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.

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—otherwise, the compiler would risk data races:

```c
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: `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

* 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).
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 \( val1 \neq val2 \) 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 employeeOfTheMonth – 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 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. Diagrams like Figure 2(a)-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.

\[
\text{domain A extend unique in } \\
\text{domain Sh B extends A in } \\
\text{domain (C, Sh B) extend unique in } \\
\text{domain Wk C and Sh C are not available}
\]

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).
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.

```plaintext
domain (CTO, Leaders) extend unique {
  let
  ceo : Person#unique = new Person,
  recruiter : Person#unique = new Person,
  programmer1 : Person#unique = new Person,
  programmer2 : Person#unique = new Person,
  cto : Person#CTO = new Person,
  teamLeader1 : Person#Leaders = new Person,
  teamLeader2 : Person#Leaders = new Person
  in ...
}
```

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, see Figure 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.

2.4.1 Moving Assignment

A moving assignment, \( y = \text{move} \ x \), 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.
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 copying 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, \((y_1, \ldots, y_n) = \text{move } x\), 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 \( y_1, \ldots, y_n \) (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 \((y_1, \ldots, y_n) = \text{move } x\) is permitted only if variables \( x, y_1, \ldots, y_n \) are all in strong or weak domains; and the target variables \( y_1, \ldots, y_n \) 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 6 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,

\[
\text{let } f : \text{Person#B} = \text{domain } A \text{ extend } B \text{ in } \{ \text{new Person#A} \}
\]

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,

\[
\text{let } f : \text{Person#Sh B} = \text{domain } Sh A \text{ extend } B \text{ in } \{ \text{new Person#Sh A} \}
\]

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.
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 \( x.f_1 \ldots f_i \) and \( y.g_1 \ldots g_j \), 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.,

\[
\text{(Leader, Members) extend U as in the code below. This domain relation allows methods to assign (Section 2.4) into the Leader and Member domains. If the parameters are at the class level, the class can put its fields in domains Leader and Members.}
\]

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

Similar to domain expressions, if a parameter is strong, then the only 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.
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

```latex
letpar x = e₁ || y = e₂ in e₃
```

runs e₁ in parallel and runs e₂ when both have finished. Analysing whether e₁ and e₂ can run in parallel is a local check that the variables used in e₁ are unaliased with the variables used in e₂ and that all objects reachable from variables in e₁ are disjoint from all objects reachable from variables in e₂.

Applying the may-alias reasoning from Section 2 and the disjointness reasoning from Section 3, this check is straightforward; Figure 8 shows an expression we might use in the running example.

```latex
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.

\[\text{score1} = \text{programmer1}.\text{performanceScore()} \parallel \text{score2} = \text{programmer2}.\text{performanceScore()}

\text{in if (score1 > score2) \{ }
\text{employeeOfMonth = programmer1 }
\text{\} else \{ }
\text{employeeOfMonth = programmer2 }
\text{\}}

The semantics requires that no access paths of the form \( x \).f₁...fn are in either \( e₁ \) or \( e₂ \) (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.
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 {  // Figure 10
    fst : DLink[(Fwd, Bwd), Data]#Fwd;  -- ↑
    lst : DLink[(Fwd, Bwd), Data]#Bwd;

def @U add(elt : Person#Data) {  // Figure 10
    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 {
            ...  // not shown
        }
    }
}
```

**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 variable `elt` 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 `elt` variable.

