Disjointness Domains for Fine-Grained Aliasing *

Stephan Brandauer Dave Clarke Tobias Wrigstad
Uppsala University
firstname.lastname@it.uu.se

Abstract

Aliasing is crucial for supporting useful implementation patterns, but it makes reasoning about programs difficult. To deal with this problem, numerous type-based aliasing control mechanisms have been proposed, expressing properties such as uniqueness. Uniqueness, however, is black-and-white: either a reference is unique or it can be arbitrarily aliased; and too global: excluding aliases throughout the entire system, making code brittle to changing requirements. *Disjointness domains*, a new approach to alias control, address this problem by enabling more graduations between uniqueness and arbitrary reference sharing. They allow expressing aliasing constraints local to a certain set of variables (either stack variables or fields) for instance that no aliasing occurs between variables *within some set* of variables but between such sets or the opposite, that aliasing occurs within that set but not between different sets. A hierarchy of disjointness domains controls the flow of references through a program, helping the programmer reason about disjointness and enforce local alias invariants. The resulting system supports fine-grained control of aliasing between both variables and objects, making aliasing explicit to programmers, compilers, and tooling. This paper presents a formal account of disjointness domains along with examples. Disjointness domains provide novel means of expressing *may-alias* kinds of constraints, which may prove useful in compiler optimisation and verification.

* This work was partially funded by the Swedish Research Council project Structured Aliasing, the EU project FP7-612985 Upscale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations, and the Uppsala Programming for Multicore Architectures Research Centre (UPMARC).

OOPSLA '15, October 25th-30th, 2015, Pittsburgh, PA, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Aliasing, two variables containing the same reference, is both a blessing and a curse. It is a blessing because it allows programmers to write efficient imperative code (any data structure that is not tree-shaped needs aliasing). It is a curse because aliasing together with mutable state makes compile-time and run-time optimisations difficult; it complicates manual and automated reasoning; and, with the advent of multi-core computers, it necessitates concurrency control and all of its complications.

Compiler optimisations depend on aliasing knowledge. For instance, the following C program may only be automatically parallelised by a compiler if all the pointers in the array are known to refer to disjoint memory locations[1] – otherwise, the compiler would risk data races:

```
int* array[N]; // N pointers to int
init(array,N);
// increment all the integers:
for (int i=0; i<N; ++i) {
  *array[i] += 1;
}
```

Reasoning about program correctness is also difficult in the presence of aliasing. The two method calls operating on two file objects, `f1.close(); f2.read()`, can only be correct if `f1` and `f2` are not aliases. If `f1` and `f2` are aliases, the file is read from after closing. In general, Hoare-style reasoning needs to consider all possible aliasing: in `val1.set(1); val2.set(2); assert(val1.get() == 1)`, the assertion provably holds only if `val1` and `val2` do not alias and the two objects don't share any state.

Experiments suggest that the vast majority of objects in object-oriented programs are in fact not aliased [32], yet compilers and verifiers alike need to err on the safe side.

A large number of type-based alias management proposals have been put forward in the last 20 years (*e.g.,*[8, 9, 12, 14, 20, 22, 26, 27, 30]). Several make use of unique references, references guaranteed to be alias-free. Uniqueness can be maintained either using destructive reads [25, 27] (reads that

[1] To explicitly allow such optimisations, the `C` language defines the `restrict` type qualifier that states that all state reachable from two such annotated pointers is disjoint. C++11 uses `rvalue` references that, intuitively, represent the return value of an expression before it has been stored into a variable. Although they unlock many optimisations, they also lack type safety.

also set the variable being read to `null`,), a swap operation [23] or static analysis [8, 22]. In such systems, a reference is either unique or shared, though generally language constructs exist for converting unique references to shared ones and vice versa [9, 12] Recently, programming languages supporting alias control have gained some traction [4, 15, 33].

This paper argues that the dichotomy of unique vs. shared is unnecessarily black-and-white, and, as a consequence, makes it impossible to use uniqueness to express local invariants without global consequences. For example, relying on uniqueness to statically enforce the property that two variables `val1` and `val2` may not alias is not possible without also prohibiting the existence of aliases elsewhere in the system. If such aliases are necessary, statically enforcing `val1 != val2` through uniqueness is no longer possible. This inhibits reasoning both about the functional correctness of code as well as the correctness of employing compiler optimisations which require reasoning about aliasing. A property as powerful as global uniqueness may often make sense, but only having shared variables as an alternative leads to loss of valuable information.

Consider, for example, using uniqueness to enforce that a list of employees contains no duplicates. This property might allow modifying the elements in parallel. Later, adding an employee-of-the month variable (which may be an alias of any of the list elements) requires a shift from unique to shared references in the list, destroying local knowledge about the alias-freedom of the list. Although parallel iteration over the employee list is still data-race free (as long as the employee-of-the-month variable is not used at the same time), this information is now lost to compiler, programmer and verification tool.

In this paper, we argue that many important aliasing invariants are local, and therefore should be enforceable without global consequences. Local alias invariants are expressed in terms of *disjointness domains*, which correspond to sets of variables and fields. Disjointness domains can express global uniqueness (in fact, it's the default) but allow fine grained middle ground between globally unique and globally shared where aliasing is introduced in piecemeal way, in a syntactically tractable fashion.

Disjointness domains provide novel machinery for expressing different kinds of *may-alias* constraints, and for adapting them as the program proceeds. Such constraints may prove useful in compiler optimisation and verification.

The paper makes the following contributions:

1. It proposes disjointness domains as a means to establish and locally reason about fine grained alias invariants (Section 2).
2. It shows how using disjointness domain parameters on classes leads to flexible local reasoning about *disjointness of objects* (whether or not there may be state shared by two objects), an ability that systems with just globally unique and globally shared variables don't have (Section 3).
3. It presents a selection of recipes for local aliasing invariants and how these can be encoded with disjointness domains (Section 4).
4. It shows that disjointness domains are strong enough to establish useful invariants. In particular, we explore disjointness domains to establish non-interference of expressions run in parallel. Disjointness domains deal with examples that other alias management systems fail to handle: a doubly-linked list with external iteration and an internal, parallel, imperative `foreach` implementation (Section 5).
5. It formalises disjointness domains in the context of a simple experimental language design (Sections 6, 7), along with their proofs (Section 8). It also shows how traditional uniqueness is a special case of local uniqueness (Section 8).

2. Overview

This section gives an overview of disjointness domains and shows how they are used to enforce various aliasing invariants. To illustrate the explanations, we are using a running example (Section 2.1). The core concepts such as disjointness domains and aliasing within or between them are introduced (Section 2.2), followed by a description of the invariants that disjointness domains enforce (Section 2.3), and the operations on variables that preserve the invariants (Section 2.4).

2.1 Running Example

As a running example, we use disjointness domains to model a company and the aliasing properties reflected in the company's structure. The company offers different jobs, represented as variables: `ceo`, `cto`, `programmer1`, `programmer2`, `teamLeader1`, `teamLeader2`, and `recruiter`. There also is an `employeeOfMonth` – but that does not count as a job.

The (aliasing) properties of the company are:

Prop. 1: No one except the CTO can have more than one job.

Prop. 2: The CTO can additionally lead at most one team.

Prop. 3: The CEO can not be employee of the month.

2.2 Core Concepts

We define the core concepts of the proposal, starting with disjointness domains (Section 2.2.1). Disjointness domains are arranged in a hierarchy (Section 2.2.2) and may be linked (Section 2.2.3). The hierarchy and linking together will determine which aliases are allowed.

2.2.1 Domains and Aliasing

A *disjointness domain* represents a set of variables satisfying some aliasing invariants. Every variable (we will use the term "variable" to refer to both stack variables and fields in the rest of the paper) is annotated with the domain to which it belongs. A type of a variable is its class and the domain, *e.g.*, `Person#unique` is a variable type that stores a globally unique reference to a `Person` object. Domains have a name and come

in three kinds: *strong*, *weak*, and *shared*. If the domain name is "D", these are written D, Wk D or Sh D, respectively.

The aliasing property of interest is whether two variables *may alias*, namely whether they can possibly contain a reference to the same object. We say that two *domains may alias* if two variables in the domain may alias.

Strong domains allow little aliasing, weak domains allow more and shared domains allow the most. In fact when changing the domain kind of a variable, aliasing grows monotonically: if a strong domain A and another domain may alias, it implies that the weak or shared domains Wk A or Sh A and the other domain may also alias. Similarly, if a weak domain Wk A and another domain may alias, it implies that Sh A and the other domain may also alias.

Domains are declared by the programmer, the only initial domain is a strong domain called **unique** which permits no aliasing at all. This domain thus captures the notion of global uniqueness. New domains can be created by *extending* existing ones, which organises domains into a hierarchy. Domains may also be *linked*, stating that they may alias.

The kind of a domains, their position in the hierarchy, and whether they are linked, all influence whether variables in different domains may alias. These aspects will be discussed in the remainder of this section.

2.2.2 Domain Hierarchy

New disjointness domains are introduced by *extending*[2] an existing one. Domains are introduced to include either just the strong kind (by introducing just a new domain name) or all three kinds (by annotating the new domain name with Sh). Domain extension induces a hierarchy of domains and domains with the same name are considered to be in the same location in this hierarchy[3]; diagrams like Figure 2(a)-Figure 2(c) depict this by putting these three domains (D, Wk D, Sh D) in the same box. As the only initially accessible domain is **unique**, any domain hierarchy is rooted in **unique**. New domains can be used to express more fine-grained aliasing patterns. The extensions of a domain are called its *children* and domains being extended are called *parents*. All domains preceding a domain in the extends relation are called its *ancestors*.

The code in Figure 1 initially introduces domain A whose parent is **unique**, and then domains B, Wk B and Sh B, whose parent is A. C and D will be discussed in Section 2.2.3.

Each disjointness domain exists for a given (lexical) scope, as indicated in the code above using curly braces. When a domain goes out of scope, it ceases to exist. This allows temporary aliasing to be expressed, and even the unaliased data to be recovered after it has been aliased for a while.

[2] Not related to extending Java classes.

[3] As the domain kind is not relevant for the location in the domain-hierarchy, syntactically we always extend from a domain name alone.

```
domain A extend unique in {
  -- domain A introduced for this scope
  domain Sh B extends A in {
    -- B, Wk B and Sh B introduced for this scope
    domain (C, Sh D) extend unique in {
      -- C and D    are linked
      -- C and Wk D are linked
      -- C and Sh D are linked
      -- Wk C and Sh C are not available
    }
  }
}
```

Figure 1. Creating domains from existing domains.

2.2.3 Linked Domains

Domains can be *linked* together. *Any two linked domains may alias.* Linked domains are expressed in code by declaring them together in the same expression (domains C and Sh D in the code above). A domain is never linked with itself or the other two domains with the same name but different kinds. In addition, if two domains of any kind A and B are linked, then A's children and B are also linked. In diagrams, such as Figure 2(a), domains created together are indicated using an arc between the arrows from their parents. In this figure, any domain in the left-most subhierarchy is linked with any domain in the center subhierarchy (due to the connecting arc). The domains in the right-most subhierarchy are not linked with any domain. *No domains in different, unlinked subhierarchies may alias.*

2.3 Disjointness Domains and their Invariants

This subsection describes the may-alias invariants imposed by disjointness domains. Whether or not two domains may alias depends on the kind of domains and on their locations in the hierarchy. We cover invariants for each domain kind: strong (Section 2.3.1), shared (Section 2.3.3) and weak (Section 2.3.2).

