we also do nothing (RESTORE-FIELD-STALE). This occurs when an expression in an inner scope assigns to the field and then later unpacks it again, meaning the returned permission represents a permission to a different object than is in the field.

$\boxed{\Delta_3 = \mathsf{merge}(\Delta_1, \Delta_2)}$

$$\textsc{Merge-Packed}\quad \frac{\pi_1 \Rrightarrow \pi_3 \otimes \pi_2}{v : (\pi_2\ s, \varnothing) = \mathsf{merge}(v : (\pi_1\ s, \varnothing), v : (\pi_2\ s, \varnothing))}$$

$$\textsc{Merge-One-Unpacked}\quad \frac{\Pi \neq \varnothing \qquad \pi_1 \Rrightarrow \pi_3 \otimes \pi_2}{v : (\pi_2\ s, \Pi) = \mathsf{merge}(v : (\pi_1\ s, \varnothing), v : (\pi_2\ s, \Pi))}$$

$$\textsc{Merge-Unpacked-Equal}\quad \frac{v : (\pi_1\ s, \Pi_3) = \mathsf{merge}(v : (\pi_1\ s, \Pi_1), v : (\pi_1\ s, \Pi_2)) \qquad \pi_1 \mathbin{.} \pi_2 \Rrightarrow \pi_4 \otimes \pi_3}{v : (\pi_1\ s, \Pi_3 \cup f : (\pi_3, i)) = \mathsf{merge}(v : (\pi_1\ s, \Pi_1 \cup f : (\pi_2, i)), v : (\pi_1\ s, \Pi_2 \cup f : (\pi_3, i))}$$

$$\textsc{Merge-Unpacked-Unequal}\quad \frac{v : (\pi_1\ s, \Pi_3) = \mathsf{merge}(v : (\pi_1\ s, \Pi_1), v : (\pi_1\ s, \Pi_2)) \qquad \pi_1 \mathbin{.} \pi_2 \Rrightarrow \pi_4 \otimes \pi_3 \qquad i_1 \neq i_2 \qquad \mathsf{fresh}(i_3)}{v : (\pi_1\ s, \Pi_3 \cup f : (\pi_3, i_3)) = \mathsf{merge}(v : (\pi_1\ s, \Pi_1 \cup f : (\pi_2, i_1)), v : (\pi_1\ s, \Pi_2 \cup f : (\pi_3, i_2))}$$

Figure 13. Merging contexts in typing `match` (selected rules).

If f is unpacked, and the identifiers match, then there are two cases. First, if restoring the permission doesn't leave a permission equal to the declared permission for f, then we update the unpacked state (RESTORE-FIELD-UNPACKED). Otherwise we pack up the field (RESTORE-FIELD-PACK) by removing it from Π. Notice we don't consider the case where $v\,.\,f$ appears in the location list but v doesn't appear in the context. Why not? There are two ways this could happen: either $v\,.\,f$ is returned as the value of the `let` expression that declares v, or $v\,.\,f$ is returned as the value of a method body, where v is `this` or the method formal parameter. In either case v must be packed after the evaluation of $v\,.\,f$ (rules METHOD in Figure 15 and LET in Figure 14). Therefore, $v\,.\,f$ can not appear in the source location list (see rule FIELD-ACCESS-PACKED in Figure 14).

Merging Contexts: Each `case` in a `match` expression may update the permissions in the context in a different way. The merge judgment from Figure 13 defines how to combine two contexts into a more general context that can be soundly used to type subsequent expressions regardless of which `case` is actually executed. We provide only the important rules for merging contexts Δ_1 and Δ_2; the rest of the rules are about pulling apart the contexts and comparing elements. In summary we use the following rules for merging the two context entries for a variable v:

1. If v is packed in both Δ_1 and Δ_2 (MERGE-PACKED), then we choose the weaker permission. The weaker permission is the one which can be defined as the residue after some permission is split from the stronger permission. If such a split does not exist, then the contexts are inconsistent and the typing fails.
2. If v is unpacked in both contexts, then we require it to have the same permission in both, and we use the field splitting rules to find the weaker of the two field permissions. If the identifier i associated with the unpacked field is the same in both contexts, then we pass it through (MERGE-UNPACKED-EQUAL), but if it is different we generate a fresh identifier (MERGE-UNPACKED-UNEQUAL). This is correct but conservative because it guarantees that no permission restore will occur to the location.
3. If v is unpacked in only one context (MERGE-ONE-UNPACKED), then we use the unpacked element, but we check that the unpacked permission for v is weaker than the packed one. Otherwise, we would have to pull a permission out of an unpacked object, which is not allowed in this language.