```
class DLink[(Fwd, Bwd)] extend U, Data extend U {  // Figure 11
    nxt : DLink[(Fwd, Bwd), Data]#Fwd;
    prv : DLink[(Fwd, Bwd), Data]#Bwd;

def @Fwd foreach(f : [D](Person#D)->Person#D) {  // Figure 11
    -- 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, 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 $e$ 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 5.1.2 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

```java
lettpar x = e_1 \parallel 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() \parallel 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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta$</td>
<td>::= $\epsilon \mid \Delta, R$ Relation environment</td>
</tr>
<tr>
<td>$R$</td>
<td>::= $DT \extend d$ Domain relation</td>
</tr>
<tr>
<td>$DT$</td>
<td>::= $(D_1, D_2)$ Domain tuple of linked domains</td>
</tr>
<tr>
<td>$D$</td>
<td>::= $K \mid d$ Domain</td>
</tr>
<tr>
<td>$K$</td>
<td>::= $St \mid Wk \mid Sh$ Dom. kind: strong/weak/shared</td>
</tr>
<tr>
<td>$d$</td>
<td>::= $dn \mid \text{unique}$ Domain: user def. or unique</td>
</tr>
<tr>
<td>$T$</td>
<td>::= $T_1 \mid \ldots$ Type: view or func (omitted)</td>
</tr>
<tr>
<td>$T_v$</td>
<td>::= $T_v.##D$ View type</td>
</tr>
<tr>
<td>$T_o$</td>
<td>::= $c[DT_t]$ Object type</td>
</tr>
<tr>
<td>$V$</td>
<td>::= $x : T$ Variable binding</td>
</tr>
<tr>
<td>$e$</td>
<td>::= $l</td>
</tr>
<tr>
<td>$l$</td>
<td>{(V_1, V_2) = e_1 \in e_2} Linked assignment</td>
</tr>
<tr>
<td>$l$</td>
<td>letpar $V_t = e_1 \parallel V_2 = e_2 \in e_3$ Parallel let</td>
</tr>
<tr>
<td>$l$</td>
<td>domain $(D_1, D_2) \extend d$ in $e$ Dom. extension</td>
</tr>
<tr>
<td>$e_A$</td>
<td>::= move $p \mid copy p \mid p$ Access expression</td>
</tr>
<tr>
<td>$p$</td>
<td>::= $x</td>
</tr>
</tbody>
</table>

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 $\Delta$ 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 $(A,B,C,D) \extend X \in \{ \ldots \}$ looks like this:

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

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 $E$ 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.

\[
\begin{align*}
\vdash \Delta & \quad \text{(Well-formed relation environment)} \\
\vdash \Delta \quad & \text{$K_1, K_2 \in \{St, Sh\}$} \\
\vdash \Delta & \text{ and } \Delta \vdash d \\
\vdash \epsilon & \quad \text{WF-RENV-EMPTY} \quad d_1 \neq d_2 \quad d_1, d_2 \notin \Delta \\
\end{align*}
\]

A well-formed relation environment $\Delta$ 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).

\[
\begin{align*}
\Delta & \vdash d \quad \text{WF-DNAME-UQ} \quad (K_1 d_1, K_2 d_2) \extend d'' \in \Delta \\
\end{align*}
\]

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

\[
\begin{align*}
\Delta & \vdash D \quad \text{(Well-formed domains)} \\
\end{align*}
\]

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 $\Delta$ (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, $\varepsilon; \Delta \vdash x_1 \perp x_2$.

6.3.1 Domain Extension

$$\Delta \vdash d_1 \prec^* d_2.$$  
(Domain name extension)

$$\text{A-EXDIRECT-LOOKUP}$$

\[
\begin{array}{l}
(K' \ d', K'' \ d'') \ \text{extend} \ d_2 \in \Delta \\
\hline
\Delta \vdash d_1 \prec d_2
\end{array}
\]

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

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

$$\text{A-NOTLINKED-SIBLING}$$

\[
\begin{array}{l}
(D_1, D_2) \ \text{extend} \ d \in \Delta \quad (D_1', D_2') \ \text{extend} \ d \in \Delta \\
(D_1, D_2) \neq (D_1', D_2') \\
K_1 \ d_1 \in \{D_1, D_2\} \quad K_2 \ d_2 \in \{D_1', D_2'\}
\hline
\Delta \vdash \text{not-linked}(d_1, d_2)
\end{array}
\]

$$\text{A-NOTLINKED-CHILD}$$

\[
\begin{array}{l}
\Delta \vdash \text{not-linked}(d_1', d_2) \\
\hline
\Delta \vdash \text{not-linked}(d_1, d_2)
\end{array}
\]

$$\text{A-NOTLINKED-SYM}$$

\[
\begin{array}{l}
\Delta \vdash \text{not-linked}(d_2', d_1) \\
\hline
\Delta \vdash \text{not-linked}(d_1, d_2)
\end{array}
\]

Rule $\text{A-NOTLINKED-SIBLING}$ expresses that two domain names $d_1, d_2$ are not linked (linked domains can not alias, as explained in Section 2.2.1) if they directly extend the same domain name $d$ but are not created together in the same domain tuple. Rule $\text{A-NOTLINKED-CHILD}$ 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 $\text{A-NOTLINKED-SYM}$.