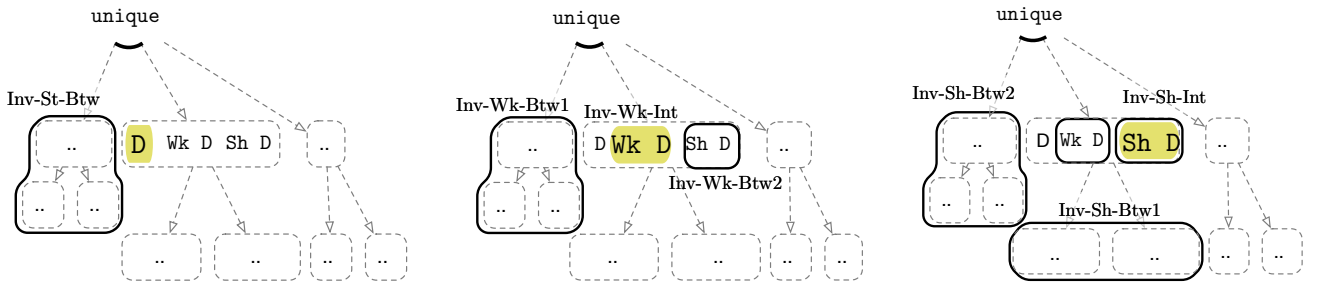
2.3.1 Strong Disjointness Domains

A *strong disjointness domain* (or simply strong domain) is a set of variables that may not alias – we call them *locally unique*. The invariants for strong domains are:

Inv-St-Int: Strong domains are internally unaliased – no two variables in the same strong domain may alias.

Inv-St-Btw: Two strong domains may alias if and only if they are linked.

Figure 2(a) illustrates the may-alias invariants for the strong domain D, indicated with the coloured shading. D may alias with the domains circled in the figure. No circle around D indicates no aliasing within D (Inv-St-Int). All domains in the linked left subhierarchy may alias D whereas domains in the right subhierarchy may not (Inv-St-Btw).



(a) The strong domain `D` may alias any of the domains circled. Arcs denote domains created together – they are linked, and so are their children. Labels next to circled domains denote the relevant invariants.

(b) Weak domain `Wk D` may alias domains that are circled. `Wk D` may alias every domain that the domain `D` may alias and additionally may alias `Sh D`.

(c) Shared domain `Sh D` may alias domains that are circled. `Sh D` may alias every domain that `Wk D` may alias and additionally may alias `Wk D` and all ancestor domains. `Sh D` may alias itself, two variables in that domain may alias.

Figure 2. Changing the domain kind of a variable from strong to weak to shared adds to the set of variables it may alias with.

Traditional uniqueness (*e.g.,*[10, 20, 27, 30]) is achieved using the strong domain `unique` which has no linked domains and therefore can not alias any other domain.

Running Example: The declarations below set up two linked domains `CTO` and `Leaders` and declare variables for each job (but not the employee of the month yet). We assume a `Person` class to be defined elsewhere. Figure 3 shows the domain hierarchy of these variables.

```
1  domain (CTO, Leaders) extend unique {
2    let
3      ceo          : Person#unique  = new Person,
4      recruiter    : Person#unique  = new Person,
5      programmer1  : Person#unique  = new Person,
6      programmer2  : Person#unique  = new Person,
7      cto          : Person#CTO     = new Person,
8      teamLeader1  : Person#Leaders = new Person,
9      teamLeader2  : Person#Leaders = new Person
10   in ...
11 }
```

We wanted to express certain properties in Section 2.1 and now take a look at the first two:

Prop. 1, that no one except the CTO can have more than one job is present: for a person to have two jobs, two job variables would need to be referring to that person, making them aliases. Some variables are unique (their types are annotated with `#unique`), and a unique variable can not have any aliases. The other variables (except `cto`, which we treat in the next paragraph) are in the `Leaders` domain (annotated with `#Leaders`) – the `Leaders` and `unique` domains are not linked, so they can not contain aliases (Inv-St-Btw). Also the `Leaders` domain is strong – strong domains are unaliased (Inv-St-Int), so the two team leaders may not alias.

Prop. 2, that the CTO can lead at most one team is present. It requires aliasing to be possible between the `cto` variable and the `teamLeader1/2` variables. Since the domains of the `cto` variable (`CTO`) and the team leader variables (`Leaders`) are linked, aliasing is indeed possible (Inv-St-Btw). The CTO can only lead *at most* one team: for the CTO to lead more than one team, the two team leaders would need to be aliases. They can't be aliases due to the fact that the leaders domain is aliased (Inv-St-Int).

2.3.2 Weak Disjointness Domains

Weak disjointness domains (simply, weak domains) invariants are very similar to those of strong domains. The difference is that weak domains may additionally alias equally-named shared domains, whereas strong domains cannot.

Inv-Wk-Int: Weak domains are internally unaliased – no two variables in the same weak domain may alias.

Inv-Wk-Btw1: If a weak domain and any other domain are linked, they may alias.

Inv-Wk-Btw2: A weak domain `Wk D` and an equally-named shared domain `Sh D` may alias.

Figure 2(b) illustrates these rules. Observe that the circled domains for weak domains are a superset of those for strong domains, aliasing grows monotonically.

It is possible to have a single alias in a weak domain and many aliases in the corresponding shared domain. The reference in the weak domain is privileged, in that it is unaliased within the weak domain. This is similar to *e.g.,* C++'s difference between stack values (two stack values are always in different locations) and references (there can be many references to the same stack value).

2.3.3 Shared Disjointness Domains

A *shared disjointness domain* (or simply shared domain) is a set of variables that may alias – we call them *locally shared*. Local sharing is, in a sense, the opposite of local uniqueness (a set of variables that may *not* alias).

Inv-Sh-Int: Shared domains are internally aliased – any two variables in the same shared domain may alias.

Inv-Sh-Btw1: If a shared domain is an ancestor of another domain, they may alias.

Inv-Sh-Btw2: If a shared domain and any other domain are linked, they may alias.

Figure 2(c) depicts the constraints imposed by these rules. Note that the set of circled domains in Figure 2(c) is a superset of the circled domains in Figure 2(a) and Figure 2(b), aliasing grows monotonically.

Global sharing is possible by putting all variables in the same shared domain.

Running Example: In Section 2.3.1, we treated the first two properties of the company, leaving the third.

Prop. 3 of the company example requires that the employee of the month variable may alias programmers, recruiters, team leaders, CTO – all jobs except the CEO.

Recalling Inv-Sh-Btw1 (if a shared domain is an ancestor of a domain, they may alias, seeFigure 2(c); we will add a new shared domain `Sh Employees` that is an ancestor of the domains of all variables that the employee of the month must be able to alias. We put, in Figure 4 and Figure 5, the `employeeOfMonth` variable inside the `Sh Employees` domain and we will move the other job variables – except `ceo` – into domains that are children of `Sh Employees`.

The domain declaration `Sh Employees` in source code line 1 indicates that all three domain kinds can be used with that name (`Employees`, `Wk Employees`, `Sh Employees`). In lines 2 and 3, we extend from `Employees`.

2.4 Moving and Copying References within and between Domains

There are four operations on references in domains: moving a reference from one variable to another; copying a reference from one variable to another; linked assignment; and recovery. This subsection describes the semantics of these operations and the constraints on them to preserve the invariants of disjointness domains.

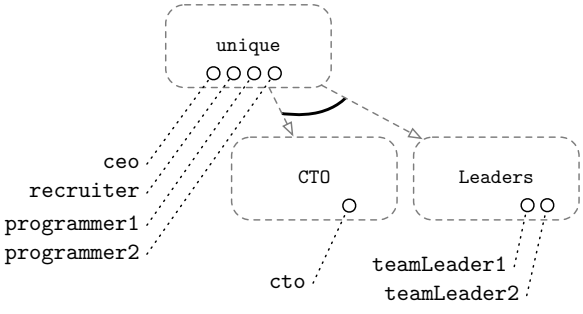


Figure 3. A domain hierarchy for **Props. 1 and 2**. Circles depict variables within their domain, they are labelled with the variable name.

```
1  domain Sh Employees   extend unique {
2  domain Rest           extend Employees {
3  domain (CTO, Leaders) extend Employees {
4    let
5     employeeOfMonth : Person#Sh Employees = null,
6     ceo             : Person#unique       = ..,
7     recruiter       : Person#Rest         = ..,
8     programmer1     : Person#Rest         = ..,
9     programmer2     : Person#Rest         = ..,
10    cto             : Person#CTO          = ..,
11    teamLeader1     : Person#Leaders      = ..,
12    teamLeader2     : Person#Leaders      = ..
13   in
14    -- we can now choose to make programmer1
15    -- the employee of the month:
16    employeeOfMonth = programmer1
17 }}}
```

Figure 4. Code implementing the domain hierarchy for **Props. 1-3**.

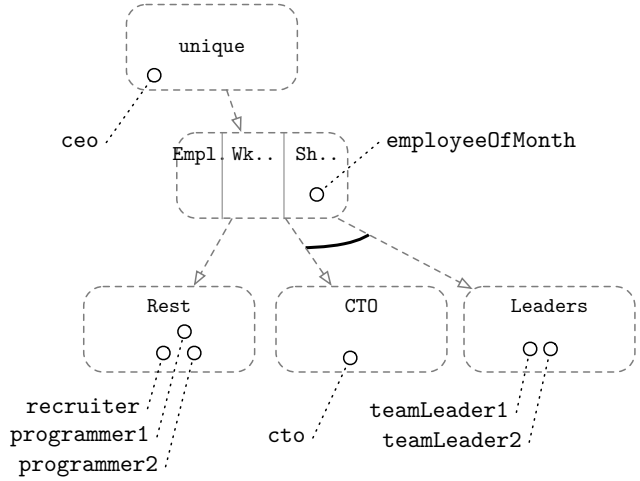
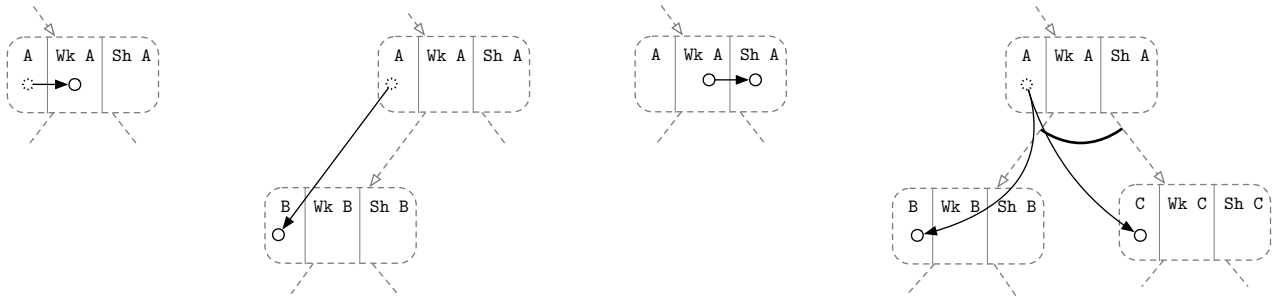


Figure 5. The final domain hierarchy for **Props. 1–3**. The `employeeOfMonth` variable may alias all variables in ancestor domains (all except `ceo`).

2.4.1 Moving Assignment

A *moving assignment*, `y = move x`, also known as destructive read, is an assignment from one variable to another that results in the original variable being destroyed. A moving assignment is permitted:

- within a strong or a weak domain, preserving alias-freedom of the domain (Inv-St-Int, Inv-Wk-Int);
- from a strong domain to the equally-named weak domain (Figure 6(a)), also preserving alias-freedom; or
- from a strong or weak domain to a strong or weak domain *down the hierarchy* (Figure 6(b)). The domain kinds do not need to match.



(a) Moving a reference, `wka = move a`, from a strong domain to an equally named weak domain.

(b) Moving a reference from a strong domain down the hierarchy (`b = move a`).

(c) Copying a reference, `sha = copy wka`, from a weak domain to a shared domain.

(d) Linked assignment `(b, c) = move a` from a parent domain to linked child domains, making `b` and `c` aliases.