Expressions: Permissions are pulled out of the linear context by typing expressions. The judgment $P; \Delta; \pi \vdash e : s; \Delta'; \ell^*$ takes the starting context Δ, a needed permission π, and an expression e that specifies where π could be pulled from. It produces the state s of the expression, an updated context Δ', and a source location list ℓ^* that contains all the possible locations in the context that the permission π may have been pulled from. The exact location is not statically known because which branch of a `match` expression is executed is determined at runtime. Figure 14 gives the rules for typing expressions which we now summarize.

Let: We use the permission π' declared for x to type e, obtaining state s, a context Δ', and a source location list ℓ^*. Then we use the needed permission π of the whole expression to type e' in the context Δ' augmented by the type binding for x. This produces an updated context Δ'' where x is left with permission π'', and a source location list ℓ'^*. We require that x be packed because it is going out of scope. We restore π'' to ℓ^*, the locations that the permission for x may have been pulled from, which generates a final context Δ''' that is returned along with the state and source locations from the body.

Match: For a single-case `match`, we type e with a needed permission `none` yielding some state s and an updated context Δ'. We disregard the source location lists since returning `none` is a no-op. We ensure that the state s has a common superstate with the state s_c named in the `case`. This ensures that s_c is a potentially valid state of e. Using Δ' and the needed permission for the entire expression, we type e' to get the state s' and source location list ℓ^* which are reported as the result of the `case`. For a multiple-case `match`, we recursively check the `match` with all but the first `case`. Then we check the `match` with the remaining `case` in isolation using the original context. Finally, we return the the least upper bound of the resulting states, the merged contexts, and the union of the source location lists from the first `case` and the remaining cases.

Method invocation: We type e in the input context with needed permission π_1 yielding a state s_1 and a context Δ_2.[3] Next we look up the method named m in state s_1. This gives us the permission π_2 required for the argument. We use π_2 to type e' in the context Δ_2, which yields a state s_2' and a context Δ_3. Now we check that s_2' conforms to the method parameter state s_2. Then we check that needed permission π can be extracted from the permission π_3 returned by the method. We generate the output context Δ_5 by restoring the output permissions specified for the argument and receiver in the method signature to their respective source location lists. The outgoing source location list is empty because our system does not track what locations the permissions returned from method calls can come from.

Variables: We require that v is packed, because its value may be assigned to another variable or stored on the heap. We also check that the existing permission can be split to give the needed permission, and we leave the residue in the type reported in the outgoing context. The source expression list contains only the variable itself.

Field access: We start by finding the type of v in the context and then get the defined type of field f in its state s. Since we are only accessing the field of v, we do not update the permission of v. However, we may need to update the unpacked state of f in v. If f

[3] π_1 is the needed permission for the receiver as determined by the signature of the method m in the state s_1 of the expression e. This permission can be determined by a pre-pass that ignores permissions and just gathers state information for all variables.

