IMPLEMENTING PERMISSION ANALYSIS

by

William S Retert

A Disertation Submitted in

Partial Fulfillment of the

Requirements for the degree of

Doctor of Engineering

in

Computer Science

at

The University of Wisconsin-Milwaukee

May 2009

IMPLEMENTING PERMISSION ANALYSIS

by

William S Retert

A Disertation Submitted in

Partial Fulfillment of the

Requirements for the degree of

Doctor of Engineering

in

Computer Science

at

The University of Wisconsin-Milwaukee

May 2009

John Boyland	Date
Graduate School Approval	Date

IMPLEMENTING PERMISSION ANALYSIS

By

William S Retert

The University of Wisconsin-Milwaukee, 2008

Under the Supervision of Professor John Boyland

ABSTRACT

Annotations allow programmers to express design intent. A system of quasi-linear fractional permissions can support a wide range of program annotations; however, checking the fractional permissions is nontrivial. Permission checking can be approximated, with formal rules and a fully implemented control flow analysis, well enough to accurately check uniqueness and effects annotations.

John Boyland Date

Contents

List of Figures

Chapter 1

Annotations and Intent

One of the largest areas of concern in computer science is the tendency of software to not behave as intended, both on the large scale and the small. Software engineering features a panoply of techniques to verify—even partially—the correctness of programs. Much of this effort is spent on empirical verification in the form of testing, or informal verification in the form of manual review of source code. However, it is also possible to attempt automated formal verification of program; often these prove the correctness of secondary annotations, ensuring the absence of particular errors. An example of this is the type system, which obviates type errors. Herein I detail one particular implementation of one particular formalism for checking the accuracy a particular regimen of annotations. Ultimately, being able to demonstrate that programs are correctly annotated will tend to increase their correctness.

1.1 Why Annotations

According to C.A.R. Hoare [Hoa69], "The most important property of a program is whether it accomplishes the intentions of its user." To this end, program verification attempts to prove programmer-provided assertions. Such assertions are typically couched in formal logic, and added as annotations to code [LBR99, KMJ02]. Unfortunately, the intentions of the user for an entire program can be complex. Additionally, the logic language used to encode the desired behavior must employ low-level descriptions of machine state for (semi-)automated verification to be practicable. Thus the resulting annotations tend to be complex and difficult to understand; this forms a strong barrier to adoption.

Conversely, simpler annotation schemes [CBS98] involving intuitively grasped concepts present less work to the programmer. Such annotations lack full expressive power and therefore cannot be used to fully verify programs, as they are insufficient to describe the intentions of the user. However, they can express some portion of the programs *design intent*. That is, they allow programmers to relate high level design intuitions, and an automated annotation-checking tool can assure that the code satisfies the model used for its design. Programmers must still show that the program as designed will satisfy the needs of the user, but are assured that the code will perform according to that design.

Additionally, as the annotations for design intent generally involve particular properties, full logical reasoning can be replaced with program analysis. When using logical predicates to describe program state, the predicates may at any time be replaced with a logically equivalent or logically consequent set; this is embodied by the rules of consequence in Hoare logic. At worst, this requires an (in general) undecidable proof

to produce intermediate steps. Program analysis (in particular, control-flow analysis and abstract interpretation) approximates program states using lattices (partial orderings). Well-developed theory [NNH99a] shows that if the partial ordering is complete and obeys the ascending chain condition, and if the program changes state monotonically, a simple worklist algorithm can find a safe fixed-point approximation of the program state. Thus, properties, such as some forms of design intent, may be checked automatically.

One area where this approach seems particularly applicable is in verifying programs with aliasing. By their nature, programs containing a large number of pointers are difficult to reason about. Moreover, by their nature, object-oriented programs contain a large number of pointers (consider Java). This has led to recent efforts to hybridize verification with pointer analysis [LARSW00] or to use more complex logics [IO01] which are aliasing aware. Equally, over the last decade, many proposals have been made concerning annotations of design intent for pointer usage. Common proposals include various forms of ownership, read-only references, and unique references. Mostly, these proposals extend object-oriented programs to better represent the encapsulation of an object's state.

1.1.1 Analysis and Assurance

The Fluid project contains divergent functionalities generally intended to assist the fluid evolution of code over time by maintaining consistency between the evolving program and the relatively consistent intent of the programmer(s) as expressed by annotations on the program. Among these are the underlying versioning infrastructure (the Fluid IR), support for control-flow analysis, a mechanism (double-checker) for assurance of code, and a system (Drop-Sea) for maintaining the connection between code and intent. A natural way to extend the system is by adding a new

assurance and also, adding a new analysis to support this assurance.

An *assurance* is the feedback a programmer receives documenting that the program does or does not meet its design intent as stated by annotations on the program. Commonly the assurance is equally referred to as an analysis; for this writing, I will make a distinction. Here, an *analysis* is a static tool which approximates the runtime state of the program. In particular, a control-flow analysis uses lattice values, iteratively propagated to a fixed point, as its approximations. For convenience, this paper will use analysis and control-flow analysis interchangeably; in practice most analyses used in Fluid are not control-flow analyses.

An assurance is usually implemented using an analysis. The analysis approximates the behavior of the program (including some behaviors assumed from specification viá annotation[1]). The assurance can prove that certain annotations accurately describe the program by comparing the program behavior as approximated by the analysis with the behavior as specified by the annotations being checked. These comparisons are nontrivial in control-flow analyses because the analysis' truth is computed by iteration; assurance is only meaningful on the final result, not interim lattice states.

Consider a simple `NonNull` control-flow analysis that passes around an intersection lattice representing the set of local variables which are known not to be null. One may desire to use this analysis to implement an assurance that a given flow unit cannot throw a `NullPointerException`. For example, given the code in Figure 1.1, the analysis could use the `if` test to determine that `x` is not null and the assurance can use the fact that `x` is not null to argue that calling `somefun()` will not generate a `NullPointerException`.

[1]Generally, this is not the annotation currently being checked.

```
@noNPE
void foo(SomeClass x){
  if(x != null) x.somefun();
}
```

Figure 1.1: A simple example

1.1.2 Some Proposed Annotations

For many years now, researchers have proposed various annotations indicating design intent on pointers.

In ownership types [BLR02, BSBR03, Cla01, CNP01, CPN98, MPH00, AC04], each object "owns" its representation. Objects are not allowed to see, use or store pointers into another object's representation.[2] Programmers communicate desired ownership relationships by annotating each object by a symbolic ownership domain. By limiting domains to those passed from above and one representing the current object one eliminates the possibility of a pointer into a separate object's representation. In addition to ownership types, there are comparable ways to prevent aliasing into encapsulated representation, such as confined types [BV99, ZPV03, ZPV06].

Alternately, one could annotate certain references as `read-only` [MPH00, BE04a, BE04b, ETT05]. The object pointed-to could be read through this pointer, but not written. Read-only pointers may be transitive; some systems provide casts to and from read-only. Objects may encapsulate state by forcing pointers from "outside" the encapsulated boundary to be read-only. However, as Boyland [Boy05] argues, read-onliness is not an ideal model because it does not prevent inappropriate information flow. It only prevents changes to encapsulated state, and not even those if the read-only gets cast away. In a similar vein, Haack [HPSS07] uses an ownership system to

[2]This is different than `private` modifiers in that privacy protects names on a per-class basis and ownership protects objects on a per-object basis.

support immutable classes.

Another model for access control is uniqueness, which will be the focus for much of the remainder of this paper. A unique pointer is, conceptually, the only reference to a particular object. This allows for objects to be de facto owned and encapsulated by placing them in unique fields. There have been many proposed implementations, with differing semantics. Most systems include several other kinds of pointers. Non-unique (shared) pointers should be allowed; not every object should have only one pointer to it. Under certain circumstances (e.g. within a method) it should be possible to violate uniqueness constraints; otherwise unique references are nigh unusable. Temporarily aliasing unique pointers is generally called "borrowing".

1.2 Uniqueness

One of the running problems with many of these proposed annotations is that they have been proposed in isolation. Each proposal includes rules proscribing the use of annotated values; however, the exact semantics of the annotation are murky. Especially murky are the semantics of using more than one proposed annotation at the same time. If pointers can be `read-only` and `unique`, then is it possible to have a field that is both unique and read-only? Can one store a unique parameter in a read-only field, or pass a unique field as a read-only parameter? Blindly following the rule for each annotation will only work if they are independent; if being read-only has no effect on being unique.

But even without worrying about interactions between annotations (yet), a well-founded semantic definition is vital. If a `unique` pointer is one that follows arbitrary rule set R, does "uniqueness" have a meaning beyond "follows the rules in R"? Or is there some property defining uniqueness which R preserves?

```
class ListItem{
  unique Object item;
  unique ListItem next;
  ListItem(unique Object o, unique ListItem n) {
    item = o; next = n;
  }
  unique Object getItem() {
    Object temp = item;
    item = null;
    return temp;
  }
  ...
}
```

Figure 1.2: Simple use of *unique*

One of the simplest and most common foundations for uniqueness is the destructive read. With a destructive read, a unique pointer's value is destroyed when it is read. This ensures that there is only one reference to the unique object, as any attempt to make a copy destroys the original. External uniqueness [CW03], for example, assumes destructive reads. Unfortunately, actually directly implementing dynamic reads requires a drastic overhaul of the runtime system.

Thus, one finds static analyses that attempt to prove that a program adheres to the semantics of destructive reads without actually implementing them. For example, the `getItem` method in Figure 1.2 performs the equivalent of a destructive read when it nullifies `item` prior to returning its value. In its original form, AliasJava [AKC02] used a liveness analysis on local variables to ensure the correct use of these 'manual' destructive reads.

But this may not actually correspond to one's intuitive notion of uniqueness. Persistent aliases of unique objects are banned; however aliases defined as temporary may in practice persist after the unique object has been handed off to somewhere else.

```
class Bad{
  unique ListItem a;
  unique Object b;
  Bad(unique Object o, unique ListItem n){
    a = new ListItem(o,n);  b = null;
  }
  unique ListItem getList(){
    ListItem temp = a;
    a = null;
    return temp;
  }
  unique ListItem send(){
    ListItem bad = a;
    ListItem ret = new ListItem(getList(),null);
    b = bad.getItem();
    return ret;
  }
}
```

Figure 1.3: 'Breaking' uniqueness?

The code in Figure 1.3 gives an example of this. Although the `ListItem` appears to have the only reference to its item, one can, as in `send`, affect the `ListItem`'s *`unique`* item from outside the `ListItem`. To this extent, uniqueness cannot actually represent encapsulation; that is, no class can be certain that it encapsulates any unique object which it is passed. (Of course, if a unique object is passed to a class, it should be encapsulated as there are "no" other references.)

1.2.1 Effects

The semantics for uniqueness thus depends on what constitutes a "temporary" alias. If methods affecting the unique field may be called while it has been borrowed, the borrowed reference can break the implied encapsulation provided by uniqueness. However, if a uniqueness analysis can track all the fields accessed by a method, it can

determine whether a borrowed reference may be safely maintained across a call to that method. For example, knowing that the call to `getList` changes `a` allows us, when analyzing `send` to mark the temporary alias `bad` as untenable after the call to `getList`. This is the insight behind Alias Burying [Boy01a].

Tracking the fields accessed across a method call requires an effects analysis [GB99, LPHZ02]. Each method is annotated with a list of which fields it reads and which fields it writes. Information hiding can be supported with data groups (also called regions). A data group is an abstraction grouping several fields together. Effects on the fields can be expressed in annotations as effects on the data groups containing the fields. For example, a class representing a circle on the Euclidean plane could have separate data groups for the `Size` and `Location` of the circle. A function to move the circle would declare an effect on the `Location` data group, whether the location was represented by x and y coordinates or a point object—provided the corresponding fields were declared to be in the `Location` group.

An effects analysis can similarly utilize uniqueness information to transfer the effects on the uniquely referenced object to the unique field. What effects should be on the `send` method in Figure 1.3? It writes the field `b` directly, and it writes the field `a` by calling `getList`. It also writes the field `a.item` when it calls `bad.getItem()`. But `a` is a unique field. Thus, the only way to access `a.item` is through `a`; declaring that we write `a.list` is redundant with declaring we write `a`. That is, we map effects on fields solely accessible through `a` as effects on `a`.

Mapping effects relies, of course, on sole access through a field, which means that we map effects onto ***`unique`*** fields. (This is not restricted to uniqueness; one can map effects on owned objects onto their owner.) Unfortunately, this means that we cannot analyze or even precisely define uniqueness and effects separately [Boy01b]. What is needed is a common semantics for both effects and uniqueness (and, ideally, many

other useful annotations).

1.3 A combined system

Fractional permissions provide a common semantics for a wide variety of annotations, not just uniqueness and effects. This section describes a regimen of annotations on a simplified Java-like language. Each of these annotations can be defined using fractional permissions; however, the actual analysis, as implemented does not check all of them.

Every method is annotated with its effects. An effect may be on `shared` (the "world") or on fields or groups of `this` or a parameter. Groups are abstractions of fields; each field and group is nested in its declared group (`All` if a group is not explicitly given). Each class inherits two special data groups, `All` and `Owned`, from `Object`; the latter holds the state of objects owned by the current object.

Fields, parameters, receivers, and return values are annotated with exactly one of the following pointer annotations (local variables have their annotation inferred by the type checker):

- An "ordinary" pointer, with no strict restrictions on (aliasing) the referenced object, is annotated ***`shared`***.

- Ownership describes an object's representation beyond the object itself. The `ListNode`s (Figure 1.4) help represent the list, although they are not stored inside the list object in the heap. References to these nodes are annotated with their ***`owner`*** (either ***`this`*** in the `List` or the class' ownership parameter in the `ListNode`s). We use a weak form of ownership which does not forbid references across ownership boundaries.

- A ***unique*** pointer represents the only access to the pointed-to object. As such it can represent a transferable form of ownership.
- A ***borrowed*** pointer *temporarily* aliases another (unique) pointer. The borrowing lasts until the unique reference is again required, as determined by effects annotations. Only method parameters and receivers may be annotated `borrowed`.
- The annotation ***from*** *effect*, as discussed in previous work [BRZ07], represents a return value that "borrows" from the named effect. For example, the `iterator()` function in Figure 1.4 returns a read iterator that borrows the effect of reading the list. When the list is written, this borrowed effect must be returned.
- A ***readonly*** pointer cannot be used to write the object it references or the objects owned or uniquely referenced by that object. The object may be written through other pointers. State that a ***readonly-owner*** reference refers to may be read but not written by the designated *owner*.
- A ***unique-write*** reference can be used to write the object it references, and it knows no one else can write that object. Any aliases must be read-only.
- An immutable object may never be written; an ***immutable*** reference points to an immutable object. Thus it is possible to pass an immutable reference where a read-only reference is expected but the reverse is illegal.
- An ***identity*** reference may be tested for equality but may not be mentioned in effects and cannot be used to access the pointed-to object in any way.

Each field (parameter, receiver, or return value) is annotated with its class, one of the above pointer annotations, and two modifiers. A reference may be annotated

non-null, which helps our type system ensure the absence of null-pointer exceptions. Of course, a *nonnull* field cannot actually be non-null upon entry to the class constructor; it is only non-null thereafter. In general, the invariants representing field annotations are not established until the constructor has finished execution. Thus, if `this` is passed as a parameter or receiver from within the constructor, the called method cannot assume the field annotations are accurate. Following Fähndrich and Leino [FL03], we term an object "raw" if it has not necessarily established its field annotations and "cooked" if it has. Each class' constructor establishes the field invariants for that class, but not for its descendants. A *raw*`[`C`]` object has its fields initialized through class C but not necessarily further; `raw` is a shorthand for *raw*`[Object]`. In general, cooked objects can be passed as *raw*`[`C`]`, and *raw*`[`C`]` objects may be passed as *raw*`[`C'`]` if $C \preceq C'$. In our examples, a reference without an explicit nullness modifier may be null, one without a rawness modifier is cooked.

Each class takes *one* ownership parameter which is passed in at construction, and may only be passed on or used in ownership annotations. (Our system can be easily generalized to any bounded number of parameters.) Unlike with some ownership systems, ownership is not mandatory—the ownership parameter may be ignored. The super constructor call implicitly passes the same ownership parameter on.

The ownership parameter may itself bear a pointer annotation describing the "owning" object; the absence of an annotation on the ownership parameter is syntactic sugar for borrowed. The annotations *unique*, *unique-write*, and *from* are not allowed on ownership parameters. The owner parameter may be null or raw. Ownership annotations are restricted to either `this` or the class' ownership parameter. Having the null owner is equivalent to being shared. In our examples, we elide an ownership parameter that is never used.

1.3.1 Example

The `List` class, its accompanying `ListNode`, and a read `ListIterator` are depicted in Figure 1.4. Together they define a simple doubly-linked list and, incidentally, illustrate many of the annotations whose semantics can be given using fractional permissions. The formal syntax used in this example is that described in Figure 2.8.

The `ListNode` accepts a single ownership parameter which will refer to its owner; the `next` and `previous` `ListNode`s that it links to must be owned by the same object. With permissions, one needs permission to access the owning `List` to access the fields of the next and previous `ListNode`s. In contrast, the data object stored in the node is annotated ***`shared`***, which corresponds to ownership by the global context: anyone with permission to access the global context may access the state of any shared object. The `ListNode` constructor initializes all three fields from identically-annotated parameters.

The `List` has a single field, a reference to a `ListNode` which is owned by the `List`. The constructor initializes this field to `null`. The `first()` method has several annotations. It is annotated with the effect that it may read any field of the `List`; in particular it reads the `head`. This is represented by the `first()` method being passed permissions for all of the fields of the `List` whenever it is called, including that of `head`. The `first()` method also has an annotation of ***`borrowed`*** at the end of the line; this annotation applies to the receiver of `first()`. A ***`borrowed`*** reference is a temporary alias; the fields of the receiver can only be accessed as described in the effects annotation. A reference to a ***`shared`*** object is returned from `first` because `datum` is annotated ***`shared`***.

The ***`writes All`*** effect on the `insert()` method is similar to the effect on `first()` except that it represents a write effect, and thus a requirement of a write permission. Here writes of objects owned by the `List` are mapped to an effect on the `List` itself.

```
class ListNode<owner>{
  owner ListNode next;
  owner ListNode prev;
  shared Object datum;

  ListNode(shared Object d, owner Node n, owner Node p)
     { super(); datum = d; next = n; prev = p;}
}
class List{
  this ListNode head;

  List(){ super(); head = null; }
  reads All
   shared Object first() borrowed
     { return head == null ? null : head.datum; }
  writes All
   identity Object insert(shared Object datum) borrowed
     { return (head = new ListNode<this>(datum, head,null);
        head.next == null ? null :(head.next.prev = head; null)) }
  reads All
   from(All) ListIterator iterator() borrowed
     { return new ListIterator<this>(head); }
}
class ListIterator<readonly-this list> extends Iterator{
  list ListNode cursor;

  ListIterator(list ListNode head) { super(); cursor = head;  }
  reads All
   identity Object hasNext() borrowed { return cursor;  }
  writes All
   shared Object next() borrowed
     { return (cursor == null) ? null : let temp = cursor.datum in
                                        (cursor = cursor.next; temp); }
}
```

Figure 1.4: Ownership in a List

When the new `ListNode` is created, it is passed the `List` as its owner. The *`shared`* annotation on the parameter and the *`borrowed`* annotation on the receiver mean the same as before. The *`identity`* annotation on the method return indicates that the returned reference may only be used in pointer identity tests; no one may read or write its fields. Given the uninteresting nature of the value returned from the function, this is appropriate.

The `iterator()` method returns an iterator over the `List`. The read effect and the *`borrowed`* receiver are exactly as described above. The *`from(All)`* annotation on the method return allows the returned iterator to temporarily use anything in the *`reads All`* effect of the list. As discussed elsewhere [BRZ07], once the iterator is no longer needed, it can be discarded and the permission recovered, after which the list may again be mutated.

The iterator uses its borrowed permission to take ownership of the list. In this example, the ownership parameter does not own the object to which it is passed; the opposite is in fact the case. The *`readonly-this`* annotation on the iterator's ownership parameter indicates that the `ListIterator` actually owns the `List`, but only for reading. The `ListIterator` features a reference to a `ListNode` owned by the `List`, which it, thus, indirectly owns. This indirect ownership is used in the `next()` method to map the effect of reading the fields of the `cursor` from the effect of writing the `ListIterator`. The other annotations in the `ListIterator` class behave as described above.

Most of the annotations used in this example interact heavily with one another. The receiver is made *`borrowed`*, forcing all permissions to be handled as effects. The write effect on the `insert()` method interacts with the ownership of the `ListNode`s, enabling `insert()` to manipulate them without declaring a representation-exposing effect. Both the ownership of the `ListNode`s by the `List` and of the `List` itself by the

`ListIterator` interact in a similar way with the write effect on `next()`. Further, the ownership of the `List` by its iterator would not be possible without the *`from(All)`* annotation, which in turn directly interacts with the effect which it "borrows." These interactions all express themselves naturally once all the annotations are given a semantics using fractional permissions.

1.4 Other Related Work

The work herein described sprawls across several different specific research topics. The two most notable areas of similar research are regimens of annotations and linear logics supporting permissions of a sort—many deal in linear predicates that may be treated identically to permissions.

1.4.1 Annotation Regimens

Uniqueness is not a new concept. On the one hand, it can be traced back to linear type systems [Wad90]. A value with linear type may only be used once. As such it is *de facto* unique. Linear types may coexist with nonlinear (normal) types, but the two cannot intermix. Additionally, the use-once property that assures uniqueness can affect ease of expression, making linear values difficult to work with using nonlinear paradigms for programming. Linearity can be eased in a syntactically limited space using the `let!` construct.

Early systems for unique fields featured this difficulty; therefore a series of proposals [Hog91, Min96, AKC02, Boy01a] began featuring what have become known as "borrowed" parameters. A borrowed parameter provided a temporary alias to a unique value. The difficulty sets in when deciding exactly what constitutes unique and what constitutes temporary.

One common paradigm (used in AliasJava [AKC02]) is that unique references have no aliases in the heap; borrowed then refers to aliases contained in the stack. However, stack references that remain live across uses of the unique field they alias may break locality, aliasing far-flung fields that are unaware that they may not be "really" unique. This eliminates one of the principle advantages of unique references: implicit encapsulation of state.

Another approach involves assuring uniqueness dynamically. External Uniqueness [CW03] contains both uniqueness and ownership types. Externally unique objects allow only one reference into their representation. In permission semantics, one can implement this similarly to uniqueness, by attaching permission for the representation to the externally unique pointer. The proposed system, however, is supported using destructive reads which are equivalent to linear semantics: reading a unique pointer destroys or nullifies it. External uniqueness causes borrowing reads to nullify the unique value for a syntactically limited time (thus keeping borrowed references off the heap). Remote consumers are given the safe, null, value. In general this has the effect of replacing uniqueness errors with `NullPointerException`s.

Destructive reads also require a change to the runtime system to destroy pointer values dynamically. Alternately, an (object-oriented) effects system [GB99, LPHZ02] may be imposed to determine statically where destructive reads would be necessary. To localize the ruin, the destruction may be imposed on the borrowed references rather than their unique progenitor. This is the essence of alias burying [Boy01a]: to bury borrowed references if their continued existence might otherwise necessitate the destruction of the unique field they alias. Unfortunately, the effects system circularly depends on uniqueness [Boy01b]. This interdependence was the genesis of the current permission-based system which deals jointly with uniqueness and effects.

The Spec# program verifier [BRLS04] is similar to the linear permission system

described here in that it provides a common semantics to a wide range of program annotations, including non-null, ownership, immutable, and effects annotations. It is implemented by transforming the annotations into logical predicates which are then verified by a general program prover [ByECD+06]. Using full program verification to check annotations has the advantage that program verification is well-established; however, it is also an intrinsically harder problem than annotation checking.

1.4.2 Linear Logics

A particularly active area in recent research has been in quasi-linear logics for reasoning about program behavior. The advantage these logics provide to automated reasoning is the same as the advantage they provide to our annotations: they enable reasoning to be local to some subset of the heap without complex frame properties describing what the procedure being analyzed *does not do* to the rest of the heap.

Adoption and Focus

The immediate progenitor of Boyland-style permissions is adoption and focus [FD02]. Vault [DF01], a language using linear types to enforce protocol (ordering) constraints, uses adoption and focus to mix linear and non-linear types. Here the linear pointer type is split into the singleton alias type [SWM00] and the capability (permission) to access it. Adoption is the permanent incorporation of one object's state into a pointer of another object (the guard). The guarded reference is sharable, while the original pointer is not. The Vault system supports this at runtime by keeping back pointers from guards to guarded objects. Focus allows access to the linear guarded state. The guard is unusable for the duration of the focus; in particular, multiple copies of the guard may not be focused on simultaneously.

There are some immediate differences between adoption and focus and the permission system of Chapter 2. Permissions allow smaller granularity; individual fields may adopt and be adopted. This is used to allow field keys to double as effects; for uniqueness, an object's state is treated as a whole. Finer granularity requires that the adopter not be completely unusable: each adopted field should only be focused on once, but the adopting field may have multiple fields focused on ("carved out") simultaneously. Fields only require the return (unfocusing?) of adoptees when required by the annotation system (e.g. to return an effect to the caller).

Other differences stem less from the formal permission system than the planned usage. Adoption and focus are expressions and as such are evaluated at runtime. We perform similar operation using permission transformation; as such, they become no-ops at runtime. However, as the type system is provably sound, the adoption operation is effectively a no-op. For the Java annotation checker, adoption occurs instantaneously on object creation and is never revisited. Thus adoption need not be expressly implemented in either the static or dynamic semantics of the annotation checker.

Separation Logic

Although the immediate inspiration for this permission system was adoption and focus, the final result closely resembles separation logic [Rey02]. Separation logic allows reasoning about heap properties by reasoning about partial heaps. The core predicate is the contents of one heap cell (akin to the alias type for a single permission). Two partial heaps are declared disjoint using the star $\star$ connector (akin to ,). The magic wand ($-\!\!\star$) operator lets the contents of one partial heap encompass another (akin to the linear implication used to represent carved-out permissions). Indeed, fractional permissions may be just as easily grafted into separation logic [BCOP05].

These similarities in form and function conceal fundamental differences.

The most obvious difference is the lack of adoption in separation logic. Equally straightforward is the difference in intent. One reasons about program properties in separation logic; one does not define extra properties (design intent), then use the mechanics of the logic to assure it.