Figure 6. Examples of legal assignments. ○ – Variable; ⁘ – Destroyed variable.

2.4.2 Copying Assignment

A *copying assignment*, `y = x`, is the standard assignment operator, which copies the reference in `x` and stores it in `y`. As copying assignment introduces aliases, only the following cases are permitted:

- within the same shared domain.
- from a weak domain to an equally-named shared domain (Figure 6(c)). This allows a weak domain to alias a shared domain.
- from a domain to a shared *ancestor* domain.

The rules above allow weak domains to alias equally named shared domains. In contrast, *strong* domains may not alias equally named shared domains as a moving assignment from the strong domain is necessary to assign to the weak domain (Section 2.4.1).

2.4.3 Linked Assignment

A *linked assignment*, `(y1, .., yn) = move x`, is the only way to create aliases in linked domains. It is a moving assignment that has multiple target variables on the left hand side. It copies the reference into each target variable, destroying the original variable. For instance, the expression above stores the reference from `x` in the variables `y1, .., yn` (making them aliases) and destroys `x` in the same step. Linked assignments can create aliases between linked strong domains. The original variable is destroyed to prevent the reference ending up in unlinked child domains through subsequent operations.

A linked assignment `(y1, .., yn) = move x` is permitted only if variables `x, y1, .., yn` are all in strong or weak domains; and the target variables `y1, .., yn` are in linked domains; and `x` is in a parent domain of those linked domains (Figure 6(d)).

2.4.4 Getting Uniqueness Back

Domain extension, as described above, introduces new domains that can be used to describe aliasing. The lifetime of the newly-introduced domains is governed by a lexical scope. At the end of that scope, all aliases in the new domains vanish – they are not well-typed any more. This allows programmers to express *temporary* aliasing. The final operation, *recovery*, allows programmers to pull some of the references in domains that go out of scope and place them into an ancestor domain.

For example, in Figure 3 we can recover variables in *either* the `CTO` *or* the `Leaders` domain to be assigned to variables in **`unique`** when both `CTO` and `Leaders` go out of scope. This does not break **`unique`**'s invariants as any two variables in the same child domain may not alias each other and may also not alias any variable in **`unique`**.

In the following example,

```
let f : Person#B =
      domain A extend B in { new Person#A }
```

the expression within the domain block has type `Person#A`. As `A` goes out of scope at the end of the block, this value is returned, but its type is changed to `Person#B`. The whole domain expression is well-typed if this resulting type is valid. For instance,

```
let f : Person#Sh B =
  domain Sh A extend B in { new Person#Sh A }
```

only works if `Sh B` is a legal domain.

The value returned from a domain extension expression can also be a tuple or any class type with domain parameters (Section 3.1), allowing more than one value of the domain going out of scope to be placed back up the hierarchy.

In the code example below, linked domains `A` and `B` are declared and used for aliasing (line †). At the end of the **`domain`** expression's body, the variables `x` and `z` are recovered to **`unique`** by returning a tuple from the **`domain`** expression, and all occurrences of `A` in the return type are cast to **`unique`**.

```
Person#unique p = new Person;
-- the return value will be recovered from
-- type (Person#A, Person#A) to
-- type (Person#unique, Person#unique):
(a,b) : (Person#unique, Person#unique) =
  domain (A, B) extend unique in {
    let x : Person#A = null,
        y : Person#B = null,
        z : Person#A = new Person in {
      (x, y) = move p; †
      x.setName('Alice');
      (move x, move z) -- return two Persons,
                       -- they will be recovered
    }
  }
```

Recovering allows to express in code that an object is aliased during one phase of its lifetime, but not later.

Recovering here is similar to recovering uniqueness after borrowing in external uniqueness [12], though recovery is more fine-grained.

3. From May-Not-Alias to Object Disjointness

The disjointness domains invariants that we presented in Section 2 are about shallow aliasing – they say nothing about state reachable from two references. But knowing that two objects `x` and `y` are *disjoint* (meaning there is no pair of access paths $\texttt{x.f}_1\texttt{..f}_i$ and $\texttt{y.g}_1\texttt{..g}_j$ that alias) is useful: for example in order to know whether two method calls can be parallelised and in order to conclude that one object's invariants can not be broken by accesses to the other object.

Classes and methods are parameterised by domains (Section 3.1) and can only use domains they get as parameters. Passing domains as parameters enables static, local reasoning about the disjointness of objects (Section 3.2).

3.1 Declaring Class Aliasing using Domain Parameters

In order to modularly reason about whether two objects may contain shared state we need to know what domains the fields of the objects (and of reachable objects) are in. Domain parameters on classes tell us exactly that: classes can only use domains that they have been *parameterised with*.

Disjointness domains can be passed through a program using parameters on classes and methods. The formal parameters are always phrased in terms of domain relations, *e.g.,* `(Leader, Members) extend U` as in the code below. This domain relation allows methods to assign (Section 2.4) into the `Leader` and `Member` domains. If the parameters are at the class level, the class can put its fields in domains `Leader` and `Members`.

```
class Team[(Leader,Members) extend U] {
  -- can only use unique, Leader, Members domains
  teamName     : String#unique;
  teamLeader   : Person#Leader;
  programmer1  : Person#Members;
  programmer2  : Person#Members;
}
```

Similar to `domain` expressions, if a parameter is strong, only the strong domain is legal, if a parameter is shared, the strong and weak domains of the same name are also legal. In addition, it is possible to bind weak domains to strong domain parameters, *e.g.,* type `Team[(Wk A, B)]` makes sense. In this case, the class will treat references stored in its fields as if they *were* in strong domains and hence cannot have fields in the `Sh A` domain. Code that uses such an object will however see the fields as being in a weak domain and therefore has the ability to create aliases that the class itself couldn't. Section 3.2 applies this knowledge in practice.

In this system, a class that has no parameters at all maintains that all of its objects are encapsulated: as the class internally can only use the `unique` domain, no aliasing from the outside is possible.

As a design tradeoff, this system does not currently allow the binding of shared domains to strong parameters since allowing shared domains to bind to strong parameters would make parallelism more complex and perhaps surprising: because `this.programmer1` and `this.programmer2` may in fact alias when a shared domain is bound to `Members`, additional features like silently converting parallel calls on the two fields to sequential calls would be required.

The advantage of allowing shared domains to strong parameters would be increased reuse: *e.g.,* a `Team` with a shared domain bound to its `Members` domain parameter could then also have the same person working both the `programmer1` and `programmer2` jobs – while the `Team` guarantees, by only requiring strong parameters, that the `Team` *will not introduce such aliasing itself*. Similarly, the list presented later in this paper could then bind a shared domain to its `Data` parameter to be able to contain the same element several times.

3.1.1 Domain Extension in Classes

Class declarations may introduce internal domains for their fields by extending any domain visible to them. This is straightforward and enables a simple form of ownership types [13], as domain names introduced internal to a class are not visible externally. Values created in external domains can be moved into internal domains, but recovering such values requires tracking when the object becomes garbage. While this is possible, this extension is left for a separate paper. Furthermore, we omit the discussion of domain extensions internal to a class, as these can in part be modelled using class parameters (see Section 5.1).

3.2 Disjointness of Objects

Propagating domains through classes as parameters (Section 3.1) seamlessly allows modular reasoning about the worst-case aliasing of an object (a conservative approximation), independent of its implementation:

If no domain that an object uses may alias a domain the other object uses, the objects are guaranteed to be disjoint.

The invariants from Section 2 are still the same, but they now seamlessly apply to object disjointness.

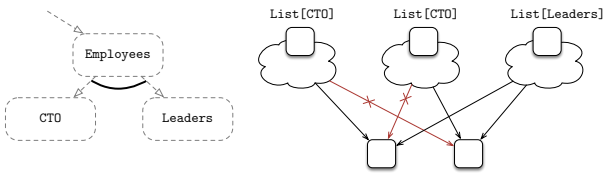


Figure 7. The two objects of type `List[CTO]` are always disjoint, but a `List[CTO]` and a `List[Leaders]` may share reachable state, as `CTO` and `Leaders` are linked and thus may alias.

Consider a hypothetical list implementation that has a strong domain parameter called `Data` that the nodes' fields that contain references to the elements are put in. In Figure 7, let two lists bind `CTO` to their `Data` parameter (putting all their element fields in the `CTO` domain) and the third list binds `Leaders` to its `Data` parameter. Just locally analysing the parameters allows concluding that the first two lists must be disjoint, but the third list may share elements with any or both of the first two lists.

4. Recipes for Local Aliasing Invariants

Disjointness domains allow aliasing properties to be expressed in a tractable way, lending itself to powerful may-alias reasoning (Section 2) and disjointness reasoning (Section 3). The table in this short section gives an intuition about how common idioms of aliasing are expressed statically using disjointness domains. The code in Section 5.1 applies some of these recipes.

Local invariant	Recipe
All the elements in a list are mutually unique	Put element fields in same strong domain `E`
All the elements in list are globally unique	Put element fields in the **`unique`** domain
All `next`-fields refer to different links	Put fields in same strong domain `D`
In addition to above, add pointer to last list element	Put `last`-field in domain linked with `D`
In addition to above, add an iterator	Make `D` above a weak domain
Share elements across two lists	Use types `List[D1]` and `List[D2]` with linked `D1`, `D2`
Reference in variable `x` in domain `D` is aliased for a scope S	Extend `D` to linked `D'`, `D''` for S and move `x` there. When S exits, recover `x`.
Send a temporary alias as argument to method m	Use a domain parameter D to m which will go out scope as the method exits
In addition to above, keep value in-place	Make D shared

5. Disjointness for Parallelism

Disjointness domains give weaker invariants than global uniqueness. These weaker invariants are still strong enough to give useful guarantees about aliasing. To demonstrate a practical use case of disjointness domains, we show how to exploit its invariants to give safe (data-race free and deterministic) parallelism.

If two expressions only access disjoint data, they are safe to run in parallel. The expression **`letpar`** `x=`e_1`||y=`e_2 **`in`** e_3 runs e_1 and e_2 in parallel and runs e_3 when both have finished. Analysing whether e_1 and e_2 can run in parallel is a local check that the variables[4] used in e_1 are unaliased with the variables used in e_2 and that all objects reachable from variables in e_1 are disjoint from all objects reachable from variables in e_2.

Applying the may-alias reasoning from Section 2 and the disjointness reasoning from Section 3.2, this check is straightforward; Figure 8 shows an expression we might use in the running example.

```
letpar
  score1 = programmer1.performanceScore() ||
  score2 = programmer2.performanceScore()
in if (score1 > score2) {
     employeeOfMonth = programmer1
   } else {
     employeeOfMonth = programmer2
   }
```

Figure 8. Scoring programmers in parallel.

Even though `programmer1` and `programmer2` may both be aliased with the previously declared `employeeOfMonth` variable, they may not be aliased with each other as they are both in the same weak domain `Wk Rest` (see Inv-Wk-Int). Also, state reachable from the `Person` object referenced by `programmer1` can not overlap with state reachable from the `Person` object referenced by `programmer2` as the person class is encapsulated (Section 3.1) and the `.performanceScore()` method calls may therefore be parallelised. For parallelism of the two calls, the `employeeOfMonth` variable is *innocuous* and the type system recognises this.

5.1 Example: Doubly Linked List

We now use disjointness domains to express the necessary aliasing properties for a doubly-linked list that supports both external iterators and a deterministic parallel internal iterator (a `foreach` method that takes a function as an argument, which is then applied to all elements). Figure 9 shows an instance of a doubly linked list and an external iterator whose `cursor` field stores a reference to one of the list's nodes.