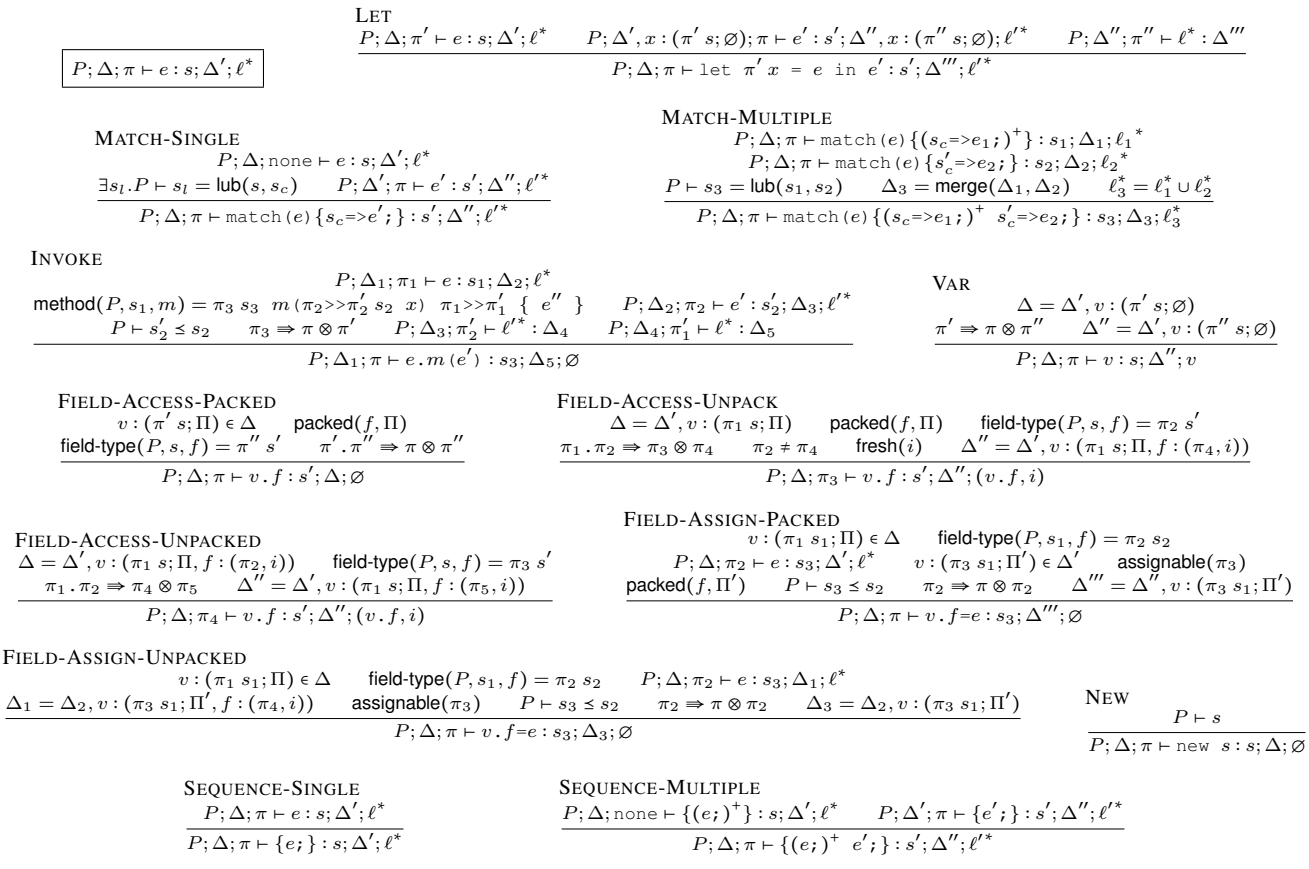

Figure 14. Typing expressions. $P \vdash s = \mathsf{lub}(s', s'')$ means s is the least state that is a superstate of both s' and s''. $\mathsf{method}(P, s, m) = M$ means that M is the method named m defined in state s for program P, and $\mathsf{field\text{-}type}(P, s, f) = T$ similarly produces the type T of field f from s in P. The predicate $\mathsf{assignable}(\pi)$ holds if π is not `none`, `immutable`, `local immutable`, or $\mathsf{borrow}(\texttt{local immutable}, \texttt{immutable}, n)$. $\mathsf{packed}(f, \Pi)$ means $\neg\exists(\pi, i).(f : (\pi, i) \in \Pi)$.

starts packed in v's type, then there are two cases to handle. First, if we can take the required permission out of the field and leave the same permission behind, then we leave the object packed (FIELD-ACCESS-PACKED). The source expression list is empty because no permission needs to be restored to a field that is packed. Second, if we need to leave a different residual permission, then we unpack the field, leaving the residual permission behind (FIELD-ACCESS-UNPACK). We also generate a fresh identifier i and report $(v\,.\,f, i)$ as the source location list. If f is already unpacked in v, then we split the needed permission from the current field permission, and replace the current permission in the unpacked field state with the residual permission (FIELD-ACCESS-UNPACKED). We report $(v\,.\,f, i)$ as the source location, using the existing identifier i.

Field assignment: If $v\,.\,f$ is packed, then to assign e to it we (1) look in the context to get the permission we need for the field; (2) type e; (3) check that we have writable permission to v in the resulting context; and (4) check that the states match (FIELD-ASSIGN-PACKED). We also ensure that the permission we need can be split off from the permission needed by the field, while retaining the field permission. If $v\,.\,f$ is unpacked, then we do the same thing, but we pack up f at the end (FIELD-ASSIGN-UNPACKED). In both cases, we do not need to return permissions to the source locations of e because what's left after assigning to $v\,.\,f$ must be `none` or symmetric and returning either is a no-op. For the same reason, the returned source location list is empty.

Object creation and expression sequence: These rules are straightforward. In SEQUENCE-MULTIPLE, notice that we pull the needed permission only for the last element in the sequence, whose value is returned by evaluating the expression; we pull `none` from the rest of the expressions in the sequence. For example, if `x` is `unique`, it is legal to request a `unique` permission from the expression `{x; x; x;}`, because `unique` is only pulled from the last `x`. Since returning a `none` permission is a no-op, we can safely discard the source location lists for these expressions.

Top-Level Program Structure: Figure 15 gives the rules for typing programs, states, fields, and methods. In rule PROGRAM we type the main expression with a needed permission `none`, because no permission to the result is needed after program execution is complete. In rule FIELD we type the initializer expression in the empty environment to ensure that `this` isn't stored to the heap by a `new` expression. Also, we require that user-declared field types cannot be declared with `local` permissions (borrow permissions are also excluded because they cannot appear in the source). This restriction ensures that a local permission is never assigned to the heap. In rule METHOD we check that the return type and parameters are valid. We check that the method body types in the context created by binding the method receiver and parameter to their types and that the resulting state is a substate of the return state. We require that the parameter and receiver be packed in the the output context Δ' and also that their permissions in Δ' agree with the output permissions given by their change types.

