Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1749

Structured Data

STEPHAN BRANDAUER

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018

ISSN 1651-6214
ISBN 978-91-513-0515-8
urn:nbn:se:uu:diva-366932

Dissertation presented at Uppsala University to be publicly examined in Room 2446, Institutionen för informationsteknologi, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Wednesday, 23 January 2019 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Doug Lea (State University of New York at Oswego).

Abstract

Brandauer, S. 2018. Structured Data. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1749. 85 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0515-8.

References are a programming language construct that lets a programmer access a datum invariant of its location.

References permit aliasing -- several references to the same object, effectively making a single object accessible through different names (or paths). Aliasing, especially of mutable data, is both a blessing and a curse: when used correctly, it can make a programmer's life easier; when used incorrectly, for example through accidental aliases that the programmer is unaware of, aliasing can lead to hard to find bugs, and hard to verify programs.

Aliases allow us to build efficient data structures by connecting objects together, making them immediately reachable. Aliases are at the heart of many useful programming idioms. But with great power comes great responsibility: unless a programmer carefully manages aliases in a program, aliases propagate changes and make parts of a program's memory change seemingly for no reason. Additionally, such bugs are very easy to make but very hard to track down.

This thesis presents an overview of techniques for controlling how, when and if data can be aliased, as well as how and if data can be mutated. Additionally, it presents three different projects aimed at conserving the blessings, but reducing the curses. The first project is disjointness domains, a type system for expressing intended aliasing in a fine-grained manner so that aliasing will not be unexpected; the second project is Spencer, a tool to flexibly and precisely analyse the use of aliasing in programs to improve our understanding of how aliasing of mutable data is used in practise; and the third project is c flat, an approach for implementing high-level collection data structures using a richer reference construct that reduces aliasing problems but still retains many of aliasing's benefits.

Keywords: Aliasing, mutable state, imperative, programming, programming languages.

Stephan Brandauer, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0515-8
urn:nbn:se:uu:diva-366932 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-366932)

To Kim and Felix.

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Disjointness Domains for Fine-Grained Aliasing
Stephan Brandauer, Dave Clarke, Tobias Wrigstad
International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015 [8]

A type system for expressing the shape of a program's data in a fine-grained manner.

II Spencer: Interactive Heap Analysis for the Masses
Stephan Brandauer, Tobias Wrigstad
International Conference on Mining Software Repositories (MSR), 2017 [10]

An interactive tool for simple, online analysis of dynamic execution traces of programs from standard program corpora. Spencer hosts large data sets and runs in a web browser, making it easy to gain insights about the behaviour of the programs.

III Mining for Safety using Interactive Trace Analysis
Stephan Brandauer, Tobias Wrigstad
Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL), 2017 [9]

An application of Spencer (Paper II) to analyse program traces for safety properties, like immutability, uniqueness, stack-boundedness, etc.

IV C♭: A Modular Approach to Efficient and Tunable Collections
Stephan Brandauer, Elias Castegren, Tobias Wrigstad
Onward!, 2018, [7]

A domain specific language and its implementation that lets programmers design high level structures. These data structures can be combined with different ways (*back-ends*) to represent data in memory, yielding collections that can be optimised by just picking the right back-end.

Reprints were made with permission from the publishers.

The Author's Contributions

I Main author. Idea and design by main author, formalisation and manuscript with co-authors.
II Main author. Sole implementer, manuscript with co-author.
III Main author. Experiments by main author, manuscript in collaboration with co-author.
IV Main author. Idea, design, implementation and evaluation by main author. Manuscript with co-authors.

Related Publications

Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore
Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, Albert Mingkun Yang, 2015
15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming

An introduction to the design of the Encore programming language. Encore is a parallel object-oriented programming language with active objects that communicate asynchronously via message passing and a type system that guarantees data race freedom.

The Joelle Programming Language: Evolving Java Programs Along Two Axes of Parallel Eval
Johan Östlund, **Stephan Brandauer**, Tobias Wrigstad
International Workshop on Languages for the Multicore Era 2012

An introduction to the Joelle programming language that explores the use of ownership types and effects, and actors for building parallel programs safely.