A doubly-linked list is an interesting use case as its forward- and backward-links alias. The iterator further needs to alias the forward links in order to iterate over the list and to alias the elements variables (a destructive read in each case

[4] The semantics requires that no access paths of the form `x.f1...fn` are in either e_1 or e_2 (for paths, temporary variables can be used). Allowing such paths in the expressions would require reasoning about race freedom to become more complex, as we'd also need to make sure that no *prefix* of any path is reachable from the other expression.

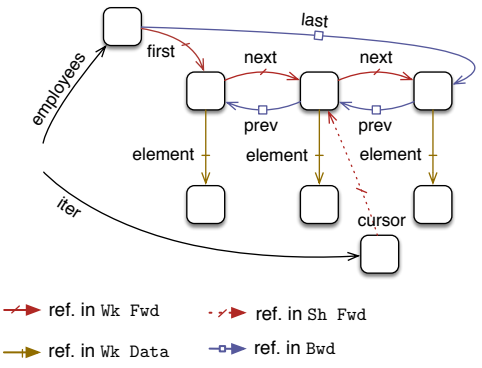


Figure 9. Object graph from the code in Figures 10 and 11 with an iterator. Decorations on the arrows denote domain membership.

would destroy the list structure). These two cases of aliasing cover *aliasing within* a data structure, and external *aliasing into* a data structure plus the storing of borrowed variables on the heap, namely the `cursor` field of the iterator.

5.1.1 The List Class

A doubly-linked list class (Figure 10) uses the linked domains `Fwd` and `Bwd` for the `nxt` and `prv` fields, respectively. As a node in a doubly linked list always has exactly two variables referring to it (one in forward- and one in backward-direction), linking the `Fwd` and `Bwd` domains allows creating these aliases.

```
class List[(Fwd,Bwd) extend U, Data extend U] {
  fst : DLink[(Fwd,Bwd),Data]#Fwd; -- †
  lst : DLink[(Fwd,Bwd),Data]#Bwd;

  def @U add(elt : Person#Data) {
    let
      newL = new DLink[(Fwd,Bwd),Data]
    in {
      if null == this.fst then {
        -- create aliases in Fwd and Bwd domain:
        (this.fst, this.lst) = move newL; -- ‡
        this.fst.elt = move elt
      } else {
        ...
      }
    }
  }
```

Figure 10. Excerpt of a doubly-linked list class.

The `add` method in Figure 10 creates a new link and stores it in a temporary variable in the **`unique`** domain. Linked assignment to two variables introduces the aliases for the forward links and backward links. The `add` method declares, using the `@U` annotation, the type of `this` to be in the `U` domain which is bound to the parent domain of the domains `Fwd` and `Bwd` are bound to. This must hold at the call-site and guarantees, internal to the method, that `this` and (for instance) `this.nxt` are not aliases, and hence that the list is not cyclic when the method is running. When not explicitly given, the annotation will default to **`unique`** as the domain of `this`.

The `this` variable is immutable and therefore the reference cannot be moved (moving the reference would reassign `this` to `null`). Consequently, syntactic sugar can reinstate the variable a method was called on after the method call returns, as if the method would have returned the `this` variable.

```
class DLink[(Fwd,Bwd) extend U, Data extend U] {
  nxt : DLink[(Fwd,Bwd),Data]#Fwd;
  prv : DLink[(Fwd,Bwd),Data]#Bwd;
  elt : Person#Data;
  def @Fwd foreach(f: [D](Person#D)->Person#D) {
    -- move from fields to stack vars, s.t. the
    -- type checker can prove disjointness
    let elt = move this.elt,
        nxt = move this.nxt
    in {
      letpar
        -- call the function with the element:
        newPerson = f(move elt) ||
        -- in parallel, map over rest of list:
        y = if null == nxt
              then null
              else nxt.foreach(f)
      in {
        -- put back the element:
        this.elt = move newPerson;
        this.nxt = move nxt;
      }
    }
  }
```

Figure 11. Excerpt of a link class for the doubly-linked list. The type of the `f` parameter, `[D](Person#D) -> Person#D` describes a lambda function that receives an `Person#D` argument in some polymorphic `D` domain, and returns the same type.

5.1.2 The Link Class

Figure 11 shows the Link class. Similarly to the list, link objects require linked `Fwd`, `Bwd` and `Data` parameters. Whereas the list only contained fields in the first two domains, the link also directly uses the `Data` parameter by putting its element `elt` field into that domain.

For the parallel `foreach` method, the **`letpar`** expression applies the lambda function to the element and calls `foreach` on the rest of the list in parallel.

For **`letpar`** to be safe (Section 5), a) all of the variables in the left expression must be unaliased with all of the variables in the right expression and b) all objects reachable from the variables in the left expression must be disjoint from all objects reachable from the right expression. If this was not the case, data races would be possible. Lambda functions can not capture any context so they may, as an exemption[5], occur in both the left-hand and right-hand side expressions

[5] Immutable data and primitive values could be similarly exempt, but we don't treat them in this paper

```
class Iterator[(Sh Fwd, Bwd) extend U,
               Sh Data       extend U] {
  cursor : DLink[(Wk Fwd,Bwd), Wk Data]#Sh Fwd;
  def init(fst :
        DLink[(Wk Fwd,Bwd), Wk Data]#Sh Fwd) {
    this.cursor = fst
  }

  def get() : Person#Sh Data {
    -- the elt field is in a weak domain,
    -- so we can copy the element:
    this.cursor.elt
  }

  def step() : bool {
    if this.hasNext() then {
      this.cursor = this.cursor.nxt;
      true
    } else {
      false
    }
  }
  ...
```

Figure 12. The iterator code for the doubly-linked list.

(as the variable `f` does). This leaves the variable `elt` in the left expression and `nxt` in the right expression (with types `Person#Data` and `DLink[(Fwd, Bwd)]#Fwd`).

The variables `nxt` and `elt` are a) unaliased as the domains `Data` and `Fwd` are not linked, as required by the class (the class requires `Fwd`, `Bwd` and `Data` to have the same parent and domains bound to these parameters must satisfy this constraint); `nxt` and `elt` are also b) disjoint, as the class type `Person` does not have any domain parameters and is therefore encapsulated.

5.1.3 The Iterator Class

Neither the list nor the link classes need shared fields for their implementation. However, the iterator in Figure 12 needs to be able to copy, non-destructively, the `nxt` field of links.

By making the `Fwd` and `Bwd` parameters strong, the list class in Section 5.1.1 denoted the strongest aliasing semantics that the *list's methods* can work with, while other code may reuse the class with weak domains bound to those strong parameters. The iterator in Section 5.1.3 reuses the list in such a way.

In order to iterate non-destructively, the iterator explicitly only accepts lists whose `Fwd` and `Data` domain parameters are bound to weak domains. That allows the iterator to copy the references to the list's nodes into its `cursor` field and returning elements without destroying the `elt` field without the list changing its interface to a more permissive one using shared parameters. A programmer can therefore first write a list implementation that only requires strong domains – without having to anticipate how the list will later be used – but then *still* reuse the same implementation with weak domains bound to the domain parameters to also get iterators.

Now we can use the list and iterators in our running example. Using the domains we have set up in Section 2.3.3, and using syntactic desugaring of the expression $(e_1 \parallel e_2)$ to `letpar x =` e_1 `|| y =` e_2 `in null`:

```
...
domain (Sh Fwd, Bwd) extend unique in {
  let
    programmers = new List[(Wk Fwd, Bwd), Wk Rest];
    iter =  new Iterator[(Sh Fwd, Bwd), Sh Rest];
  in {
    employees.add(new Person);
    -- can copy fst, because it's in Wk Fwd:
    iter.init(employees.fst);
    -- employee1 has inferred type Person#Sh Empl:
    let employee1 = iter.get() in {
      -- parallel access is racy:
      (employee1.modify() ||
         employees.map(...)); -- ERROR †
      -- sequential access is safe:
      (employee1.modify();
         employees.map(...))  -- COMPILES
    }
  }
}
```

The example uses straightforward type inference in the let expressions (we do not annotate the types of the variable bindings). We create the necessary domains from `unique`, making sure to link `Fwd` and `Bwd` but not `Data` as the list's parameters require. The race in line † and below rightfully fails to compile because that `employee1` is disjoint from `employees` cannot be derived.

6. Semantics of Disjointness Domains

The core idea of disjointness domains is simple. Variables (and fields) are grouped into sets which have aliasing invariants attached to them. The sets are propagated through the program to maintain these invariants, and allow local reasoning about aliasing. Relations between domains capture how variables in one may be assigned from variables in another allowing cheap reasoning about the possible aliasing of variables in a source location.

Shared, weak and strong domains allow proving increasingly strong properties about the aliasing in a program, and also capture patterns of aliasing. Strong domains express a form of ownership: a variable *owns* a certain share of the object it refers to. A variable in `unique` is the *sole owner* of the object it refers to. Weak domains are a weaker form of ownership: variables also own a share of the object they refer to, but they explicitly allow borrowing. Variables in shared domains are borrowed references that do not encode any ownership (after all, there can be an unlimited number of shared aliases) and are useful, for example, to keep track of positions in larger data structures like in the iterator.

In this section, we give a formal account of our design in the context of the Java-like language Gift. In particular, we show how domain information is propagated, used to reason about domain-relations and aliasing; we finally show how

this information can be used to reason about non-interference of expressions to enable static checking of their safe parallelisation.

Apart from destructive reads, there is nothing special in Gift that is required for disjointness domains to be soundly implemented. For clarity, we use explicit move and copy keywords for assignment with and without movement semantics.

A full table of the judgements is in Appendix B.

6.1 Core Syntax

Table 1 contains the syntax of the core language that we will use in this section.

Δ	::=	$\epsilon \mid \Delta, R$	Relation environment
R	::=	DT extend d	Domain relation
DT	::=	(D_1, D_2)	Domain tuple of linked domains
D	::=	$K\ d$	Domain
K	::=	St | Wk | Sh	Dom. kind: strong/weak/shared
d	::=	dn | `unique`	Domain: user def. or `unique`
T	::=	$T_v \mid \ldots$	Type: view or func (omitted)
T_v	::=	$T_o \# D$	View type
T_o	::=	$c[\overline{DT_i}^{\,i}]$	Object type
V	::=	$x : T$	Variable binding
e	::=		Expression
		$\mid e_a$	Access expression
		$\mid (V_1, V_2) = e_1$ in e_2	Linked assignment
		$\mid$ letpar $V_1 = e_1 \parallel V_2 = e_2$ in e_3	Parallel `let`
		$\mid$ domain (D_1, D_2) extend d in e	Dom. extension
e_a	::=	`move` $p \mid$ `copy` $p \mid p$	Access expression
p	::=	$x \mid$ `this` $\mid p.f \mid v$	Access path

Table 1. Core syntax. Meta variables: x, y, f, g for variable- and field-names, c for class names, dn for user-defined domain names.

A *relation environment* Δ contains *domain relations*. A domain relation introduces a *domain tuple* (two linked sibling domains) with two domains that extend a parent domain name. The parent is a domain name, as the parent's domain kind does not matter. The formalism handles only 2-tuples of new domains WLOG: when only one domain is desired, the other domain may be ignored, when more than two linked domains are desired, we can use 2-tuples to construct them, *e.g.,* a construction with the same aliasing semantics as `domain (A,B,C,D) extend X in {...}` looks like this:

```
domain (A, Tmp) extend X in {
  domain (B, Tmp') extend Tmp in {
    domain (C, D) extend Tmp' in { ... }}}
```

A *view type* T_v is the local view of an object used for variables. It is an object-type T_o decorated with the domain of the variable and there may be aliases referring to the same object with different view types, but they will all have the same object type T_o.

A *domain* is a domain kind (either `St`, `Wk`, or `Sh`) and a domain name.

6.2 Well-Formedness

We present well-formedness judgement by judgement, with an explanation following each of them. The static variable environment $\mathcal{E}$ is a standard map from variables to types and we omit its definition. We further omit some judgements (like well-formed programs, well-formed class definitions, and well-formed method definitions) for not being necessary to understand disjointness domains.