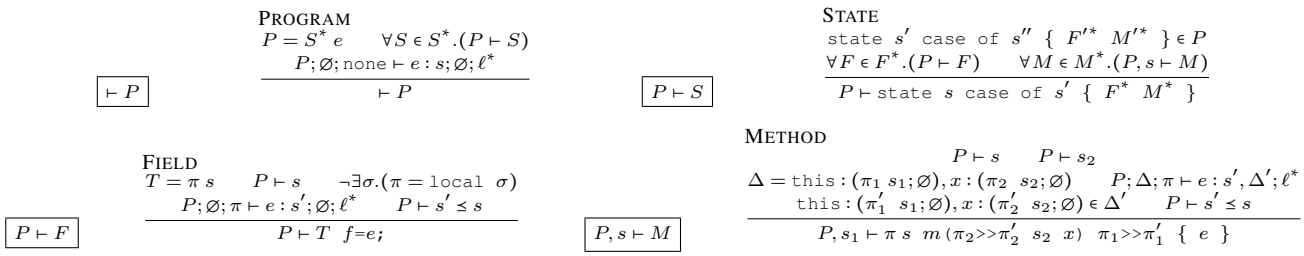

Figure 15. Programs, states, fields, and methods.

For simplicity, we implicitly disallow overriding, shadowing, and/or overloading of fields and methods. We also disallow cycles in the inheritance hierarchy and treat the set of states as a forest of trees, where a state s is a root if it has itself as a parent.

3.3 Dynamic Semantics

An execution state $H; e$ includes a heap H and an expression e.

Heap: Our model of the heap H is a partial function from object references to object identifiers and from object identifiers to objects. The addition of *object references* provide an additional level of indirection that allows us to explicitly track and reason about different aliases to a single object as in [28]. These serve a formal purpose only and are not necessary in an implementation. Consistent with this indirection, the fields of an object dF^* map field names to object references. Every object reference in $\mathsf{dom}(H)$ maps to an object identifier except the special reference null, which is used during object initialization.

$$\begin{array}{rcl} H & ::= & \mathsf{null} \mid o \mapsto O, H \mid O \mapsto s\{dF^*\}, H \\ dF & ::= & f \mapsto o \end{array}$$

Expressions: We enhance our expression language to support partially-executed programs. We add object references o as well as the new form $\texttt{alias}(o)$ as expressions. $\texttt{alias}(o)$ gives computational meaning to splitting a permission of o by creating a new alias to hold the split permission. As alias expressions will be substituted for bound variables in let bodies, we allow them to appear wherever variables can. Finally, we add a partial let form that keeps track of the scope in which an object reference o is bound.

$$\begin{array}{rcl} e & ::= & \ldots \mid \texttt{o} \mid \texttt{alias}(o) \mid \texttt{alias}(o)\texttt{.}f \mid \texttt{alias}(o)\texttt{.}f\texttt{=}e \mid \\ & & \texttt{let } o \texttt{ in } e \end{array}$$

Reduction Rules: We formalize program execution using a small step operational semantics. Given a program P, the reduction rules shown in Figure 16 take one execution state to another.

Congruence: We specify reduction of subexpressions using an evaluation context E defined below:

$$\begin{array}{rcl} E & ::= & \Box \mid \texttt{let } \pi\ x\texttt{=}E \texttt{ in } e \mid \texttt{let } o \texttt{ in } E \mid \\ & & \texttt{match}(E)\{(s\texttt{=>}e\texttt{;})^+\} \mid E\texttt{.}m(e) \mid \\ & & o\texttt{.}m(E) \mid \texttt{alias}(o)\texttt{.}f\texttt{=}E \mid \{E\texttt{;}\ (e\texttt{;})^*\} \end{array}$$

As usual, $\Box$ is a "hole" that is filled in with the subexpression under evaluation to construct the entire expression being evaluated.

Alias: Reducing an $\texttt{alias}(o)$ expression creates a fresh object reference that points to the same object identifier as the original object reference.

Let: Once the bound expression is reduced to an object reference o we substitute $\texttt{alias}(o)$ for all occurrences of the bound variable x in the body. Thus, any use of x in the body must create a fresh alias. We also keep track of the scope of o by reducing to the partial let form. Once the body of a partial let has been evaluated to an object reference we remove the scoping annotation.

Match: Once the selector expression becomes an object reference o, we select the first match case, if any, that is a superstate of the state of o. If there is no match, then execution gets stuck.