6.3.3 Reference Flow in Assignment

$$\Delta \vdash D_1 \xrightarrow{\text{copy}} D_2$$  
(Reference flow)

$$\text{A-CANMOVE-STWK}$$

\[
\begin{array}{l}
K, K' \in \{\text{St, Wk}\} \\
K' = \text{St} \implies K = \text{St}
\hline
\Delta \vdash K' \ d
\end{array}
\]

$$\text{A-CANCOPY-S}$$

\[
\begin{array}{l}
K \in \{\text{Wk, Sh}\} \\
\Delta \vdash \text{Sh} \ d
\hline
\Delta \vdash K \ d \xrightarrow{\text{copy}} \text{Sh} \ d'
\end{array}
\]

$$\text{A-CANCOPY-UP}$$

\[
\begin{array}{l}
\Delta \vdash K \ d \quad \Delta \vdash \text{Sh} \ d' \\
\hline
\Delta \vdash K \ d \xrightarrow{\text{copy}} \text{Sh} \ d'
\end{array}
\]

$$\text{A-CANLINKED-DIRECT}$$

\[
\begin{array}{l}
K \in \{\text{St, Wk}\} \\
\Delta \vdash K \ d
\hline
\Delta \vdash K \ d \xrightarrow{\text{linked}} (\text{St} \ d_1, \text{St} \ d_2)
\end{array}
\]

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 $\text{A-CANMOVE-STWK}$ 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 $\text{A-CANCOPY-S}$ allows her to copy from a weak or shared domain to a shared domain with the same name. The rule $\text{A-CANCOPY-UP}$ 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 $\text{A-CANLINKED-DIRECT}$ 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 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

\[ \Delta \vdash D_1 \perp D_2 \]  
(Domains may not alias)

- **A-DMNA-SAMENAME**
  \[ K_1 = \text{St } \lor (K_1 \neq \text{Sh} \land K_2 \neq \text{Sh}) \]
  \[ \Delta \vdash K_1 d \quad \Delta \vdash K_2 d \]
  \[ \Delta \vdash K_1 d \perp K_2 d \]

- **A-DMNA-SYMM**
  \[ \Delta \vdash D_1 \perp D_2 \]
  \[ \Delta \vdash D_2 \perp D_1 \]

- **A-DMNA-NOTLINKED**
  \[ \Delta \vdash K_1 d_1 \perp K_2 d_2 \]
  \[ \Delta \vdash K_1 d_1 \perp K_2 d_2 \]

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

By rule \[ \text{A-DMNA-SAMENAME} \] 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 \[ \text{A-DMNA-SYMM} \] the relation is symmetric.