$\boxed{\vdash \Delta}$ *(Well-formed relation environment)*

$$\textsc{wf-renv-empty}\ \frac{}{\vdash \epsilon} \qquad \textsc{wf-renv-ext}\ \frac{\vdash \Delta \qquad K_1, K_2 \in \{\mathsf{St}, \mathsf{Sh}\} \qquad d_1 \neq d_2 \qquad d_1, d_2 \notin \Delta}{\vdash \Delta, (K_1\ d_1, K_2\ d_2)\ \mathsf{extend}\ d}$$

A well-formed relation environment Δ contains domain relations of the form DT extend d as they are introduced by either `domain` expressions or by domain-parameters. The domain kinds of the linked domains DT are either St (strong) or Sh (shared), reflecting how domains are created (see Section 2.3.3, running example). An empty relation environment is well-formed (rule WF-RENV-EMPTY). Extending a relation environment is only well-formed if none of the newly introduced domain names already exist (rule WF-RENV-EXT).

$\boxed{\Delta \vdash d}$ *(Well-formed domain names)*

$$\textsc{wf-dname-uq}\ \frac{\vdash \Delta}{\Delta \vdash \texttt{unique}} \qquad \textsc{wf-dname-lkp}\ \frac{(K_1\ d_1, K_2\ d_2)\ \mathsf{extend}\ d'' \in \Delta \qquad d \in \{d_1, d_2\} \qquad \vdash \Delta}{\Delta \vdash d}$$

According to rule WF-DNAME-UQ, the domain name `unique` is well-formed in any well-formed relation environment Δ. Other domain names are well-formed if and only if the relation environment contains a relation that introduces the domain name.

$\boxed{\Delta \vdash D}$ *(Well-formed domains)*

$$\textsc{wf-d-lkp}\ \frac{(D_1, D_2)\ \mathsf{extend}\ d \in \Delta \qquad \vdash \Delta \qquad D \in \{D_1, D_2\}}{\Delta \vdash D} \qquad \textsc{wf-d-st}\ \frac{\Delta \vdash \mathsf{Sh}\ d}{\Delta \vdash \mathsf{St}\ d} \qquad \textsc{wf-d-wk}\ \frac{\Delta \vdash \mathsf{Sh}\ d}{\Delta \vdash \mathsf{Wk}\ d}$$

A domain D is well-formed if it is found in the well-formed domain relation environment (rule WF-D-LKP). If a shared domain is well-formed, it also implies that its strong and weak counterparts are well-formed (rules WF-D-ST and WF-D-WK); `unique` is well-formed in any well-formed Δ (rule not shown).

6.3 Aliasing

In this section, we formalise reasoning about aliasing and object disjointness using disjointness domains. We start with simple building blocks related to the hierarchy of domains and finish with object disjointness, $\mathcal{E}; \Delta \vdash x_1 \perp\!\!\!\perp x_2$.

6.3.1 Domain Extension

$\boxed{\Delta \vdash d_1 \prec / \prec^* d_2,}$ *(Domain name extension)*

A-EXDIRECT-LOOKUP
$$\frac{(K'\ d', K''\ d'') \,\mathsf{extend}\, d_2 \in \Delta \quad d_1 \in \{d', d''\}}{\Delta \vdash d_1 \prec d_2}$$

A-EXTR-DIRECT
$$\frac{\Delta \vdash d_1 \prec d_2}{\Delta \vdash d_1 \prec^* d_2}$$

A-EXTR-TR
$$\frac{\Delta \vdash d_1 \prec^* d' \quad \Delta \vdash d' \prec^* d_2}{\Delta \vdash d_1 \prec^* d_2}$$

A-EXTR-UQ
$$\frac{\Delta \vdash d_1}{\Delta \vdash d_1 \prec^* \texttt{unique}}$$

The extension-relations (direct extension $\prec$ and transitive extension $\prec^*$) express the partial order in which domain names were introduced. This is used to reason about what domains may contain aliases.

6.3.2 Linked Domains

$\boxed{\Delta \vdash \mathsf{not\text{-}linked}(d_1, d_2)}$ *(Domain names are not linked)*

A-NOTLINKED-SIBLING
$$\frac{\vdash \Delta \quad (D_1, D_2)\,\mathsf{extend}\, d \in \Delta \quad (D_1', D_2')\,\mathsf{extend}\, d \in \Delta \quad (D_1, D_2) \neq (D_1', D_2') \quad K_1\ d_1 \in \{D_1, D_2\} \quad K_2\ d_2 \in \{D_1', D_2'\}}{\Delta \vdash \mathsf{not\text{-}linked}(d_1, d_2)}$$

A-NOTLINKED-CHILD
$$\frac{\Delta \vdash d_1 \prec d_1' \quad \Delta \vdash \mathsf{not\text{-}linked}(d_1', d_2)}{\Delta \vdash \mathsf{not\text{-}linked}(d_1, d_2)}$$

A-NOTLINKED-SYMM
$$\frac{\Delta \vdash \mathsf{not\text{-}linked}(d_2, d_1)}{\Delta \vdash \mathsf{not\text{-}linked}(d_1, d_2)}$$

Rule A-NOTLINKED-SIBLING expresses that two domain names d_1, d_2 are *not linked* (linked domains can not alias, as explained in Section 2.2.2) if they directly extend the same domain name d but are not created together in the same domain tuple. Rule A-NOTLINKED-CHILD expresses that if two domains d_1' and d_2 are not linked, then d_1''s child d_1 and d_2 are also not linked. The relation is symmetric by rule A-NOTLINKED-SYMM.

6.3.3 Reference Flow in Assignment

$\boxed{\Delta \vdash D_1 \xrightarrow{\text{copy}} / \xrightarrow{\text{move}} / \xrightarrow{\text{linked}} D_2}$ *(Reference flow)*

A-CANMOVE-STWK
$$\frac{K, K' \in \{\mathsf{St}, \mathsf{Wk}\} \quad K' = \mathsf{St} \implies K = \mathsf{St} \quad \Delta \vdash K'\ d}{\Delta \vdash K\ d \xrightarrow{\text{move}} K'\ d}$$

A-CANCOPY-S
$$\frac{\Delta \vdash \mathsf{Sh}\ d \quad K \in \{\mathsf{Wk}, \mathsf{Sh}\}}{\Delta \vdash K\ d \xrightarrow{\text{copy}} \mathsf{Sh}\ d}$$

A-CANCOPY-UP
$$\frac{\Delta \vdash d \prec^* d' \quad \Delta \vdash K\ d \quad \Delta \vdash \mathsf{Sh}\ d'}{\Delta \vdash K\ d \xrightarrow{\text{copy}} \mathsf{Sh}\ d'}$$

A-CANLINKED-DIRECT
$$\frac{K \in \{\mathsf{St}, \mathsf{Wk}\} \quad \Delta \vdash K\ d \quad (K_1\ d_1, K_2\ d_2)\,\mathsf{extend}\, d \in \Delta}{\Delta \vdash K\ d \xrightarrow{\text{linked}} (\mathsf{St}\ d_1, \mathsf{St}\ d_2)}$$

As explained in Section 2.4, assignment of references from variable to variable is governed by the domains the target and the source variable are in. Assigning using the `move` operator from a variable in domain D_1 to a variable in domain D_2 is only possible if $\Delta \vdash D_1 \xrightarrow{\text{move}} D_2$ can be derived.

The rule A-CANMOVE-STWK allows a programmer to move from a strong domain to its weak counterpart, or from a strong domain to itself or from a weak domain to itself. Moving down the hierarchy is omitted w.l.o.g, as linked assignment allows moving down the hierarchy.

Rule A-CANCOPY-S allows her to copy from a weak or shared domain to a shared domain with the same name. The rule A-CANCOPY-UP allows her to copy from a domain to a shared domain that is an *ancestor*, as required by the $\prec^*$ relation. Having this rule forces us to recover strong child domains only to *weak* parent domains if there exists a weak parent domain: an alias of a variable in a strong child domain could have been created in the shared parent domain. Recovering the child variable to a variable in the strong parent domain would be unsound as now there would be aliases in the strong and shared domains of the same name.

Last, rule A-CANLINKED-DIRECT allows her to execute a linked assignment down the hierarchy. Similarly, if two domains are created in the same expression, linked-assignment from `unique` to linked domains is always possible (rule not shown).

We omit convenience rules that exist solely to prevent temporary variables, *e.g.,* the rule that makes the $\xrightarrow{\text{move}}$ relation transitive, the rule that allows a move to either domain in a domain tuple if the linked assignment to the domain tuple is allowed ("moving down the hierarchy") and the rule that allows linked assignments also to weak domains down the hierarchy.

6.3.4 May-Not-Alias for Domains

$\boxed{\Delta \vdash D_1 \perp D_2}$ *(Domains may not alias)*

A-DMNA-SAMENAME
$$\frac{K_1 = \mathsf{St} \vee (K_1 \neq \mathsf{Sh} \wedge K_2 \neq \mathsf{Sh}) \quad \Delta \vdash K_1\ d \quad \Delta \vdash K_2\ d}{\Delta \vdash K_1\ d \perp K_2\ d}$$

A-DMNA-SYMM
$$\frac{\Delta \vdash D_2 \perp D_1}{\Delta \vdash D_1 \perp D_2}$$

A-DMNA-NOTLINKED
$$\frac{\Delta \vdash \mathsf{not\text{-}linked}(d_1, d_2) \quad \Delta \vdash K_1\ d_1 \quad \Delta \vdash K_2\ d_2}{\Delta \vdash K_1\ d_1 \perp K_2\ d_2}$$

A-DMNA-ANCESTOR
$$\frac{\Delta \vdash d_2 \prec^* d_1 \quad K_1 \in \{\mathsf{St}, \mathsf{Wk}\} \quad \Delta \vdash K_1\ d_1 \quad \Delta \vdash K_2\ d_2}{\Delta \vdash K_1\ d_1 \perp K_2\ d_2}$$

The judgement $\Delta \vdash D_1 \perp D_2$ means that two domains may not be aliased.

By rule A-DMNA-SAMENAME, domains with the same name but possibly different domain kinds may not alias if a) one domain kind is strong or b) none is shared. See Inv-St-Int.

By rule A-DMNA-SYMM, the relation is symmetric.

By rule A-DMNA-NOTLINKED, unlinked domains may not alias. This is so as no reference can have been stored in two variables in both unlinked subhierarchys – it could only have been moved into one of the two. See Inv-St-Btw, Inv-Wk-Int.

If two domains are related by extension (transitive), rule A-DMNA-ANCESTOR expresses that if the ancestor domain $K_1\ d_1$ is strong or weak, that is proof that the reference has not been moved down in the hierarchy yet – therefore, there can't be aliasing. See Inv-St-Btw, Inv-Wk-Btw1.

The invariant Inv-Wk-Btw2 says that a weak domain `Wk D` and an equally-named shared domain `Sh D` may alias. The rules therefore do *not* allow deriving $\Delta \vdash \mathsf{Wk}\ d \perp \mathsf{Sh}\ d$. Similarly for Inv-Sh-Int (any two variables in a shared domain may alias), Inv-Sh-Btw1 (if a shared domain is an ancestor of a domain, they may alias) and Inv-Sh-Btw2 (if a shared domain and any other domain are linked, they may alias).

6.3.5 Object Disjointness

$\boxed{\mathcal{E}; \Delta \vdash x_1 \perp\!\!\!\perp x_2}$ *(Object disjointness)*

A-DISJOINT-CANNOTSHARE
$$\frac{\mathcal{E}; \Delta \vdash x_1 : T_1 \qquad \mathcal{E}; \Delta \vdash x_2 : T_2 \quad \forall D_1 \in \mathsf{domains}(T_1), D_2 \in \mathsf{domains}(T_2).\ \Delta \vdash D_1 \perp D_2}{\mathcal{E}; \Delta \vdash x_1 \perp\!\!\!\perp x_2}$$

The objects referred to by two stack variables x_1, x_2 are *disjoint* if there are no two paths $x_1.f_1 \ldots f_i$ and $x_2.g_1 \ldots g_j$ (with $i, j \geq 0$) that may be aliases.