Method invocation: Once we have object references for the receiver and argument of the method, we substitute the method body surrounded by let expressions binding the receiver and argument. This allows let reduction to handle the substitution and scoping.

Object creation: We create a new object identifier O and give its fields the reference null, which will be replaced by the proper values after running the initializer expressions. We reduce to a sequence expression which initializes each field of the object in turn through a fresh object reference o. The final expression in the sequence returns o.

Field access and assignment: A field access evaluates to a fresh alias to the object reference the heap associates with the field. Field assignment replaces the object reference in the field with object reference from evaluating the right-hand side. It returns a fresh alias of this same reference. Since we do not use the target of the field read or assignment except to access its field, we do not need a new alias to it, so we leave the $\texttt{alias}(o)$ in the target unevaluated.

Sequence: Each expression in the sequence is evaluated in order until there are no further expressions at which point the sequence is reduced to the object reference resulting from the reduction of the final expression.

3.4 Soundness Results

In this section, we present the main soundness results and supporting definitions for our system. Our technical report [23] contains the full definitions and proofs.

Typing of Dynamic Expressions: We state and prove soundness in terms of a standard dynamic typing. To do this, we need rules for typing object references, alias expressions, and partial lets.

First, we enhance the definition of the linear context to map object references to types. We also add entries of the form $\ell \Leftarrow o$ to the context which indicate that o returns its permission to the source location ℓ when it leaves scope. We will also want to be able to return permissions to object references, so we extend the definition of source locations to include them.

$$\begin{array}{rcl} \Delta & ::= & \ldots \mid o : (\pi\ s, \Pi), \Delta \mid \ell \Leftarrow o, \Delta \\ \ell & ::= & \ldots \mid o \mid (o.f, i) \end{array}$$

The rules for typing partial expressions are found in Figure 17. The ALIAS rule says that typing $\texttt{alias}(o)$ proceeds by typing o like a variable. In particular, the source location is o. A raw object reference o is typed using rule OBJECT-REF which is also similar to the VAR rule with two important exceptions. First, we look in the context to find the return location specified for o. This is because we need to maintain the typing after an alias expression steps to an object reference. In particular, after $\texttt{alias}(o)$ steps to o', typing o' should still return o as the source location. Second, the binding for the type of o is removed from the outgoing context to ensure that o is not used later in the program which would break

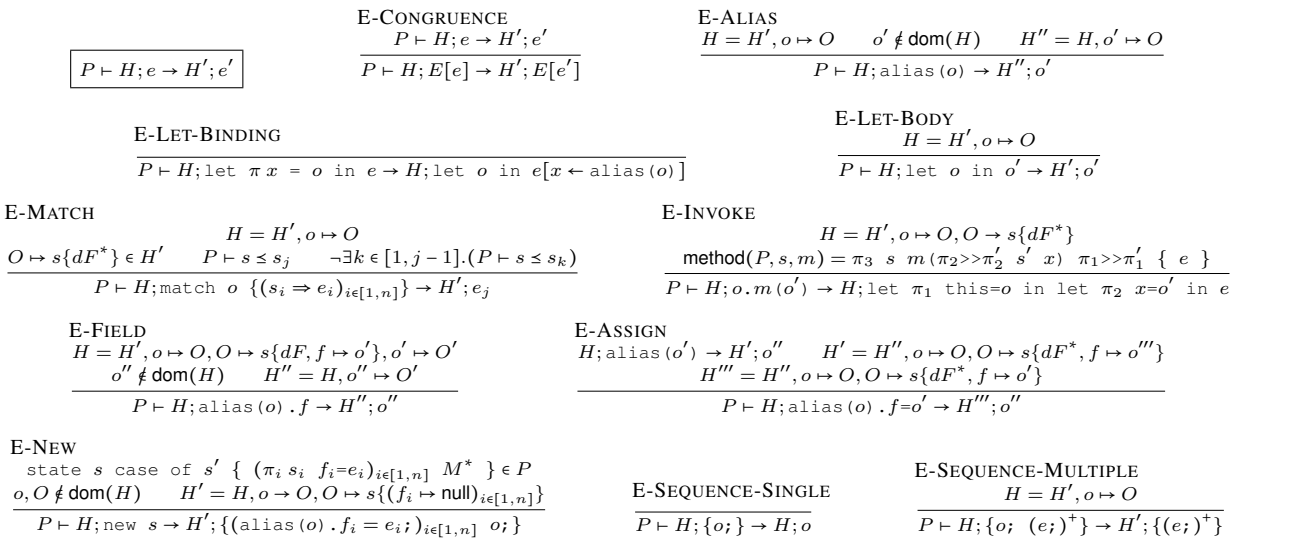

Figure 16. Reduction rules.