Contents

Summary in Swedish

Referenser är en programspråkskonstruktion som abstraherar datorns minne genom att låta program manipulera data som är beläget på abstrakta platser. Detta tillåter en programmerare att skriva kod som behandlar data oavhängigt dess placering i minnet och skapa datastrukturer som kan växa och krympa dynamiskt genom att flera datum länkas samman. En graf kan till exempel representeras som ett antal objekt (noder) som håller referenser (kanter) till andra objekt. Referenser tillåter vidare *aliasering* – att ett objekt är nåbart i ett program via flera olika namn. Utöver dess uppenbara nytta för att representera strukturer – som grafer – där cykler eller flera vägar till samma plats förekommer naturligt används aliasering typiskt för att effektivt dela minne mellan delar av ett program eller traversera en datastruktur, t.ex. följa en serie av länkar mellan objekt för att söka efter ett element i en viss ordning. En negativ sidoeffekt av detta bruk av referenser är att både programlogik och prestanda knyts till hur data är representerat i minnet – ju fler länkar mellan objekt, desto effektivare kan vi nå fram till en vissa plats i strukturen, men till priset av mer komplexa referensstrukturer som måste underhållas och som gör det svårare att resonera om programs korrekthet, som vi snart skall se.

Referenskonstruktionen har sin upprinnelse i språket PL/I på 60-talet, och blev tidigt känd som både kraftfull och behäftad med problem. Dessa problem uppkommer särskilt när referenser används för att *dela föränderligt data*. Om ett objekt O delas mellan – aliaseras av – flera stukturer, A och B säg, blir all förändring via A också synlig från B och tvärtom. Som en konsekvens av detta måste därför alla förändringar av O ta hänsyn till alla förväntningar på O från både A och B (och alla andra stukturer som O är en del av). Att hålla reda på vilka strukturer ett objekt är del av vid varje given tidpunkt, eller, mer generellt, vem som blir påverkad av en förändring, blir därför av yttersta vikt för ett programs korrekthet. Detta försvåras av att strukturer byggs upp och förändras dynamiskt under körning – och att det inte är synligt i koden huruvida en förändring av O är synlig för bara A, för bara B eller för både A och B.

Ämnet för denna avhandling är referenser: tekniker för att undvika problem med referenser genom att undvika aliasering, eller aliasering av föränderligt data, och tekniker som tillåter aliasering under kontrollerade former.

I inledningen ger vi en översikt och en kategorisering av existerande tekniker för programmering med referenser. Vår hypotes är att programspråk ger friheter att använda referenser som de flesta program inte använder eller behöver och att detta gör det svårare för programmerare, verktyg och kompilatorer att resonera om ett programs beteende.

För att undersöka denna hypotes utvecklar vi ett verktyg – Spencer – som spårar hur referenser skapas, används och förstörs i program, och använder det på flera välkända program. Vi finner att trots att alla referenser i de flesta programspråk tillåts vara alias, och till yttermera visso förändras, är detta ovanligt, och följer ofta vissa mönster. Spencer är byggt för reproducerbar forskning – data lagras i molnet, resultat delas enkelt mellan forkare, och det är enkelt att bygga vidare på och förfina någons resultat.

Vi skapar ytterligare två system: först “disjointness domains” som är ett system som låter programmerare uttrycka vilken aliasering som är möjligt i ett program på ett precist sätt. Detta system är kraftfullt, och bevisat korrekt, men svårt att utvärdera fullt ut eftersom det är skapat för ett enkelt forskningsspråk. Vi bygger vidare på resultaten från Spencer och disjointness domains; istället för att förhindra aliasering eller kontrollera förändringar abstraherar vi referenskonstruktionen ytterligare i “C♭”, och tillåter endast att referenser används för traversera en dataastruktur i enlighet med ett fördefinierat protokoll (t.ex., som om datstrukturen var ett träd). Detta tillåter oss att låta referenser peka på abstrakta objekt, vilket i sin tur gör det möjligt att separera logiken som implementerar en abstrakt datatyp från hur den är representerad i minnet. Detta gör det möjligt att justera prestanda utan att ändra i logiken, och utan komplicerade referensstrukturer.

Thanks, but not Goodbye

I want to take some time to thank the people that have made my time during my PhD studies in Uppsala so much more enjoyable.

Tobias, you have been a great advisor over the last five years! Other than always having my back, you’ve taught me a lot about teaching, where I made some of my most memorable experiences as a PhD student. Our group is so tight-knit, we treat each other more like friends than like colleagues. I think that your influence has been responsible for a lot of this positive climate. Thank you for your patience with me!

Dave, I will miss the weekly meetings, the fun we had, the design discussions about Encore (needs more monads!). I’m happy that you joined our daily step-count competition, but I’m not so thrilled about the distances you are walking!

Kim, I can not describe how glad I am to have you. During these five years, there were ups and downs in my work, and you’ve always been there for me. I hope that one day, I can repay you for your never-ending kindness, your patience, the sourdough bread, the smiles that you put on my face when I’m tired. I’m not sure if I’ll manage, but I have a lifetime to try my best. I love you!