However, the limited syntax for permissions (predicates) conceals another, more fundamental distinction. Permission forms were limited syntactically to allow a mechanical flattening process to determine whether they are consistent with memory. All transformations of permission states (e.g. carving out a permission) are safe because the result is consistent in any memory in which the progenitor is consistent. That is, changes in permission state never break consistency with memory. On the other hand, separation logic is founded on the logic of bunched implications [OP99]. Transformations of predicate state are allowed when they are sound according to this logic.

Smallfoot [BCO05] provides (some) automatic verification for concurrent programs using separation logic. It uses a system of symbolic execution which resembles the permission type system mentioned herein. However, it has differences too, both in intent—the system verifies low-level predicates specifying data structures—and in technical detail. Evaluation reduces to entailments in separation logic; neither fractions nor adoption are present.

Like the annotations described herein, separation logic can be used to verify particular usage patterns. For example, there are several ways [Bie06, Kri06] to associate separation logic predicates with Java iterators to avoid their misuse. This differs from *`from`* in that *`from`* is a general-purpose annotation which can be used to help assure correct iterator behavior; however, it is not specific to iterators. The observer pattern can be similarly checked [KAB07].

Chapter 2

Permissions

It is possible to provide a single semantics for many of the proposed pointer annotations using a system of fractional permissions with adoption. The particular system of fractional permissions described in this chapter is joint work of John Boyland, Yang Zhao, and myself. The chapter is largely a compendium of our joint work [Boy03, BR05, BRZ, Boy07, BRZ07]. It contains a formal review of permissions, their syntax and semantics, how they provide a semantics for annotations, and how they may be type-checked non-algorithmically.

2.1 Permissions Described

A permission represents the right to access some portion of state. "State" here refers to the values of the fields of the objects in memory; for pointer fields, these values are the memory locations of the pointed-to objects. In particular, the state referred to is the value of a single field; this granularity is necessary to express field-level annotations. Accessing state means either reading or writing the value of some field. As access is restricted to possessors of the associated permission, controlling the

distribution of permissions is equivalent to controlling access, which is the essential goal of the pointer annotations.

Permissions are *linear*. Every field has exactly one permission associated with it. To write state, one must possess this permission; possession of this permission conversely entails that *no one else* could possibly write the same state. However, while it is generally advantageous to prevent multiple writers of the same field, one may wish to enable multiple readers. To this end, a permission may be split in two. Possession of a partial permission allows one to read but not write the corresponding field. Partial permissions may be further divided or re-combined. If *all* the partial permissions are rejoined, one may once again use it to write state. Determining that all pieces have been reassembled requires some accounting; therefore when a permission is divided, each piece is assigned a fraction of the divided permission. A "fraction" of 1 denotes write permission; a "fraction" of 0 denotes absence of accessibility. (In practice zero is not allowed as a fraction; absence of access equates to absence of permission entirely.)

Permissions cannot be duplicated. This ensures the uniqueness of a field's write permission, but it poses a problem when trying to share access to or even information about a field. We therefore allow the adoption (or nesting) of one permission (the nested permission) into another (the nester permission). Then anyone who has the nester permission (and only someone with the nester permission) also has the *nested* permission. The fact that this nesting has taken place does not represent any permission itself; it merely indicates that the nested permission is available if one has the nester permission. Since the fact of the adoption is not itself a permission, it may be duplicated. By analogy: I put my life savings in a piggy bank. Telling people "I put my life savings in a piggy bank" does not double my money, nor does it give them access to my money. To access my life savings, one first needs access to the

piggy bank, after which the money can be removed. The process of removing a nested permission from its nester is referred to as "carving-out." (Fändrich and DeLine use the term "focus.") This carving-out leaves a hole in the nester that must be replaced before the nester permission may be used again.

Adoption is used for abstraction and convenience. Related fields may be adopted into the same data group (or region), allowing the data group permission to stand in for all the fields. The entire state of an object can (and for our purposes is) adopted into one such group, providing an easy handle for the complete state of the object. Further, adoption facts provide a sharable way of expressing field annotations, and forcing the carved-out holes to be filled enforces the presence of the adopted permissions and therefore the correctness of the annotation.

2.1.1 Permission Syntax

On some level, permissions are essentially syntactic entities. A permission is defined by its structure. The semantics of fractional permissions, described intuitively above, is given formally in Section 2.1.4. The separation of syntax and semantics enables us to provide an analysis that operates solely on syntactic fractions, yet is provably correct relative to the semantics. Figure 2.1 gives a (partial) context-free-grammar defining the syntax of fractional permissions.

A *formula* Γ represents a fact whose truth (if known) is preserved throughout execution. Facts can be treated "nonlinearly"; they can be duplicated freely. As well as basic predicate logic (`true`, conjunction, negation, existentials and named predicates), we have reference equality tests, nesting facts and object type assertions that state that the object at ρ is of class C (or a subclass).

A permission (Π) may take on many forms. There is exactly one *unit permission* associated with every field in the heap. If ρ refers to some object on the heap, f

$\rho ::= o \mid r$		*literal reference, variable*	$\Pi ::=$		*permission:*
$k ::= \rho\,.\,f$		*key (field instance)*		Γ	*formula*
				$k{\rightarrow}\rho$	*unit permission*
				v	*permission variable*
$\xi ::=$		*fraction:*		$\emptyset$	*no permissions*
	q	*literal* ($0 < q \leq 1$)		$\Pi + \Pi$	*combination*
	z	*fraction variable*		$\xi\Pi$	*fraction scaling*
				$\Gamma\,?\,\Pi : \Pi$	*conditional*
				$\exists r \cdot \rho.f{\rightarrow}r + \Pi$	*existential*
$\Gamma ::=$		*formula:*		$\Pi{-\!\!+}\Pi$	*implication*
	`true`	*true*			
	$\Gamma{\wedge}\Gamma$	*conjunction*			
	$\neg\Gamma$	*negation*	$P ::= \{\ldots, p(\overline{\rho}) = \Gamma, \ldots\}$		*predicate defns*
	$\exists\delta \cdot \Gamma$	*existential*	$\delta ::= r \mid z \mid v$		*any variable*
	$p(\overline{\rho})$	*named predicate*	$\Delta ::= \{\ldots, \delta, \ldots\}$		*variables*
	$\rho = \rho$	*reference equality*	$E ::= \Delta\,;\Pi$		*environment*
	$\Pi \prec k$	*nesting*	$\alpha ::= \forall\Delta; \Pi \longrightarrow \exists\Delta; \Pi$		*procedure type*
	$\rho \in C$	*object typing*	$\sigma ::= \{r \mapsto \rho, z \mapsto \xi, v \mapsto \Pi, \ldots\}$		*substitution*
	$\Gamma \implies \Gamma'$	$\triangleq \neg(\Gamma \wedge \neg\Gamma')$			

Figure 2.1: Permission Syntax.

is a field and ρ' is the contents of that field, then we can give the syntax of a unit permission as one kind of permission:

$$\Pi ::= \rho.f \rightarrow \rho' \mid \ldots$$

The unit permission gives the right to access the state, reading or writing the field. If the field is written, the unit permission changes to reflect the new value stored in the field.

Permissions can be combined in a number of different ways:

$$\Pi ::= \Pi + \Pi \mid \exists r \cdot \Pi \mid \Gamma \,?\, \Pi : \Pi \mid \ldots$$

A *compound permission* $\Pi + \Pi'$ gives one all the rights associated with both of the permissions being compounded. The identity of the compounding operator ("+") is the trivial permission $\emptyset$ which gives no rights to *any* state. *Existential permissions* represents a permission where a location r is unspecified. In a *conditional permission* $\Gamma \,?\, \Pi_1 : \Pi_2$, Γ is a boolean formula (not a permission). If the formula evaluates to true, one has the first permission; if false, then the second permission.

A permission may be *scaled* by a fraction ξ:

$$\Pi ::= \ldots \mid \xi\Pi \mid \ldots$$

Here ξ represents a positive fraction (rational number) usually less than 1. It may be a fraction variable, but will never be zero. Fractions give a way to get multiple permissions for a single state; a permission may be split in two:

$$\Pi \equiv \frac{1}{2}\Pi + \frac{1}{2}\Pi \equiv \frac{1}{2}\Pi + \frac{1}{4}\Pi + \frac{1}{4}\Pi \equiv \frac{3}{4}\Pi + \frac{1}{4}\Pi \equiv ...$$

Scaling distributes through compounds, conditionals and existentials. A scaled *unit* permission gives *read* access to the field thus referred to.

One last kind of permission is a form of *linear implication*:

$$\Pi ::= \ldots \mid \Pi \multimap \Pi$$

A linear implication $\Pi_1 \multimap \Pi_2$ means that one has the rights of the consequent Π_2, except for the ones of the antecedent Π_1. The rights implicit in a linear implication cannot be used until it is combined with its antecedent using a linear form of *modus ponens*:

$$\Pi_1 + (\Pi_1 \multimap \Pi_2) \Rightarrow \Pi_2$$

Here the $\Rightarrow$ symbol is meta-implication: the permissions on the left may be transformed into the right form as needed during type checking.

A permission may be *nested* in a unit permission, or more precise, in the field itself. This means that anyone who has the unit permission (the *nester*) also has the *nested* permission. We represent the information that Π is nested in $\rho.f$ by a formula

$$\Gamma ::= \ldots \mid \Pi \prec \rho.f \mid \ldots$$

The formula does not represent any permission itself; it merely indicates that the nested permission Π is available if one has the nester permission $\rho.f \rightarrow \rho'$.

The picture here shows the composition of two permissions being put together ($\rightsquigarrow$) by nesting the darker permission Π in the lighter one k, after which case the

$\Pi +$ $k \rightarrow \rho'$ $k \rightarrow \rho'$ $\Pi \prec k$ $\frac{3}{4}k \ldots +$ $\frac{1}{4}k \ldots$ $\Pi \prec k$

darker permission is no longer (directly) accessible, but a new nesting fact $\Pi \prec k$ is known. Then this nester permission is split into two pieces, thus implicitly splitting the nested permission. Here k is a meta-variable referring to a field of some object $k \equiv \rho.f$. The nesting fact $\Pi \prec k$ is immutable and thus need not be scaled.

Access to the nested permission requires *carving* it out of the nester permission (Fähndrich and DeLine use the term "focus"). The carve operation leaves a hole in the nester permission. The hole is represented by linear implication (*nested* —+ *nester*). The nester permission will not be considered complete until the nested permission is "replaced."

$\frac{3}{4}k{\rightarrow}\rho'$ $\quad$ $\left(\frac{3}{4}\Pi \multimap (\frac{3}{4}k{\rightarrow}\rho')\right) + \frac{3}{4}\Pi$ $\quad$ $\frac{3}{4}k{\rightarrow}\rho'$

$\Pi \prec k$ $\quad$ $\Pi \prec k$ $\quad$ $\Pi \prec k$

As seen in the picture, permission carving handles fractions transitively: if one has only a fraction of the nesting permission, one can only get a fraction of the nested permission. Notationally, we have:

$$(\Pi \prec k) + (\xi k{\rightarrow}\rho) \Rightarrow (\Pi \prec k) + \xi\Pi + (\xi\Pi \multimap (\xi k{\rightarrow}\rho))$$

When one is done with the nested permission, it can be replaced using the linear *modus ponens* rule explained above.

A permission may include a permission variable v used to refer to an unknown permission treated parametrically by a method. Also a formula Γ may be treated as a permission with no rights, as long as the formula is true. We restrict existentials to increase their precision by forcing the variable (r) to be stored in a known field $\rho.f$

(we do not permit $\rho = r$). This reduces the expressiveness of existentials, but also prevents certain degenerate cases.

Permissions share inspiration with separation logic [Rey02] and there are several parallels. The combination operator (+) is the equivalent of $\star$ in separation logic, while $\emptyset$ is equivalent to "**emp**." We do not use separation logic's syntax because the metaphor is wrong ("$\star$" would be addition for fractions) and to avoid confusion in how we treat non-linear terms: "`true`" corresponds to **emp**, not to **true**. We also use a semantics for $-\!\!+$ that is more restricted than for $-\!\!\star$.

The environment $E = (\Delta; \Pi)$ in which a term is permission checked (see Section 2.3.1) has two parts: a type context Δ which is a set of variables drawn from R (reference variables), Z (fraction variables), and V (permission variables); and a bag of permissions Π. For an environment $E = (\Delta; \Pi)$, we normally require that all free variables in Π are in Δ. For brevity purposes, this restriction is left implicit.

A procedure type $\forall \Delta\,; \Pi \longrightarrow \exists \Delta'\,; \Pi'$ is polymorphic over variables in Δ. It accepts the permission Π and returns the permission Π', using perhaps some new variables Δ' (as well as the existing variables Δ).

Substitutions (σ) of variables map each kind of variable to the appropriate kind. Substitutions are made total (being the identity on variables not in the original domain) and are lifted to apply to all syntactic entities in the normal manner.

2.1.2 Fractional heaps

Separation logic is given semantics using heaps [IO01]; fractional permissions with "fractional heaps." To handle fractions, the heap is not simply either defined or undefined for each location, but rather if defined, is defined to a fractional extent between 0 exclusive and 1 inclusive. The heaps of separation logic can then be seen as a special case where every fraction is one.

Formally, a (fractional) heap h is a finite (partial) map of locations to pairs of a positive fraction and a value (for us, an object reference from the countably infinite set O):

$$h : L \rightharpoonup (\mathbf{Q}^+ \times O)$$

Here L is the (countably infinite) set of locations in the heap. We use $\hat{\emptyset}$ to refer to the empty heap that is defined nowhere. A fractional heap where every fraction is 1 is called a *memory* (written μ).

We apply our permission types to object-oriented languages and thus for us locations are fields of objects ($L = O \times F$, where F is a finite set of field names); every object's field has its own individual permission since fields of (mutable) objects can be updated independently. The fraction represents the permission to access the heap at that location (field); it does *not* grant access to the object whose reference is stored in the heap at that location (field). We say that one heap is included in another $h_1 \leq h_2$ if for every fraction in the first, it matches the second with at most that fraction:

$$h_1 \leq h_2 = \forall_{(q_1,o)=h_1(l)}\, q_1 \leq q_2 \text{ where } (q_2, o) = h_2(l)$$

Heaps can be combined by adding together the corresponding fractions, but *only if the values match*: If there is a location l such that $h_i(l) = (q_i, o_i)$ and $o_1 \neq o_2$ then $h_1 \hat{+} h_2$ is undefined. Otherwise it is defined as follows:

$$(h_1 \hat{+} h_2)(l) = \begin{cases} h_1(l) & \text{if } h_2(l) \text{ is undefined} \\ h_2(l) & \text{if } h_1(l) \text{ is undefined} \\ (q_1 + q_2, o) & \text{where } (q_i, o) = h_i(l) \end{cases}$$

It is clearly the case that $h \hat{+} \hat{\emptyset} = \hat{\emptyset} \hat{+} h = h$ for all h.

Two fractional heaps that can be added are called *compatible*. Because we permit fractions to exceed 1, a fractional heap is always compatible with itself.

Heaps can be *scaled* by any fraction $q > 0$:

$$(qh)l = (qq', o) \text{ where } hl = (q', o)$$

The ability to scale heaps (and also permissions) is required because of nesting, as explained above.

Ambiguity in the semantics comes from having multiple heaps that model the same permission. Unrestricted ambiguity can make the fundamental fraction intuition $\frac{1}{2}\Pi + \frac{1}{2}\Pi \equiv \Pi$ unsound since the two occurrences of Π could refer to different heaps.

Most desirable would be if a permission were "precise." We adopt the concept of "precision" from separation logic: A permission is *precise* if any two compatible fractional heaps h_1 and h_2 that model Π must be equal. Unfortunately however, there are some permissions, such as the first example of an implication $\Pi_2 \multimap \Pi_1$ that are imprecise, as will be shown in Section 2.1.4.

2.1.3 Equivalence

Figure 2.2 defines a relation $\Rrightarrow$ on permissions used to define equivalence:

Definition 2.1.1 The $\equiv$ relation is the transitive, symmetric and reflexive closure of the $\Rrightarrow$ relation. We write $\Pi \geq \Pi'$ if and only if there exists Π'' where $\Pi \equiv \Pi' + \Pi''$.

The equivalence relation induces a partition on permissions. We choose a particular representative as "canonical."

$$\text{Q-Identity}\quad \Pi + \emptyset \Rrightarrow \Pi \qquad \text{Q-Commute}\quad \Pi + \Pi' \Rrightarrow \Pi' + \Pi \qquad \text{Q-Associate}\quad \Pi + (\Pi' + \Pi'') \Rrightarrow (\Pi + \Pi') + \Pi''$$

$$\text{Q-Combine}\quad \frac{\Pi_1 \Rrightarrow \Pi_1' \qquad \Pi_2 \Rrightarrow \Pi_2'}{\Pi_1 + \Pi_2 \Rrightarrow \Pi_1' + \Pi_2'} \qquad \text{Q-Zero}\quad q\emptyset \Rrightarrow \emptyset \qquad \text{Q-One}\quad 1\Pi \Rrightarrow \Pi \qquad \text{Q-Distribute}\quad q(\Pi + \Pi') \Rrightarrow q\Pi + q\Pi'$$

$$\text{Q-Multiply}\quad \frac{q, q' > 0 \qquad qq' = q''}{q(q'\Pi) \Rrightarrow q''\Pi} \qquad \text{Q-Add}\quad \frac{q, q' > 0 \qquad q + q' = q''}{q\Pi + q'\Pi \Rrightarrow q''\Pi}$$

Figure 2.2: Permission Equivalence (helper relation).

Definition 2.1.2 Given an arbitrary total ordering $<$ on permissions, a permission Π is in *canonical* form, if it is in the form

$$(q_1\pi_1 + (q_2\pi_2 + \ldots (q_n\pi_n + \emptyset)\ldots))$$

for $n \geq 0$, where for every $0 < i < j \leq n$, we have $\pi_i < \pi_j$ using this arbitrary order, and each π (called a "unit" permission) has the form

$$\pi \quad ::= \quad \Gamma \mid o.f \rightarrow o \mid \Gamma \,?\, \Pi : \Pi \mid \exists r \cdot \Pi \mid \Pi \multimap \Pi$$

Boyland [Boy07] has proven the following using machine-checked proofs and the Twelf theorem checker.

Lemma 2.1.3 *The following rule is "admissible"*

$$\text{Q-Scale}\quad \frac{\Pi \Rrightarrow \Pi'}{q\Pi \Rrightarrow q\Pi'}$$

In other words, adding it would not change the definition of equivalence.

Theorem 2.1.4 *For every* Π*, there exists exactly one canonical* Ψ *such that* $\Pi \equiv \Psi$*.*

2.1.4 Permission Semantics

This section defines the semantics of fractional permissions in terms of fractional heaps. The semantics depends on the current nesting situation N, a complete map on locations to the permissions nested in that location:

$$N : L \to \{\Pi\}$$

We require that this map yields the empty permission $\emptyset$ except for a finite number of locations.

Permissions are used to analyze stateful programs. At the beginning, we start with the empty nesting situation: $N_0(l) = \emptyset$. While the program is running, new permissions may be nested in locations; thus N may grow (but never shrink). We define the $\leq$ operation on nesting situations $N_1 \leq N_2$:

$$\forall l \cdot N_1(l) \leq N_2(l)$$

where $\leq$ on permissions is defined using equivalence (see Defn. 2.1.1).

Formula Evaluation

For the semantics of permissions, we need an evaluation of formula to boolean values written $A; N \vdash \Gamma \Downarrow b$ where $b \in \{\text{true}, \text{false}\}$. The A set is a set of assumptions used for simulated coinduction. It starts off empty. Formula evaluation need not be defined for all formulae, but it must be **deterministic**: a formula cannot evalate to

$$\text{B-True}\quad A; N \vdash \top \Downarrow \text{true}$$

$$\text{B-Neg}\quad \frac{A; N \vdash \Gamma \Downarrow b}{A; N \vdash \neg\Gamma \Downarrow \neg b}$$

$$\text{B-AndFalse1}\quad \frac{A; N \vdash \Gamma_1 \Downarrow \text{false}}{A; N \vdash \Gamma_1 \wedge \Gamma_2 \Downarrow \text{false}}$$

$$\text{B-AndFalse2}\quad \frac{A; N \vdash \Gamma_2 \Downarrow \text{false}}{A; N \vdash \Gamma_1 \wedge \Gamma_2 \Downarrow \text{false}}$$

$$\text{B-AndTrue}\quad \frac{A; N \vdash \Gamma_1 \Downarrow \text{true} \qquad A \vdash \Gamma_2 \Downarrow \text{true}}{A; N \vdash \Gamma_1 \wedge \Gamma_2 \Downarrow \text{true}}$$

$$\text{B-Equal}\quad A; N \vdash o{=}o' \Downarrow (o = o')$$

$$\text{B-Nest}\quad \frac{N(l) \geq \Psi}{A; N \vdash \Psi \prec l \Downarrow \text{true}}$$

$$\text{B-Exist}\quad \frac{A; N \vdash [\delta \mapsto X]\Gamma \Downarrow \text{true}}{A; N \vdash \exists\delta.\Gamma \Downarrow \text{true}}$$

$$\text{B-Axiom}\quad \frac{\Gamma \in A}{A; N \vdash \Gamma \Downarrow \text{true}}$$

$$\text{B-Pred}\quad \frac{A \cup \{p(\overline{o})\}\,; N \vdash [\overline{r \rightarrow o}]P(p) \Downarrow \text{true}}{A; N \vdash p(\overline{o}) \Downarrow \text{true}}$$

Figure 2.3: Evaluation rules for boolean formulae: $A; N \vdash \Gamma \Downarrow b$

both true and false in the same context, and it must be **stable**:

$$N_1 \leq N_2 \wedge (A; N_1 \vdash \Gamma \Downarrow b) \Rightarrow (A; N_2 \vdash \Gamma \Downarrow b)$$

In other words, a formula cannot change its value during execution. This permits non-linear reasoning even in the face of state changes: once true, always true; once false, always false.

Figure 2.3 defines evaluation rules for the formulae forms defined in Figure 2.1. Of some interest is the fact that conjunction can avoid evaluating a sub-formula, which may be necessary because a formula may be undefined in the current nesting situation. For instance, B-Nest shows that a nesting fact can only be true, never false. Since nestings increase monotonically, we can never depend on a nesting *not* being true.

A similar situation applies to existential formulae and recursive predicates. An

$$\textsc{S-Implication}\ \frac{h; \Psi + \Psi' \models^C_N \Pi}{h; \Psi \models^C_N \Psi' \multimap \Pi} \qquad \textsc{S-Obligation}\ \hat{\emptyset}; \Psi \models^C_N \Psi \qquad \textsc{S-Formula}\ \frac{\emptyset; N \vdash \Gamma \Downarrow \text{true}}{\hat{\emptyset}; \emptyset \models^C_N \Gamma}$$

$$\textsc{S-Cond}\ \frac{\emptyset; N \vdash \Gamma \Downarrow b \qquad h; \Psi \models^C_N \Pi_b}{h; \Psi \models^C_N \Gamma \multimap \Pi_{\text{true}} \Pi_{\text{false}}} \qquad \textsc{S-Exist}\ \frac{h; \Psi \models^C_N [\delta \mapsto X]\Pi}{h; \Psi \models^C_N \exists \delta.\Pi} \qquad \textsc{S-Fraction}\ \frac{h; \Psi \models^C_N \Pi}{qh; q\Psi \models^C_N q\Pi}$$

$$\textsc{S-Combine}\ \frac{h_1; \Psi_1 \models^C_N \Pi_1 \qquad h_2; \Psi_2 \models^C_N \Pi_2}{h_1 \hat{+} h_2; \Psi_1 + \Psi_2 \models^C_N \Pi_1 + \Pi_2} \qquad \textsc{S-Equiv}\ \frac{\Psi \equiv \Psi' \qquad \Pi \equiv \Pi' \qquad h; \Psi' \models^C_N \Pi'}{h; \Psi \models^C_N \Pi}$$

$$\textsc{S-Field}\ \frac{h; \Psi \models^{C \cup \{(h;\Psi) \prec l\}}_N N(l) \qquad u = [l \mapsto (1, o)]}{h \hat{+} u; \Psi \models^C_N l \rightarrow o} \qquad \textsc{S-Field-Co}\ \frac{(h; \Psi) \prec l \in C \qquad u = [l \mapsto (1, o)]}{h \hat{+} u; \Psi \models^C_N l \rightarrow o}$$

Figure 2.4: Semantics of Fractional Permissions with Nesting

existential or predicate call can only be evaluated as true. We assume that $P(p)$ gives the definition of predicate p and that we never call a predicate with the wrong number of arguments. Reflecting standard practice, we use an overline to represent multiple formals $\overline{r}$ and actuals $\overline{o}$. The form $[\overline{r \rightarrow o}]P(p)$ means to substitute the actuals for the formals in the body of the predicate. Rules B-Pred and B-Axiom together implement coinduction: while evaluating the body of the predicate, we assume the result will be true. If we permitted both true and false assumptions, a recursive formula could evaluate to both true and false, violating determinism.

Theorem 2.1.5 *Boolean evaluation is deterministic.*

Theorem 2.1.6 *Boolean evaluation is stable.*

Permissions

Figure 2.4 shows the relation that defines when a heap models a permission $h; \Psi \models^{C}_{N} \Pi$. We have already discussed the N qualification. We now discuss the presence of Ψ on the left-side of the relation. The C qualification is explained below.

The Ψ on the left of the modeling relation is an "obligation." The obligation starts empty. Recall that $\Pi_1 \multimap \Pi_2$ means "everything permitted by Π_2 except that permitted by Π_1." This intuition is expressed in making Π_1 an obligation that must be discharged *symbolically* while expanding Π_2. This can be seen in the rules S-Implication and S-Obligation in Fig. 2.4. Thus in S-Obligation, if we are looking for heap to model a permission that exactly matches the obligation, the obligation is dicharged, and thus the empty heap fits. New obligations are added in a sub-goal in S-Implication.

The rules S-Field and S-Field-Co deal with field permissions. In the first rule, the nesting situation N is used to determine the current nesting for the location. The resulting heap is added to a unit permission for the field. The C set is used to get a co-inductive effect of the first rule by permitting it to help establish itself. The elements of the C set are triples written $(h; \Psi) \prec l$. The rule S-Field-Co uses the C set to avoid recursively looking at the nested permissions.

2.1.5 Transformation

Once we have a semantic basis for permissions, we can define the transformation relation to enables us to convert permissions from one form to another. Ultimately, this will be used by the type system to produce the exact permission needed when checking reads, writes, and method annotations.