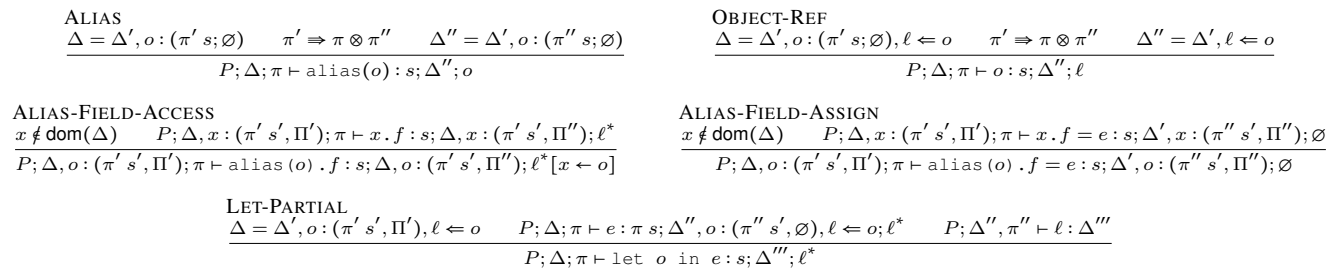

Figure 17. Intermediate Typing Rules.

our model of execution where each variable and field is tracked as a separate object reference. To type a field read or assignment after an `alias` has been substituted for the target variable ALIAS-FIELD* undoes the substitution with a fresh variable x that replaces `alias`(o). We give x the type of o from the context when typing the updated expression and then return the resulting outputs with o put back for x in the context and source location list. LET-PARTIAL assumes the scoped object reference o is already in the context along with a return location for it. After the body is typed, the remaining permission to o is restored to its return location.

Soundness Judgments: Figure 18 shows the soundness judgments for the system. $P; \Delta; \pi \vdash H; e : \pi\ s; \hat{\Delta}; \ell^*$ is the top-level judgment and says that the runtime environment $H; e$ is well-typed with respect to a a program P, a context Δ, and a permission π. The premises of this judgment are as follows. First, the context and the heap must map the same set of object references. Second, the incoming context must be able to be partitioned into two distinct parts: Δ_e to type the expression (which produces the outputs of the judgment), and Δ_f which contains all of the object references stored in fields f in H. Third, three invariants must hold for the context and the heap: (1) consistent object permissions, (2) consistent object references, and (3) distinct fields.

Consistent object permissions: The key soundness condition of our system is that for each O, the set of references o that point to it must have *consistent* permissions in Δ. This condition captures the meaning of the permissions in our system. Figure 19 defines this consistency condition. The judgment $\mathsf{perms}(\Delta, H, O) = [\pi^*]$ forms the list of all the permissions π given to object references o that point to O in H. The judgment $[\pi^*]$ **consistent** places three requirements on this list: (1) if `unique` appears in the list, then it appears only once, and all other permissions are `none`; (2) if a $\mathsf{borrow}(\mathtt{unique}, \sigma, n)$ permission appears in the list, then the number of `local` σ permissions must match the sum of the counts of all borrow permissions in the list; and (3) no list may contain both `shared` and `immutable` permissions (including the `local` and borrow versions).

Consistent object references: Object references must also be internally consistent. We say that $P; \Delta; H \vdash o$ **ok** if the representation of the object reference o in H is consistent with the type of o in Δ. For the representation to be valid, we first need for the actual states of the object and its fields to be substates of the states given by the type of o. Second, we need to ensure that any permission that can be statically pulled from a field f of the state s through o can also be pulled from the permission of the object reference that represents the field in H. Given the field type $\pi_o\,.\,\pi_f$ of f in o, we say that π_r *fulfills* $\pi_o\,.\,\pi_f$ (represented by $\pi_r \rhd \pi_o\,.\,\pi_f$) if any permission that can be split from $\pi_o\,.\,\pi_f$ can also be split from π_r. Both judgments are formalized in Figure 18.

Distinct fields: In order to carry out the typing of a partially evaluated expression we must ensure that no object reference o ever appears (1) both in a field and in the expression being typed or (2) in two different fields in H. Otherwise, typing one location could have a non-local impact on the permission in the other location that