Felix, since you were born, everything is different. You have changed my life completely and I love every bit of my new life. I can’t wait to go on parental leave with you! I love you!

Elias, we started our PhD at roughly the same time, and I always considered the two of us to be “in this together”. It was great to have you as a colleague to share an office with; go to the best summer school ever with (and the second best); argue about ketchup; invent “clapital letters” with; work with at the whiteboard and in the class room. Thank you for having been such a great colleague, and thank you for teaching me so much over the years.

Kiko, Albert, Phuc: I’m really glad to have had you as colleagues. We had plenty of fun and I learned lots from you guys. I will always remember how much fun we had at the lab outings, the coffee breaks we had together. Thank you!

Andre, Greg: the two of you have so often managed to take my mind off work, I owe you. Thank you for all the fun we had together, thank you for the late night conversations, thank you for being such great uncles to Felix.

Andreas, Anke, Arve, Astrid, Beatrice David, Greg, Gustaf, Kjell, Magnus (Lång, Norgren, and Själander), Mihail, Stavros, and those I have forgotten: I’m glad to have been here with you all. Lets stay in contact!

Mum and dad, you have always believed in me and supported me. Without you, so much would not have been possible. Thank you for all of that!

My big brothers: the influence of one of you got me interested in computers; the influence of the other got me interested in academia. But these are only the little things! Growing up with you has defined me in so many good ways – you’re awesome!

Part I:
Introduction

1. Overview

A reference in programming is a language construct that can be used to indirectly access a datum at any location in memory. References are handles that can be stored in variables or fields and that allow accessing an object at a different location in memory. Several references can exist that refer to the same datum. When several references to a datum exist, we call the references, as well as the variables containing them, *aliases*, and the datum *aliased*.

Definition 1 (Alias) *Two references* `x` *and* `y` *are* aliases *if they refer to the same object. The term "aliasing" stems from the fact that a single object has several names,* `x` *and* `y`*. If two variables contain these references, we also call the variables aliases.*

References criss-crossing the mutable state of a running program make understanding the program hard. This is a problem for all parties involved in writing, maintaining, and executing software: programmers inadvertently introduce bugs because they have a hard time keeping in mind which parts of a system are affected by updating a datum; compilers are limited in the amount of optimisations they can do; the hardware finds itself slowed down because dereferencing a pointer often leads to cache misses (and requires slowly loading data from main memory).

The situation is, we think, reminiscent of unstructured code: in the 1970s, programmers slowly moved away from "spaghetti code" and started using techniques of structured programming [21] —*i.e.,* loops and function calls. This thesis is about structured data. At the heart of the issue we are addressing lies a fundamental disconnect, similar to the one proponents of structured programming found: while most programs use data that is structured (most objects have few aliases, encapsulation is common, etc.), programming languages provide little means of documenting, understanding, or optimising that structure. Code that uses much aliasing looks similar to code that uses little.

1.1 Background

There is a vast amount of techniques to mitigate the risks that aliasing of mutable state brings with it. This introduction chapter covers a fraction of these techniques. The range of ideas presented here is vary wide, we will cover type-based abstractions, code generation techniques, libraries for systems programming languages, pure functional programming, etc. Coherently presenting these techniques is no small task, but we attempt to make it easier to achieve a unified understanding by classifying them into a coherent, informal, system that we will introduce in Section 1.4 and Section 1.5. The purpose of this classification is, in part, to hide details that are irrelevant for this thesis (even though these details are far from irrelevant in programming practice): *e.g.,* we will spend very little time talking about type soundness because we are interested in the core idea of an abstraction, not its implementation (which may be sound or unsound).

Our main goal for this chapter is to make the reader understand two things:

1. How large the number of techniques is that researchers and practitioners have come up with to try and solve this problem and how small in comparison the number of basic concepts being used is (most notably: uniqueness, encapsulation, and immutability in various forms);
2. How the research we have conducted fits into this context by relying on many of the same ideas.

1.2 Contributions

The contributions of this thesis are presented in the subsequent chapters. The work covers a wide range of research that is all aiming to understand, and cope with, aliasing in imperative programming.

In particular, the thesis make the following contributions:

1. a novel type system for expressing structure invariants of programs;
2. a tool to analyse dynamic program traces that we apply to look for evidence that program memory is highly structured in practice; and
3. an embedded domain specific language for data-structure implementations that removes aliasing from its semantics altogether, but recovers much of the performance benefits of aliasing by using high-level optimisations of whole data structures.