The domains helper function returns a conservative (maximally aliased) set of all the domains that a type may use for fields and the domain of the original variable; for view-types $T_v ::= T_o\#D$, these are the domain parameters of the object type T_o and the domain D, while for a function type it is an empty set (reflecting that functions can not capture any state and are freely sharable).

For $i = 0 \vee j = 0$, A-DISJOINT-CANNOTSHARE requires that the two variables may not alias directly (and that neither is reachable *from* the other) by using the domains-function that also returns the domain D of the original variable.

For $i > 0 \wedge j > 0$, A-DISJOINT-CANNOTSHARE looks at a conservative approximation of the domain parameters D_1 that the x_1 object can use for its fields and all the domains D_2 that the x_2 object can use: if there is no domain from x_1 that may alias any domain in x_2 and vice versa, then there cannot be any field in the aggregate after x_1 that reaches the object at the end of x_2 or any object in the aggregate after it (or vice versa).

For instance, when parallelising `foreach` in Section 5.1.2: $\mathsf{domains}(\texttt{DLink[(Fwd,Bwd),Data]\#Fwd}) = \{\texttt{Data}, \texttt{Fwd}, \texttt{Bwd}\}$, $\mathsf{domains}(\texttt{Person\#Data}) = \{\texttt{Data}\}$. Running an expression involving such a `DLink` object in parallel with an expression involving such an `Person` object is allowed as all pairs of domains may not alias:

$\Delta \vdash \texttt{Data} \perp \texttt{Data}$ (by rule A-DMNA-SAMENAME)
$\Delta \vdash \texttt{Fwd} \perp \texttt{Data}$ (by rule A-DMNA-NOTLINKED)
$\Delta \vdash \texttt{Bwd} \perp \texttt{Data}$ (by rule A-DMNA-NOTLINKED)

6.4 Well-Typedness

6.4.1 Well-Formed Type

$\boxed{\Delta \vdash T_o}$ *(Well-formed object type)*

WF-OT-MATCH
$$\frac{\Delta \vdash \mathsf{classpmatch}(T_o) \dashv \sigma}{\Delta \vdash T_o}$$

AUX-DEFPARAMS-LKP
$$\frac{\mathsf{class}\ c[\overline{R_i}^i]\{\overline{V_j}^j\ \overline{M_k}^k\} \in P}{\mathsf{defparams}(c) = \overline{R_i}^i}$$

AUX-CP-MATCH
$$\frac{\mathsf{defparams}(c) = \overline{R_i}^i \quad \Delta; \mathsf{id} \vdash \mathsf{pmatch}(\overline{DT_i}^i, \overline{R_i}^i) \dashv \sigma}{\Delta \vdash \mathsf{classpmatch}(c[\overline{DT_i}^i]) \dashv \sigma}$$

AUX-P-MATCH
$$\frac{\vdash \Delta \qquad (D_1, D_2)\ \mathsf{extend}\ d \in \Delta \quad \mathsf{canbind}(D_1', D_1) \qquad \mathsf{canbind}(D_2', D_2) \quad \sigma' = [D_1' \mapsto D_1, D_2' \mapsto D_2] \circ [\forall K.K\ d' \mapsto K\ d] \quad \Delta \backslash (D_1, D_2)\ \mathsf{extend}\ d = \Delta' \quad \Delta'; \sigma \circ \sigma' \vdash \mathsf{pmatch}(\overline{DT_j}^j, \overline{R_i'}^i) \dashv \sigma''}{\Delta; \sigma \vdash \mathsf{pmatch}((D_1, D_2)\overline{DT_j}^j, (D_1', D_2')\ \mathsf{extend}\ d'\overline{R_i'}^i) \dashv \sigma''}$$

AUX-CANBIND-KINDS
$$\frac{K_{decl} = \mathsf{Sh} \implies K_{use} = \mathsf{Sh} \quad K_{decl} = \mathsf{St} \implies (K_{use} = \mathsf{St} \vee K_{use} = \mathsf{Wk})}{\mathsf{canbind}(K_{decl}\ d_{decl}, K_{use}\ d_{use})}$$

An object type T_o is well-formed (rule WF-OT-MATCH) if we can match the class' declaration site relations to the type parameters (rule AUX-CP-MATCH). The judgement $\Delta; \sigma \vdash \mathsf{pmatch}(\overline{DT_i}^i, \overline{R_i}^i) \dashv \sigma'$ matches bound domain tuples (and their parent domains, which are implicitly present in Δ) to declaration site domain relations and returns a substitution σ' which maps declaration-site domains to use-site domains. The composition of substitutions $\sigma \circ \sigma'$ is undefined if there exists a domain D such that $\sigma(D) \neq \sigma'(D)$.

When matching parameters, we need to make sure that the declaration site domain kinds and the use site domain parameters are compatible: shared domains at declaration site *require* a shared use site domain while strong domains at declaration site can be bound to a strong *or* weak parameter, as explained in Section 3.1. The rule AUX-CANBIND-KINDS enforces this rule. By removing the matched relation from the environment and matching the rest of the parameters with Δ' ($\Delta \backslash (D_1, D_2)$ extend $d = \Delta'$), we avoid matching the same use-site domains to different parameters at declaration site. If we would not do that, we could bind the same use site domain to two declaration site domains that are not linked. Binding, for instance, the same shared domain to notlinked declaration site domains could lead to two variables in those domains being aliases when from the class' view point they must not be.

6.4.2 Well-Typed Path and Expression

$\boxed{\mathcal{E}; \Delta \vdash p : T}$ *(Well-typed path)*

WT-P-X
$$\frac{\Delta \vdash T \qquad \Delta \vdash \mathcal{E} \qquad x : T \in \mathcal{E}}{\mathcal{E}; \Delta \vdash x : T}$$

WT-P-PF
$$\frac{\Delta \vdash T_o \# D \qquad \mathcal{E}; \Delta \vdash p : T_o \# D \qquad f : T_v \in \mathsf{fields}(\Delta, T_o)}{\mathcal{E}; \Delta \vdash p.f : T_o \# D}$$

A variable's type is stored directly in the variable environment $\mathcal{E}$ (rule WT-P-X). In rule WT-P-PF, the type of a path $p.f$ relies on a field lookup: $f : T_v \# D \in \mathsf{fields}(\Delta, T_o)$. The field lookup simply substitutes the domains bound for an object's declared parameters, similarly to type substitution in *e.g.,* Java's generics.

$\boxed{\mathcal{E}; \Delta \vdash e : T}$ *(Well-typed expression)*

WT-E-MOVE
$$\frac{p \neq \texttt{this} \qquad \mathcal{E}; \Delta \vdash p : T_o \# D' \qquad \Delta \vdash D' \xrightarrow{\text{move}} D}{\mathcal{E}; \Delta \vdash \texttt{move}\ p : T_o \# D}$$

WT-E-COPY
$$\frac{\mathcal{E}; \Delta \vdash p : T_o \# D' \qquad \Delta \vdash D' \xrightarrow{\text{copy}} D}{\mathcal{E}; \Delta \vdash \texttt{copy}\ p : T_o \# D}$$

WT-E-LINKED
$$\frac{\mathcal{E}; \Delta \vdash e_1 : T_o \# K\ d \qquad \Delta \vdash K\ d \xrightarrow{\text{linked}} (D_1, D_2) \qquad \mathcal{E}, x_1 : T_o \# D_1, x_2 : T_o \# D_2; \Delta \vdash e_2 : T}{\mathcal{E}; \Delta \vdash (x_1 : T_o \# D_1, x_2 : T_o \# D_2) = e_1 \text{ in } e_2 : T}$$

WT-E-LETPAR
$$\frac{\mathsf{starts}(e_1) = [\overline{x_i}^i] \qquad \mathsf{starts}(e_2) = [\overline{y_j}^j] \qquad \forall x \in \overline{x_i}^i, y \in \overline{y_j}^j.\ \mathcal{E}; \Delta \vdash x \perp\!\!\!\perp y \qquad \mathcal{E}; \Delta \vdash e_1 : T_1 \qquad \mathcal{E}; \Delta \vdash e_2 : T_2 \qquad \mathcal{E}, x : T_1, y : T_2; \Delta \vdash e_3 : T}{\mathcal{E}; \Delta \vdash \mathsf{letpar}\ x : T_1 = e_1 \parallel y : T_2 = e_2 \text{ in } e_3 : T}$$

WT-E-DOMAIN
$$\frac{\Delta \vdash d' \qquad \mathcal{E}; \Delta, (K_1\ d_1, K_2\ d_2) \text{ extend } d' \vdash e : T' \qquad \vdash \Delta, (K_1\ d_1, K_2\ d_2) \text{ extend } d' \qquad T = \mathsf{recover}_{d_1 \mapsto d'}(T') \qquad \Delta \vdash T}{\mathcal{E}; \Delta \vdash \mathsf{domain}(K_1\ d_1, K_2\ d_2) \text{ extend } d' \text{ in } e : T}$$

Rules WT-E-MOVE, WT-E-COPY, and WT-E-LINKED use the $\xrightarrow{\text{move}}$, $\xrightarrow{\text{copy}}$, $\xrightarrow{\text{linked}}$ judgements. They constrain reference-flow between variables based on the domains they are in and were described in detail in Section 6.3.

The rule WT-E-LETPAR looks at all pairs of stack variables x_1, x_2 where x_1 is a variable used in e_1 and x_2 is a variable used in e_2. If all the objects reachable through these variables are disjoint from each other ($\perp\!\!\!\perp$, as defined by rule A-DISJOINT-CANNOTSHARE), the two expressions can be safely run in parallel since there is no shared memory accessed by both. The starts function takes an expression and returns all the paths that are used in the expression. We require all of these paths to be variables.

The rule WT-E-DOMAIN introduces a new domain relation and makes it available to its inner expression e. We arbitrarily choose the first child domain and recover it in the return type by replacing it with the parent domain. The $\mathsf{recover}_{d_1 \mapsto d'}$ function is the substitution $[\mathsf{St}\ d_1 \mapsto \mathsf{Wk}\ d', \mathsf{Wk}\ d_1 \mapsto \mathsf{Wk}\ d', \mathsf{Sh}\ d_1 \mapsto \mathsf{Sh}\ d']$ if $\mathsf{Sh}\ d'$ is well-formed and the substitution $[\mathsf{St}\ d_1 \mapsto \mathsf{St}\ d', \mathsf{Wk}\ d_1 \mapsto \mathsf{St}\ d']$ otherwise. In the former case, we recover from strong domains only to weak domains as an alias in the linked domain $\mathsf{Wk}\ d_2$ might have been assigned to a variable in $\mathsf{Sh}\ d'$ using rule A-CANCOPY-UP. Giving up this rule would allow to recover fully but make the system less expressive. In the latter case, if $\mathsf{Sh}\ d'$ is not well-formed, we can recover strong, as the rule A-CANCOPY-UP could not have been used for lack of a variable in $\mathsf{Sh}\ d'$. We can furthermore recover the weak domain to the strong parent as the shared variables are about to go out of scope and that means that the strong and weak child domains can be merged into one strong domain without violating any invariants.

7. Dynamic Semantics

The dynamic semantics is a small-step reduction semantics. Table 2 defines heaps $\mathcal{H}$ as maps from locations ι to objects. Objects $\mathcal{O}$ are tuples of object types T_o and a list of fields mapping to values v .