By rule \[ \text{A-DMNA-NOTLINKED} \] unlinked domains may not alias. This is so as no reference can have been stored in two variables in both unlinked subhierarchies – 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 \[ \text{A-DMNA-ANCESTOR} \] 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 \( \text{Sh} \) \( d \) and an equally-named shared domain \( \text{Sh} \) \( d \) may alias. The rules therefore do not allow deriving \( \Delta \vdash \text{Wk} d \perp \text{Sh} d \). 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

\[ \mathcal{E}; \Delta \vdash x_1 \perp x_2 \]  
(Object disjointness)

- **A-DISJOINT-CANNOTSHARE**
  \[ \mathcal{E}; \Delta \vdash x_1 : T_1 \]
  \[ \forall D_1 \in \text{domains}(T_1) \exists D_2 \in \text{domains}(T_2). \Delta \vdash D_1 \perp D_2 \]
  \[ \mathcal{E}; \Delta \vdash x_1 \perp x_2 \]

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 \( T_o := T_o \# D \), 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 \( i = 0 \lor j = 0 \), \[ \text{A-DISJOINT-CANNOTSHARE} \] 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 \( i > 0 \land j > 0 \), \[ \text{A-DISJOINT-CANNOTSHARE} \] 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, domains \( \{ \text{Data} \} \) \( \text{Wk} \) \( \text{Sh} \) \( \text{Bwd} \) \( \text{Fwd} \) \( \text{Fwd} \) \( \text{Bwd} \), domains \( \{ \text{Data} \} \) \( \{ \text{Data} \} \) \( \{ \text{Data} \} \) \( \{ \text{Data} \} \) \( \{ \text{Data} \} \) \( \{ \text{Data} \} \)

Running an expression involving such a DLink object in parallel with an expression involving such a Person object is allowed as all pairs of domains may not alias:

- \( \Delta \vdash \text{Data} \perp \text{Data} \) (by rule \[ \text{A-DMNA-SAMENAME} \])
- \( \Delta \vdash \text{Fwd} \perp \text{Fwd} \) (by rule \[ \text{A-DMNA-NOTLINKED} \])
- \( \Delta \vdash \text{Bwd} \perp \text{Bwd} \) (by rule \[ \text{A-DMNA-NOTLINKED} \])

6.4 Well-Typedness

6.4.1 Well-Formed Type

\[ \Delta \vdash T_o \]  
(Well-formed object type)

- **WF-OT-MATCH**
  \[ \Delta \vdash \text{classmatch}(T_o) \vdash \sigma \]

- **AUX-DELPARAMS-LKP**
  \[ \text{class } c[R_i^j][V_j^k] \in P \]

- **AUX-CP-MATCH**
  \[ \text{defparams}(c) = R_i^j \]
  \[ \Delta; \text{id } \vdash \text{pmatch}(DT_i^j, R_i^j) \vdash \sigma \]

- **AUX-P-MATCH**
  \[ \Delta \vdash \text{classmatch}(c[DT_i^j]) \vdash \sigma \]

- **AUX-CANBIND-KINDS**
  \[ K_{dec} = \text{Sh} \implies K_{use} = \text{Sh} \]
  \[ K_{dec} = \text{St} \implies (K_{use} = \text{St} \lor K_{use} = \text{Wk}) \]

- **AUX-CANBIND-KINDS**
  \[ \text{canbind}(K_{dec} \ d_{dec}, K_{use} \ d_{use}) \]
An object type \( T_0 \) 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 \( \Delta; \sigma \vdash \text{match}(DT'; \overline{R}_{i}) \) \( \vdash \sigma' \) matches bound domain tuples (and their parent domains, which are implicitly present in \( \Delta \)) to declaration site domain relations and returns a substitution \( \sigma' \) 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 5.1. The rule \[AUX-CANBIND-KINDS\] enforces this rule. By removing the matched relation from the environment and matching the rest of the parameters with \( \Delta' \) (\( \Delta' \setminus (D_1, D_2) \) 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

\[
\begin{align*}
\varepsilon; \Delta \vdash p : T & \quad \text{(Well-typed path)} \\
\text{WT-P-PF} & \\
\Delta \vdash T & \\
\Delta ; \vdash x : T & \\
\varepsilon ; \Delta \vdash x : T & \\
\text{WT-P-X} & \\
\Delta ; \vdash p : T_0 \# D & \\
\Delta ; \vdash f : T_e \in \text{fields}(\Delta, T_0) & \\
\varepsilon ; \Delta \vdash p : f : T_o \# D & \\
\text{WT-P-F} & \\
\end{align*}
\]

A variable’s type is stored directly in the variable environment \( \varepsilon \) (rule \[WT-P-X\]). In rule \[WT-P-F\], the type of a path \( p \).\( f \) relies on a field lookup: \( f : T_e \# D \in \text{fields}(\Delta, T_0) \). The field lookup simply substitutes the domains bound for an object’s declared parameters, similarly to type substitution in e.g., Java’s generics.