1.3 A Short History of References

Before we can talk about how references are problematic, and what to do about that, we need to understand the reasons for why references exist in the first place. As references have been in programming use for so long, it is easy to forget those reasons.

Harold W. Lawson received the IEEE Computer Pioneer Award in 2000 for the invention of pointers, and their implementation in the PL/I programming language in 1964–65 [37]. Pointers are variables that store the memory address of a datum. Pointers are a *kind of* reference (but references may be implemented in other ways, like as integer indexes into an array).

The pointer concept was ground-breaking at the time as it enabled several important features, amongst them:

- The implementation of dynamically sized data structures in a high level language (Section 1.3.1);
- constant-time passing of large data structures (Section 1.3.2);
- sharing of mutable data from several locations (Section 1.3.3).

1.3.1 References for Dynamically Sized Data Structures

Using references, we can implement dynamically sized data structures. A pointer always has the same size (the length of a virtual memory address) in memory, *regardless of the size of the datum pointed at*. This is crucial for determining the size of a recursive data type[1], a type that contains values of type `T` or pointers to `T`. As an example, consider a linked list. A node in a linked list consists of a value (here called "element"), and a pointer to a following node—it is a recursively defined data type, because the node contains a field pointing to another node.

In the C programming language a node could be declared like this:

```
struct {
  int element;
  struct node *next; // next points at the following node, or is NULL
} node;
```

The size of such a list in memory might be defined simply as the sum of the size of its constituent fields[2]. In this case: the sum of the size of an integer (for the `element` field) and a pointer (for the `next`), `sizeof(struct node)= sizeof(int)+ sizeof(struct node *)` – no matter whether `next` is `NULL`, refers to a tail list of length 1, 2, or any other number.

If `next` would not be a reference type, but rather declared as `struct node next`, then the size of a node would be undefined, due to the sum's definition becoming recursive: `sizeof(struct node)= sizeof(int)+sizeof(struct node)`. When the program allocates space for a node, it needs to know the size of a node in bytes. In C, defining a data type with such a recursive size definition is, for this reason, prohibited.

In conclusion, references are useful for the constant size they have, no matter what the size of the referent is; this knowledge is helpful to implement, as presented, recursive data types. Recursive data types *could be implemented*

[1]If we squint and look at dynamically sized arrays of `T` elements and length N as tuples of a `T` value and an array of length $N - 1$, we see that the references are also generally necessary for dynamically sized arrays just like other recursive data types.

[2]We are skipping over data structure alignment in C for simplicity.

without pointers, but the implementations we are aware of end up relying on a form of references, like integer indices into an array of nodes.

1.3.2 References For Constant Time Passing of Large Data

There are at least two reasons to pass a reference to a parameter, rather than a copy of the parameter:

1. It can be more efficient to pass a reference to a large object as a parameter than making an expensive copy,
2. passing a reference to an object as a function parameter, rather than a copy can be used to let the function change the object.

Passing values by reference in a high-level language was first implemented in the language PL/I. In PL/I, it was possible for pointers to refer to the location of a stack variable, as well referring to heap-allocated data.

1.3.3 References For Sharing of Mutable State

References allow reaching the same object from several other variables or objects. When this object happens to be mutable, changes to it can be perceived through all aliases.

This behaviour gives rise to useful programming idioms, like iterators:

```
class StringListNode {
 private StringListNode next;
 private String element;

  ...

 public void allStringsToUpper() {
  StringListNode n = this;

  do {
   n.element = n.element.toUpper(); // ‡
   n = n.next;
  } while (current ≠ null);
 }
}

class StringListIterator {
 private StringListNode currentNode; // †
 public String getNext() { ... }
 public void advance() { ... }
}
```

In the example, the field `currentNode` (defined at †) ranges over all the nodes in the list, if the `advance` method is called often enough. Importantly, the contents of `currentNode` is a reference to a node that is also part of the list. This means that the changes caused to the *list object* (at ‡) will affect the values a *list iterator* returns. This is a form of communication between objects, and intended behaviour. It is possible due to the use of shared references to mutable state.

Referencing a mutable object from several fields can be useful for asymptotically better performance: mutating an object that is reachable from N other objects indirectly changes the behaviour of these other objects as well—but requires only a constant amount of work. This quite abstract principle has concrete applications in data structures. In a linked list, maintaining a last-reference gives it an $\mathcal{O}(1)$ append operation by exploiting the aliasing of the last node of the list. Due to the aliased last pointer in Figure 1.1b, an append operation can find the end of the list in constant time and update it immediately. In Section 2.2.3, we will compare how immutable data structures can handle a requirement for fast appends, and how immutability makes this task harder to achieve.