There are two forms of transformation, transformation by implication, and transformation by nesting. These can be written

$$\textsc{Tr-Implies}\quad \frac{E \Rightarrow E'}{E \rightsquigarrow E'} \qquad \textsc{Tr-Nest}\quad \Delta; \Pi + \Pi' \rightsquigarrow \Delta; \Pi + (\Pi' \prec k)$$

The nesting rule says that one may give up any permission and receive in its place a "nesting fact." The place of nesting is arbitrary.

With implication, one permission can be transformed into another if, for every fractional heap consistent with a memory that models the former permission, there is a second fractional heap, consistent with the first, that models the latter permission.

$$\frac{\forall h, \Psi, N, \mu \quad h \leq \mu \implies \left(h; \Psi \models^{\emptyset}_{N} \sigma\Pi \Rightarrow \exists h' \leq h, \sigma' \supseteq \sigma \text{ such that } h'; \Psi \models^{\emptyset}_{N} \sigma'\Pi'\right)}{\Pi \implies \Pi'}$$

This rule is sufficient to ensure that implication is safe. That is, no one can use transformation with implication to gain access to something to which they had previously lacked permission. In fact, the rule is written to allow *any* safe implication.

2.2 Representing Pointer Annotations using Permissions

The permissions detailed in Section 2.1 provide a powerful system for highly granular control over users' access to fields. The detail is useful for its expressive power but also provides a bar to the system's utility. It is not reasonable to expect programmers to provide annotations in fully detailed semantics. As stated initially, the aim is to provide programmers with intuitive annotations and the machinery necessary to

assure them. Permissions are useful because their expressive power is sufficient to provide a semantic basis for many common pointer annotations.

It seems counterintuitive to use permissions to define pointers given that permissions reflect the right to access a piece of state (that of the pointed-to object). However, permissions are convenient for defining pointer annotations precisely because they *uniquely*[1] control access to the pointed-to object. Semantics, like capabilities [BNR], which define pointer annotations by qualities of the pointers themselves cannot be localized because the pointer property itself is not local. A unique pointer is not unique because of any quality it possesses; it is unique because *no other pointer refers to the same object*. This is a global property of the system, as is allowing other pointers to the same object (shared). Permissions, by being local to the pointed-to object, implicitly make these global claims; if one holds both a "unique" pointer and permission to all fields of the pointed-to object, no other useful pointer in the system can reference the same object.[2]

2.2.1 Field annotations, Class Invariants, and Inheritance

The conditions that the annotations impose on fields are called "unary field invariants" because they are object instance invariants that involve one field at a time. We handle unary field invariants through nesting. A field is always nested using a particular permission. This permission expresses the possible nullness, the enclosed permissions (if any) as well as the ownership of the state of the object pointed to. Thus when the field's permission is nested in its data group, it must meet all the required conditions. If one knows of the nesting and has the permission to the data group, one knows that its unary field invariants (expressed through annotations) are

[1] Remember, permissions are linear.

[2] Pointers with no permission are still allowed.

valid. We express all the fields' invariants as a named predicate whose body consists of a conjunction of nesting facts. This is the class invariant.

A "data group" is modeled by a field with an uninteresting type; in this paper, a `group` declaration is converted into a field that is always null. Thus permission to the `All` data group of object r is written $r.\text{All}\rightarrow 0$. The direct children (fields or other data groups) of this data group have their permission nested within the data group. Therefore, when a data group is used for an effect, the exact nature of what fields are used is rightfully hidden from the caller, which only must provide the permission for the group which includes the permission for any fields. In addition to the `All` and `Owned` data groups in `Object`, there are two global data groups (groups on the null pointer), one for immutable state and one for state owned by the "world[3]."

The class invariant must be established by the constructor. The constructor is modeled by a method that takes the permissions for each field individually and returns the permission for the "All" data group after establishing the class invariant for its own and all superclasses. The permission to "All" and the class invariant jointly imply that all fields are consistent with their types.

Once the invariant is established in the constructor, it can be broken by carving the field out of its data group and assigning it a value that does not fit. But then the data group permission cannot be restored until the required unary field invariant is restored. Boogie uses a similar semantics for invariants [BDF+04]. In JML [LBR99], all invariants of fully-constructed objects must be valid in every method, except that a "helper" method may be called on an object whose invariant is currently broken. With permissions, one has a looser semantics: the only objects whose invariants must be true when the method starts are those whose state can be observed using the permissions passed into the method. JML designers are considering an "accessible"

[3]For simplicity, we use the `Owned` data group of the null pointer.

clause to limit the state that a method is permitted to observe; this would essentially be a read effect and would permit the kind of analysis done here using permissions.

2.2.2 Raw and Cooked

When a method call (dynamic dispatch) is invoked on the object under construction (whether directly from the constructor, or indirectly through a method called with the under-construction object as a parameter), that call may be passed a reference whose class invariant is not yet established. For example, no field is non-null on entry to the constructor. This can lead to unsoundness if one assumes that invariants are always true. On the other hand, if the requirement on the invariant is an explicit part of the method signature, then the method cannot be overridden in a subclass wanting a stronger invariant.

We solve this conundrum through the use of raw types (borrowed from Fähndrich and Leino [FL03]). Instead of treating "raw" as a primitive, we express it (or rather its inverse, "cooked") using permissions. A "cooked" pointer is one for which the invariant is true for every dynamic type that the object possesses:

$$\begin{aligned}\texttt{cooked}(r, o) = (r \in C_1 \implies C_1(r, o)) \\ \wedge \ldots \wedge (r \in C_n \implies C_n(r, o))\end{aligned}$$

Here $r \in C$ states the object at location r is of class C or one of its descendants, and $C(r, o)$ is the class invariant for class C (where o is the ownership parameter). The body of the `cooked` predicate ranges over all classes in the system. Thus a method can require that its receiver be "cooked" and permit an overriding method to use the same predicate to provide a stronger invariant. A reference with a `raw[`C`]` annotation only guarantees those from C up the class hierarchy.

Rawness Annotation (ra)	Facts ($\Gamma_{ra,r}$)
`raw[`C`]`	$\mathrm{C}(r, r_{\mathrm{owner}})$
`cooked`	$\mathrm{cooked}\,(r, r_{\mathrm{owner}})$

Figure 2.5: Translation of Rawness Annotation ($ra \to \Gamma_{ra,r}$)

Pointer Annotation (pa)	Permissions ($\Pi_{pa,r}$)
`unique`	$r.\mathrm{All}{\to}0$
`shared`	$r.\mathrm{All} \prec 0.$`Owned`
`immutable`	$\exists z.zr.\mathrm{All} \prec 0.$`Immutable`
`readonly`	$\exists z.zr.\mathrm{All} \prec 0.$`Owned`
`unique-write`	$\frac{1}{2}r.\mathrm{All}{\to}0+$ $\frac{1}{2}r.\mathrm{All} \prec 0.$`Owned`
owner	$r.\mathrm{All} \prec r_{\mathrm{owner}}.$`Owned`
`readonly-`*owner*	$\exists z.zr.\mathrm{All} \prec r_{\mathrm{owner}}.$`Owned`
`borrowed`	
`identity`	
`from(`$x\,.\,f$`)`	$r.\mathrm{All}{\to}0+$ $r.\mathrm{All}{\to}0 \multimap\!+ \xi r_x.f + v$
`readonly-from(`$x\,.\,f$`)`	$\frac{1}{2}\xi r.\mathrm{All}{\to}0+$ $\frac{1}{2}\xi r.\mathrm{All}{\to}0 \multimap\!+ \xi r_x.f + v$

Figure 2.6: Translation of Pointer Annotations ($pa \to \Pi_{pa,r}$)

Every field permission includes a predicate ($\Gamma_{ra,r}$) for the pointed-to object stored in r, which indicates how much of the class invariant is known to be established. The form this predicate takes for each possible rawness annotation is shown in Figure 2.5.

2.2.3 Pointer Annotations

Field annotations are represented in the class annotation by nesting the existentially-qualified permission to the field in the appropriate data group. The various pointer annotations are given semantics by the permissions included in the existential closure along with the field permission. These are summarized in Figure 2.6. As before,

r represents the object to which the field points. The last four annotations in the diagram may annotate method parameters or return values, but not fields.

Uniqueness

A pointer is unique if permission to all of the pointed-to object's state is stored in the closure. This is accomplished using permission to the All data group (r.All). Because there is exactly 'one' permission for each location and because every permission for an object is ultimately nested within its All data group the unique field has the only permission allowing access to the object to which it refers. Thus no one has permission to access the uniquely pointed-to object except through the unique field.

If we have a linked list of unique nodes,

```
class Node{
  ...
  unique Node n;
}
```

the class annotation would then be in part

$$\text{Node}(r_t, r_o) = \ldots \exists r \cdot \left(\begin{array}{l} r_t.\text{n} \rightarrow r + r.\text{All} \rightarrow 0 \\ + \text{cooked}\,(r, 0) + r \in \text{Node} \end{array} \right) \prec r_t.\text{All} \ldots$$

Therefore, one can carve out the permission for the next `Node` from the current `Node`, but must restore it before reasserting the class invariant. The provided annotation is inaccurate in that it ignores possible nullity (see below).

Ownership

Ownership is expressed by nesting the "All" group of the owned object in the "Owned" group of the owning object ($r.\text{All}{\rightarrow}0 \prec r_{\text{owner}}.\text{Owned}$). Thus, the class invariant for the `List` (in Figure 1.4), would be in part

$$\text{List}(r_t, r_o) = \ldots \exists r \cdot \left(\begin{array}{l} r_t.\text{head}{\rightarrow}r + r.\text{All}{\rightarrow}0 \prec r_t.\text{Owned} \\ + \text{cooked}\,(r, r_t) + r \in \text{ListNode} \end{array} \right) \prec r_t.\text{All}\ldots$$

Ownership works similarly in the `ListNode`, however the owning object changes.

$$\text{ListNode}(r_t, r_o) = \ldots \exists r \cdot \left(\begin{array}{l} r_t.\text{prev}{\rightarrow}r + r.\text{All}{\rightarrow}0 \prec r_o.\text{Owned} \\ + \text{cooked}\,(r, r_o) + r \in \text{ListNode} \end{array} \right) \prec r_t.\text{All}\ldots$$

Carving out the nested permission for the `ListNode`s from the permission for the `List` is what allows us to map effects on the nodes to effects on the `List` itself. The permanence of nesting fits well with most ownership type systems in which the owner of an object is fixed. Unlike some ownership systems, we do not fix an owner upon creation; an object is created unique. Optionally, it may receive ownership at a later time. And if the owner is discarded, the object may be considered unique again, or given a new owner. In the example, the `List` becomes unowned once the `ListIterator` is no longer in use.

Permission nesting only models the hierarchical structure of the ownership graph. Additional rules are necessary if one wishes to prevent *access* to the nested permissions by outsiders who have access to the owner.

Shared

A shared pointer is one whose object's state is owned by the "world," represented here by the (fictive) null object. More precisely, the "All" data group of the pointed-to object is nested in the "Owned" group of the null pointer. (r.All$\rightarrow 0 \prec 0$.Owned). If a method wishes to read or write the state of `shared` objects, it must declare an effect on `shared` (`null.Owned`). This suffices for single-threaded programs, but `shared` state is problematic in concurrent programs.

Immutability

All immutable state has a fraction that is nested into a special globally known "immutable" group. ($\exists z.zr$.All$\rightarrow 0 \prec 0$.Immutable) Implicitly, every time a method is called, it passed permission for a (small) fraction of this field. Thus, we do not need to declare method effects for reading immutable state. "Final" fields can be modeled by nesting the field permission in the immutable group.

Read-Only

With fractional permissions, we can nest a *fraction* of a permission in a data group instead of the whole thing. A *read-only* pointer refers to state made globally readable by having some fraction of it owned by the "world" ($\exists z.zr$.All $\prec 0$.`Owned`). The pointer to an object can be broadcast widely together with the knowledge that its state is read-only. In this case, `readonly` is not precisely transitive, since any state nested or existentially closed within the read-only state is not technically `readonly`: no fraction of its permission is necessarily directly owned by the "world." But the transitivity of fractions over permission nesting achieves the same effect: the subordinate state cannot be mutated using this reference.

Unique-Write

The difference between `readonly` and `immutable` is a difference in convention between the shared state and the "immutable" state: a method will *never* be given the whole permission to the special "immutable" field. On the other hand, a class may "know" (as part of its invariant), that half of the permission to some object is stored in the field pointing to it and the other half belongs to the "world" ($\frac{1}{2}r.\text{All}\rightarrow 0, \frac{1}{2}r.\text{All} \prec 0.\texttt{Owned}$). In this case, a method in the class can request the whole "world" permission with the annotation "writes shared," and then combine the obtained permission with the permission in the field to get a whole permission, with which it can write referred-to the object. This is what we call `unique-write`: any method with access to the shared state can read the state, but some privileged methods (inside the class with the `unique-write` field) have the ability to write it as well. Requiring the privileged methods to declare an effect on the shared state ensures that interference cannot escape notice.

The distinction between a `unique-write` and a `unique` variable is clear from its definition: a unique variable can be mutated without reference to the shared state, but a `unique-write` variable needs to ensure that no one else is in the process of reading the variable and thus must access the shared state.

2.2.4 Maybe-null and Non-null

Forming an existential closure with permission to access the state of the pointed-to object only makes sense when there is a pointed-to object; that is, when the pointer is not null. This can be established in two ways. First, the pointer can be declared not to be null, which avoids the problem. Alternately, the enclosed permissions can be made conditional on whether the pointer is null. These correspond to `nonnull`

Non-null Annotation (na)	Precondition ($\Gamma_{na,r}$)
`nonnull`	`true`
`maybenull`	$\neg(r = 0)$

Figure 2.7: Translation of Nullity Annotation ($na \rightarrow \Gamma_{na,r}$)

and `maybenull` annotations respectively. For simplicity, we always represent nullity as a precondition, using `true` as the precondition when the field is `nonnull`, as shown in Figure 2.7.

2.2.5 Complete Annotation Translation

In general, the full permission for any annotated field ($ra\ na\ pa\ C\ f$) is:

$$\Gamma_{na,r}\ ?\ (\Pi_{pa,r}, \Gamma_{ra,r}, r \in C) : \emptyset$$

This shows up in the class invariant existentially qualified for r and nested in the appropriate data group.

Our unique linked list node then becomes

$$\text{Node}(r_t, r_o) = \ldots \exists r \cdot \neg(r = 0)\ ?\ \left(\begin{array}{l} r_t.\text{n} \rightarrow r + r.\text{All} \rightarrow 0 \\ \\ + \text{cooked}\,(r, 0) + r \in \text{Node} \end{array} \right) : \emptyset \prec r_t.\text{All}$$

Now if we carve out permission for the next node, the invariant may be restored either by restoring that permission (as before) or by pointing to null. The full class

invariant for the `List` class in Figure 1.4 is

$$\mathrm{List}(r_t,r_o) = r_t.\mathrm{Owned}{\rightarrow}0 \prec r_t.\mathrm{All}\wedge$$
$$\exists r \cdot \neg(r=0)\ ?\ \left(\begin{array}{l} r_t.\mathrm{head}{\rightarrow}r + r.\mathrm{All}{\rightarrow}0 \prec r_t.\mathrm{Owned} \\ +\,\mathrm{cooked}\,(r, r_t) + r \in \mathrm{ListNode} \end{array}\right) : \emptyset \prec r_t.\mathrm{All}$$

2.2.6 Effects and Method Annotations

A method is passed permissions to enable it to access state. Thus an effects annotation on a method can translate directly as the permission to access the location described in the effect. This permission must be passed in when the function is called and must be returned to the caller after it returns. A write effect is passed (and returns) the full permission to access the field. A read effect is similar, except it requires any non-zero fraction of permission (e.g. $z\exists r.r_{\mathrm{this}}.x{\rightarrow}r$) that the caller can spare during execution. The same fraction is returned when the method returns. The variable r_{this} refers to the object which is the receiver, r_{x} to parameter `x`, while the variable r_{ret} stores the return value.

Unlike previous systems with method effects (notably that of Greenhouse and Boyland [GB99]), a method effect for us is given a *linear* semantics: it precludes the existence of an incompatible effect elsewhere. Thus while a method possesses the permission passed in as a read-effect, the state referred to is immutable. Thus we get the semantics of Pechtchanski's "context immutability" [PS02] (without needing a whole-program analysis or run-time checking). In other words, the read effect prevents dangerous aliasing with read-write state. This does not overly strengthen the semantics, because allowing such aliasing should be considered an error. Such situations are called "erroneous" in Ada; they are deemed bad, but the Ada compiler does not prevent them, statically or dynamically.

Pointer Annotations and Methods

Pointer annotations on parameters and method returns are handled differently than effects: the appropriate permissions are passed into the function for a parameter and out of the function for a return value. These permissions are the same as used with fields, but lacking the existential closure. Thus, the method annotation for the `first` method in the `List` in Figure 1.4 would be

$$\forall z; zr_{\text{this}}.\text{All}{\rightarrow}0 + \text{cooked}\,(r_{\text{this}}) + r_{\text{this}} \in \text{List} \longrightarrow \exists\emptyset; zr_{\text{this}}.\text{All}{\rightarrow}0 + \neg(r_{\text{ret}} = 0)\ ?\ r_{\text{ret}}.\text{All}{\rightarrow}0 \prec 0.\text{Owned} + \text{cooked}\,(r_{\text{ret}}) + r_{\text{ret}} \in \text{Object} : \emptyset$$

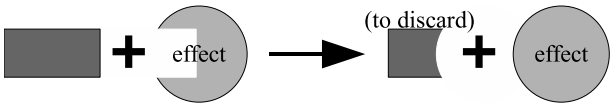
The same fraction of the permission for the `All` data group is returned from the method as was passed in.

(As shorthand, I will hereafter adopt some conventions in permission examples. An omitted fraction on a permission is equivalent to a fraction of 1. A data group—which always points to null—may have its pointer information elided. Thus, $r_{\text{this}}.\text{All}$ is equivalent to $1r_{\text{this}}.\text{All}{\rightarrow}0$.)

Returning Permissions From Effects

The effects of a method are the permissions that a method needs to operate; they are returned when the method returns. In some cases, the return value may be annotated as having acquired permissions "from" some of the method effects. When this happens the permissions are actually carved out from the effects, and thus the method effects are not returned until the return value is no longer needed. The `ListIterator` in Figure 1.4 borrows permission to read the `List` from the read effect on the `iterator` method. The method returns (among other things) a linear implication showing that permission to the `Iterator` has been carved out of the permission representing the

effect. The method annotation for `Iterator` is



$$\forall z; z r_{\text{this}}.\text{All}{\rightarrow}0 + \text{cooked}\,(r_{\text{this}}) + r_{\text{this}} \in \text{List} \longrightarrow \exists \emptyset;$$
$$r_{\text{ret}}.\text{All}{\rightarrow}0 + \text{cooked}\,(r_{\text{this}}) + r_{\text{this}} \in \text{ListIterator}+$$
$$(r_{\text{ret}}.\text{All}{\rightarrow}0 — + z r_{\text{this}}.\text{All}{\rightarrow}0 + \text{cooked}\,(r_{\text{this}}) + r_{\text{this}} \in \text{List} + v)$$

The v represents the "rest" of the iterator, beyond what it borrowed from the `List`. We can statically infer when to restore the permission to the list, but doing so consumes the iterator permission, rendering the iterator useless [BRZ07].

The annotation ***`borrowed`*** is represents a pointer with no permissions associated with the location to which it refers. Thus, this location is not existentially qualified. Borrowed annotations are placed on method parameters, receivers, and returns; any permission for the referred-to object will be passed separately using the method effects. The annotation ***`identity`*** has the same permission semantics as ***`borrowed`***, but holds a different meaning in the language. Something is annotated with ***`identity`*** to indicate that it has an uninteresting value; ***`identity`*** can also be used for boolean values, by testing the nullity of the pointer. Both annotations are listed at the bottom of Figure 2.6.

2.3 A simple language

Figure 2.8 describes a simple object-oriented language with several pointer annotations. However, we do not actually check permissions in this language. Instead, as

D ::= `class` C`<`$pa\,x$`>extends` C' `{`$\overline{F}$ K $\overline{M}$ `}` — *class definition*

F ::= — *field declaration:*

- t f [`in` G];
- `group` G [`in` G'];

K ::= fx C(`this` $\overline{t\,x}$) `{super(`$\overline{e}$`);`e`;}` — *constructor definition*

M ::= fx t m(`this` $\overline{t\,x}$) a `{return` e`;}` — *method definition*

fx ::= `writes` $\overline{g}$ `reads` $\overline{g}$ — *effects*

g ::= e`.`f — *target*

`shared` $\triangleq$ `null.Owned`

t ::= a C — *annotated type*

a ::= ra na pa — *annotations*

pa ::= — *pointer annotations:*

- `unique` | `shared`
- `borrowed` — *(only on parameters)*
- `readonly` | `unique-write`
- `immutable`
- *`owner`* | `readonly-`*`owner`* — *(`this` or ownership parameter)*
- `from(`e`)` | `readonly-from(`e`)` — *(only on return types)*
- `identity`

na ::= — *nullity modifier:*

- `nonnull`
- `maybenull`

ra ::= — *raw modifier:*

- `cooked` — *invariant fully established*
- `raw[`C`]` — *invariant through* C
- `raw` $\triangleq$ `raw[Object]`

e ::= — *terms:*

- x — *variable*
- `null` — *null location*
- e`;`e — *sequencing*
- e`==`e `?` e `:` e — *conditional*
- e `instanceof` C `?` e `:` e — *instanceof*
- `let` x`=`e `in` e — *local*
- e`.`f — *field read*
- e`.`f`=`e — *field write*
- `(`C`)`e — *downcast*
- `new` C — *allocation*
- e`.`m`(`$\overline{e}$`)` — *method dispatch*
- e`.`C`#`m`(`$\overline{e}$`)` — *method call*
- `new` C`<`x`>(`$\overline{e}$`)` $\triangleq$ `new` C`.`C`#`C`(`x`,`$\overline{e}$`)`
- `super.`m`(`$\overline{e}$`)` $\triangleq$ `this.`C'`#`m`(`$\overline{e}$`)`

Figure 2.8: High-Level Syntax.

noted above, we provide permission annotations for each method, and define a predicate for each class. The type system in Figures 2.9 and 2.10 uses these, and ignores the high-level annotations.

2.3.1 Type System

Figures 2.9 and 2.10 shows the formal type rules used for permission-checking programs in our simple language. We include brief explanations of some of the less-obvious rules.

For IF, after evaluating e_1 and e_2 respectively, the type system makes the equality or inequality being tested available in the corresponding branch. At the end of it, we create a fresh pointer variable r to be the type for the result. The output environment will have r equal to different actual types depending on the result of the condition. We use conditional permissions to come up with a single output permission. This permission may need to be simplified/restructured by TRANS (q.v.) before it can be useful. The rule IFINSTANCEOF is similar.

The CAST rule checks that the reference has the correct type before permitting the cast; if necessary, the programmer will have to insert an explicit `instanceof` check around the cast.

For READ, we require that at least we should have partial permission to access the field of an object, which will be represented as $\xi\rho.f \rightarrow \rho_f$. But for WRITE, we require the whole permission $\rho_1.f \rightarrow \rho_f$ for the field that will be updated.