\[
\begin{align*}
\varepsilon; \Delta \vdash e : T & \quad \text{(Well-typed expression)} \\
\text{WT-E-MOVE} & \\
p \neq \text{this} & \\
\varepsilon; \Delta \vdash p : T_0 \# D' & \\
\text{WT-E-COPY} & \\
\Delta ; \vdash D' \text{ move} \rightarrow D & \\
\varepsilon; \Delta \vdash p : T_0 \# D & \\
\text{WT-E-LINKED} & \\
\varepsilon; \Delta \vdash e_1 : T_0 \# K d & \\
\Delta \vdash K \# d \text{ linked} \rightarrow (D_1, D_2) & \\
\varepsilon; x_1 : T_0 \# D_1, x_2 : T_0 \# D_2, \Delta \vdash e_2 : T & \\
\varepsilon; \Delta \vdash (x_1 : T_0 \# D_1, x_2 : T_0 \# D_2) = e_1 \text{ in } e_2 : T & \\
\end{align*}
\]

### 7. Dynamic Semantics

The dynamic semantics is a small-step reduction semantics. Table 2 defines heaps \( H \) as maps from locations \( s \) to objects. Objects \( O \) are tuples of object types \( T_0 \) and a list of fields mapping to values \( v \).

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

$$\Gamma; \Delta; \mathcal{H}; \mathcal{E}; e \vdash e : T$$

(Store typing)

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

$$\Delta \vdash \Gamma$$

(Well-formed store environment $\Gamma$)

A store environment $\Gamma$ 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 a set of locations in this case).

$$\Gamma; \Delta \vdash \mathcal{O}$$

(Well-formed object $\mathcal{O}$)

An object $\mathcal{O}$ is well-formed, $\Gamma; \Delta \vdash \mathcal{O}$, if all of its fields are either null ($v_i \neq i_t$) 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).

---

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

<table>
<thead>
<tr>
<th>Syntax Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{H}$</td>
<td>::= $e \mid \mathcal{H}, i \mapsto O$ (Heap)</td>
</tr>
<tr>
<td>$O$</td>
<td>::= $(T_o, f_i \mapsto v_i)$ (Dynamic Object)</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>::= $e \mid \Gamma, T_o$ (Store Environment)</td>
</tr>
<tr>
<td>$V$</td>
<td>::= $i # D \mid null$ (Stack value)</td>
</tr>
<tr>
<td>$\mathcal{F}$</td>
<td>::= $e \mid \mathcal{F}, x \mapsto v \mid \mathcal{F}, x \mapsto e_\lambda$ (Frame)</td>
</tr>
<tr>
<td>$\mathcal{C}fg$</td>
<td>::= $\langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle \mid ERROR$ (Configuration)</td>
</tr>
</tbody>
</table>

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.

$$\mathcal{C}fg \mapsto \mathcal{C}fg'$$

(Reduction step, selection)

**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 $\Gamma$ in Table 2 map locations to their object type $T_o$. Values also denote the domain they are in, used in the preservation proof.

$$\Gamma \vdash \mathcal{C}fg : T$$

(Well-typed configuration)

**META-LOCAL-UNIQUENESS**

$\forall i \# D_i, i \# D_j \in \text{allrefs}(\mathcal{F}, \mathcal{H})$.

$i \neq j \land (\Delta \vdash D_i \perp D_j) \implies (v_i \neq v_j)$

$\Delta \vdash \text{localuniqueness}(\mathcal{F}, \mathcal{H})$

**META-WT-CONFIG**

$\Gamma; \Delta \vdash \mathcal{H}$

$\Gamma; \Delta; \mathcal{H}; \mathcal{F} \vdash \mathcal{C}fg$

$\Delta \vdash \text{localuniqueness}(\mathcal{F}, \mathcal{H})$

$\Gamma; \Delta; \mathcal{H}; e \vdash e : T$

$\Gamma \vdash \langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle : T$
A heap is well-formed (rule \textsc{meta-wf-heap}, if no location maps to several objects (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 \(\langle H; \Delta; \mathcal{F}; e \rangle\).

\[\Gamma; \Delta \vdash \mathcal{H} / \mathcal{F}\]

The relation \(\mathcal{H} \supseteq \mathcal{H}'\) denotes that \(\mathcal{H}\) is extended monotonically: every location that is in \(\mathcal{H}\), is also in \(\mathcal{H}'\) and it will map to the same object type \(T\):

\[\Gamma' \supseteq \Gamma \iff \forall i. \; ((\mathcal{H}(i) = \mathcal{O}) \wedge \Gamma; \Delta \vdash \mathcal{O})\]

**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 uniqueness 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 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 \(\text{ERROR}\)): \(\Gamma; \Delta \vdash \langle \mathcal{H}; \mathcal{F}; e \rangle : T \implies (e = v \lor \exists Cfg'. \langle \mathcal{H}; \Delta; \mathcal{F}; e \rangle \implies Cfg')\).