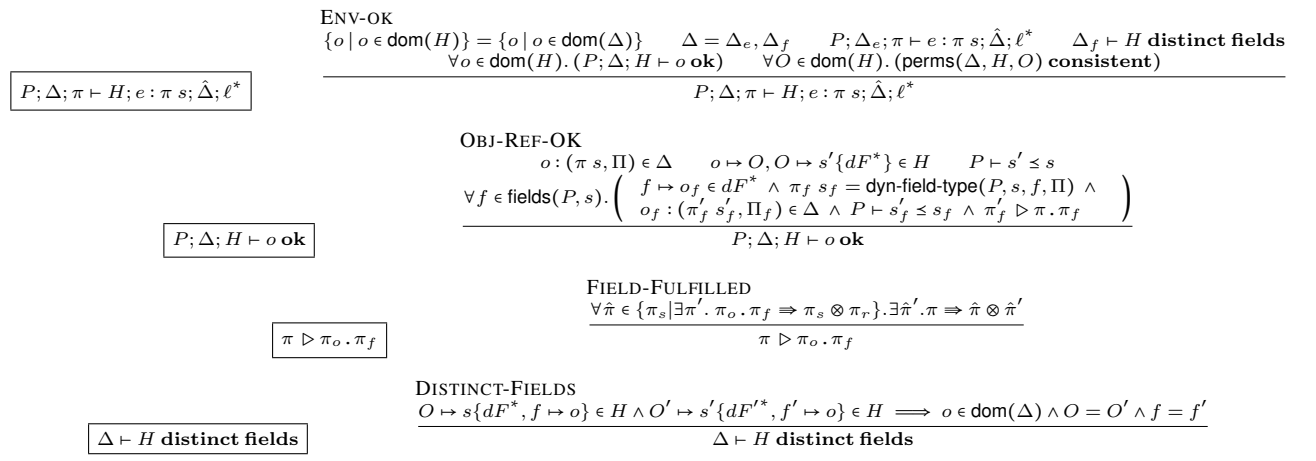

Figure 18. Soundness judgments. $\mathsf{dyn\text{-}field\text{-}type}(P, s, f, \Pi)$ reports the permission for f in Π if it is recorded there, otherwise the declared permission of f in the definition of state s.

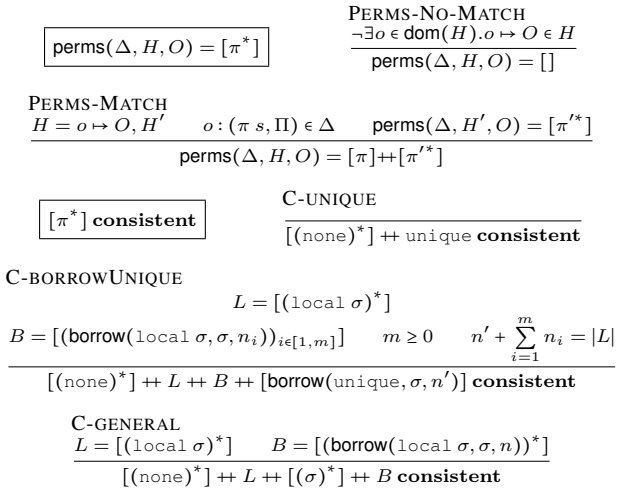

Figure 19. Permission lists and consistent permissions. The notation $L = [\pi^*]$ means that L is a list of permissions (with duplicates allowed). The symbol $+\!\!+$ denotes list concatenation.

would be difficult to track. This invariant is captured formally by the judgment $\Delta \vdash H$ **distinct fields** in Figure 18.

Context Splitting: As an expression is evaluated, more definite information about the execution state becomes available for use by the typing rules. For instance, after reducing away a `match` expression the needed permission may be pulled from fewer locations in the context, leaving the resulting context stronger than before the reduction. We introduce the concept of *context splitting* to account for this increase in permissions in the output context.

The judgment $P \vdash \Delta_1 \Rightarrow \Delta_2$ shown in Figure 20 states that Δ_1 splits into Δ_2. The judgment holds if we can transform Δ_1 into Δ_2 by doing one or both of the following to each location (variable or field) appearing in both Δ_1 and Δ_2: (1) pull a permission out of the location in Δ_1 such that the residue is the permission of the location in Δ_2; and (2) replace the state of the location in Δ_1 with the superstate from Δ_2. The helper judgments $P \vdash \Delta_1 \Rightarrow \Delta_2@v$ and $P \vdash \Delta_1 \Rightarrow \Delta_2@v.f$ define these operations formally for variables and fields.

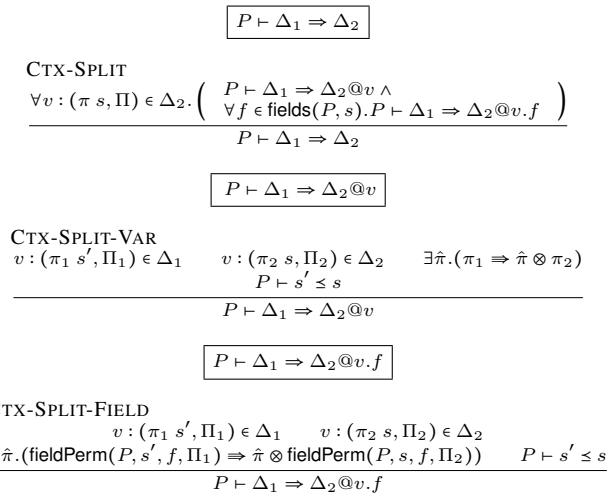

Figure 20. Context splitting.