In DISPATCH, when we try to type check a method invocation, we don't know which method will be picked until figuring out the type for the receiver. The type rule will pick some type that the system knows for the receiver and check using that. For most precision, one would pick the "best" static type, but safety requires only *a* possible type. The rule for methods checks overriding to ensure that picking a less

$$\textsc{Variable}\ \frac{r_x \in \Delta}{\Delta; \Pi \vdash x \Downarrow r_x \dashv \Delta; \Pi} \qquad \textsc{ObjLoc}\ E \vdash o \Downarrow o \dashv E \qquad \textsc{Seq}\ \frac{E \vdash e_1 \Downarrow \rho_1 \dashv E' \vdash e_2 \Downarrow \rho_2 \dashv E''}{E \vdash e_1\,;\ e_2 \Downarrow \rho_2 \dashv E''}$$

$$\textsc{If}\ \frac{\begin{array}{c} E \vdash e_1 \Downarrow \rho_1 \dashv E' \vdash e_2 \Downarrow \rho_2 \dashv E'' \qquad E'' + \rho_1 = \rho_2 \vdash e_3 \Downarrow \rho_3 \dashv \Delta_1; \Pi_1 \\ E'' + \neg(\rho_1 = \rho_2) \vdash e_4 \Downarrow \rho_4 \dashv \Delta_2; \Pi_2 \qquad r \notin \Delta_1 \cup \Delta_2 \end{array}}{E \vdash e_1 \texttt{==} e_2\ \texttt{?}\ e_3 \texttt{:} e_4 \Downarrow r \dashv \Delta_1 \cup \Delta_2 \cup \{r\}\,; \rho_1 = \rho_2\ ?\ (\Pi_1 + (r = \rho_3)) : (\Pi_2 + (r = \rho_4))}$$

$$\textsc{IfInstanceof}\ \frac{\begin{array}{c} E \vdash e \Downarrow \rho \dashv E' \\ E' + \rho \in C \vdash e_1 \Downarrow \rho_1 \dashv \Delta_1; \Pi_1 \qquad E' + \neg(\rho \in C) \vdash e_2 \Downarrow \rho_2 \dashv \Delta_2; \Pi_2 \\ r \notin \Delta_1 \cup \Delta_2 \qquad \Pi_1' = (\Pi_1 + (r = \rho_1)) \qquad \Pi_2' = (\Pi_2 + (r = \rho_2)) \end{array}}{E \vdash e\ \texttt{instanceof}\ C\ \texttt{?}\ e_1 \texttt{:} e_2 \Downarrow r \dashv \Delta_1 \cup \Delta_2 \cup \{r\}\,; \rho \in C\ ?\ \Pi_1' : \Pi_2'}$$

$$\textsc{Local}\ \frac{E \vdash e_1 \Downarrow \rho_1 \dashv \Delta_1; \Pi_1 \qquad r_x \notin \Delta \qquad \Delta_1 \cup \{r_x\}\,; \Pi_1 + (\rho_1 = r_x) \vdash e_2 \Downarrow \rho_2 \dashv \Delta_2; \Pi_2}{E \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 \Downarrow [r_x \mapsto \rho_1]\rho_2 \dashv \Delta_2 \setminus \{r_x\}\,; [r_x \mapsto \rho_1]\Pi_2}$$

$$\textsc{Cast}\ \frac{E \vdash e \Downarrow \rho \dashv E' \qquad E' = (\Delta'; \Pi' + \rho \in C)}{E \vdash \texttt{(}C\texttt{)}e \Downarrow \rho \dashv E'}$$

$$\textsc{Read}\ \frac{E \vdash e \Downarrow \rho \dashv E' \qquad E' = \Delta'; \Pi' + \xi\rho.f{\rightarrow}\rho_f}{E \vdash e\,.\,f \Downarrow \rho_f \dashv E'}$$

$$\textsc{Write}\ \frac{E \vdash e_1 \Downarrow \rho_1 \dashv E' \vdash e_2 \Downarrow \rho_2 \dashv E'' \qquad E'' = \Delta''; \Pi'' + \rho_1.f{\rightarrow}\rho_f}{E \vdash e_1\,.\,f\texttt{=}e_2 \Downarrow \rho_2 \dashv \Delta''; \Pi'' + \rho_1.f{\rightarrow}\rho_2}$$

$$\textsc{New}\ \frac{r \notin \Delta \qquad \Pi' = \Pi + r \in C + r.f{\rightarrow}0 \mid f \in \text{fields}(C) + \neg(r \in C') \mid C \not\preceq C'}{\Delta; \Pi \vdash \texttt{new}\ C \Downarrow r \dashv \Delta \cup \{r\}\,; \Pi'}$$

$$\textsc{Dispatch}\ \frac{E \vdash e_0 \Downarrow \rho_0 \dashv E_0 \qquad E_0 = (\Delta; \Pi_0 + \rho_0 \in C) \qquad E \vdash e_0\,.\,C\texttt{\#}m\texttt{(}\overline{e}\texttt{)} \Downarrow r \dashv E'}{E \vdash e_0\,.\,m\texttt{(}\overline{e}\texttt{)} \Downarrow r \dashv E'}$$

Figure 2.9: Permission Type Rules (Part 1)

$$\textsc{Trans}\quad \frac{E_1 \rightsquigarrow E_2 \vdash e \Downarrow \rho \dashv E_3 \rightsquigarrow E_4}{E_1 \vdash e \Downarrow \rho \dashv E_4}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}|\overline{e}| = n \qquad E \vdash e_0 \Downarrow \rho_0 \dashv E_0 \vdash e_1 \Downarrow \rho_1 \dashv \ldots \vdash e_n \Downarrow \rho_n \dashv E_n \\ E_0 = (\Delta; \Pi_0 + \rho_0 \in C) \qquad \mathrm{mbody}(C, m) = (\overline{x}, e, (\forall \Delta'; \Pi' \longrightarrow \exists \Delta''; \Pi'')) \\ |\overline{x}| = n \qquad \Delta''' \cap \Delta = \emptyset \qquad \sigma : \Delta' \to \Delta \qquad \sigma'' : \Delta''' \to \Delta'' \qquad E_n = \Delta; \Pi + \sigma\Pi' \\ \Pi'' = \sigma''\Pi''' \qquad \forall_i \sigma(r_{x_i}) = \rho_i \qquad \sigma(r_{\mathtt{this}}) = \rho_0 \qquad r \in \Delta''' \qquad \sigma''(r) = r_{\mathtt{ret}}\end{array}}{E \vdash e_0 \,.\, C\#m(\overline{e}) \Downarrow r \dashv \Delta \cup \Delta'''; \Pi + \sigma\Pi'''}$$

$$\textsc{Method}\quad \frac{\begin{array}{c}\Delta_1; \Pi_1 + r_{\mathtt{this}} \in C \vdash e \Downarrow \rho \dashv \Delta'; \sigma\Pi_2 \\ \{\overline{r_x}, r_{\mathtt{this}}\} \subseteq \Delta_1 \qquad \Delta' \cap \Delta_2 = \emptyset \qquad \sigma : \Delta_2 \to \Delta' \qquad r_{\mathtt{ret}} \in \Delta_2 \\ \sigma(r_{\mathtt{ret}}) = \rho \qquad \forall_{C \preceq C'} \mathrm{mbody}(m, C') = (\overline{x'}, e', (\forall \Delta_1'; \Pi_1' \longrightarrow \exists \Delta_2'; \Pi_2')) \Rightarrow \\ |\overline{x'}| = |\overline{x}| \wedge [\overline{x' \mapsto x}]\Pi_1' \overset{*}{\rightsquigarrow} \Pi_1 \wedge \Pi_2 \overset{*}{\rightsquigarrow} [\overline{x' \mapsto x}]\Pi_2'\end{array}}{\vdash \forall \Delta_1; \Pi_1 \longrightarrow \exists \Delta_2; \Pi_2\, m(\overline{x})\,\mathtt{:=}e \text{ is defined in class } C}$$

Figure 2.10: Permission Type Rules (Part 2)

precise type does not subvert the permission type system. Then Dispatch delegates to Call.

In Call, after we check the type for the receiver, we use it to fetch the procedure type $\alpha = \forall \Delta'; \Pi' \longrightarrow \exists \Delta''; \Pi''$. Once we have checked each actual parameter, we can form a substitution σ to substitute the procedure variables. The permissions after evaluating all arguments are split into two parts: one which matches the substituted input permissions $\sigma\Pi'$ and one which contains the remaining permissions. The latter are combined with the substituted output permissions to create the resulting environment.

On the other side of the method call boundary, Method type checks the body of the method in the input environment of the method type, with the additional fact that the receiver has the type of the class. At the end of the method body,

the output permissions must match a substitution of the output environment. In addition, if the method overrides another method, it should satisfy the standard covariant/contravariant condition, specified using environment transformation.

2.3.2 Operational Semantics

The operational semantics is given in Figure 2.11 in terms of a small-step evaluation. Term evaluation is defined by a relation of the form $(\mu, e) \rightarrow (\mu', e')$ where μ is a store which partially maps locations (object address and field name pairs) to other addresses in memory:

$$\mu : (O \times F) \rightharpoonup O$$

The evaluation rules are straightforward. The rules that simply move the evaluation to a subterm are collected into a single rule E-Common using an evaluation context T[•]. T[•] shows which subterm will experience evaluation next.

We suppose the (allocated *and* unallocated) object space is partitioned by class such that class(o) always gives the precise object type for any object reference o. We also assume an unlimited supply of objects of any type. Thus the only possible errors (causing evaluation to get stuck) are

- A failed cast;
- Attempting to access (read or write) non-existing state;
- Calling an undefined method;
- Calling a method with the wrong number of parameters.

The well known error of "dereferencing a null pointer" is subsumed by the error of accessing non-existent state.

$\mathrm{T}[\bullet] ::= \bullet;\ e \mid \bullet == e\ ?\ e : e \mid o == \bullet\ ?\ e : e \mid \bullet \text{ instanceof } C\ ?\ e : e \mid \text{let } x = \bullet \text{ in } e \mid \bullet.f \mid \bullet.f = e \mid o.f = \bullet \mid (C)\ \bullet \mid \bullet.m(\overline{e}) \mid o.m(\overline{o}, \bullet, \overline{e})$

E-Common
$$\frac{(\mu; e) \to (\mu'; e')}{(\mu; \mathrm{T}[e]) \to (\mu'; \mathrm{T}[e'])}$$

E-Seq
$$(\mu; o; e_2) \to (\mu; e_2)$$

E-IfTrue
$$\frac{o_1 = o_2}{(\mu; o_1 == o_2 ? e_3 : e_4) \to (\mu; e_3)}$$

E-IfFalse
$$\frac{o_1 \neq o_2}{(\mu; o_1 == o_2 ? e_3 : e_4) \to (\mu; e_4)}$$

E-IfInstanofTrue
$$\frac{\mathrm{class}(o_1) \preceq C}{(\mu; o_1\ \texttt{instanceof}\ C ? e_2 : e_3) \to (\mu; e_2)}$$

E-IfInstanofFalse
$$\frac{\mathrm{class}(o_1) \not\preceq C}{(\mu; o_1\ \texttt{instanceof}\ C ? e_2 : e_3) \to (\mu; e_3)}$$

E-Let
$$(\mu; \texttt{let}\ x = o_1\ \texttt{in}\ e_2) \to (\mu; [x \mapsto o_1]e_2)$$

E-Read
$$\frac{\mu(o, f) = o'}{(\mu; o.f) \to (\mu; o')}$$

E-Write
$$\frac{\mu(o, f) = o' \qquad \mu' = \mu[(o_1, f) \mapsto o_2]}{(\mu; o_1.f = o_2) \to (\mu'; o_2)}$$

E-Cast
$$\frac{\mathrm{class}(o) \preceq C}{(\mu; (C)o) \to (\mu; o)}$$

E-New
$$\frac{\forall_f\ (o, f) \notin \mathrm{Dom}(\mu)}{(\mu; \texttt{new}\ C) \to (\mu[(o, f) \mapsto 0 \mid f \in \mathrm{fields}(C)]; o)}$$

E-Call
$$\frac{\mathrm{mbody}(m, \mathrm{class}(o_0)) = (\overline{x}, t, \alpha)}{(\mu; o_0.m(\overline{o})) \to (\mu; [r_{\texttt{this}} \mapsto o_0, \overline{x \mapsto o}]e)}$$

Figure 2.11: Operational Semantics.

Michael Welch is currently attempting, in separate but closely related work, to prove, using Twelf [PS], that the above type system is sound with respect to this operational semantics.

2.4 Semantics of Annotations

Having described how we can define sundry annotations using fractional permissions, it is worth pausing to examine how well these semantics correspond to our intuitive notions of the annotations.

The essential property of fractional permissions is that they are linear. There is exactly one permission for each piece of state (field or data group). Thus, if a method possesses a full write permission to a field, say by declaring a write effect on the field, we *know* that no one else could possibly have *any* permission to access the field. The write effect cannot interfere with any other effect. Similarly, possession of a read permission ensures that, while others may simultaneously read the field with a separate fraction of the same permission, no one could write the field. Again, interference is obviated if the permissions type-check. This is enforced in the type system using the rules for reading and writing fields, and also in passing permissions corresponding to the effects to each methods.

That permissions represent effects is unsurprising given that they are duals [BR05]. The more interesting question is how they support annotations traditionally associated with heap state, such as uniqueness. However, the underlying intent behind pointer annotations is to preserve restrictions on accessing state. Thus the traditional model, which does not restrict use of any pointer, instead controls access by permitting or not permitting the existence of certain references. With permissions, the locus of control is shifted: all references are allowed, but not all may be used.

2.4.1 Permission Semantics of `unique`

Let us examine what this shift means for ***`unique`*** pointers. The "standard" definition for a uniqueness is that the unique reference is the only persistent (that is, stored in the heap) pointer to the object in question. With permissions, the permission for the unique field encloses the permission for the object. On the surface, these two definitions are incompatible, but they express nearly the same intuition. We think of a (non-null) ***`unique`*** pointer as 'the only pointer that points to that location in memory.' A ban on the existence of other pointers accomplishes this intent, but can be difficult to enforce while allowing flexible usage. Only restricting persistent pointers alleviates usability constraints, but somewhat can weaken the meaning of uniqueness. With permissions, we alter the semantics slightly to make a unique pointer 'the only usable pointer that points to that location in memory.' That is, other references can exist, but (permission for) the unique pointer is required to actually access that location.

Further, the requirement that effects be returned, combined with the requirement that all fields be given some annotation, means that in practice, programs that create persistent aliases will generally fail to type-check when using permissions.

```
class UniqueDemo{
  nonnull unique Object o1;
  nonnull unique Object o2;
  UniqueDemo(){ o1 = new Object(); o2 = new Object(); }

  writes this.o1, this.o2
  void bad1(){
    o1 = o2;
```

```
  }
}
```

This violates uniqueness because the object, initially pointed-to by only `o1` has two fields pointing to it after `bad1` is called. Here `bad1` will fail to type-check because the linear permission for the `Object` to which `o2` points cannot be attached to both fields. Even were the effect on `o2` a read effect, returning the effects from this method would require more than 1 permission to the shared object.

The requirement that permissions corresponding to the effects be returned from the method forces both fields to enclose the (single) permission. That is, the effects on the method translate into the permission annotation:

$$\forall r_{\text{this}} \cdot \left(\begin{array}{c} \exists r \cdot r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0 \\ + \exists r \cdot r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0 \end{array} \right) \longrightarrow \exists r_{\text{ret}} \cdot \left(\begin{array}{c} \exists r \cdot r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0 \\ + \exists r \cdot r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0 \end{array} \right)$$

(Recall that r_{ret} refers to the return value, and is unused in a `void` function.) Performing the assignment will require us to unpack the first existential to write `o1` and to unpack the second to read `o2`. Before the assignment, then, the type system should possess permissions equivalent to

$$r_{\text{this}}.\texttt{o1} \rightarrow r_1 + r_1.\text{All} \rightarrow 0 + r_{\text{this}}.\texttt{o2} \rightarrow r_2 + r_2.\text{All} \rightarrow 0$$

After the assignment, we get

$$r_{\text{this}}.\texttt{o1} \rightarrow r_2 + r_1.\text{All} \rightarrow 0 + r_{\text{this}}.\texttt{o2} \rightarrow r_2 + r_2.\text{All} \rightarrow 0$$

From these permissions we can form the existential $\exists r \cdot r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0$ or the existential $\exists r \cdot r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0$ but not both, as each requires the single linear

permission $r_2 \rightarrow .\text{All}$.

(The permissions shown here are somewhat simplified. We ignore for the moment class information, raw and cooked. Also, the fields are ***nonnull***. Thus, the full input permission for, say, `o1` should be $\exists r \cdot r_{\text{this}}.o1 \rightarrow r + \text{true}\,?\,r.\text{All} \rightarrow 0 + \text{cooked}\,(r) + r \in \text{Object} : \emptyset$ but this is equivalent—ignoring classes—to the stated permission $\exists r \cdot r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0$.)

This example also highlights why most persistent aliases will show up as type errors. Pointers in the heap are generally fields; fields have some pointer annotation dictating the disposal of the permission for the pointed-to state. The permission for state uniquely pointed-to cannot be put anywhere else; if the unique field encloses the permission, no other annotation can be satisfied using that permission (with the exception of ***identity***). That is, wherever `o2` is assigned, $r_2.\text{All} \rightarrow 0$ cannot be packed up with `o2` and also packed with some other pointer or adopted into some other data group.

2.4.2 A More Detailed Example

Because permissions underlie both uniqueness and effects, they model the interactions between the two. In the example in Figure 1.3, we saw how a temporarily borrowed reference could affect a ***unique*** field after the object had been handed off. What does this look like when using permissions? First, the method `send` would gain ***write*** effects for `a` and `b`, while `getList` writes only `a`. Once more ignoring class

types, we get annotations of

$$\forall r_{\text{this}} \cdot \left(\begin{array}{c} \exists r \cdot r_{\text{this}}.\texttt{a} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset \\ + \exists r \cdot r_{\text{this}}.\texttt{b} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset \end{array} \right) \longrightarrow \exists r_{\text{ret}} \cdot \left(\begin{array}{c} \exists r \cdot r_{\text{this}}.\texttt{a} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset \\ + \exists r \cdot r_{\text{this}}.\texttt{b} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset \\ + r_{\text{ret}} \neq 0 \,?\, r_{\text{ret}}.\text{All} \rightarrow 0 : \emptyset \end{array} \right)$$

for `send` and

$$\forall r_{\text{this}} \cdot \left(\exists r \cdot r_{\text{this}}.\texttt{a} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset \right) \longrightarrow \exists r_{\text{ret}} \cdot \left(\begin{array}{c} \exists r \cdot r_{\text{this}}.\texttt{a} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset \\ + r_{\text{ret}} \neq 0 \,?\, r_{\text{ret}}.\text{All} \rightarrow 0 : \emptyset \end{array} \right)$$

for `getList`. Then, we can examine what the permission environment looks like when type-checking `send`.

Figure 2.12 shows an approximation of the permission typing for the send method. (As before we ignore class types.) Before each line of code there are two permissions. The first is the permission that resulted from the previous line of code, while the second is a transformation of the first into a form that can be used to check the following line.

Initially we start with the two input permissions, derived from the effects declaration on the method. Immediately, we transform these permissions to unpack that of `a` prior to assigning it to the local variable `bad`. After this assignment, we gain the additional permission fact that `bad` refers to the same location as the unpacked field `a`.

```
writes a, b
nonnull unique ListItem send(){
```

$$\begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \end{pmatrix} a \rightsquigarrow \begin{pmatrix} r_{\text{this}}.\texttt{a} \to r_a \\ +r_a \neq 0 \,?\, r_a.\text{All} \to 0 : \emptyset \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \end{pmatrix}$$

```
  ListItem bad = a;
```

$$\begin{pmatrix} r_{\text{this}}.\texttt{a} \to r_a \\ +r_a \neq 0 \,?\, r_a.\text{All} \to 0 : \emptyset \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \end{pmatrix} \rightsquigarrow \begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \end{pmatrix}$$

```
  ListItem t = getList();
```

$$\begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{t}} \neq 0 \,?\, r_{\text{t}}.\text{All} \to 0 : \emptyset \end{pmatrix} \rightsquigarrow \begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{t}} \neq 0 \,?\, r_{\text{t}}.\text{All} \to 0 : \emptyset \\ +0 \neq 0 \,?\, 0.\text{All} \to 0 : \emptyset \end{pmatrix}$$

```
  ListItem ret = new ListItem(t,null);
```

$$\begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{ret}}.\text{All} \to 0 \end{pmatrix} \rightsquigarrow \begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{bad}} = r_a \\ +r_{\text{this}}.\texttt{b} \to r_b \\ +r_b \neq 0 \,?\, r_b.\text{All} \to 0 : \emptyset \\ +r_{\text{ret}}.\text{All} \to 0 \end{pmatrix}$$

```
  b = bad.getItem();  // WE CANNOT MAKE THIS CALL--no permission for bad.item
```

$$\begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{bad}} = r_a \\ +r_{\text{this}}.\texttt{b} \to r_? \\ +r_? \neq 0 \,?\, r_?.\text{All} \to 0 : \emptyset \\ +r_{\text{ret}}.\text{All} \to 0 \end{pmatrix} \rightsquigarrow \begin{pmatrix} \exists r \cdot (r_{\text{this}}.\texttt{a} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\texttt{b} \to r + r \neq 0 \,?\, r.\text{All} \to 0 : \emptyset) \\ +r_{\text{ret}} \neq 0 \,?\, r_{\text{ret}}.\text{All} \to 0 : \emptyset \end{pmatrix}$$

```
  return ret;
}
```

Figure 2.12: Approximate Permission Typing of `send`

Next we need to call the method `getList`. To satisfy the effects declaration on this method, we must pass it the (existentially packed) permission for the field `a`. Thus we immediately transform our permission to re-pack `a`. (In the original example, this call was nested as a parameter to the constructor. A separate local variable is used here to clarify the steps involved, which only change in the naming of the permission variable used for the temporary return value.) After the call, `t` is assigned the ***`unique`*** return value. (In theory, `t` and the returned value have separate location variable which happen to be equal; the same variable is used for both for simplicity.)

After this, the temporary ***`unique`*** value and `null` are sent to the constructor as ***`unique`*** parameters. Sending `null` as a ***`maybenull unique`*** value is possible because the condition of the conditional permission is counterfactual. The constructor call returns a new item of which we possess the All permission. It is possible to also receive the fact that the newly created object is not null; however, this fact can be derived from the possession of the permission—we cannot have the All permission of `null`. (We have again conflated the location variable for the returned object with that of the local variable storing it.)

Then we can unpack the field `b` in anticipation of assigning it a value. But then we run into a problem calling the `getItem` method of `bad`. This method writes the `item` field of its host; the caller must pass it a write permission for this field. We cannot transform our permission to obtain a write permission for `bad.item`! Were `a` still unpacked, we could carve the permission for r_a.`item` out of the permission r_a.All→0, which suffices as $r_{\text{bad}} = r_a$. But it is not unpacked. If we unpack this permission again, we will not get the same location variable as `a` may not, and in fact *does* not, refer to the same location as before. This highlights the general pattern for borrowing: a unique pointer may be indiscriminately aliased until some event requiring its associated permissions, such as passing it as a parameter or closing the

existential to pass it as an effect, after which lingering aliases lack all permission to access the object.

Once we cannot make the method call, we are in some sense done: `send` does not type-check. For purposes of explanation, however, we can look at what would have happened if the call had succeeded. (In practice, the analysis will also continue after reporting an error, in the hopes of identifying further problems.) The `getItem` method returns a unique return value (here named $r_?$) and the nonexistent (and therefore ignored) permission to write `bad.item`. The uniqueness of the returned value ensures we have sufficient permissions to pack the existential permission for `b`. As we know r_{ret} is non-null, we can transform r_{ret}.All→0 to the syntactically appropriate form for a ***`unique`*** return value. (Here we conflate r_{ret} the location stored in the local variable `ret` with r_{ret} the location of the return value of the method. While in general these may be separate locations, in this example `ret` is the returned value.) In this final form, we can drop the superfluous (once `bad` leaves scope) equality fact and have precisely the permissions we need to return from the method.

2.4.3 Effects and Uniqueness

The example in the previous section illustrates how expressing both effects and uniqueness with linear permissions requires them to interact as desired. This occurs most obviously in the inability to call `getItem`. The immediate point is that the type system detects the uniqueness error when it cannot satisfy the effects annotation. The reason it cannot is that a previous effect effectively killed the (utility of the) borrowed local variable `bad`. The mere act of checking both effects and uniqueness with permissions naturally expressed the interaction by which effects writing the unique field kill its aliases.

There is another interaction between effects and uniqueness attempted, if not

accomplished, here. To see it, let's look at a 'corrected' version of the method:

```
writes this.a, this.b
unique ListITem send(){
  ListItem bad = a;
  b = bad.getItem();
  ListItem t = getList();
  ListITem ret = new ListItem(t,null);
  return t;
}
```

This method uses uniqueness correctly in that it modifies the `ListItem` in `a` through its alias prior to handing it off to the returned `ListItem`. In this version. the permissions present prior to calling `getItem` resemble

$$\left(\begin{array}{l} r_{\text{this}}.\mathtt{a}{\rightarrow}r_a \\ +r_a \neq 0\ ?\ r_a.\text{All}{\rightarrow}0 : \emptyset \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\mathtt{b}{\rightarrow}r + r \neq 0\ ?\ r.\text{All}{\rightarrow}0 : \emptyset) \end{array}\right)$$

To call `getItem`, we need permission to write the field `bad.item`. Where can we get this permission? Since $r_{\text{bad}} = r_a$ we can transform the previous permission int

$$\left(\begin{array}{l} r_{\text{this}}.\mathtt{a}{\rightarrow}r_a \\ +r_{\text{bad}} \neq 0\ ?\ r_{\text{bad}}.\text{All}{\rightarrow}0 : \emptyset \\ +r_{\text{bad}} = r_a \\ +\exists r \cdot (r_{\text{this}}.\mathtt{b}{\rightarrow}r + r \neq 0\ ?\ r.\text{All}{\rightarrow}0 : \emptyset) \end{array}\right)$$

Furthermore, once the method call has begun, we can infer that `bad` is non-null: were `bad` null, the call would fail with a `NullPointerException` before we would need to pass any permissions. This allows the following transformations:

$$\left(\begin{array}{l} \quad r_{\text{this}}.\texttt{a}{\rightarrow}r_a \\ +r_{\text{bad}} \neq 0 \ ?\ r_{\text{bad}}.\text{All}{\rightarrow}0 : \emptyset \\ +r_{\text{bad}} = r_a \\ +r_{\text{bad}} \neq 0 \\ +\exists r \cdot (r_{\text{this}}.\texttt{b}{\rightarrow}r + r \neq 0 \ ?\ r.\text{All}{\rightarrow}0 : \emptyset) \end{array}\right)$$

$$\rightsquigarrow \left(\begin{array}{l} \quad r_{\text{this}}.\texttt{a}{\rightarrow}r_a \\ +r_{\text{bad}}.\text{All}{\rightarrow}0 \\ +r_{\text{bad}} = r_a \\ +r_{\text{bad}} \neq 0 \\ +\exists r \cdot (r_{\text{this}}.\texttt{b}{\rightarrow}r + r \neq 0 \ ?\ r.\text{All}{\rightarrow}0 : \emptyset) \end{array}\right)$$

We now have permission for $r_{\text{bad}}.\text{All}$, but not $r_{\text{bad}}.\texttt{item}$. However, we know from the class annotation for the `ListItem` class (elided here for simplicity, as with all other

class information) that $\exists r \cdot (r_{\text{bad}}.\texttt{item} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset) \prec r_{\text{bad}}.\text{All}$. This enables us to carve the permission for `bad.item` out of the permission for `bad.`All.

$$\left(\begin{array}{l} \quad r_{\text{this}}.\texttt{a} \rightarrow r_a \\ + r_{\text{bad}}.\text{All} \rightarrow 0 \\ + \exists r \cdot (r_{\text{bad}}.\texttt{item} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset) \prec r_{\text{bad}}.\text{All} \\ + r_{\text{bad}} = r_a \\ + r_{\text{bad}} \neq 0 \\ + \exists r \cdot (r_{\text{this}}.\texttt{b} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset) \end{array}\right)$$

$$\rightsquigarrow \left(\begin{array}{l} \quad r_{\text{this}}.\texttt{a} \rightarrow r_a \\ + \exists r \cdot (r_{\text{bad}}.\texttt{item} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset) \\ + \exists r \cdot (r_{\text{bad}}.\texttt{item} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset) \multimap + r_{\text{bad}}.\text{All} \rightarrow 0 \\ + \exists r \cdot (r_{\text{bad}}.\texttt{item} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset) \prec r_{\text{bad}}.\text{All} \\ + r_{\text{bad}} = r_a \\ + r_{\text{bad}} \neq 0 \\ + \exists r \cdot (r_{\text{this}}.\texttt{b} \rightarrow r + r \neq 0 \,?\, r.\text{All} \rightarrow 0 : \emptyset) \end{array}\right)$$

Now we have the necessary permissions to call `getItem`. The process can be reversed upon method return.

The above transformations enable us to call a method (`getItem`) which has a write effect on `bad.item` (that is, `a.item`) from within the `send` method without declaring a write effect on `a.item` on `send`. Carving permission for `bad.item` from the permission `a.`All corresponds directly with mapping the effects of a uniquely pointed-to object onto the ***`unique`*** field. Indeed, it works *because* the permission for `a`, as a ***`unique`*** field, encloses the permission for `a.`All, from which any field permissions may be

carved. Thus permissions encompass mapping effects onto unique fields as well as burying aliases to a unique field after writes detected by effects.