A configuration is a heap, a relation environment Δ, a stack frame $\mathcal{F}$, and an expression e – or ERROR (in case of a null pointer dereference). A frame maps variable names to

$\mathcal{H}$	::=	$\epsilon \mid \mathcal{H}, \iota \mapsto \mathcal{O}$	Heap
$\mathcal{O}$	::=	$(T_o, \overline{f_i \mapsto v_i}^{\,i})$	Dynamic Object
Γ	::=	$\epsilon \mid \iota \mapsto T_o$	Store Environment
v	::=	$\iota \# D \mid \mathsf{null}$	Stack value
$\mathcal{F}$	::=	$\epsilon \mid \mathcal{F}, x \mapsto v \mid \mathcal{F}, x \mapsto e_\lambda$	Frame
Cfg	::=	$\langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle \mid \mathsf{ERROR}$	Configuration

Table 2. Syntax extensions for the dynamic semantics and meta semantics.

stack values or to lambda expressions and we avoid name clashes by replacing all new variable names and argument names that go in the frame by replacing them by fresh names.

$\boxed{\mathit{Cfg} \hookrightarrow \mathit{Cfg}'}$ *(Reduction step, selection)*

DYN-STEP-ASSIGNVAL

$$\frac{\mathsf{fresh}\ x_1' \qquad \mathsf{fresh}\ x_2' \qquad e_2' = e_2[x_1 \mapsto x_1', x_2 \mapsto x_2']}{\begin{array}{c}\langle \mathcal{H}; \Delta; \mathcal{F}; (x_1 : T_o \# D_1, x_2 : T_o \# D_2) = \iota \# D \ \mathsf{in}\ e_2 \rangle \hookrightarrow \\ \langle \mathcal{H}; \Delta; \mathcal{F}, x_1' \mapsto \iota \# D_1, x_2' \mapsto \iota \# D_2; e_2' \rangle\end{array}}$$

DYN-STEP-LETPARLEFT

$$\frac{\langle \mathcal{H}; \Delta; \mathcal{F}; e_1 \rangle \hookrightarrow \langle \mathcal{H}'; \Delta'; \mathcal{F}'; e_1' \rangle}{\begin{array}{c}\langle \mathcal{H}; \Delta; \mathcal{F}; \mathsf{letpar}\ x : T_1 = e_1 \parallel y : T_2 = e_2 \ \mathsf{in}\ e_3 \rangle \hookrightarrow \\ \langle \mathcal{H}'; \Delta'; \mathcal{F}'; \mathsf{letpar}\ x : T_1 = e_1' \parallel y : T_2 = e_2 \ \mathsf{in}\ e_3 \rangle\end{array}}$$

Rule DYN-STEP-ASSIGNVAL creates two aliases once the right hand side expression is fully evaluated and rule DYN-STEP-LETPARLEFT evaluates the left hand side expression e_1 of a `letpar` expression. Dynamic semantics, other than destroying variables that are moved are standard and we omit most rules for brevity.

8. Meta-Theory

We prove progress and preservation. Preservation maintains a well-typed configuration. The conditions for well-typedness are mostly standard, but also require that no aliasing exists in the system outside of that allowed by the domains and their associated variables.

Store type environments Γ in Table 2 map locations to their object type T_o. Values also denote the domain they are in, used in the preservation proof.

$\boxed{\Gamma \vdash \mathit{Cfg} : T}$ *(Well-typed configuration)*

META-LOCAL-UNIQUENESS

$$\frac{\begin{array}{c}\forall \iota_i \# D_i, \iota_j \# D_j \in \mathsf{allrefs}(\mathcal{F}, \mathcal{H}). \\ i \neq j \wedge (\Delta \vdash D_i \perp D_j) \implies (\iota_i \neq \iota_j)\end{array}}{\Delta \vdash \mathsf{localuniqueness}(\mathcal{F}, \mathcal{H})}$$

META-WT-CONFIG

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash \mathcal{H} \qquad \Gamma; \Delta; \mathcal{H} \vdash \mathcal{F} \\ \Delta \vdash \mathsf{localuniqueness}(\mathcal{F}, \mathcal{H}) \\ \Gamma; \Delta; \mathcal{H}; \epsilon \vdash e : T\end{array}}{\Gamma \vdash \langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle : T}$$

The local uniqueness property demands that all the references in all variables in both heap $\mathcal{H}$ and frame $\mathcal{F}$ (a multiset, returned by the allrefs function – if N aliases to an object exist, even with the same domain, they will be present in this multiset N times) only alias if the type system allows their domains to contain aliases.

A configuration is then well-typed if both its heap and frame are well-formed (explained later in this section), its expression is well-typed (using the store environment) and the local uniqueness property holds.

$\boxed{\Gamma; \Delta; \mathcal{H}; \mathcal{E} \vdash e : T}$ *(Store typing)*

META-STYPED-REPLACELOC

$$\frac{\begin{array}{c}e' = e[\forall D\ \mathsf{s.t}\ \Delta \vdash D.\ (\iota \# D \mapsto \mathsf{fresh}\, x_{\iota \# D})] \\ \mathcal{E}' = \mathcal{E}, \forall D\ \mathsf{s.t}\ \Delta \vdash D.\ x_{\iota \# D}.T_o \# D \\ \Gamma; \Delta; \mathcal{H}; \mathcal{E}' \vdash e' : T\end{array}}{\Gamma, \iota \mapsto T_o; \Delta; \mathcal{H}; \mathcal{E} \vdash e : T}$$

META-STYPED-TRIVIAL

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash \mathcal{H} \\ \mathcal{E}; \Delta \vdash e : T\end{array}}{\Gamma; \Delta; \mathcal{H}; \mathcal{E} \vdash e : T}$$

The store typing rules META-STYPED-REPLACELOC and META-STYPED-TRIVIAL replace values by fresh variables with the same type of the expression that produced the value. This enables well-typed configuration to involve localuniqueness even though localuniqueness does not have access to the store environment Γ.

$\boxed{\Delta \vdash \Gamma}$ *(Well-formed store environment Γ)*

META-WF-STOREENV

$$\frac{\begin{array}{c}\vdash \Delta \qquad \mathsf{unambiguous}(\Gamma) \\ \forall T_o \in \mathsf{range}(\Gamma).\ \Delta \vdash T_o\end{array}}{\Delta \vdash \Gamma}$$

A store environment Γ is well-formed if each location is contained only once (the unambiguous-judgement demands that no location maps to several object types T_o). Additionally, all object types in the store environment (the range function returns the range of a map, which is) must be well-formed.

$\boxed{\Gamma; \Delta \vdash \mathcal{O}}$ *(Well-formed object $\mathcal{O}$)*

META-WF-OBJ

$$\frac{\begin{array}{c}\mathcal{O} = (T_o, \overline{f_i \mapsto v_i}^{\,i}) \qquad \Delta \vdash T_o \qquad \Delta \vdash \Gamma \\ \forall i.\ (v_i = \iota_i \implies \iota_i \in \mathsf{domain}(\Gamma))\end{array}}{\Gamma; \Delta \vdash \mathcal{O}}$$

An object $\mathcal{O}$ is well-formed, $\Gamma; \Delta \vdash \mathcal{O}$, if all of its fields are either `null` ($v_i \neq \iota_i$) or contain references to objects that actually exist in the store environment (the domain function returns the domain of a map, which is a set of locations in this case).

$\boxed{\Gamma; \Delta \vdash \mathcal{H} / \Gamma; \Delta; \mathcal{H} \vdash \mathcal{F}}$ *(Well-formed heap $\mathcal{H}$/ frame $\mathcal{F}$)*

META-WF-HEAP
$$\frac{\Delta \vdash \Gamma \qquad \mathsf{unambiguous}(\mathcal{H}) \qquad \forall \mathcal{O} \in \mathsf{range}(\mathcal{H}).\ \Gamma; \Delta \vdash \mathcal{O}}{\Gamma; \Delta \vdash \mathcal{H}}$$

META-WF-FRAME
$$\frac{\Gamma; \Delta \vdash \mathcal{H} \qquad \mathsf{unambiguous}(\mathcal{F}) \qquad \forall \iota \# D \in \mathsf{range}(\mathcal{F}).\ \iota \in \mathsf{domain}(\mathcal{H}) \wedge \mathcal{H}(\iota) = \mathcal{O} \wedge \Gamma; \Delta \vdash \mathcal{O}}{\Gamma; \Delta; \mathcal{H} \vdash \mathcal{F}}$$

A heap is well-formed (rule META-WF-HEAP), if no location maps to several objects ($\mathsf{unambiguous}(\mathcal{H})$) and it contains only well-formed objects.

A frame is well-formed if it is unambiguous (has each variable name only once) and only contains well-formed objects.

8.1 Type Soundness

We prove type soundness in the standard way of progress plus preservation and present sketches for brevity. A Gift programme must define a well-typed starting expression e. The initial configuration is defined as $\langle \epsilon; \epsilon; \epsilon; e \rangle$.

8.1.1 Preservation

In this context, preservation means that if $\Gamma \vdash \mathit{Cfg} : T$ and $\mathit{Cfg} \hookrightarrow \mathit{Cfg}'$ then $\mathit{Cfg}' = \mathsf{ERROR} \vee \exists \Gamma'.\ (\Gamma' \supseteq \Gamma \wedge \Gamma' \vdash \mathit{Cfg}' : T)$.

The relation $\Gamma' \supseteq \Gamma$ denotes that Γ is extended monotonically: every location that is in Γ, is also in Γ' and it will map to the same object type T_o:

$$\Gamma' \supseteq \Gamma \iff \forall \iota.\ ((\Gamma(\iota) = T_o) \implies (\Gamma'(\iota) = T_o))$$

Proof The proof is by induction on expressions. Local uniqueness, required in the premises of a well-formed configuration, requires the most attention in the proof. Local uniquness is maintained by using the move operation where appropriate – the key insight is that *moving* a reference from one variable to another does not change the multi set that $\mathsf{allrefs}$ returns.

8.1.2 Progress

A well-typed configuration whose expression is not a value is able to take a step leading to another configuration (possibly ERROR): $\Gamma \vdash \langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle : T \implies (e = v \vee \exists \mathit{Cfg}'.\ \langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle \hookrightarrow \mathit{Cfg}')$.

Proof The dynamic semantics have no non-standard feature other than the move operation, the proof is straightforward by induction on expressions.

8.2 Uniqueness

From type soundness follow two interesting theorems.

8.2.1 Local Uniqueness

The first theorem states that all aliasing that can be observed in the system is allowed by the type system:

Theorem 1 (Local Uniqueness)**.** $\Gamma \vdash \langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle : T \wedge i \neq j \wedge [\ldots \iota_i \# D_i \ldots \iota_j \# D_j \ldots] = \mathsf{allrefs}(\mathcal{H}, \mathcal{F}) \wedge \Delta \vdash D_i \perp D_j \implies \iota_i \neq \iota_j$.

Proof Well-formed configuration is the main workhorse in the proof of this theorem, since it relies on rule META-LOCAL-UNIQUENESS.

8.2.2 Global Uniqueness

The second theorem states that references in the `unique` domain are never aliased.

Theorem 2 (Global Uniqueness)**.** $\Gamma \vdash \langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle : T \wedge i \neq j \wedge [\ldots \iota_i \# D_i \ldots \iota_j \# D_j \ldots] = \mathsf{allrefs}(\mathcal{H}, \mathcal{F}) \wedge D_i = \mathsf{unique} \implies \iota_1 \neq \iota_2$.

Proof The theorem holds as an instance of Theorem 1: as it is not possible to have a well-formed domain that may alias `unique` no aliases of variables in this domain may exist.

Note that the only things special about the unique domain is that it is the root, so there is no other domain linked to it — thus, we conclude that global uniqueness is merely a special case of local uniqueness. This is not surprising: the larger the scope of the domain, the more powerful it's invariants. The unique domain has a global scope, hence it's alias-freedom guarantees are also global.