**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). \(\Gamma; \Delta \vdash \langle \mathcal{H}; \mathcal{F}; e \rangle : T \land i \neq j \land [\ldots t_i \# D_i \ldots t_j \# D_j \ldots] = \text{allrefs}(\mathcal{H}, \mathcal{F}) \land D_i = \text{unique} \implies t_i \neq t_j\).

**Proof** Well-formed configuration is the main workhorse in the proof of this theorem, since it relies on rule \textsc{meta-wf-heap}.

#### 8.2.2 Global Uniqueness

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

**Theorem 2** (Global Uniqueness). \(\Gamma; \Delta \vdash \langle \mathcal{H}; \mathcal{F}; e \rangle : T \land i \neq j \land [\ldots t_i \# D_i \ldots t_j \# D_j \ldots] = \text{allrefs}(\mathcal{H}, \mathcal{F}) \land D_i = \text{unique} \implies t_i \neq t_j\).

**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 \texttt{letpar} construct follows from the disjointness of the state on the left and right hand expressions of the \texttt{letpar} expression. The \texttt{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], OOXML [21], Vault [19], External Uniqueness [12], AtlasJava [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 “adopter” 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—i.e., 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] and other purposes [6, 18, 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, 30] 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


A. Auxiliary Rules

\[ T_o \# D f \in \text{fields}(\Delta, c[DT_i^+]) \]  
(Field type lookup)

\[
\text{AUX-FIELD-TYPE-LKP}
\begin{align*}
\text{class} & \ c[\mathcal{F}^+_i] \{ V_j^f : T_o \# D V_j^f \rightarrow \mathcal{M}_k^c \} \in P \\
\Delta & \vdash \text{classpmatch}(c[DT_i^+]) \vdash \sigma \\
D' & = \sigma(D)
\end{align*}
\]

\[ f : T_o \# D' \in \text{fields}(\Delta, c[DT_i^+]) \]  

\[ \text{allrefs}(\mathcal{F}, \mathcal{H}) = \{ i \# D_i^+ \} \]  
(All references in fields and variables)

\[
\text{AUX-ALL-REFS}
\begin{align*}
i \# D_i & = \{ i \# D | i \# D \in \text{range}(\mathcal{F}) \} \\
i' \# D_j & = \{ i' \# D' | \mathcal{H}(i).f = i' \# D', i \in \text{domain}(\mathcal{H}) \}
\end{align*}
\]

\[ \text{allrefs}(\mathcal{F}, \mathcal{H}) = \{ i \# D_i^+, i' \# D_j^+ \} \]

B. Table of Judgements

\[ \vdash \Delta \]  
Well-formed relation env.

\[ \Delta \vdash d \]  
Well-formed domain name

\[ \Delta \vdash D \]  
Well-formed domain

\[ \epsilon; \Delta \vdash p : T \]  
Well-typed path

\[ \epsilon; \Delta \vdash e : T \]  
Well-typed expression

\[ \Delta \vdash \text{not-linked}(d_1, d_2) \]  
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 \( D_2 \)

\[ \Delta \vdash \text{linked} \rightarrow DT \]  
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

\[ \epsilon; \Delta \vdash x \perp y \]  
Objects are disjoint

\[ Cfg \leftrightarrow Cfg' \]  
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

\[ f : T_o \# D \in \ldots \]  
Field type lookup

\[ \ldots \text{fields}(\Delta, T_o) \]  
Field access

\[ \mathcal{H}(i).f = v \]  
Field assignment

\[ \mathcal{H}' = (\mathcal{H}(i) := v) \]  
Starting paths of expression

\[ \text{starts}(e) = \{ \mathcal{F}^+ \} \]  
Locations in map’s domain are unambiguous (no key maps to several values)

\[ \text{domains}(T) \]  
The domains an object’s fields can be in (over-approximation)