Progress and Preservation: We use these definitions to state the soundness of our system via standard progress and preservation theorems.

Theorem 3.1 (Progress)**.** *If $P; \Delta; \pi \vdash H; e : \pi\ s; \hat{\Delta}; \ell^*$, then either $e = o$ or $P \vdash H; e \to H'; e'$ or execution is stuck at a match statement where the matched expression has been evaluated to an object, but there is no matching case.*

Theorem 3.2 (Preservation)**.** *If $P; \Delta; \pi \vdash H; e : \pi\ s; \hat{\Delta}; \ell^*$, and $P \vdash H; e \to H'; e'$, then $\exists \Delta'$ such that*

1. *$P; \Delta'; \pi \vdash H'; e' : \pi\ s'; \hat{\Delta}', \ell'^*$ where $P \vdash \hat{\Delta}' \Rightarrow \hat{\Delta}$ and $P \vdash s' \preceq s$*
2. *$P, \hat{\Delta}, \pi \vdash (\ell^* \setminus \ell'^*) : \hat{\Delta}_r$ where $P \vdash \hat{\Delta}' \Rightarrow \hat{\Delta}_r$*

The first condition of preservation requires that the new environment after reduction is still well-formed, with the extra restriction that the context $\hat{\Delta}'$ generated by typing the updated expression e' must split into the context $\hat{\Delta}$ produced by typing the original e. However, this is not strong enough because it does not account for the permissions that were pulled out as a part of typechecking

which may be returned to the context later. The second condition provides the extra power we need. Consider the source location list $\ell^* \setminus \ell'^*$ representing all the source locations from which the needed permission π was pulled when typing e but that were not used to get π when typing e'. There is potentially more permission at these locations in $\hat{\Delta}'$ than in $\hat{\Delta}$ because π was not split from them during the typing of e'. However, are we guaranteed that if we restore the pulled permission to these locations in $\hat{\Delta}$ using the judgment defined in Figure 12, the resulting context $\hat{\Delta}_r$ can still be split from $\hat{\Delta}'$? The second condition of preservation says, "Yes." This fact is necessary to prove the splitting condition on $\hat{\Delta}'$ from the first condition in some cases such as when E-CONGRUENCE is used to reduce a `let` expression. This and other details of the proofs of these theorems are discussed in our accompanying technical report [23].

4. Related Work

Wadler first introduced the concept of temporarily converting a linear (`unique`) reference into a non-linear reference using the `let!` construct [27]. Other early uniqueness type systems built on his work and added support for borrowing as a special annotation, such as "borrowed" or "lent" [1, 19, 22]. 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") [7]. 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 [3], 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 [8]. 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 [3], Chalice [17], and VeriFast [21] 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 [10]. 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 [2]. While the technical details are somewhat different, the system presented in [2], 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 [2]. Our system additionally supports share permissions, match expressions and sequences, and our system tracks permissions taken out of individual fields. Unlike [2], 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 [25] is a functional language with support for unique references. However, since the language is functional, there is no concept for returning permissions to a previous location as in our system.

The Vault programming language [13] 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 [18] 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 [28] 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 [26]. 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.

Acknowledgements

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. We thank the anonymous reviewers for their helpful feedback.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

[2] K. Bierhoff. Automated program verification made SYMPLAR: SYMbolic Permissions for Lightweight Automated Reasoning. In *Onward!*, 2011.

[3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.

[4] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *OOPSLA*, 2009.

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

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

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

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

[9] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, 2001.

[10] J. T. Boyland and W. Retert. Connecting Effects and Uniqueness With Adoption. In *POPL*, 2005.

[11] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

[12] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, 2004.

[13] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

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

[15] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, 2002.

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

[17] S. Heule, R. Leino, P. Müller, and A. Summers. Fractional permissions without the fractions. In *FTfFP*, 2011.

[18] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, 2004.

[19] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*, 1991.

[20] R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy. *The Turing Language: Design and Definition*. Prentice-Hall, 1988.

[21] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, 2011.

[22] N. H. Minsky. Towards alias-free pointers. In *ECOOP*, 1996.

[23] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. Technical Report CMU-CS-11-142, Computer Science Department, Carnegie Mellon University, December 2011.

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

[25] S. Smetsers, E. Barendsen, M. van Eekelen, and R. 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.

[26] J. A. Tov and R. Pucella. Practical affine types. In *POPL*, 2011.

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

[28] R. Wolff, R. Garcia, Éric Tanter, and J. Aldrich. Gradual typestate. In *ECOOP*, 2011.