2.4.4 The Null-ness Modifier

The example also highlights the functionality of ***maybenull***. When given a pointer that may be `null`, the permissions associated with that pointer's annotation are conditional on the pointer not being null. In the example, this is the unpacked permission $r_a \neq 0 \,?\, r_a.\text{All}{\rightarrow}0 : \emptyset$. We can only get the permission $r_a.\text{All}{\rightarrow}0$ from this if we can prove that $r_a \neq 0$, as we do not have that permission for the null pointer.

$$r_a \neq 0 \,?\, r_a.\text{All}{\rightarrow}0 : \emptyset \not\rightsquigarrow r_a.\text{All}{\rightarrow}0$$

$$r_a \neq 0 \,?\, r_a.\text{All}{\rightarrow}0 : \emptyset + r_a \neq 0 \rightsquigarrow r_a.\text{All}{\rightarrow}0$$

(In the previous section our not throwing a `NullPointerException` when calling a function 'proved' `bad` was non-null.) However, we can *form* the conditional permission from either a null reference *or* one for which we have the appropriate permission:

$$r_a = 0 \rightsquigarrow r_a \neq 0 \,?\, r_a.\text{All}{\rightarrow}0 : \emptyset$$

$$r_a.\text{All}{\rightarrow}0 \rightsquigarrow r_a \neq 0 \,?\, r_a.\text{All}{\rightarrow}0 : \emptyset$$

This is even simpler for ***nonnull***, as $\Pi \Leftrightarrow \text{true} \,?\, \Pi : \emptyset$. Thus we can always get the associated permission for a ***nonnull*** pointer, and only form a ***nonnull*** conditional if we have that permission (which in turn implies that the pointer cannot be null).

2.4.5 Ownership

Ownership systems create spheres of control for objects. Other objects may be within the sphere of their owner and thus may only be accessed by their owner. For example, a linked list class may be established as the owner of the node objects making up the list. The purpose of the ownership system is to prevent use of the nodes outside of the list. Traditional ownership system do this by forbidding persistent references from outside the objects to those objects it owns.

With permissions we represent ownership slightly differently. As with uniqueness we permit the existence of references that cross ownership barriers, but limit access using those references. In particular, owned objects can only be accessed if we have permission to access their owner. This is slightly weaker than standard ownership as it still allows access from outside the owner. In particular, owned state may be borrowed in the same manner as unique state, passing permission to the owner as an effect. As with unique state, kept references to owned state will become useless once the effect is returned, making ownership using permissions the same as standard ownership in practice.

2.4.6 Fractions

Fractions are used to distinguish reads and writes. Writing a field requires the entire base permission for that field. Reading a field only requires some fraction. The primary advantage of using fractional permissions to distinguish reads and writes is that, after one splits the write permission into several read permissions (allowing read-read parallelism), one can combine them again to recreate the original write permission, as long as one can account for all fractions. Having the whole permission for the write ensures that writing the field cannot interfere with any other access, as no other access

```
class EZ{
  nonnull unique Object f;

  EZ(nonnull unique Object x){
    f = x;
  }

  writes f
  void foo(nonnull unique Object x){
    bar();
    f = x;
   }

  reads f
  void bar(){
    System.out.println(f);
  }
}
```

Figure 2.13: A Simple Example Using Fractions

is permitted. Previous systems, such as the calculus of capabilities [CWM], allowed the distribution of a write permission into several reads, but had no way to recover the original write permission once it was divided.

The `foo` method in Figure 2.13 demonstrates this use of fractional permissions. The annotation for `foo` (again simplified to ignore class types) is

$$\forall r_{\text{this}}, r_x; \exists r \cdot (r_{\text{this}}.f{\rightarrow}r + r.\text{All}{\rightarrow}0) + r_x.\text{All}{\rightarrow}0 \longrightarrow \exists; \exists r \cdot (r_{\text{this}}.f{\rightarrow}r + r.\text{All}{\rightarrow}0)$$

while that of `bar` is

$$\forall r_{\text{this}}, z; z\exists r \cdot (r_{\text{this}}.f{\rightarrow}r + r.\text{All}{\rightarrow}0) \longrightarrow \exists; z\exists r \cdot (r_{\text{this}}.f{\rightarrow}r + r.\text{All}{\rightarrow}0)\,.$$

(On the right-hand side of both procedure types, there is an $\exists$; after the arrow; this

represents the empty list of existentially-qualified permission and variables used in the method. It may be omitted for brevity.) Therefore when permission-checking `foo`, we start with the permission

$$\exists r \cdot (r_{\text{this}}.f \rightarrow r + r.\text{All} \rightarrow 0) + r_x.\text{All} \rightarrow 0$$

and, to call `bar` at the first statement, transform it into

$$\frac{1}{2}\exists r \cdot (r_{\text{this}}.f \rightarrow r + r.\text{All} \rightarrow 0) + \frac{1}{2}\exists r \cdot (r_{\text{this}}.f \rightarrow r + r.\text{All} \rightarrow 0) + r_x.\text{All} \rightarrow 0$$

One half of the permission for `f` can then passed to `bar` as a effect, and subsequently returned, resulting in the same permission as before the call. The subsequent write of `f` requires the entire permission. Fortunately, this transformation is possible; we can combine the two halves to get

$$\exists r \cdot (r_{\text{this}}.f \rightarrow r + r.\text{All} \rightarrow 0) + r_x.\text{All} \rightarrow 0$$

and then unpack the existential

$$r_{\text{this}}.f \rightarrow r_f + r_f.\text{All} \rightarrow 0 + r_x.\text{All} \rightarrow 0$$

We can now perform the assignment, resulting in permissions

$$r_{\text{this}}.f \rightarrow r_x + r_f.\text{All} \rightarrow 0 + r_x.\text{All} \rightarrow 0$$

These can be existentially repackaged to return at the end of the method, together

```
class Split{
  nonnull unique Object o1;
  nonnull unique Object o2;

  Split(){
    o1 = new Object();
    o2 = new Object();
  }

  reads this.All
  void test(){
    if(o1 == o2){
      throw SomeError("Aliased unique values!!  Panic!");
    }
  }

  writes this.All
  void evil(){
    o1 = o2;
    test();
    o2 = new Object();
  }
}
```

Figure 2.14: A Surprising Use of Fractional Permissions

with an extra permission for r_f, which is discarded:

$$\exists r \cdot (r_{\text{this}}.f \rightarrow r + r.\text{All} \rightarrow 0) + r_f.\text{All} \rightarrow 0$$

Thus the method returns safely. The write permission for `f` was divided into two read permissions, then recombined into the write permission again allowing the field to be written. The recombination was possible because adding $\frac{1}{2} + \frac{1}{2} = 1$ ensured we had the entire write permission restored.

Sometimes, fractional permissions can result in surprising situations, as in Figure 2.14. Initially, it appears that the `test` function will never be able to throw its exception—the two ***`nonnull unique`*** fields could never point to the same object as that object would not be ***`unique`***. However, careful examination of the `evil` method shows how it could be done. The annotation for `test` is (approximate modulo class predicates)

$$\forall z, r_{\text{this}}; z r_{\text{this}}.\text{All}{\rightarrow}0+$$
$$(\exists r \cdot (r_{\text{this}}.o1{\rightarrow}r + r.\text{All}{\rightarrow}0) \prec r_{\text{this}}.\text{All} \wedge \exists r \cdot (r_{\text{this}}.o1{\rightarrow}r + r.\text{All}{\rightarrow}0) \prec r_{\text{this}}.\text{All})$$
$$\longrightarrow \exists; z r_{\text{this}}; r_{\text{this}}.\text{All}{\rightarrow}0$$

The annotation for `evil` is similar:

$$\forall r_{\text{this}}; r_{\text{this}}.\text{All}{\rightarrow}0+$$
$$(\exists r \cdot (r_{\text{this}}.o1{\rightarrow}r + r.\text{All}{\rightarrow}0) \prec r_{\text{this}}.\text{All} \wedge \exists r \cdot (r_{\text{this}}.o1{\rightarrow}r + r.\text{All}{\rightarrow}0) \prec r_{\text{this}}.\text{All})$$
$$\longrightarrow \exists; r_{\text{this}}; r_{\text{this}}.\text{All}{\rightarrow}0$$

We now examine the permission-checking of the method `evil`. Given the initial permissions from the procedure annotation, we transform them to enable the assignment between the two nested fields. First the field permissions are carved from the data group, then the existentials are unpacked to allow the assignment to occur.

$$\rightsquigarrow \left(\begin{array}{l} r_{\text{this}}.\text{All}{\rightarrow}0 \\ +\exists r \cdot (r_{\text{this}}.\texttt{o1}{\rightarrow}r + r.\text{All}{\rightarrow}0) \prec r_{\text{this}}.\text{All}{\rightarrow}0 \\ +\exists r \cdot (r_{\text{this}}.\texttt{o2}{\rightarrow}r + r.\text{All}{\rightarrow}0) \prec r_{\text{this}}.\text{All}{\rightarrow}0 \end{array} \right)$$

$$\rightsquigarrow \left(\begin{array}{l} \quad (\exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) + \exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0)) \multimap + r_{\text{this}}.\text{All} \rightarrow 0 \\ + \exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) \\ + \exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0) \end{array}\right)$$

$$\rightsquigarrow \left(\begin{array}{l} \quad (\exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) + \exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0)) \multimap + r_{\text{this}}.\text{All} \rightarrow 0 \\ + r_{\text{this}}.\texttt{o1} \rightarrow r_1 \\ + r_1.\text{All} \rightarrow 0 \\ + r_{\text{this}}.\texttt{o1} \rightarrow r_2 \\ + r_2.\text{All} \rightarrow 0 \end{array}\right)$$

After the assignment, the permissions are almost identical, except that both fields are now referencing the same object. This is legal, but the `evil` method will not be able to return when the permissions are in this state, as both cannot be replaced. However, we can split both the carved-out field permissions *and* the linear implications to produce a fraction of the All permission for the receiver. This fraction is sufficient to call the test method.

$$\left(\begin{array}{l} \quad \left(\begin{array}{r} \exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) \\ + \exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0) \end{array}\right) \multimap + r_{\text{this}}.\text{All} \rightarrow 0 \\ + r_{\text{this}}.\texttt{o1} \rightarrow r_2 \\ + r_1.\text{All} \rightarrow 0 \\ + r_{\text{this}}.\texttt{o1} \rightarrow r_2 \\ + r_2.\text{All} \rightarrow 0 \end{array}\right)$$

$$
\rightsquigarrow
\left(\begin{array}{l}
\quad \frac{1}{2}\left(\left(\begin{array}{r} \exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) \\ +\exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0) \end{array}\right) \multimap r_{\text{this}}.\text{All} \rightarrow 0\right) \\
+\frac{1}{2}\left(\left(\begin{array}{r} \exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) \\ +\exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0) \end{array}\right) \multimap r_{\text{this}}.\text{All} \rightarrow 0\right) \\
+\frac{1}{2} r_{\text{this}}.\texttt{o1} \rightarrow r_2 + \frac{1}{2} r_{\text{this}}.\texttt{o1} \rightarrow r_2 \\
+r_1.\text{All} \rightarrow 0 \\
+\frac{1}{2} r_{\text{this}}.\texttt{o1} \rightarrow r_2 + \frac{1}{2} r_{\text{this}}.\texttt{o1} \rightarrow r_2 \\
+\frac{1}{2} r_2.\text{All} \rightarrow 0 + \frac{1}{2} r_2.\text{All} \rightarrow 0
\end{array}\right)
$$

$$
\rightsquigarrow
\left(\begin{array}{l}
\quad \frac{1}{2}\left(\left(\begin{array}{r} \exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) \\ +\exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0) \end{array}\right) \multimap r_{\text{this}}.\text{All} \rightarrow 0\right) \\
+\frac{1}{2}\left(\begin{array}{r} \exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) \\ +\exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0) \end{array}\right) \multimap \frac{1}{2} r_{\text{this}}.\text{All} \rightarrow 0 \\
+r_1.\text{All} \rightarrow 0 \\
+\frac{1}{2} r_{\text{this}}.\texttt{o1} \rightarrow r_2 + \frac{1}{2} r_2.\text{All} \rightarrow 0 \\
+\frac{1}{2} r_{\text{this}}.\texttt{o1} \rightarrow r_2 \\
+\frac{1}{2} r_{\text{this}}.\texttt{o2} \rightarrow r_2 + \frac{1}{2} r_2.\text{All} \rightarrow 0 \\
+\frac{1}{2} r_{\text{this}}.\texttt{o2} \rightarrow r_2
\end{array}\right)
$$

$$
\rightsquigarrow \left(\begin{array}{l}
\frac{1}{2}\left(\left(\begin{array}{r}\exists r\cdot(r_{\text{this}}.\texttt{o1}{\rightarrow}r + r.\text{All}{\rightarrow}0)\\ +\exists r\cdot(r_{\text{this}}.\texttt{o2}{\rightarrow}r + r.\text{All}{\rightarrow}0)\end{array}\right) -\!\!+ r_{\text{this}}.\text{All}{\rightarrow}0\right)\\
+\frac{1}{2}\left(\begin{array}{r}\exists r\cdot(r_{\text{this}}.\texttt{o1}{\rightarrow}r + r.\text{All}{\rightarrow}0)\\ +\exists r\cdot(r_{\text{this}}.\texttt{o2}{\rightarrow}r + r.\text{All}{\rightarrow}0)\end{array}\right) -\!\!+ \frac{1}{2}r_{\text{this}}.\text{All}{\rightarrow}0\\
+r_1.\text{All}{\rightarrow}0\\
+\frac{1}{2}\exists r\cdot(r_{\text{this}}.\texttt{o1}{\rightarrow}r + r.\text{All}{\rightarrow}0)\\
+\frac{1}{2}r_{\text{this}}.\texttt{o1}{\rightarrow}r_2\\
+\frac{1}{2}\exists r\cdot(r_{\text{this}}.\texttt{o2}{\rightarrow}r + r.\text{All}{\rightarrow}0)\\
+\frac{1}{2}r_{\text{this}}.\texttt{o2}{\rightarrow}r_2
\end{array}\right)
$$

$$
\rightsquigarrow \left(\begin{array}{l}
\frac{1}{2}\left(\left(\begin{array}{r}\exists r\cdot(r_{\text{this}}.\texttt{o1}{\rightarrow}r + r.\text{All}{\rightarrow}0)\\ +\exists r\cdot(r_{\text{this}}.\texttt{o2}{\rightarrow}r + r.\text{All}{\rightarrow}0)\end{array}\right) -\!\!+ r_{\text{this}}.\text{All}{\rightarrow}0\right)\\
+\frac{1}{2}r_{\text{this}}.\text{All}{\rightarrow}0\\
+r_1.\text{All}{\rightarrow}0\\
+\frac{1}{2}r_{\text{this}}.\texttt{o1}{\rightarrow}r_2\\
+\frac{1}{2}r_{\text{this}}.\texttt{o2}{\rightarrow}r_2
\end{array}\right)
$$

Once the fractional permission is returned for the field, this sequence of transformations can be reversed:

$$
\begin{array}{l}
\left(\begin{array}{l}
\quad \frac{1}{2}\left(\left(\begin{array}{r}\exists r \cdot (r_{\text{this}}.\texttt{o1}{\rightarrow}r + r.\text{All}{\rightarrow}0) \\ +\exists r \cdot (r_{\text{this}}.\texttt{o2}{\rightarrow}r + r.\text{All}{\rightarrow}0)\end{array}\right) \multimap r_{\text{this}}.\text{All}{\rightarrow}0\right) \\
+\frac{1}{2} r_{\text{this}}.\text{All}{\rightarrow}0 \\
+r_1.\text{All}{\rightarrow}0 \\
+\frac{1}{2} r_{\text{this}}.\texttt{o1}{\rightarrow}r_2 \\
+\frac{1}{2} r_{\text{this}}.\texttt{o2}{\rightarrow}r_2
\end{array}\right) \\
\rightsquigarrow \left(\begin{array}{l}
\quad \left(\begin{array}{r}\exists r \cdot (r_{\text{this}}.\texttt{o1}{\rightarrow}r + r.\text{All}{\rightarrow}0) \\ +\exists r \cdot (r_{\text{this}}.\texttt{o2}{\rightarrow}r + r.\text{All}{\rightarrow}0)\end{array}\right) \multimap r_{\text{this}}.\text{All}{\rightarrow}0 \\
+r_{\text{this}}.\texttt{o1}{\rightarrow}r_2 \\
+r_1.\text{All}{\rightarrow}0 \\
+r_{\text{this}}.\texttt{o2}{\rightarrow}r_2 \\
+r_2.\text{All}{\rightarrow}0
\end{array}\right)
\end{array}
$$

Then, to enable the safe return from the `evil` method, the two ***`unique`*** fields must be made to refer to different objects once more. This is done easily by assigning one

a newly allocated object.

$$\left(\begin{array}{l} \left(\begin{array}{l} \exists r \cdot (r_{\text{this}}.\texttt{o1} \rightarrow r + r.\text{All} \rightarrow 0) \\ + \exists r \cdot (r_{\text{this}}.\texttt{o2} \rightarrow r + r.\text{All} \rightarrow 0) \end{array}\right) - + r_{\text{this}}.\text{All} \rightarrow 0 \\ + r_{\text{this}}.\texttt{o1} \rightarrow r_2 \\ + r_1.\text{All} \rightarrow 0 \\ + r_{\text{this}}.\texttt{o2} \rightarrow r_{\text{new}} \\ + r_{\text{new}}.\text{All} \rightarrow 0 \\ + r_2.\text{All} \rightarrow 0 \end{array}\right)$$

It is now possible to re-pack the existential permissions, replace them where they had been carved from, and return with no errors.

Chapter 3

Design of Approximating Analysis

The previous chapter explains how we can rewrite pointer annotations in terms of permissions and statically type-check the resulting permission annotations such that errors in checking the permissions correspond directly to disagreement between the stated design intent from the code annotations and the actual behavior of the program. Unfortunately, the type system provided relies on the purely semantic rule for transformation, which allows any safe transformation without providing an algorithm for selecting the transformations actually needed. Any implemented analysis will therefore need an algorithmic transformation operation.

In this chapter, rules describing one such algorithmic type system are described. These rules form an intermediate system, caught between the formal semantics of permissions and the actual implemented behavior of the control-flow analysis checking permissions. The latter is described in Chapter 4.

ATR-Direct
$$\frac{U;\Gamma;\Pi' \gg_{\Pi} \langle \Pi_1 \mid \Gamma';\Pi_2\rangle}{\langle \Pi_1 \mid \Gamma';\Pi_2\rangle \ll_{\Pi} U;\Gamma;\Pi'}$$

ATR-Carve
$$\frac{U;\Gamma';\Pi \gg_{\xi k\to\rho} \langle \xi' k\to\rho' \mid \Gamma'';\Pi'\rangle \qquad \xi' < \xi \qquad \forall k' \cdot \Gamma \models \xi_1 k\to\rho' \prec k' \qquad \langle \xi_2 k'\to\rho_2 \mid \Gamma';\Pi_1\rangle \ll_{k'\to\rho_1} U;\Gamma;\Pi \qquad \xi' + (\xi_1\xi_2) \geq \xi}{\langle \xi k\to\rho' \mid \Gamma'';\Pi_1 + (\xi - (\xi' + \xi_1\xi_2))k\to\rho' + \xi_2(\xi_1 k\to\rho' \multimap k'\to\rho')\rangle \ll_{\xi k\to\rho} U;\Gamma;\Pi}$$

ATR-Carve-Existential
$$\frac{U;\Gamma';\Pi \gg_{\xi k\to\rho} \langle \xi' k\to\rho' \mid \Gamma'';\Pi'\rangle \qquad \xi' < \xi \qquad \forall k' \cdot \Gamma \models \exists r \cdot k\to\rho' + \Gamma_2;\Pi_2 \prec k' \qquad \langle \xi_2 k'\to\rho_2 \mid \Gamma';\Pi_1\rangle \ll_{k'\to\rho_1} U;\Gamma;\Pi \qquad \xi' + (\xi_2) \geq \xi}{\langle \xi k\to\rho' \mid \Gamma'';\Pi_1 + (\xi - (\xi' + \xi_2))k\to\rho' + \xi_2(k\to\rho' \multimap k'\to\rho') + \Pi_2\rangle \ll_{\xi k\to\rho} U;\Gamma;\Pi}$$

Figure 3.1: Algorithmic Type Rules for Carving

3.1 Algorithmic Transformation

Algorithmic transformation of permissions uses several forms of one simple relation. In the relation $U;\Gamma;\Pi_1 \gg_{\Pi} \langle \Pi' \mid \Gamma';\Pi_2\rangle$, we are attempting to produce the permission Π from the permission Π_1, given facts Γ. Two permissions are "returned" as results (of the transformation); the Π' is our current best approximation of the requested permission while $\Gamma';\Pi_2$ contains the facts known after searching for the requested permission and the permissions unrelated to the requested permission. The set U is explained below.

The analysis proceeds over several layers. The outermost layer is presented in Figure 3.1. It uses the relation $AlgFrom\Pi U;\Gamma;\Pi_1\langle \Pi' \mid \Gamma';\Pi_2\rangle$. The reversed order is used to distinguish operations on this level from those on other levels. Here we attempt to find a permission first by running the algorithm from the next layer (Figure 3.2), as shown in the rule ATR-Direct. If the permission cannot be obtained in this manner, we see if it has been nested—whereupon it could be carved

ATR-Fact
$$\frac{\Gamma \models \Gamma'}{U; \Gamma; \Pi \gg_{\Gamma'} \langle \emptyset \mid \Gamma; \Pi \rangle}$$

ATR-Adopt
$$\frac{\Gamma \not\models \Pi_1 \prec \Pi_2 \qquad U; \Gamma; \Pi \gg_{\Pi_1} \langle \Pi_1 \mid \Gamma'; \Pi' \rangle}{U; \Gamma; \Pi \gg_{\Pi_1 \prec \Pi_2} \langle \emptyset \mid \Gamma' \wedge \Pi_1 \prec \Pi_2; \Pi' \rangle}$$

ATR-Nothing
$$\frac{}{U; \Gamma; \Pi \gg_{\emptyset} \langle \emptyset \mid \Gamma; \Pi \rangle}$$

ATR-Base-ND
$$\frac{U; \langle \Gamma; \Pi \mid \Pi \rangle \gg_{k \to \rho} \langle \xi k \to \rho \mid \Gamma'; \Pi' \rangle; S \qquad \langle \xi' k \to \rho \mid \Gamma''; \Pi'' \rangle \in S \cup \{\langle \xi k \to \rho \mid \Gamma'; \Pi' \rangle\} \quad \text{where } \xi' = \max(\{\xi_1 \mid \langle \xi_1 k \to \rho \mid \Gamma''; \Pi'' \rangle \in S \cup \{\langle \xi k \to \rho \mid \Gamma'; \Pi' \rangle\}\})}{U; \Gamma; \Pi \gg_{k \to \rho} \langle \xi' k \to \rho \mid \Gamma''; \Pi'' \rangle}$$

ATR-Plus
$$\frac{U; \Gamma; \Pi \gg_{\Pi_1} \langle \Pi_1 \mid \Gamma'; \Pi' \rangle \qquad U; \Gamma; \Pi' \gg_{\Pi_2} \langle \Pi_2 \mid \Gamma''; \Pi'' \rangle}{U; \Gamma; \Pi \gg_{\Pi_1 + \Pi_2} \langle \Pi_1 + \Pi_2 \mid \Gamma''; \Pi'' \rangle}$$

ATR-IfThen
$$\frac{U; \Gamma \wedge \Gamma'; \Pi \gg_{\Pi_1} \langle \Pi_1 \mid \Gamma_1; \Pi_1' \rangle \qquad U; \Gamma \wedge \neg \Gamma'; \Pi \gg_{\Pi_2} \langle \Pi_2 \mid \Gamma_2; \Pi_2' \rangle}{U; \Gamma; \Pi \gg_{\Gamma' \,?\, \Pi_1 : \Pi_2} \langle \Gamma' \,?\, \Pi_1 : \Pi_2 \mid \Gamma' \,?\, \Gamma_1 : \Gamma_2; \Gamma' \,?\, \Pi_1' : \Pi_2' \rangle}$$

ATR-Existential
$$\frac{\begin{array}{c} U; \Gamma; \Pi \gg_{k \to \rho} \langle k \to \rho' \mid \Gamma_2; \Pi_2 \rangle \\ U; \Gamma_2; \Pi_2 \gg_{[r \mapsto \rho']\Pi'} \langle [r \mapsto \rho']\Pi' \mid \Gamma_2; \Pi_3 \rangle \end{array} \qquad \Gamma_3 \models [r \mapsto \rho']\Gamma'}{U; \Gamma; \Pi \gg_{\exists r \cdot k \to \rho + \Gamma'; \Pi'} \langle \exists r \cdot k \to \rho + \Gamma'; \Pi' \mid \Gamma_3; \Pi_3 \rangle}$$

ATR-Scale
$$\frac{U; \Gamma; \Pi \gg_{\Pi_1} \langle \xi' \Pi_1 \mid \Gamma'; \Pi' \rangle \qquad \xi' \geq \xi}{U; \Gamma; \Pi \gg_{\xi \Pi_1} \langle \xi \Pi_1 \mid \Gamma'; (\xi' - \xi)\Pi_1 + \Pi' \rangle}$$

Figure 3.2: Algorithmic Lookup Rules

ATR-Base-Found

$$\frac{\Gamma \models \rho_1 = \rho_2}{U; \langle \Gamma; \Pi \mid \rho_2.f{\rightarrow}\rho' \rangle \gg_{\rho_1.f\rightarrow\rho} \langle \rho_1.f{\rightarrow}\rho' \mid \emptyset \rangle ; \emptyset}$$

ATR-Base-NotEqual

$$\frac{\Gamma \not\models \rho_1 = \rho_2}{U; \langle \Gamma; \Pi \mid \rho_2.f{\rightarrow}\rho' \rangle \gg_{\rho_1.f\rightarrow\rho} \langle 0\rho_1.f{\rightarrow}\rho \mid \rho_2.f{\rightarrow}\rho' \rangle ; \emptyset}$$

ATR-Base-WrongField

$$\frac{}{U; \langle \Gamma; \Pi \mid \rho_2.g{\rightarrow}\rho' \rangle \gg_{\rho_1.f\rightarrow\rho} \langle 0\rho_1.f{\rightarrow}\rho \mid \Gamma; \rho_2.g{\rightarrow}\rho' \rangle ; \emptyset}$$

ATR-Base-Existential-Match

$$\frac{\Gamma \models \rho_1 = \rho_2 \qquad \rho' \text{ fresh}}{U; \langle \Gamma; \Pi \mid \exists r \cdot \rho_2.f{\rightarrow}r + \Gamma'; \Pi' \rangle \gg_{\rho_1.f\rightarrow\rho} \langle \rho_1.f{\rightarrow}\rho' \mid \Gamma \wedge [r \mapsto \rho']\Gamma'; [r \mapsto \rho']\Pi' \rangle ; \emptyset}$$