8.3 Corollary: Deterministic Parallelism

The determinism of the `letpar` construct follows from the disjointness of the state on the left and right hand expressions of the `letpar` expression. The `letpar` construct requires that all reachable objects in sub-expressions on the left and right hand side of || do not share objects through fields. This is achieved by requiring that the objects in point do not have domains in their type which would allow such pointers.

9. Related Work

Hogg's seminal Island's paper [25] uses unique references to achieve a strong notion of encapsulation and disjointness. This work introduced the notion of a destructive read. Subsequent work on uniqueness includes Balloons [3], Capabilities for sharing [10], Eiffel* [27], OOFX [21], Vault [19], External Uniqueness [12], AliasJava [1], SafeJava [7], and derivatives (*e.g.,* [20, 22, 31, 34]). They all differ from disjointness domains in that they treat uniqueness as a global property (although generally temporary borrowing of references is allowed with special limitations to prevent borrowed pointers from leaking to the heap). In contrast, disjointness domains can express global uniqueness, but also allow to fall back to local uniqueness once requirements change. Vault [19] invents adoption, where a unique reference to an "adoptee" can be converted to one of a sharable type and an

"adopter" stores (invisibly) a reference to the adoptee. Uniqueness can then be regained *either* temporally by "focusing" on such a sharable reference (during focusing, the type checker will not allow accesses to possible aliases) *or* forever by freeing the adopter (which will make the aliases ill-typed and return the invisible, unique reference). Using disjointness domains, a form of adoption could be implemented by storing a strong alias to the "adoptee" in a field of an "adopter" and using aliases in a shared, linked domain as the sharable aliases. Uniqueness can then be regained using recovering. Disjointness domains are more flexible, but the flexibility also means that the programmer needs to implement some alias management herself.

Systems that structure the heap (Ownership types [13, 14], Universes [29], DPJ [6] and others) enforce object encapsulation statically by imposing a hierarchical structure on the heap. They are concerned with what part of the system may alias what other part – if two variables are tagged with the same owner, they may alias. These system are not concerned with individual references, for example expressing a tree in ownership types would require explicitly expressing the same nesting structure at the type level which makes balancing and deleting hard [11].

Uniqueness is applied, in combination with encapsulation, to parallelism in SafeJava, [7], Joelle [31] and other systems [20, 34]. Haller and Odersky [22] use global uniqueness and disjointness of objects for copyless message passing in an actor-based concurrency model. Aliases between sending and receiving actors would result in data races.

Unique references are important for work that performs strong updates, such as session types [17], type state [2, 35] and other purposes [16, 18, 28, 33]. Such work generally requires global uniqueness. Trying to combine disjointness domains and such systems is an interesting direction for future work.

Bocchino and Aldrich [5] introduce the notion of reference groups, which like disjointness domains, are sets of fields and variables which are locally unique. Reference groups are not related to each other, and there is no support for reasoning about may-aliasing or disjointness between different reference groups.

Fractional permissions and descendants [9, 24, 36] address the creation of aliases by implementing single-writer/multiple-readers at a type system level. Fractional permissions are simpler than disjointness domains, but also less powerful. For instance, they do not provide non-shallow insights like object disjointness (Section 3). Immutability is useful, but immutability *for the sake of parallelism* may hinder reuse and result in brittle code – imperative code cannot be reused in a shared setting, where the mutable parts of class interfaces are not usable any longer. Westbrook et al. [36] allow task-local aliasing, similar what our `letpar` expression implements.

10. Conclusion

Disjointness domains are a descriptive static means for alias control that support a notion of local uniqueness, which allows a more gradual approach to uniqueness than simply unique or shared references. Disjointness domains allow a programmer or compiler to reason about may-alias properties of different variables based on the domains in which variables are placed, the kinds of those domains, and the relationship between them. Reasoning about aliasing is complicated, using a type system or not. Benefits of using a type system is that programmer intent is captured, thereby giving insights to readers, compilers and runtimes alike.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 311–330. ACM Press, 2002.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-Oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. .

[3] PauloSérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63089-0. .

[4] Brian Anderson, Lars Bergstrom, David Herman, Josh Matthews, Keegan McAllister, Manish Goregaokar, Jack Moffitt, and Simon Sapin. Experience Report: Developing the Servo Web Browser Engine using Rust. *CoRR*, abs/1505.07383, 2015.

[5] Robert Bocchino and Jonathan Aldrich. Reference Groups for Local Uniqueness. Technical report, Technical Report, CMU-ISR-14-100, to appear.

[6] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. .

[7] Chandrasekhar Boyapati. *Safejava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[8] John Boyland. Alias Burying: Unique Variables Without Destructive Reads. *Software - Practice and Experience*, 31 (6):533–553, 2001. ISSN 00380644. .

[9] 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.

[10] John Boyland, James Noble, and William Retert. Capabilities for Aliasing: A Generalisation of Uniqueness and Read-Only. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pages 2–27, Berlin, Heidelberg, New York, 2001. Springer.

[11] Elias Castegren, Johan Östlund, and Tobias Wrigstad. Refined Ownership: Fine-grained Controlled Internal Sharing. In *Formal Methods for Multicore Programming*. 2015.

[12] Dave Clarke and Tobias Wrigstad. External Uniqueness Is Unique Enough. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40531-3. .

[13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership Types: A Survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36945-2. .

[14] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 48–64. ACM Press, 1998.

[15] Sylvan Clebsch and Sophia Drossopoulou. Fully Concurrent Garbage Collection of Actors on Many-Core Machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 553–570, 2013. .

[16] Robert DeLine and Manuel Fähndrich. The Fugue protocol checker: Is your software Baroque? Technical report, Technical Report MSR-TR-2004-07, Microsoft Research, 2004.

[17] Mariangiola Dezani-ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In *In Proceedings of ECOOP'06, LNCS*, pages 328–352. Springer, 2006.

[18] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic Object Reclassification. In JørgenLindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42206-8. .

[19] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24. ACM, May 2002.

[20] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. *SIGPLAN Not.*, 47(10):21–40, October 2012. ISSN 0362-1340. .

[21] Aaron Greenhouse and John Boyland. An Object-Oriented Effects System. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, number 1628 in Lecture Notes in Computer Science, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[22] Philipp Haller and Martin Odersky. Capabilities for Uniqueness and Borrowing. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14106-5. .

[23] D.E. Harms and B.W. Weide. Copying and Swapping: Influences on the Design of Reusable Software Components. *Software Engineering, IEEE Transactions on*, 17(5):424–435, May 1991. ISSN 0098-5589. .

[24] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Abstract Read Permissions: Fractional Permissions without the Fractions. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 315–334. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35872-2. .

[25] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. .

[26] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the Treatment of Object Aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, April 1992. ISSN 1055-6400. .

[27] Naftaly H. Minsky. Towards Alias-Free Pointers. In Pierre Cointe, editor, *ECOOP '96 — Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61439-5. .

[28] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-Based Typed Assembly Language. In *Journal of Functional Programming*, pages 28–52. Springer-Verlag, 1998.

[29] Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, FernUniversität Hagen, 2001.

[30] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64737-9. .

[31] Johan Östlund, Stephan Brandauer, and Tobias Wrigstad. The Joelle Programming Language : Evolving Java Programs Along Two Axes of Parallel Eval. International Workshop on Languages for the Multi-core Era 2012, 2012.

[32] Alex Potanin, James Noble, and Robert Biddle. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004. ISSN 1532-0634. .

[33] Francois Pottier and Jonathan Protzenko. Programming with Permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

[34] Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs. In *Proceedings of the Workshop on Determinism and Correctness in Parallel Programming*, 2013.

[35] R.E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *Software Engineering, IEEE Transactions on*, SE-12(1):157–171, Jan 1986. ISSN 0098-5589. .

[36] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. Practical Permissions for Race-Free Parallelism. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 614–639. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31056-0. .

A. Auxiliary Rules

$\boxed{T_o\#D\ f \in \mathsf{fields}(\Delta, c[\overline{DT_i}^i])}$ *(Field type lookup)*

AUX-FIELDTYPE-LKP

$$\frac{\mathsf{class}\ c[\overline{R_i}^i]\{\overline{V_j}^j f : T_o\#D\,\overline{V'_j}^j\,\overline{M_k}^k\} \in P \quad \Delta \vdash \mathsf{classpmatch}(c[\overline{DT_i}^i]) \dashv \sigma \quad D' = \sigma(D)}{f : T_o\#D' \in \mathsf{fields}(\Delta, c[\overline{DT_i}^i])}$$

$\boxed{\mathsf{allrefs}(\mathcal{F}, \mathcal{H}) = [\overline{\iota_i\#D_i}^i]}$ *(All references in fields and variables)*

AUX-ALL-REFS

$$\frac{\overline{\iota_i\#D_i}^i = [\iota\#D \mid \iota\#D \in \mathsf{range}(\mathcal{F})] \quad \overline{\iota'_j\#D'_j}^j = [\iota'\#D' \mid \mathcal{H}(\iota).f = \iota'\#D', \iota \in \mathsf{domain}(\mathcal{H})]}{\mathsf{allrefs}(\mathcal{F}, \mathcal{H}) = [\overline{\iota_i\#D_i}^i, \overline{\iota'_j\#D'_j}^j]}$$

B. Table of Judgements

$\vdash \Delta$	Well-formed relation env.
$\Delta \vdash d$	Well-formed domain name
$\Delta \vdash D$	Well-formed domain
$\mathcal{E}; \Delta \vdash p : T$	Well-typed path
$\mathcal{E}; \Delta \vdash e : T$	Well-typed expression
$\Delta \vdash \mathsf{not\text{-}linked}(d_1, d_2)$	Domain names not linked
$\Delta \vdash D_1 \xrightarrow{\text{copy}} D_2$	Can copy from D_1 to D_2
$\Delta \vdash D_1 \xrightarrow{\text{move}} D_2$	Can move from D_1 to D_2
$\Delta \vdash D_1 \xrightarrow{\text{linked}} DT$	Can assign linked fr. D_1 to DT
$\Delta \vdash d_1 \prec d_2$	D. name d_1 extends d_2 directly
$\Delta \vdash d_1 \prec^* d_2$	D. name d_1 extends d_2 transitively
$\Delta \vdash D_1 \perp D_2$	D_1, D_2 may not contain aliases
$\mathcal{E}; \Delta \vdash x \perp\!\!\!\perp y$	Objects are disjoint
$Cfg \hookrightarrow Cfg'$	Reduction step
$\Gamma \vdash Cfg : T$	Well-typed configuration
$\Delta \vdash \Gamma$	Well-formed store environment
$\Gamma; \Delta \vdash \mathcal{O}$	Well-formed object
$\Gamma; \Delta \vdash \mathcal{H}$	Well-formed heap
$\Gamma; \Delta; \mathcal{H} \vdash \mathcal{F}$	Well-formed frame
$\Gamma; \Delta; \mathcal{H}; \mathcal{E} \vdash e : T$	Store-typed expression
$\vdash P$	Well-formed program
$\vdash C$	Well-formed class
$T_o; \Delta \vdash M$	Well-formed method
$\Delta \vdash T$	Well-formed type
$\Delta \vdash T_o$	Well-formed object-type
$f : T_v\#D\ \in \ldots$ $\ldots \mathsf{fields}(\Delta, T_o)$	Field type lookup
$\mathcal{H}(\iota).f = v$	Field access
$\mathcal{H}' = (\mathcal{H}(\iota) := v)$	Field assignment
$\mathsf{starts}(e) = [\overline{p_i}^i]$	Starting paths of expression
$\mathsf{unambiguous}(..)$	Locations in map's domain are unambiguous (no key maps to several values)
$\mathsf{domains}(T)$	The domains an object's fields can be in (over-approximation)