ATR-Base-Existential-NotEqual

$$\frac{\Gamma \not\models \rho_1 = \rho_2}{U; \langle \Gamma; \Pi \mid \exists r \cdot \rho_2.f{\rightarrow}r + \Gamma'; \Pi' \rangle \gg_{\rho_1.f\rightarrow\rho} \langle 0\rho_1.f{\rightarrow}\rho \mid \Gamma; \exists r \cdot \rho_2.f{\rightarrow}r + \Gamma'; \Pi' \rangle; \emptyset}$$

ATR-Base-Existential-WrongField

$$\frac{}{U; \langle \Gamma; \Pi \mid \exists r \cdot \rho_2.g{\rightarrow}r + \Gamma'; \Pi' \rangle \gg_{\rho_1.f\rightarrow\rho} \langle 0\rho_1.f{\rightarrow}\rho \mid \Gamma; \exists r \cdot \rho_2.g{\rightarrow}r + \Gamma'; \Pi' \rangle; \emptyset}$$

Figure 3.3: Comparing Base Permissions

$$\text{ATR-Base-Empty}\quad \frac{}{U; \langle \Gamma; \Pi \mid \emptyset \rangle \gg_{k\to\rho} \langle 0k\to\rho \mid \Gamma; \emptyset \rangle ; \emptyset}$$

$$\text{ATR-Base-Plus}\quad \frac{U; \langle \Gamma; \Pi \mid \Pi_1 \rangle \gg_{k\to\rho} \left\langle \xi' k\to\rho_1 \mid \Gamma_1; \Pi_1' \right\rangle ; S_1 \qquad U; \langle \Gamma; \Pi \mid \Pi_2 \rangle \gg_{k\to\rho} \left\langle \xi'' k\to\rho_2 \mid \Gamma_2; \Pi_2' \right\rangle ; S_2}{U; \langle \Gamma; \Pi \mid \Pi_1 + \Pi_2 \rangle \gg_{k\to\rho} \left\langle (\xi' + \xi'')k\to\rho \mid \Gamma_1 \wedge \Gamma_2 \wedge \rho_1 = \rho_2; \Pi_1' + \Pi_2' \right\rangle ; S_1 \cup S_2}$$

$$\text{ATR-Base-Scale}\quad \frac{U; \left\langle \Gamma; \Pi \mid \Pi' \right\rangle \gg_{k\to\rho} \left\langle \xi' k\to\rho \mid \Gamma'; \Pi'' \right\rangle ; S}{U; \left\langle \Gamma; \Pi \mid \xi\Pi' \right\rangle \gg_{\xi k\to\rho} \left\langle \xi\xi' k\to\rho \mid \Gamma'; \xi\Pi'' \right\rangle ; \xi S}$$

$$\text{ATR-Base-Cond-True}\quad \frac{\Gamma \models \Gamma' \qquad U; \langle \Gamma, \Pi \mid \Pi_1 \rangle \gg_{k\to\rho} \left\langle \xi' k\to\rho \mid \Gamma''; \Pi' \right\rangle ; S}{U; \left\langle \Gamma; \Pi \mid \Gamma' \,?\, \Pi_1 : \Pi_2 \right\rangle \gg_{k\to\rho} \left\langle \xi' k\to\rho \mid \Gamma''; \Pi' \right\rangle ; S}$$

$$\text{ATR-Base-Cond-False}\quad \frac{\Gamma \models \neg\Gamma' \qquad U; \langle \Gamma, \Pi \mid \Pi_2 \rangle \gg_{k\to\rho} \left\langle \xi' k\to\rho \mid \Gamma''; \Pi' \right\rangle ; S}{U; \left\langle \Gamma; \Pi \mid \Gamma' \,?\, \Pi_1 : \Pi_2 \right\rangle \gg_{k\to\rho} \left\langle \xi' k\to\rho \mid \Gamma''; \Pi' \right\rangle ; S}$$

$$\text{ATR-Base-Cond-Unknown-Match}\quad \frac{\Gamma \not\models \Gamma' \qquad \Gamma \not\models \neg\Gamma' \qquad U; \langle \Gamma, \Pi \mid \Pi_2 \rangle \gg_{k\to\rho} \left\langle \xi_2 k\to\rho_2 \mid \Gamma_2; \Pi_2' \right\rangle ; S_2 \qquad U; \langle \Gamma, \Pi \mid \Pi_1 \rangle \gg_{k\to\rho} \left\langle \xi_1 k\to\rho_1 \mid \Gamma_1; \Pi_1' \right\rangle ; S_1 \qquad \Gamma \models \rho_1 = \rho_2}{U; \left\langle \Gamma; \Pi \mid \Gamma' \,?\, \Pi_1 : \Pi_2 \right\rangle \gg_{k\to\rho} \left\langle \min(\xi_1, \xi_2)k\to\rho_1 \mid \Gamma' \,?\, \Gamma_1 : \Gamma_2; \Gamma' \,?\, \Pi_1' : \Pi_2' \right\rangle ; \emptyset}$$

$$\text{ATR-Base-Cond-Unknown-NoMatch}\quad \frac{\Gamma \not\models \Gamma' \qquad \Gamma \not\models \neg\Gamma' \qquad U; \langle \Gamma, \Pi \mid \Pi_2 \rangle \gg_{k\to\rho} \left\langle \xi_2 k\to\rho_2 \mid \Gamma_2; \Pi_2' \right\rangle ; S_2 \qquad U; \langle \Gamma, \Pi \mid \Pi_1 \rangle \gg_{k\to\rho} \left\langle \xi_1 k\to\rho_1 \mid \Gamma_1; \Pi_1' \right\rangle ; S_1 \qquad \Gamma \not\models \rho_1 = \rho_2}{U; \left\langle \Gamma; \Pi \mid \Gamma' \,?\, \Pi_1 : \Pi_2 \right\rangle \gg_{k\to\rho} \left\langle 0k\to\rho' \mid \Gamma; \Gamma' \,?\, \Pi_1 : \Pi_2 \right\rangle ; \emptyset}$$

$$\text{ATR-Base-Hypothesize-Match}\quad \frac{\Pi_1' \neq 0\Pi_1 \qquad \begin{array}{c} U \cup \{\Pi_1\}; \Gamma; \Pi \gg_{\Pi_1} \left\langle \Pi_1' \mid \Gamma_1; \Pi' \right\rangle ; S \\ U \cup \{\Pi_1\}; \left\langle \Gamma_1; \Pi' \mid \Pi_2 \right\rangle \gg_{k\to\rho} \left\langle \xi' k\to\rho \mid \Gamma_2; \Pi_2' \right\rangle ; S' \end{array} \qquad \Pi_1 \notin U}{U; \langle \Gamma; \Pi \mid \Pi_1 \multimap \Pi_2 \rangle \gg_{k\to\rho} \langle 0k\to\rho \mid \Gamma; \Pi_1 \multimap \Pi_2 \rangle ; \left\{ \left\langle \xi' k\to\rho \mid \Gamma_2 \Pi_2' \right\rangle \right\} \cup S'}$$

$$\text{ATR-Base-Hypothesize-NoMatch}\quad \frac{U \cup \{\Pi_1\}; \Gamma; \Pi \gg_{\Pi_1} \left\langle 0\Pi_1 \mid \Pi' \right\rangle ; S \qquad \Pi_1 \notin U}{U; \langle \Gamma; \Pi \mid \Pi_1 \multimap \Pi_2 \rangle \gg_{k\to\rho} \langle 0k\to\rho \mid \Gamma; \Pi_1 \multimap \Pi_2 \rangle ; \emptyset}$$

$$\text{ATR-Base-HypothesizeAlready}\quad \frac{\Pi_1 \in U}{U; \langle \Gamma; \Pi \mid \Pi_1 \multimap \Pi_2 \rangle \gg_{k\to\rho} \langle 0k\to\rho \mid \Gamma; \Pi_1 \multimap \Pi_2 \rangle ; \emptyset}$$

Figure 3.4: Finding Base Permissions

$$\textsc{ATR-Impl-Found}\quad \frac{}{U; \Gamma; \Pi_1 \multimap \Pi_2 \ggg_{\Pi_1 \multimap \Pi'} \langle \Pi_1 \multimap \Pi' \mid \emptyset \rangle}$$

$$\textsc{ATR-Impl-Plus-Left}\quad \frac{U; \Gamma; \Pi \ggg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_2 \mid \Pi'' \rangle}{U; \Gamma; \Pi + \Pi' \ggg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_2 \mid \Pi'' + \Pi' \rangle}$$

$$\textsc{ATR-Impl-Plus-Right}\quad \frac{U; \Gamma; \Pi \ggg_{\Pi_1 \multimap \Pi_2} \langle \emptyset \mid \Pi \rangle \qquad U; \Gamma; \Pi' \ggg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_2 \mid \Pi'' \rangle}{U; \Gamma; \Pi + \Pi' \ggg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_2 \mid \Pi + \Pi'' \rangle}$$

$$\textsc{ATR-Impl-Skip}\quad \frac{\Pi \neq \Pi' + \Pi'' \qquad \Pi \neq \Pi_1 \multimap \Pi'''}{U; \Gamma; \Pi \ggg_{\Pi_1 \multimap \Pi_2} \langle \emptyset \mid \Pi \rangle} \qquad \textsc{ATR-Impl-Match}\quad \frac{U; \Gamma; \Pi \ggg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_2 \mid \Pi'' \rangle}{U; \Gamma; \Pi \gg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_2 \mid \Pi'' \rangle}$$

$$\textsc{ATR-Impl-Match-withCarve}\quad \frac{U; \Gamma; \Pi \ggg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_3 \mid \Pi'' \rangle \qquad U; \Gamma; \Pi \gg_{\Pi_3 \multimap \Pi_2} \langle \Pi_3 \multimap \Pi_2 \mid \Pi'' \rangle}{U; \Gamma; \Pi \gg_{\Pi_1 \multimap \Pi_2} \langle \Pi_1 \multimap \Pi_2 \mid \Pi'' \rangle}$$

$$\textsc{ATR-Impl-NoMatch}\quad \frac{U; \Gamma; \Pi \ggg_{\Pi_1 \multimap \Pi_2} \langle \emptyset \mid \Pi'' \rangle}{U; \Gamma; \Pi \gg_{\Pi_1 \multimap \Pi_2} \langle \emptyset \mid \Pi'' \rangle}$$

Figure 3.5: Pass-through rules for linear implications

out. The rules ATR-Carve and ATR-Carve-Existential provide for carving out data groups and field permissions respectively. We assume that only data groups or field permissions would be carved out, and then only from a data group. These are the sole nestings required by the annotation system; while the formal semantics allow for arbitrary nesting, only these will be actually required/useful. These rules would be non-algorithmic were we to arbitrarily select an adoption. Thus the universal quantification across all adoption facts. In practice, of course, we need only test adoption facts until we find one that 'works,' that is, which gives us sufficient permission.

The rules in Figure 3.2 describe a recursion on the structure of the requested permission. Rather than attempt to immediately find a compound permission, we find low-level components (generally base permissions) and assemble the compound. Thus ATR-Plus searches separately for the two permissions being combined and returns the joint permission. ATR-IfThen attempts to find both branches (true and false) and forms the appropriate permission. The rule ATR-Existential returns an existential if it can find the instantiated permissions, while ATR-Scale searches for *at least* the required permission, then trims it to fit.

The other three rules deal with what to do after recursing to a permission which is not compound. For facts, ATR-Fact 'finds' and returns the fact if it can be proven from the facts currently known. Additionally, ATR-Adopt allows spontaneous adoption, provided the adopted permission may be found. An empty permission is trivially located. The interesting case is the base permission; to find a base permission we switch over to the rules in Figure 3.4.

To search a base permission ($k \rightarrow \rho$), we recurse on the structure of the permission we are searching for the requested permission. The relation, from Figure 3.4, is somewhat different; it is now written: $U; \langle \Gamma; \Pi \mid \Pi_1 \rangle \gg_{k \rightarrow \rho} \langle \xi k \rightarrow \rho' \mid \Gamma'; \Pi_2 \rangle$ Most

parts of the relation have the same meaning as before; the change of the first part of the returned permission to $\xi k \rightarrow \rho'$ follows as only a (fraction of a) base permission serves as an approximation to a base permission. The value of the reference and the returned fraction are approximate because we do not know where the field points prior to finding the appropriate permission. Thus, the requested permission points to a free location variable.

The major additions to the relation itself when searching a base permission are the set S which is also returned, and splitting the permission being searched into two parts ($\langle \Gamma; \Pi \mid \Pi_1 \rangle$). The first part is a preserved copy of the original permission being searched; the second is the actual permission to which we have currently recursed. Both these changes and the set U are used to handle linear modus ponens. The idea is that rather than guess (spontaneously or heuristically or what-have-you) whether we need to apply linear modus ponens to get the permission we seek, we try to find the permission both ways; finding the antecedant from the full permission and storing the result in our set S of possible output pairs. Each element in S is then the result of applying some particular combination of linear implications. If two such implications have mutually exclusive antecedents (for example, both require the same permission with fraction 1), they are tried in parrallel as separate elements of S. The set U meanwhile tracks which antecedents we have already found, to prevent infinite regress.

Most of the rules are straightforward recursion on the structure of the searched permission. The first three rules form the base case of the recursion, testing a base permission to see if it is a match (ATR-Base-Found) or if it has the wrong key. The keys won't match if it is a different host object (ATR-Base-NotEqual) or a different field (ATR-Base-WrongField). As with base permissions, existential permissions break down into three cases: the enclosed permission matches the

permission we are searching for, and we instantiate the existential to get it (ATR-Base-Existential-Match), the enclosed permission does not match because it has the wrong field name (ATR-Base-Existential-WrongField), or it does not match because the host object is different (ATR-Base-Existential-NotEqual). We need not search the additional permissions enclosed in the existential with the base permission; the only permissions inside will be relative to the existential variable and so cannot match the permission we are trying to find. In the event that we are trying to access fields of the object pointed-to in the existential, the existential will already be unpacked to access the object itself, as the receiver.

Recursing to the empty permission (ATR-Base-Empty) results in no permission being returned. The rules ATR-Base-Plus and ATR-Base-Scale recursively search over permission combination and permission scaling, respectively. Conditional permissions can easily be recursed into if we can determine whether the conditional is true (ATR-Base-Cond-True or ATR-Base-Cond-False). If we cannot determine the truth of the precondition (ATR-Base-Cond-Unknown), we can in theory derive the maximum permission that is on both sides; a empty permission will always be safe. (In practice when conditionals are created, any permission that could be on both sides is kept aside initially. Thus the empty permission suffices.)

The ATR-Base-Hypothesize- rules handle linear modus ponens for linear implications. If it is possible to use the *full* set of permissions to derive the antecedent of a linear implication, the rule ATR-Base-Hypothesize-Match performs the modus ponens and stores the result of this operation separately, in the set S, while adding nothing to the current result. This operation is skipped if the antecedant cannot be found (ATR-Base-Hypothesize-NoMatch) or if we have already tested filling it (ATR-Base-HypothesizeAlready). The list of already hypothesized linear implications is tracked using U. All possible combinations of filled linear implications

will be tried; and ultimately stored in S—except the case of not filling any, which will be the "normal" return value. Unfortunately, this approach leads to a minor moment of non-algorithmic tension when selecting which of several valid possible outputs to use in ATR-Base. Any actual selection mechanism is allowable here as long as it maximizes the fraction of the sought-after base permission. For simplicity, we will select the first occurrence of the maximum output fraction using some arbitrary ordering of the set (as with, say, a Java iterator). (In practice this corresponds to testing combinations of implications until one reveals sufficient permission. The choice often will be simpler yet because the consequent of most linear implications either leads to the sought permission entirely or not at all. Thus, only filling that particular implication is sufficient.)

When no permission is returned, it is represented an the base permission with a fraction of 0. This vastly simplifies the rules combining fractional parts of the returned permission, but requires extending the definition of fractions to include 0 and syntactic polynomials.

$\xi ::=$	*fraction:*
q	*literal* $(0 < q \leq 1)$
z	*fraction variable*
0	*no permission*
$\xi + \xi$	*sum*
$\xi \times \xi$	*product*

These polynomial fractions are used here in the formal rules but not in the actual implementation. Section 3.2 discusses how we approximate them using abstract fractions. We also alter permission syntax by extracting all facts that are in the permission and keeping them separately, as the pair $\Gamma; \Pi$ instead of embedding the Γ in the

permission. This is purely a move toward economy; this extraction could as easily be accomplished on demand.

Permissions constructed with a linear implication form a special case. The only time a permission including a linear implication will be demanded is when attempting to match a *from* annotation. In other cases, implications may be created as byproducts of carving, but not as part of the permission actively being sought. Even with *from*, the permission being carved is also explicitly demanded; searching for this first may also produce the implication as a byproduct. However, in some cases, the linear implication in the *from* does not correspond to a by-the-numbers adoption and carve. Here, we can be smarter in choosing which permissions to demand, and perform a 'safe' transformation into the form required by the *from* offstage; that is, as part of checking the *from*, rather than part of the transformation rules.

However, the carving occasionally predates the *from*. This happens often when the carved item is being passed through from another function. (A common example is an object returning an iterator over an owned container as an iterator over the owner.) Here, then, we need rules to find a linear implication which already exists within our known permission. This is provided for using the rules in Figure 3.5. The rule ATR-Impl-Found represents finding a matching antecedant, where matching implies that the same permission has been carved from it. The two ATR-Impl-Plus- rules perform a short-circuited evaluation across combining operators. Any other permission is ignored, as indicated in the rule ATR-Impl-Skip. In the end, the 'returned' permission must meet one of three cases.

1. No implication is found (ATR-Impl-NoMatch). Then, the empty permission is returned.

2. The exact implication is found (ATR-Impl-Match). This permission is then

returned.

3. A implication in which the same permission has been carved from somewhere else is found (ATR-Impl-Match-withCarve). Here, we assume the permission is carved indirectly; this corresponds to the case described above, where we return an iterator to an owned list as our own. Permission for the iterator is carved *`from`* the list, permission for the list is carved from an effect on the owner. Upon finding the linear implication carving the iterator from the list, we can seek out the permission carving the list from its owner, and eventually combine them. This will fail if the extra carving occurs for other reasons.

Here, though, we are treating linear implications as a unit; no rule exists that performs a carving solely to produce the appropriate implication.

3.1.1 Soundness

The algorithmic type rules are sound with respect to transformation. That is, they will produce no permission which could not also be produced by transformation. We can see this if we look at each rule using some particular fractional heap: at no point does a rule 'return' permissions that are incompatible with a heap with which the searched permission is compatible.

Lemma 3.1.1 *Suppose we have permissions* Π *and* Π'*, facts* Γ*, memory* μ*, fractional heap* $h \leq \mu$*, obligations* Ψ*, nesting assumptions* N *(*$\Gamma \models N$*) and a substitution* σ *such that* $h; \Psi \models^{\emptyset}_{N} \sigma(\Pi + \Gamma)$*. Then, if*

- $\langle \Pi_1 \mid \Gamma'; \Pi_2 \rangle \ll_{\Pi'} U; \Gamma; \Pi$*,*
- $U; \Gamma; \Pi \gg_{\Pi'} \langle \Pi_1 \mid \Gamma'; \Pi_2 \rangle$*,*
- $U; \langle \Gamma; \Pi \mid \Pi'' \rangle \gg_{\Pi'} \langle \Pi_1 \mid \Gamma'; \Pi_2 \rangle ; S$ *(where* $\Gamma + \Pi \rightsquigarrow \Pi''$*), or*

- $U; \Gamma; \Pi \ggg_{\Pi'} \langle \Pi_1 \mid \Gamma'; \Pi_2 \rangle$

there exist $h' \leq h$ *and* $\sigma' \supseteq \sigma$ *such that* $h'; \Psi \models^{\emptyset}_{N} \sigma'(\Pi_1 + \Pi_2 + \Gamma)$.

PROOF Straightforward structural induction. □

Once each rule maintains compatibility within an individual heap, it is relatively straightforward to extend compatibility across all possible heaps (and thus transformation in general).

Theorem 3.1.2 *Suppose we have permissions* Π *and* Π' *and facts* Γ*; then, if there exist permissions* Π_1*,* Π_2*, and facts* Γ' *such that* $\langle \Pi_1 \mid \Gamma'; \Pi_2 \rangle \ll_{\Pi'} \emptyset; \Gamma; \Pi$*,*

1. $\Pi + \Gamma \rightsquigarrow \Pi_1 + \Pi_2 + \Gamma'$*, and*
2. $\exists \sigma$ *such that* $\Pi_1 = \sigma(\Pi')$.

PROOF The first conclusion follows immediately from Lemma 3.1.1. The second follows from straightforward structural induction. □

The second point is merely an assurance that the demand mechanics work as advertised; searching for a particular permission will return a similar permission (or nothing).

3.1.2 Completeness

This algorithm for permission type checking is not complete. Figure 2.14 lists a simple counterexample to its completeness; the method `evil` can be type-checked using standard typing (as described in Section 2.4.6), but cannot be checked using the algorithmic rules provided in this section. The key difference is that the algorithm

here attempts to fill the carved-out permissions for the linear implication and only then split off the fractional read permission for the call to `test`. Successful permission-checking of the method requires using identical fractional splits for both the linear implications and existential permissions needed as their antecedents.

In practice, however, this particular form of example is not common. As indicated in Chapter 5, for most common programming idioms using these annotations, the direct approach embodied in these algorithmic type rules suffices. This follows largely because the permissions generated by annotations are highly restricted in form. The transformations needed for checking them likewise tend to form a fairly restricted set. The algorithmic type rules presented here were selected in part because they encompass the vast majority of the transformations necessary, for all that a few rare corner cases remain. When testing the analysis implementation, we found no examples in which the incompleteness was a factor.

3.2 Abstracting Fractions

How often do we need the exact value of a fraction? When checking for read permission, the exact value does not matter, just that it is non-zero. When passing a permission for a read effect on a method, the exact value passed does not matter. When returning a permission from a method, the exact value matters only in as far as whether it is the same fraction as was passed in for the effect; what the numerical value of the fraction is is immaterial. Thus the only place where the numerical value of fractions is central is when forming those annotations which mention an explicit number.

Figure 3.6 defines an abstract representation for fractions. Figures 3.7 and 3.8 describe the connection between fraction polynomials and these abstract fractions.

$\Xi ::=$	*abstract fraction:*
0	*no permission*
s	*smaller read*
r	*read*
w	*write*
E	*error*

Figure 3.6: Syntax of Abstract Fractions

The abstraction is reasonable because we generally do not need exact fraction values. Thus, it is enough to know that we have some read permission—the precise fraction is immaterial. This in some ways resembles the handling of read and write capabilities in the calculus of capabilities [CWM].

A major addition to that theory, however, is the inclusion of the *smaller* permission. "Smaller" read are required to help abstract our ability to split permissions; they can best be understood in relation to ordinary read permissions. The primary genesis is from effects. With effects, we care that the same fraction is returned from a method as was passed into the method initially. A "read" abstract fraction means that the fraction is of exactly the value passed into the method (or carved out of a permission that was actually passed in). A "smaller" abstract fraction indicates that there still is some read permission, but not as much as was initially passed in.

The fraction passed in will differ on exit only when it is split somewhere in the method. Checking use when accessing fields would never require us to actually split a permission. This has no effect on permissions passed as effects to method calls because we *know* that the returned effect will be the same; thus we can split off the fraction, pass it to the method, return it from the method, and add it back together offstage. As long as we can infer when it is possible to safely perform these operations, we do not need to actually carry them out.

$$\frac{q < 1}{q \triangleright r} \qquad \frac{q = 1}{q \triangleright w} \qquad z \triangleright r \qquad 0 \triangleright 0 \qquad \frac{\xi_1 \triangleright \Xi_1 \qquad \xi_2 \triangleright \Xi_2}{\xi_1 + \xi_2 \triangleright \Xi_1 \hat{+} \Xi_2} \qquad \frac{\xi_1 \triangleright \Xi_1 \qquad \xi_2 \triangleright \Xi_2}{\xi_1 \times \xi_2 \triangleright \Xi_1 \hat{\times} \Xi_2}$$

Figure 3.7: Rules for Abstracting Fractions

A read (or write) permission is noticeably split when it is passed into a method *and not returned*. Given that the permissions passed into and out of a method call are determined by the annotations on the method, we can identify the cases where this occurs.

1. An actual parameter is passed as immutable which had not previously been immutable. This forces the permanent adoption of a fraction of the permission into the `Immutable` data group. It is now impossible to return the same fractional permission for the effect. This may still type check if the permission for the field mentioned in the effect is restored by some other means (say by assigning a null value or a different object).

2. An actual parameter is passed as `read-only` or `unique-write`, which was not previously read-only. This is essentially the same case as before, only involving a different data group.

3. The return value is `readonly-from` a passed-in effect. In this case, we can split the read effect, as before. Here, after it is returned from the method, it cannot be immediately merged again, because some other permission has been carved out of it. While this carving endures, we cannot restore the original read permission.

In each of these cases we now have a smaller permission than previously, and mark this with the smaller read abstract fraction.

$\hat{+}$	0	s	r	w	E
0	0	s	r	w	E
s	s	s	r	E	E
r	r	r	r	E	E
w	w	E	E	E	E
E	E	E	E	E	E

$\hat{\times}$	0	s	r	w	E
0	0	0	0	0	0
s	0	s	s	s	E
r	0	s	s	r	E
w	0	s	r	w	E
E	0	E	E	E	E

Figure 3.8: Adding and Scaling Abstract Fractions

3.2.1 Lack of Recovery

Using abstract fractions in place of actual fraction values simplifies permission analysis immensely, as one can now analyze a method without mathematical computations. (Using full fractions with the particular algorithmic analysis detailed above is even more problematic as it entails symbolically solving polynomials over $\mathbb{Q}$.) However, they represent a step backwards in representational power.

The major advantage given by using fractions to represent read (vs. write) permissions is that the sole write permission can be divided into several read permissions, *which can merged back into the write permission* as long as all are present. We cannot do this directly using the abstract fractions. While it is still safe to divide an abstract write permission into any number of abstract read permissions, no number of abstract read permissions can ever be combined to get the abstract write back. Permission accounting fails when no one keeps the books in sufficient detail.

The inability to re-combine read permissions to restore the write permission does not destroy the ability to analyze programs featuring annotations with permission semantics. This is partially true because the vast majority of the time, splitting a fraction occurs either in a situation, such as passing a fraction of a field permission as an effect, in which the permission can be restored 'immediately' and always (the effect always returns the same fraction), or in situations, such as passing a permission into, but not out of a function, where the permission will never be made whole

regardless. In the first case, where we 'know' the fractions will add up, we can perform the addition without checking the math. In the second case, the exact fraction is immaterial: it is only necessary to enable the restoration of the original write permission. If we are not restoring the original write permission, we need not know the exact fraction.

The problematic cases, then, are those where a permission is split for a noticeable duration (several statements in the program), and then restored. One such example is a *readonly-from* iterator. Suppose we have a `List` class with the method

```
reads this.All
readonly-from(this.All) Iterator iterator();
```

This method has (highly approximated) permission annotation

$$\forall r_{\text{this}}, z; z r_{\text{this}}.\text{All}{\to}0 \longrightarrow \exists r_{\text{ret}}; r_{\text{ret}}.\text{All}{\to}0 + r_{\text{ret}}.\text{All}{\to}0 \multimap z r_{\text{this}}.\text{All}{\to}0 + v$$

Thus, if we call it while possessing a full write permission to the list ($r_{\text{list}}.\text{All}{\to}0$), this permission is split prior to the call, and after the call we have

$$\frac{1}{2} r_{\text{list}}.\text{All}{\to}0 + r_{\text{iter}}.\text{All}{\to}0 + r_{\text{iter}}.\text{All}{\to}0 \multimap \frac{1}{2} r_{\text{list}}.\text{All}{\to}0 + v$$

When the iterator finishes, the linear implication can be satisfied, and the two halves of the list permission rejoined. However, using abstract fractions, the result is

$$r r_{\text{list}}.\text{All}{\to}0 + r_{\text{iter}}.\text{All}{\to}0 + r_{\text{iter}}.\text{All}{\to}0 \multimap r r_{\text{list}}.\text{All}{\to}0 + v$$

The disadvantage here, of course, is that even after the iterator permission is restored, the write permission for the list cannot be restored.

This particular situation can be resolved, however, by extending the linear implication for the ***from*** to handle the permission math. If the resulting permission (after the call to `iterator`) is

$$rr_{\text{list}}.\text{All}{\rightarrow}0 + r_{\text{iter}}.\text{All}{\rightarrow}0 + (r_{\text{iter}}.\text{All}{\rightarrow}0 + rr_{\text{list}}.\text{All}{\rightarrow}0) \multimap wr_{\text{list}}.\text{All}{\rightarrow}0 + v$$

the write permission can again be recreated when the iterator is complete. This change is easily implemented as an adaptation of the procedure for checking ***from*** annotations. I am unaware of other situations which similarly require fractional parts of a write permission to be separated for some duration within the same procedure; any other cases, however, will be amenable to a similar solution.

Chapter 4

Implementation Issues

A type checker using the algorithmic transformation rules of the previous section is not how the permission checker is implemented. The actual implementation is as a control-flow analysis in the Fluid analysis framework. This necessarily results in both a different representation for permissions and a different algorithm for checking that they are used correctly. Some of these differences are due to the requirements of a flow analysis, some to the particular requirements of both flow analyses and result reporting imposed within Fluid, and some are idiosyncratic design decisions intended to simplify the coding and/or execution of the analysis.

4.1 Flow analysis

Following Nielson, Neilson, and Hankin [NNH99b],

> a *Monotone Framework* [for intraprocedural data flow analysis] consists of:
>
> - a complete lattice, L, that satisfies the Ascending Chain Condition, and we write $\sqcup$ for the least upper bound operator; and

- a set F of monotone functions from L to L that contains the identity function and that is closed under function composition.

In addition, we need a representation of the control flow unit which we are analyzing (e.g. a Control-Flow Graph), an initial lattice value for analysis entries, and a means of associating transfer functions with particular transitions in the flow unit (edges in the CFG).

The infrastructure provided by Fluid simplifies this task; to specify a flow analysis, one must define the lattice representation and all non-identity transfer functions. The lattice must be functional: the analysis cannot mutate a lattice object. Methods are defined to represent transfer functions over a standardized set of control flow operations representing the Java programming language. (They correspond to lower-level operations than the source, generally mimicking stepwise execution.) These may be overridden to perform non-identitity transfer functions; the choice of overridden method associates the transfer function with the appropriate edges.

4.1.1 Lattice Structure for Permission Analysis

The control flow analysis for permission analysis builds its lattice structure by aggregating several smaller, special-purpose lattices. At the bottom level, there are two basic lattices; one for locations (location variables) and one for abstract fractions. The lattice for abstract fractions is a straightforward chain lattice; the lattice features a finite enumeration of values in a total order

$$u \sqsubseteq w \sqsubseteq r \sqsubseteq s$$

where the lattice value u represents 0. The lattice element u functions as the bottom element of the lattice, while s is $\top$.

The other basic building block of the lattice structure is the `SimpleLocation`. These lattice objects represent locations in memory, or sets of locations in memory. There are three dedicated values: the null location and top and bottom values for the lattice. Other values can be created for any expression. In practice, this is used to assign incoming parameters, field values, method returns, and other values as they 'enter' the method under analysis for the first time. Lattice values are also lazily created when an otherwise undefined join operation is performed. In theory, this lattice is unbounded; however, for any analyzed method, some finite number of lattice elements will be created.

One way other lattice structures are built atop these building block is with a map lattice, which represents an infinite mapping of all possible objects to some lattice. The join operation is defined as $(f \sqcup g)(x) = f(x) \sqcup g(x)$. More realistically, we look at only the sub-lattice where all but a finite number of objects map to either the top or bottom element of the lattice (but not a mixture of the two). This enables us to track only 'interesting' entries. In particular base permissions are implemented as mappings; the base permission $\xi k \rightarrow \rho$ is represented as entries in two distinct mappings. The first maps the key to the `SimpleLocation` representing the pointed-to location, the second maps the key to the appropriate abstract fraction. Thus a pair of map lattices represent the collection of all base permissions.

Similarly, linear implications can be represented as a mapping from the consequent to the set of all keys for permissions carved from it. Here, the set lattice needs to be defined with the join operation representing set union. Thus, if a permission is carved out in only one branch, it remains carved out after joining the branch with one that did not perform that carving.

One consequence of simulating Java evaluation at a low level is that the transfer functions need to simulate stack operations. This requires the use of a `StackLattice`;

a lattice comprised of a stack of elements of some other lattice. The join operation is only defined between stacks of the same height, which join corresponding elements. (In formal theory, joining stacks of unequal height should result in the top element $\top$ of the `StackLattice`. In practice, a well-formed analysis will never do this, so we throw an exception.) The elements of the stack, being evaluated terms, are `SimpleLocation`s.

The last part of the lattice is the collection of known facts. Facts have three varieties: that two locations are equal, that two locations are not equal, or that some key is nested in another key. (In the formal system, the permission type of the nested key needs to be maintained as part of the nesting fact. This is not necessary here because the permission type of the field is determined by its static type, and field binding information is maintained separately by Fluid.) These facts are kept in a set representing a conjunction. Here the join operation is intersection. This allows for facts to be lost on merging; thus for an `if`, the condition can be kept as a fact in the true branch, its negation as a fact in the false branch, and both are dropped when the branches join.

However, the lattice is made more complex to allow for greater precision. In addition to the main conjunction, there is a collection (disjunction) of other sets (conjunctions) of facts or-ed together. The result resembles $\bigwedge \{f\} \wedge (\bigvee \{(\bigwedge \{f\})\})$. This structure enables the maintenance of additional information when joining lattices together—at the expense of a more complex join function for the full lattice. The join operation must first examine what happens when the stack and the location map join; in particular, we note which locations change as a result of the join and use these to generate two sets of 'substitutions,' one for the left hand operand of the join, and one for the right. These two are added as extra disjunctive equality facts in all possible cases. For a vastly simplified example, consider a lattice with just a location

map and a set of facts:

$$L_1.\mathrm{f}{\rightarrow}L_2 \circ \{L_1.\mathrm{f} \prec L_1.\mathrm{All}\} \sqcup L_1.\mathrm{f}{\rightarrow}L_3 \circ \{L_1.\mathrm{f} \prec L_1.\mathrm{All}\}$$
$$= L_1.\mathrm{f}{\rightarrow}L_4 \circ \{L_1.\mathrm{f} \prec L_1.\mathrm{All} \wedge \{\{L_2 = L_4\} \vee \{L_3 = L_4\}\}\}$$

This has the advantage of greater precision as we know that L_4 must be either equal to L_3 or L_2. we may now, for instance, more accurately answer some aliasing questions.

More than a complicated join function, the downside of this approach is worse algorithmic efficiency. If we join as follows:

$$L_1.\mathrm{f}{\rightarrow}L_4 \circ \{L_1.\mathrm{f} \prec L_1.\mathrm{All} \wedge \{\{L_2 = L_4\} \vee \{L_3 = L_4\}\}\} \sqcup L_1.\mathrm{f}{\rightarrow}L_5 \circ \{L_1.\mathrm{f} \prec L_1.\mathrm{All}\}$$
$$= L_1.\mathrm{f}{\rightarrow}L_6 \circ \{L_1.\mathrm{f} \prec L_1.\mathrm{All} \wedge \left\{ \begin{array}{l} \{L_2 = L_4 \wedge L_4 = L_6\} \\ \vee \{L_3 = L_4 \wedge L_4 = L_6\} \\ \vee \{L_5 = L_6\} \end{array} \right\}$$

we see that the number of disjoined elements will be linear in the number of merges. After the join, there are as many elements in the disjunction as the on both sides previously (or possibly one more if there had been none previously.) The size of each grows relative to the product of the number of join points on the path it represents and the number of locations that differed at each. In programs with complex control flow, this can grow somewhat large, which affects how quickly we can test the appropriate lattice element to see if we have the permissions required for some operation (described below).

This structure leaves out some of the existing permission forms; notably, we are missing existential closures and conditional permissions. Existential closures are left out by design; the transfer functions are set up such that they are considered to be

```
Node n = head;
while(n != null){
  // do something with node n
  n = n.next;
}
```

Figure 4.1: Standard loop for linked-lists

open (unpacked) always. When one needs to be closed, say to pass as an effect, the result of the packing and the subsequent unpacking after return are interpreted directly, without explicitly representing the intermediate form containing the existential.

Most uses of conditional permissions are subsumed by the generalized join operation (that is, $\Gamma\,?\,\Pi_1 : \Pi_2$ is imprecisely represented as $\Pi_1 \sqcup \Pi_2$). The exception to this approach is when forming *`maybenull`* closures. Here, we first attempt to prove the enclosed location is equal to `null`. If this fails, we attempt to find the appropriate permissions. This choice is hard-coded into the act of forming the existential; as with existential permissions, the conditional permission which results is never explicitly represented.

4.1.2 Loops

One omission in the earlier permission-based type system (Figures 2.92.10) is loops. Permission-typing loops is non-trivial. Even the simple loop in Figure 4.1 will cause difficulties for the step-by-step approach. Assuming the `head` and `next` pointers are unique, we will unpack an existential upon each iteration of the loop, with permission to an additional `Node` for each. The process does not automatically reach a fixed-point.

Recursive functions are easier to formalize than loops because they are annotated. That is, there is an explicit, user-provided statement generalizing the state of the

permissions on each recursive call. The equivalent for a loop is the loop invariant—a predicate which is true on each pass of the loop. In the example, a possible invariant is to existentially qualify `n`:

$$\exists r_n \cdot r_n = 0 \ ? \ \emptyset : r_n.\text{All}{\rightarrow}0$$

Each pass of the loop, `n` will refer to some possibly-null unique node. This is accurate as far as it goes, but will render it impossible to restore the original linked list. Indeed, we need to both indicate that the list as exists is incomplete (some node permissions are carved out) and that restoring these existentially-qualified permissions will restore the list. This requires additional permissions to accomplish (here, r_{this}`.head`$\rightarrow r_h$).

$$\exists r_n \cdot \left(\begin{array}{c} (r_n = 0 \ ? \ \emptyset : r_n.\text{All}{\rightarrow}0) \\ \\ + ((\exists r \cdot r = 0 \ ? \ \emptyset : r.\text{All}{\rightarrow}0) \multimap r_h.\text{All}{\rightarrow}0) \end{array} \right)$$

This approach cannot be followed directly in the control-flow analysis as it lacks both general existential quantification and user-supplied invariants. Existentially-qualified permissions can be handled with renaming, as previously; for the other, the analysis must iteratively reach a fixed-point invariant rather than be given it. To this end, we can rename those variables which are modified in the execution of the loop as part of the loop merge. In the example, the (partial) analysis state after the initial assignment of `n` from `head` would be

$$L_1.\text{head}{\rightarrow}L_2; 0.\text{n}{\rightarrow}L_2$$

$$\circ L_1.\text{head} : w; L_2.\text{All} : w.$$

This makes explicit the separation between the map for locations and permissions

(abstract fractions); each is on its own line ($\circ$ is used as a separator). We also refer to the local variable `n` as a field of the null pointer, rather than its own variable. At the first loop merge, we meet this lattice value with top, which results in the non-top value. At the end of the loop, we have a different lattice value.

$$\begin{aligned} & L_1.\text{head}{\rightarrow}L_2; 0.\text{n}{\rightarrow}L_3; L_2.\text{next}{\rightarrow}L_3 \\ \circ\, & L_1.\text{head} : w; L_2.\text{All} : w(L_3.\text{All}); L_3.\text{All} : w \\ \circ\, & L_2 \neq 0 \end{aligned}$$

The notation $L_2.\text{All} : w(L_3.\text{All})$ indicates that the permission for $L_3.\text{All}$ was carved out of that for $L_2.\text{All}$. In the next pass the control flow analysis joins the two values shown previously.

$$\begin{aligned} & \begin{aligned} & L_1.\text{head}{\rightarrow}L_2; 0.\text{n}{\rightarrow}L_2 \\ \circ\, & L_1.\text{head} : w; L_2.\text{All} : w. \end{aligned} \\ & \quad \sqcup \begin{aligned} & L_1.\text{head}{\rightarrow}L_2; 0.\text{n}{\rightarrow}L_3; L_2.\text{next}{\rightarrow}L_3 \\ \circ\, & L_1.\text{head} : w; L_2.\text{All} : w(L_3.\text{All}); L_3.\text{All} : w \\ \circ\, & L_2 \neq 0 \end{aligned} \\ & \quad = \begin{aligned} & L_1.\text{head}{\rightarrow}L_2; 0.\text{n}{\rightarrow}L_4; L_2.\text{next}{\rightarrow}L_3 \\ \circ\, & L_1.\text{head} : w; L_2.\text{All} : w(L_3.\text{All}); L_3.\text{All} : w \\ \circ\, & L_4 = L_2 \vee L_4 = L_3 \end{aligned} \end{aligned}$$

Because `n` points to two different values, a new location is generated, which must have one or the other of these values. The two sets of permissions entering the join are logically but not syntactically equal; the latter syntax is selected by the join operation as more precise. The control flow analysis will continue iterating until it reaches a fixed point. Because this example is so small, the fixed point is quickly reached; indeed, it is the result of the first join operation.

This lattice value corresponds closely with the desired invariant, with L_4 filling the role of r_n. This is unsettling in that L_4 is treated as a concrete location within the loop and an existential outside. However, within the control flow analysis, these two contexts never collide: L_4 is used for both the abstract and concrete variables, but never both at the same time. It is as if the existential variable L_4 were always being unpacked as the location L_4. Additionally, the issue has yet to present a problem in practice.

4.1.3 Drop-sea

Results in Fluid are not reported as simple messages. Rather, to better coordinate the interactions of diverse assurance results, the results are entered as *drops* in the Drop-Sea truth management system. A Drop is a particular piece of information which can be either true or false. In practice, there are two essential kinds of drops: promise drops, which record those facts resulting from user annotations (e.g. "This field is `unshared`"), and result drops which record those facts arising from assurances (e.g. "This method preserves the unshared character of that field"). A Sea in Drop-Sea is a collection of Drops, which are implicitly added to the sea when they are instantiated.

Drops exhibit several relationships. Promise drops *support* result drops by providing information used to generate that result. More precisely they provide facts to the analysis which the assurance uses to generate the result drops. Similarly, result drops support the promise drops they are assuring. Additionally, a drop may depend on any other drop; this means the fact represented by the former is a consequent of the fact represented by the latter. Negating the latter will negate the former.

Drop-Sea mediates the relations between assurances by making explicit the dependencies between assurances. That is, if one assurance relies on certain promises that

cannot be verified (or worse are provably inconsistent) by a second assurance, this uncertainty is propagated to the results of the first analysis. By using the Sea as the primary instrument to display assurance result, these dependencies are explicated for the user. Drop-Sea adds the burden of connecting assurance results to annotations used and verified to the assurance itself.

4.1.4 UI for Permission Assurance

The interface for the permission assurance, then can be described in terms of which promise drops it supports with which result drops. The general assurance mechanism for the `double-checker` package ties user-selected assurances to the act of building a package; that is, building the package will cause the selected assurances to be run on the package. The assurances are responsible for providing support for the promise drops in the annotated code. The permission assurance can support the following types of promises:

- promises that a field will be ***`unshared`*** or that a parameter, receiver or return value will be ***`unique`***,
- promises that a value has been ***`borrowed`***, and
- promises describing the effects of a method.

Notice that there is no promise that a value is ***`shared`***; this owes to ***`shared`***-ness being the default assumption for values without annotation. This is handled by creating promise drops for shared fields/parameters/receivers/return values that are not associated with any actual annotation, but are tied to the appropriate declaration. Promises for ***`owned`***, ***`readonly`***, ***`immutable`*** and ***`from`*** are not yet supported.

The following drops are generated by the analysis to support various annotations.

- Drops representing reading a field that support the effects of the method containing the read. If a read is attempted without finding read permission for the field, the drop shows up as an error; the effect annotation is not supported. The corresponding messages are "`Read permission for field ??  present`" and "`Read permission for field ??  absent`," where `??` is the name of the field.

- Drops representing writing a field; these behave like reads. The possible messages are "`Write permission for field ??  present`" if the write permission for the field is and "`Write permission for field ??  absent` if it is not."

- Drops for passing permissions to called methods as effects; these behave as the drops for reading or writing in the calling method. The messages are "`Annotated ??  permission for field ??  present`" and "`Annotated ??  permission for field ??  absent`." Here, the first `??` is either read or write, as appropriate.

- Drops for checking that a value really is unique. These are generated at any boundary and involve showing that (for *`maybenull`* unique) the value is either null or the appropriate All permission is present. Failure to satisfy one of these conditions results in an error. The drops support the *`unique`* or *`unshared`* promise for the appropriate value. The messages are "`??  is unique`" and "`Cannot make ??  be unique`." Again, the `??` is the field name.

- Drops for checking that a value is *`shared`*. These behave as those for unique values, except they support the artificial shared drops, as opposed to drops actually attached to annotations. The messages are "`??  is shared`" and "`Cannot make ??  be shared`."

- Drops that always support *`borrowed`*. As borrowed is always correct, each borrowed annotation is given a (always) correct result drop proclaiming that fact. The only possible message is that "`borrowed requires no assurance`."

Next, we discuss how these messages are generated by the control-flow analysis, first in the general case, and then in ways specific to the permission analysis.

4.2 Integrating CFG-Analysis, Drop-Sea

These drops are generated as part of assuring that the program annotations describe its behavior accurately. The control-flow analysis merely uses them to decide what the approximate state of the program is at any given point. For permission analysis, this involves deciding which, if any, permissions are available, carved out, fractional, etc. at any given point of execution. This is, to some extent, separate from deciding whether the permissions which are present indicate that the pointer annotations on the program are correct. Thus, we need to consider how to generate assurance (that is, the result drops described above) from the control-flow analysis. In general, there are several possible approaches to consider.

4.2.1 Post-pass Assurance

One may, of course, separate assurance from analysis. A control-flow analysis can be run to completion over the code, and then the assurance may make a second pass, probably as a tree-walk, querying analysis results on appropriate edges. The assurance would be solely responsible for reporting evidence that the program did or did not meet its specification. There are to my knowledge no assurances in fluid currently using this approach.

One advantage of such an approach is that it can be easier to frame an analysis without worrying about resulting assurances. Also, error reporting from tree walks is simpler; in particular, the CFG analysis will have already iterated to a fixed point before any assertions are made. However, this requires two separate, yet linked, code analyses. Coordinating the information gathered with that needed for reporting results is nontrivial.

4.2.2 Example Revisited

The `NonNull` analysis/`noNPE` assurance pair naturally expresses itself as a post-pass analysis. The CFG analysis can determine a set of local variables which are not null, and, more generally, a general formula for determining whether an expression is not null. Then, an assurance can perform a tree walk in which, if it is not currently within a `try` block which catches `NullPointerException`s, checks all receiver expressions against the analysis to see whether they must be non-null. If a receiver outside such a `try` could be null, this is reported as a negative assurance for the `@noNPE` annotation. That is, first a fixed point of non-nullness information is computed, then a separate tree walk assures that this information supports the annotation (or not). This approach seems reasonable for this assurance because the necessary tests for being in a `try` that catches `NullPointerException`s are easily integrated into a tree walker, but add noticeable complexity to the lattice of non-null local variables.

4.2.3 Poisoned Lattices

It is tempting to consolidate the analysis of the program and the provision of assurance into one pass over the code by performing assertions while iteratively calculating analysis results. This runs the risk of generating *false* results when asserting against

incomplete analysis results. In particular, a premature positive assurance of code is unsound. The easy way to avoid this is to incorporate the assertion results into the lattice computation. Only if the final result lattice is "good" will positive assurance be provided. Any assurance failure will lead to an inescapable "bad" state (a *poisoned* lattice), resulting in negative assurance. `UniqueAnalysis` in fluid uses poisoned lattices.

The advantage of poisoning lattices is that it allows assurance to be performed concurrently with analysis in a semantically sound manner. Unfortunately, because assurance is not determined until after iteration is complete, all the final result tells one is whether the assurance was successful overall or not. If the code does not assure, the nature of the problem is known, but not *exactly* where it occurs. In order to give useful feedback to the user, a separate analysis is needed anyway, to climb the syntax tree and find the point where the analysis went "bad" to report the problem to the user with any precision. For `UniqueAnalysis`, this task is performed by `UniqueAssurance`. Additionally, the analysis is only capable of pinpointing one error per flow path, because once in the bad state, further errors cannot be detected.

4.2.4 Side Effects

These problems would not occur if assertion failures that occurred while analyzing a flow unit reported results as they occur without altering the analysis lattice. For convenience, we will assume that analysis results are simple messages (strings) for now. Similarly, good results can be reported from passed assertions. All results, good and bad, can be cached in a repository for later reporting. The repository can link the message resulting from the assertion with the node being analyzed when the assertion is made to provide useful error messages. `PermissionAnalysis` in fluid attempts to follow this model; it originally used the `AssuranceLogger` as its repository.

The problem with a side-effecting analysis, of course, is that side effects (messages) can be generated when the analysis is in an incomplete state. The `AssuranceLogger` is configured as a map from the `IRNode`s at which assurances are made to the 'good' or 'bad' messages resulting from these assurances. This should be sound but is imprecise. Additionally, it requires that every positive assurance have a dual, negating negative assurance.

A better approach is to only generate side effect *after* a fixed point has been reached, but still from within the analysis. Thus the analysis would consist of two passes: a *work* pass which iterates to a fixed point in the lattice, and a *rework* pass which revisits each node exactly once. Side effects should only be generated when *rework*ing the analysis; after the fixed point is reached. One way to ensure this is to always generate side-effects, but to attach a big switch to the repository of assurance results (side effects). The repository only records results when the switch is on. Analysis results can be compiled by

1. turning the switch off
2. working the analysis to get a fixed point
3. turning the switch on
4. reworking the analysis to get recorded side effects

This procedure is apparently similar to the separate post-pass; however, the assurance results are still being calculated within the analysis, allowing tight coordination. Also, reworking is currently supported by the flow-analysis infrastructure; post-pass assurances are ad-hoc, based on the particulars of the analysis/assurance.

4.2.5 Control-flow Analysis Asea

How can we connect control-flow analysis results to Drop-Sea? If using a post-pass assurance, either explicitly or to decipher poison lattice results, the separate tree-walking assurance makes the appropriate connections; this is equivalent to generating results for a tree-walking analysis and is out of the scope of this discussion. Drops are generated within the control-flow analysis itself only when performing a side-effecting control-flow analysis.

The obvious design is to use the Sea as the side-effect repository. Side effects show up as instantiated result Drops instead of posted messages. Because the Sea lacks a switch to disable reporting until a fixed-point is reached, the switch must be added separately. A *mediator* must be placed between the analysis and the Sea to handle the switching semantics. Drops are not generated within the analysis proper, rather it directs the mediator to produce drops; the mediator is equipped with a switch enabling it to only generate effects when reworking.

This approach suffices for generating isolated result drops; however, for the Sea to be effective, the result drops must be associated with the (drops associated with the) promises they support and the (drops associated with the) promises they rely on for information. The former connections are straightforward to make because assertions are only made, and thus results are only generated, where necessary to support some promise. The result can be immediately known to support the associated promise.

Determining which promises are used to produce results is trickier because the connection is non-local. Promise information is given the analysis at method entry and possibly at other points throughout the control-flow analysis. It is used at points important to the supported promises, which are not in general where the supporting promise information is found. Fortunately, within a version, and an analysis is always run within a single version, promises and promise drops are constant. Therefore, the

promise drops used to generate particular pieces of analysis information may be added to the appropriate lattice values. Merging flows complicate the process; a lattice value that is the result of a merge may contain information from two possible sources. This translates to a union lattice of promise drops which support an associated lattice value.

So what happens at an assertion? The assertion is associated with some promise which it helps support. The assertion is made against the current state of the CFG lattice. Some information in the lattice value representing the analysis' approximation of the execution of the flow unit causes the assertion to (for instance) pass. Both the supported promise drop and the set of supporting promise drops are passed with a message to the mediator instructing it to generate the result drop associated with passing the current assertion.[1] If and only if the mediator is set to report (that is, if the analysis is reworking) it then generates the appropriate result drop and connects it to the passed-in promise drops for view in the reporting framework.

4.2.6 Example Re-Revisited

Indeed, even the simple analysis/assurance combination in Section 4.2.2 runs into this problem with non-locality of information source. In Figure 1.1, the `@noNPE` assurance can generate a result drop when checking that the receiver of the method call is non-null (as our `NonNull` analysis informs us) which supports the `@noNPE` promise drop. But what supports our analysis result? The analysis reaches its conclusion because when control flow follows the **true** branching from the `x != null` test, `x` can be added to the set of local variables know to be non-null. This can be simulated in Drop-Sea by adding a (pseudo?) promise drop associated with the node for the test, and using

[1] In practice, other information needs to be passed in. For example, the `IRNode` at which the assertion is made gets passed in so the result drop can accurately report where in the program the result was generated.

```
@noNPE
void foo(SomeClass x, SomeClass y){
  if(x != null && y != null && x.f != null){
    y.f = null;
    x.f.somefun();
  }
}
```

Figure 4.2: Now with fields!

that drop to source the assurance of the method call. To get the pseudo-promise drop from the part of the AST where it comes into being as part of the CFG analysis to where it is used requires the drop be passed along with `x` in the set of non-null local variables. This makes the CFG lattice something other than (strictly) a set of non-null local variables. In practice, some kind of map lattice mapping all local variables to a set (lattice) of non-null-supporting "promise" drops, with an empty set representing possible null-ness, will likely be necessary.

Unfortunately the simple `NonNull` analysis of Section 4.2.2 runs into trouble when dealing with fields. Consider the code in Figure 4.2. At first, this seems to be a straightforward extension of the first example. However, if `x` and `y` reference the same location the code will in fact throw a `NullPointerException`. The code in Figure 4.2 cannot be assured with the information present in the simple `NonNull` analysis.

There appear to be two obvious ways to extend `NonNull` analysis to include fields. First, a may-alias analysis could be run, either separately or as part of a more complex non-null analysis, to produce slightly less conservative aliasing results. Using a permission analysis as a backbone for a non-null assurance is one example of this. The other extension is the addition of extra annotations (e.g. `@nonNull`) on fields

```
class SomeClass{

  @nonNull SomeClass f;

  . . .

  @noNPE
  void bar(@nonNull SomeClass x, @nonNull SomeClass y){
    y.f = null;
    x.f.somefun();
  }
}
```

Figure 4.3: Now with `@NonNull`!

(and parameters and method returns and so on[2]). This results in something like Figure 4.3.

In this example, there are three places where the `@noNPE` tree walker provides assurance based on analysis results. The first two are on dereferencing fields of the `@nonNull` parameters `x` and `y`. These will have been added to the set of non-null locals at method entry, by mapping them promise drops reflecting their annotations. Similarly, `x.f` will not generate a `NullPointerException` when used as a receiver because the field is annotated as being `@nonNull`. [3] Because the receiver is not a local variable, this result comes directly from the field annotation: the inference chain can be calculated locally.

The addition of a new annotation necessitates the creation of a corresponding assurance. That is, now that we have `@nonNull` annotations, we must assure them. For the code in Figure 4.3 this assurance fails because a `@nonNull` field is given a

[2]Indeed, the CFG analysis can be obviated by forcing `@nonNull` annotations onto local variables and enforcing a strict doctrine on assignments.

[3]The `@nonNull` annotation has differing meanings when applied to parameters and fields. A `@nonNull` parameter is not null at method entry; after that its status is determined by analysis, not specification. A `@nonNull` field can never hold a null value after the class constructor has successfully executed.

(very possibly) null value. How should this new assurance be accomplished? The tree walker performing the `@noNPE` assurance could be extended with additional checks at field assignments, method invocation and method returns to assure the `@nonNull` assurances at these locations. However, this presumes that `@nonNull` assurance will solely be conducted in the context of the `@noNPE` assurance; this assumption is likely unwarranted.

The combined `@nonNull` assurance/`nonNull` analysis can consist of three parts.[4] The analysis has two parts: the analysis entry point class and the associated class of transfer functions. The former contains methods to instantiate and initialize a new flow analysis for a given flow unit, and to report on the potential null-ness of a given expression. The transfer functions will both iterate to a fixed point on the lattice which is mapping from non-null local variables to the promises (and pseudo-promises) proving them as such and, as a side-effect, check potential null-ness of field assignments, method returns and parameters to method calls.[5] These checks will pass the drop for the `@nonNull` promise being checked, the `boolean` determination of whether the annotation has been met, and the set of promise drops supporting that conclusion[6] to the mediator, which will generate appropriate result drops on the rework pass only. Six different results drops are possible: `@nonNull` parameters, field assignments and returns can each be provably non-null or not. The mediator is also responsible for correctly attaching the drop to the passed in promises and to the node where the check took place. As a courtesy, the overall non-null checking function in the entry point should return false if any `@nonNull` assurance checks fail.

[4]Four really, but the lattice can be an off-the-shelf instantiation of the existing `PartialMapLattice` class

[5]This check is against the local lattice value and not in general, as with the superficially similar check in the main class.

[6]Expressions which are not local variables must be able to spontaneously locate promises or generate pseudo-promises locally supporting or denying their non-nullness.

4.3 Fluid Decisions in Permission Lattice

The permission analysis is implemented as a side-effecting control-flow analysis. All assurances are generated in the `PermissionDropMediator` object, which is attached to the analysis as a whole. Result drops are generated from assertions against the lattice representing the current state of permissions. A boolean value representing the success of the assertion is passed to a method generating the appropriate result drop; the value of the boolean determines whether the drop is positive or negative feedback; the message is assigned accordingly. It is also the responsibility of the mediator to only actually create drops when the analysis has reached its fixed point. Thus, assurance becomes relatively straightforward: for each method or constructor being checked, reporting is disabled, the analysis iterates to a fixed point, reporting is activated, and every transition is reworked exactly once, producing the appropriate drops. The interesting questions are where and how one makes the assertions.

Where assertions are made depend on which assertion is being made. Assertions for successful reading and writing of fields are made at the uses and assignments of those fields, respectively. All other checks are made only at the boundaries of the method—what permissions exist make no difference until someone outside the method examines them. Thus method calls must make a variety of checks. First, they require read or write permissions, as appropriate, for their read or write effects; further, each of those fields (and/or each field nested in the named data groups) must be checked for its appropriate annotation (currently only ***`unshared`*** and the implicit ***`shared`***, but true for other added annotations). Further, any annotations of ***`unique`*** or ***`shared`*** (implicitly annotated) on parameters or the receiver must be checked. Permission used for ***`unique`*** is removed as it is sent to the method. As the effects are checked, they must also be removed from the permission set, to avoid using

the same permission twice. For read effects, we reduce the contributing permission to a *smaller* permission, enabling read parallelism. After all checks are complete, the permissions removed to satisfy the effects are restored; excepting that `unique` permissions are renamed (given a new location and a new All permission) to simulate packing and then unpacking the existential closure. The other important locale for checks is the end of the method body. Here, effects annotations on the method itself need to be checked, as the effects are returned when the method finishes. Again we need permission for the effects, their annotations, and the annotations on fields nested in the effects. (Nesting of fields in data groups will always be of some finite depth.) In addition, we must check that the annotation on the return value can be satisfied by the value left atop the execution stack at return.

4.3.1 Assertions and Claims

Checking for required permissions appears trivial, especially in the case of checking a base permission. We could just look up the key in question in the set of fractions and see if sufficient permission exists. Unfortunately, this ignores possible effects from equality facts. That is, the (simplified) permission lattice

$$\{L_1.\mathrm{f} \mapsto r\}$$

gives us read permission for the field L_1.f, but so does

$$\{L_2.\mathrm{f} \mapsto r \circ \{L_1 = L_2\}\}$$

and, for that matter, so does

$$\{L_3.\mathrm{f} \mapsto r; L_4.\mathrm{f} \mapsto w\} \circ \{L_1 = L_2 \wedge \{\{L_2 = L_3\} \vee \{L_2 = L_4\}\}\}\,.$$

Searches through the space of equality facts are handled using *claims*. Basically, a claim is a unary predicate that can be asserted about some `SimpleLocation`. In this case, the claim is that the current set of permissions contains at least read permission for the f field of the location. (Also necessary is that nothing has been carved from this field.) This claim is tested first on the required location L_1—remember we are looking for permission to read $L_1.f$. In the first lattice, this test suffices. If that claim fails (returns false), we generate the set of all locations, from the main conjunction in the fact lattice, that are equal to L_1 ($\{L_1, L_2\}$ in the second example) and test them until one succeeds or all fail. In the second case, the claim would succeed for $L_2.f$ and therefore succeed in general. If the test is still unsuccessful, we test *all* branches of the disjunction (together with the standard shared conjunction) and if each succeeds, the claim succeeds. Thus, for the third example, we would try $\{L_1, L_2, L_3\}$ (which succeeds for $L_3.f$) and $\{L_1, L_2, L_4\}$, which succeed because we can write $L_4.f$. As both branches succeed, there is always sufficient permission to read f.

Nesting complicates things further. The permission lattice

$$\{L_3.\mathrm{f} \mapsto r; L_4.\mathrm{g} \mapsto w\} \circ \{L_1 = L_2 \wedge L_4.f \prec L_4.g \wedge \{\{L_2 = L_3\} \vee \{L_2 = L_4\}\}\}.$$

should also succeed. Thus some claims, like those for permissions are permitted to extend 'upward' on nesting facts as well. Here, after the test for the f field of $\{L_1, L_2, L_4\}$ fails, it extends to the g field of each of $\{L_1, L_2, L_3\}$. and eventually succeeds.

Additional complications ensue because some checks need to remove the permissions. For example, once a permission is used for a check when passing an effect, we want to remove it from use (or make it smaller for read effects) Thus, the claim object needs to track which actual keys were used to pass the check, so that they can

be updated as appropriate. Of course, this is unnecessary when checking permission to read a field locally; thus the two checks have different claims, different assertions, and even different result drops.

The success of an assertion thus depends on the success of one or more attempts at its associated claim. Which result drop is generated, in turn, depends on the success or failure of the assertion. Assertions for uniqueness and shared feature their own claims, separate from those used for permission reads and writes. The mechanism is general enough, however, to be usable for other purposes; for example, aliasing questions can be rephrased as claims.

4.3.2 A Word on Null-ness

Although the formalism allows for, and even requires, modifier for nullness on fields, there is not (to my knowledge) actual support for `nonnull` and `maybenull` promises in Fluid. This means that we cannot directly oblige the standard handling of null-ness.

One solution would be to annotate all references as `maybenull` by default. This approach is safe; no reference is given too strong a condition. It becomes untenable because of a drawback in the design of the lattice: we do not explicitly represent conditional permissions. As such, we cannot represent permissions conditioned on the inequality of a reference to null, which is how `maybenull` permissions are represented. Representing all fields as `nonnull` is feasible in the analysis, but unsound—some fields really are null.

The current implementation is an uncomfortable compromise. When a field is unpacked, it is assumed to be `nonnull`: all associated facts and permissions are immediately added. When it is packed, it is treated as `maybenull`: the value is

tested for null-ness and the associated permissions are only requisitioned if it is non-null.

It is clear that, absent *nonnull* annotations, treating fields as *maybenull* when packing them is safe. Less clear is whether treating all fields initially as *nonnull* is safe. So when is treating a *maybenull* field as *nonnull* unsafe? The danger is in gaining access rights via an included permission when the reference is or could be null. However, any access rights given to a null reference will only be usable on fields of that reference: attempts to access them on a null pointer will fail from a `NullPointerException`. Passing the permission to another function is acceptable, as that use is also *maybenull*. Thus, this approach, while not entirely sound, will not result in any permission errors, which suffices.

4.3.3 Putting It All Together

What all, exactly, happens when running the analysis? The process follows several stages, complicated somewhat by design issues touched on herein.

- The `PermissionAssurance` is the upper-level entry point for the analysis. The Fluid double-checker works as a builder in Eclipse, so that whenever a project (for which Fluid assurance is enabled) is built, user-selected assurances are also run on the project. `PermissionAssurance` is one such. When given a compilation unit by the infrastructure, `PermissionAssurance` traverses the abstract syntax tree until it reaches a unit of control flow, generally a method, constructor, or static initializer. At this point, the assurance creates an instance of the control-flow analysis and runs it on the method.

- The analysis first disables error reporting, then iteratively applies the transfer functions to find a fixed-point for permission lattice values. The initial lattice is

set based on the effects and other method annotations passed in. Most transfer functions directly mirror their evaluation functionality using location lattice objects in the place of references.

- Checks are made against the lattice at function calls, field accesses, and at function return to see if requisite permissions are present. As error reporting is off, this may appear fruitless; however, the internal analysis state may be tied to the results of the checks. For example, passing a unique parameter requires both finding and removing the associated All permission.

- Most checks involve creating the appropriate `Claim` object and testing that claim on several locations, depending on the known facts.

- These facts are sometimes created on object instantiation, establishing its field nestings; sometimes after a method call, for both the returned value and any returned effects; and from the control flow itself. For example, conditional expression will be true or false at the appropriate exit port.

- Once a fixed point is reached, error (and success) reported is enabled, and the analysis is re-worked; each edge of the control-flow graph is revisited in order. Because a fixed point has been reached, the lattice values will not change. However, revisiting them will enable all of the permission checks to be made again; this time creating drops for the results in Drop-Sea.

- At this point, the analysis is done. The `PermissionAssurance` continues and may itself be invoked again on other compilation units. Eventually, when all assurances have been run on all appropriate compilation units, all drops become available for view. They are structured by the type of annotation being assured, as a tree, with positive or negative reports about an annotation as children of

that annotation.

The control flow analysis described in this paper uses a limited representation of permission. The intent is to provide sufficient representation to check annotations and no more. Not all possible permissions can be directly represented. Similarly, low-level permission transformation operations, even at the level described by the algorithmic type rules, are not always practical. Instead, the results of those particular transformations demanded by joining and/or asserting permissions are interpreted on the limited lattice representation. The result is an analysis that determines whether a permission analysis would work rather than immediately applying that permission analysis itself.

Chapter 5

Experimental Evaluation

As permission analysis has been implemented as a control-flow analysis, it is now both possible and desirable to actually run the analysis on sample code. We can compare the results with those of the preexisting (separate) analyses for effects and uniqueness and with our expectation from hand evaluation of the existing permission formalism. Then, there are two kinds of sample programs on which we should test the analysis: small examples that are simple enough to type-check by hand using the formal permission system and large complex systems, preferably consisting of actual production code. The latter are useful both in presenting the analysis with a wide range of idioms, not all of which were anticipated by the author and in providing evidence of the scalability of the analysis.

5.1 Small Examples

A small example consisting of several short functions is useful for evaluation for two principal reasons. The first is the ability to compute expected results for the analysis by hand; this permits precise determination of correctness. The second is that with

a small enough example we can "open the hood" and look not just at the final but at the evolving state of the analysis; this ability is vital for debugging. As such there are two classes of small examples on which the analysis has run: demonstration files to verify the correctness of the analysis, and isolated functionality from larger examples, introduced for the purpose of debugging. This discussion will be restricted to the former.

The first such example predates the creation of the analysis. It is a variation on the example in Figure 1.3, which is useful for demonstrating both that the analysis can detect the particular misuse of uniqueness highlighted in that example and that it can positively assure a simple linked-list implementation. The full text for the example is in Figure 5.1. Running this example through the analysis leads to all positive assurance with three exceptions: the write of `m.next` on line 41 cannot happen because we no longer have permission for the field, and uniqueness produces errors for `n` and `head` at the end because we cannot recreate the existential closures needed for to return the *`unique`* effects. The latter error seems somewhat unnecessary; however, it cascades from the effect error—commenting out line 38 fixes all problems.

Most of the other smaller examples were added to test specific pieces of functionality as they were implemented. Test cases exist for effects on regions (data groups) covering the fields inside the groups, proper handling of the array element region, static fields and methods, and the use of facts to handle aliasing across merges. The tests on statics field do not behave correctly, because of an annotation error on my part, but excepting those, all of these small test cases produce their expected results.

```
1       class Node{
2
3               @InRegion("Instance")
4               @Unique Node next;
5
6               Node(){}
7       }
8
9       class List{
10
11               @Unique Node head;
12
13              @RegionEffects("writes head")
14              void mostly_clear(){
15                      head.next = null;
16              }
17              @RegionEffects("writes head")
18              void prepend(@Unique Node n){
19                      n.next = head;
20                      head = n;
21              }
22       }
23
24       public class BB {
25
26              @Unique        Node n;
27
28              @RegionEffects("writes n,l:head")
29              void add(List l){
30                      l.prepend(n);
31                      n = null;
32              }
33              @RegionEffects("writes n,l:head")
34              void bad(List l){
35                      Node m = n;
36                      m.next = null;
37                      add(l);
38                      m.next = null;
39              }
40       }
```

Figure 5.1: Actual “Bad” Uniqueness Example

5.2 jEdit

jEdit is a Java-based open source text editor. Version 4.1 was used as a case study for Fluid assurances, primarily lock analysis and thread-coloring. As such, it forms a relatively large (15kB of source code) sample of annotated Java code on which to run the analysis. Because of its size, I could not calculate expected results for the permission analysis by hand, and instead compared it to the results from the pre-existing effects and uniqueness analyses.

On the whole, the results from both forms of analysis are comparable. *Finish this paragraph.*

In many ways, jEdit is not a good choice for evaluating a permission analysis. While it has been annotated, most of those annotations are for thread and lock analyses. Six fields have been annotated as unique; the protection provided by a lock is extended to uniquely referenced state, using the implicit ownership provided by uniqueness. With one exception, these unique annotations are known not to assure; they exist to be trusted by the lock analysis. The one exception also annotates a method return as unique to support the uniqueness of the field. No methods are explicitly annotated with method effects.

However, the methods are then treated as if annotated to write everything. As such, the permission analysis runs on each of these and assures that all writes are legal—in particular that no permissions are lost when analyzing each method. Both the large number of methods and the existence of many methods with particularly convoluted control flow give the analysis many opportunities to actively demonstrate that code without problems has no problems. That is, the analysis is actually tested in a wide range of situations.[1]

The other advantage to testing the analysis on a large code base is to investigate

[1]Several analysis bugs were discovered in this manner.

the scalability of the analysis and in this regard testing was less successful. On methods with particularly convoluted control-flow, analysis finishes in a noticeable amount of time (10-30 minutes). Worse, the cumulative memory requirements have been large enough, even with a maximum of 2 gigabytes of space, to stall out running the analysis with threshing when run on the complete `jEdit` project. To combat these problems, the analysis has been run on all methods individually, but not collectively.

This approach is satisfactory for testing purposes, but inadequate for deployment. Engineering solutions may help. A more optimized implementation would universally reduce the degree of the problem. Additionally, the assurance mechanism can be optimized to only run the control flow analysis on methods which are explicitly annotated. A more theoretical solution is to produce a hierarchy of permission analysis of varying efficiency, and only running the less efficient analyses when the faster analyses could not determine a result. This would not speed up the analysis per se, but would run it on fewer methods.

Chapter 6

Conclusion

The intention of this work was to produce an annotation checker for effects and uniqueness annotations using semantics based on fractional permissions. On the whole, this has been accomplished.

6.1 Further Work

The work described in this paper is a very small thread in a very large tapestry. Fractional permissions are a powerful tool and more can be done with them than is described here. Additions and improvements are possible to the analysis presented here; some of them principally issues of engineering, while others will require revisiting the design and possibly even the theory of the analysis. Also, more work can be done with fractional permissions outside the context of the particulars of implementing this analysis.

6.1.1 Additional Annotations

The permission formalism describes the semantics of more annotations than are currently assured. For some of these annotations, notably ownership, *readonly*, *immutable* and *nonnull*, the analysis as implemented should be powerful enough to check them also. However, there is no annotation infrastructure for them. Given the annotation objects, promise drops for the annotations, and revisions in the parser to handle these annotations, the analysis itself would only need cosmetic modification to also assure them. The principle additions would be at method boundaries, where the new annotations would need to be checked and/or accounted-for in the analysis state; in the creation of new claims specific to those annotations, and in the interface to Drop-Sea, for reporting purposes. Ideally *from* annotations could be added similarly, but the machinery needed to generate the correct lattice state for `from` is sufficiently complex that I cannot be certain.

The trickiest to add will likely be *raw* annotations. The current analysis relies on a separate binder (which links uses to definitions) to look up annotations directly, rather than passing them as additional predicates. Thus adding a full *raw*ness analysis would require adding named predicates for classes and cooked. A conservative approximation of a rawness analysis could be done by adding checks against 'leaky' constructors, which hand out *raw* objects.

6.1.2 Concurrency

The analysis and even the permission formalism is presented here entirely in the context of single-threaded programs. However, fractional permissions also can provide semantics across multiple threads. In fact, their linearity enables them to support

race-free parallel programs. It remains, however, to extend the analysis infrastructure to support annotations describing the design intent of parallelism. Already, annotations exist in Fluid describing locking regimens; they should be checkable using a permission-based analysis. Yang Zhao **??** has designed one such. Similarly, one should be able to create a permission semantics for atomicity.

6.1.3 Efficiency

As mentioned in the previous chapter, one could improve the efficiency of the assurance mechanism noticeably by altering the assurance mechanism to only perform control-flow analysis on methods that are "interesting" to assure. Methods with no annotations, no declared design intent, can be safely ignored. This requires a small secondary analysis to determine whether the method is interesting to assure or not.

Along similar lines, one can build both faster, less precise analyses and more direct theorem-prover based analyses. The assurance mechanism could then start with the light, fast, analysis, and only run the more complex analyses when the less complex ones are insufficient. It also opens the door to experimental determination of how heavyweight an analysis needs to be to handle most or all programs.

6.2 In Summary

Fractional permissions provide a powerful basis for describing and constraining interactions among the states of objects in object-oriented programs. They are extremely low-level, to the extent that they are impractical to use directly. Rather, we use them to support programmer-level annotations of design intent, notably effects and uniqueness annotations. Additionally, we require tool support to check these annotations. Implementing this tool required a series of approximations, from algorithmic type

rules that approximate transformation to lattice operations in a control-flow analysis that approximate the algorithmic type rules. However, the annotations themselves only use permissions in a stylized manner. For the purpose of checking annotations, the approximation provided by the control flow analysis appears sufficient.

Bibliography

[AC04] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *In ECOOP*, pages 1–25. Springer-Verlag, 2004.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 37, pages 311–330, New York, November 2002. ACM Press.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. Ohearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In Proceedings of FMCO05, volume 4111 of LNCS*, pages 115–137. Springer, 2005.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, New York, NY, USA, 2005. ACM Press.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Ru stan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *J. Object Technology*, 3:27–56, 2004.

[BE04a] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 35–49, New York, NY, USA, 2004. ACM Press.

[BE04b] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *In OOPSLA*, pages 35–49. ACM Press, 2004.

[Bie06] Kevin Bierhoff. Iterator specification with typestates. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 79–82, New York, 2006. ACM Press.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 37, pages 211–230, New York, November 2002. ACM Press.

[BNR] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. pages 2–27.

[Boy01a] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[Boy01b] John Boyland. The interdependence of effects and uniqueness. Paper from Workshop on Formal Techniques for Java Programs, 2001, June 2001.

[Boy03] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[Boy05] John Boyland. Why we should not add "read-only" to Java (yet). In *Informal Proceedings of "Workshop on Formal Techniques for Java-like Programs"*, June 2005.

[Boy07] John Boyland. Semantics of fractional permissions with nesting. Technical report, University of Wisconsin–Milwaukee, 2007.

[BR05] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 283–295, New York, NY, USA, 2005. ACM Press.

[BRLS04] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. pages 49–69. Springer, 2004.

[BRZ] John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In preparation for OOPSLA 2009.

[BRZ07] John Boyland, William Retert, and Yang Zhou. Iterators can be independent "from" their collections. In *ECOOP 2007 Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, July 2007.

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, volume 38, pages 324–337, New York, May 2003. ACM Press.

[BV99] Boris Bokowski and Jan Vitek. Confined types. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 34, pages 82–96, New York, October 1999. ACM Press.

[ByECD+06] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *In FMCO 2005, volume 4111 of LNCS*, pages 364–387. Springer, 2006.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering(ICSE '98)*, pages 167–176, Los Alamitos, California, 1998. IEEE Computer Society.

[Cla01] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[CNP01] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In Jørgen Lindskov Knudsen, editor, *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76, Berlin, Heidelberg, New York, 2001. Springer.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 33, pages 48–64, New York, October 1998. ACM Press.

[CW03] David Clarke and Tobias Wrigstad. External uniqueness. In Benjamin C. Pierce, editor, *Informal Proceedings of International Workshop on Foundations of Object-Oriented Languages 2003 (FOOL 10)*. January 2003.

[CWM] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. pages 262–275.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, volume 36, pages 59–69, New York, May 2001. ACM Press.

[ETT05] D. Ernst, Matthew S. Tschantz, and Matthew S. Tschantz. Javari: Adding reference immutability to java. In *In OOPSLA*, pages 211–230. ACM Press, 2005.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practial linear types for imperative programming. In *Proceedings of the ACM SIGPLAN'02Conference on Programming Language Design and Implementation*, volume 37, pages 13–24, New York, May 2002. ACM Press.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03*, 2003.

[GB99] Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 26, pages 271–285, New York, November 1991. ACM Press.

[HPSS07] C. Haack, E. Poll, J. Schfer, and A. Schubert. Immutable objects for a java-like language. Technical report, European Symposium on Programming, volume 4421 of LNCS, 2007.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of the Twenty-eighth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 14–26, New York, 2001. ACM Press.

[KAB07] Neelakantan R. Krishnaswami, Jonathan Aldrich, and Lars Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *In Proceedings of FTfJP*, 2007.

[KMJ02] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245, New York, NY, USA, 2002. ACM Press.

[Kri06] Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 83–86, New York, 2006. ACM Press.

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 26–38, New York, NY, USA, 2000. ACM Press.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston,Massachussetts, USA, 1999.

[LPHZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN'02Conference on Programming Language Design and Implementation*, volume 37, pages 246–257, New York, May 2002. ACM Press.

[Min96] Naftaly Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96 — Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Berlin, Heidelberg, New York, July 1996. Springer.

[MPH00] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In Sophia Drossopolou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *2nd ECOOP Workshop on Formal Techniques for Java Programs*, 2000.

[NNH99a] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Berlin, Heidelberg, New York, 1999.

[NNH99b] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Berlin, Heidelberg, New York, 1999.

[OP99] P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[PS] Frank Pfenning and Carsten Schürmann. The twelf project.

[PS02] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Joint ACM-ISCOPE Java Grande Conference*, 2002.

[Rey02] John Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74, Los Alamitos, California, July22–25 2002. IEEE Computer Society.

[SWM00] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In Gert Smolka, editor, *ESOP'00 — Programming Languages and Systems, 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Heidelberg, New York, 2000. Springer.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. Elsevier, North-Holland, 1990.

[ZPV03] Tian Zhao, Jens Palsber, and Jan Vitek. Lightweight confinement for Featherweight Java. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 135–148, New York, NY, USA, 2003. ACM Press.

[ZPV06] Tian Zhao, Jens Palsber, and Jan Vitek. Type based confinement. *Journal of Functional Programming*, 2006.