1.3.4 Unintended Sharing of Mutable State

While references are a powerful construct, they are also dangerous. Even the justification for the computer pioneer award acknowledges that, quote:

> “ *When utilised in a non-professional manner, pointer variables have led to software reliability problems. These problems are evident in enormous software composites containing significant unnecessary complexity.*
>
> — IEEE Comp. Soc. [37]

Programmers are tasked with avoiding these problems, not languages and tools:

> “ *By not properly engineering software, complex pointer relationships and status can arise that are difficult to predict and trace thus leading to difficulties. When will we ever learn that engineering discipline is required to produce high quality software? Pointer variables when properly utilised have provided hundreds of thousands of software engineers with an essential enabling tool of the trade.*
>
> — IEEE Comp. Soc. [37]

It is true that pointers (or generally, references) are "an essential enabling tool of the trade". But the fact that their use remains, after decades of industry use, a reliability problem might come from the fact that references make it too easy to do the wrong thing: references can, by way of aliasing and mutability, introduce a hidden connection from one aggregate object to another; if a programmer is not aware that such a connection exists, they can easily—believing they are modifying only one of the objects—modify both of them. This accidental modification of an object is a bug, and finding that bug can be very hard, as it may not be clear where changes are coming from. A programmer needs to understand aliasing of mutable state in order to modify a program she herself has not written.

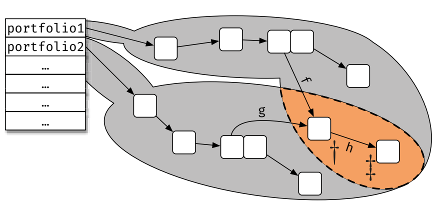
As an example, consider a program's memory in Figure 1.1a: there are two stack variables `portfolio1` and `portfolio2` that contain the addresses of different objects. However, the transitive closures of these objects are not disjoint. In particular, we can reach the object marked † from an object in `portfolio1`'s transitive closure (via the field `f`) and as well from an object in `portfolio2`'s transitive closure (via the field `g`). Any object we can reach from †, like the ‡ object, is therefore also reachable from both `portfolio1` and `portfolio2`. Therefore, modifying the data reached from `portfolio1` can indirectly change the state of `portfolio2` unexpectedly, and vice-versa. This may lead to methods being called on `portfolio2` that break invariants associated with `portfolio1`. To use an example by Hogg et al. [34][3] of a class modelling portfolios of financial assets that contain checking accounts (and perhaps other assets). Portfolios have the `transferTo(Portfolio other, int amount)` method that will transfer money from one portfolio to another. Consider the following statement:

```
portfolio1.transferTo(portfolio2, 100);
```

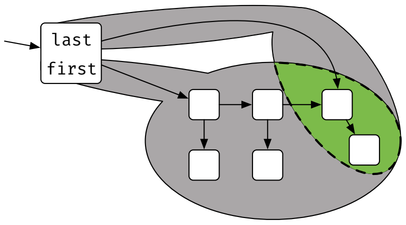
Will executing this code decrease the amount of money in `portfolio1`? This depends: if `portfolio1` and `portfolio2` are aliases, the statement will not have the expected effect. In this case, it is easy to guard against: the check **`assert`**`(portfolio1 ≠ portfolio2)` will prevent the problem from breaking an invariant. However, what happens if the portfolios are not the same object, but each happens to contain the same checking account, like in Figure 1.1a? The behaviour will be equally unexpected, but the simple dynamic assertion does not solve the issue any longer. If the portfolio class makes its internal state inaccessible by outside code, the problem can no longer be solved without modifying the portfolio class.

Instead of dynamically enforcing aliasing invariants, we could consider static analysis to avoid unintended sharing of mutable state. Pointer analysis (analysis of whether or not two pointers may refer to the same object) was shown [40, 41, 54] to be undecidable—no general algorithm exists that computes precisely whether or not two pointer variables `x` and `y` may contain the same pointer at

[3] We are translating the code to a Java-like syntax from the original Smalltalk code.



(a) Unintented sharing.



(b) Intended sharing in a linked list.

Figure 1.1. Pointers can be used for sharing. a) Programmers might not know about these connections between mutable state, thereby causing bugs by first writing to the state reachable from `x` and then reading from the state reachable from `y` (or vice-versa). b) When sharing is intended, it can have a positive effect, like a $\mathcal{O}(1)$ add operation in a linked list that needs a `last` pointer and therefore aliasing.

a given location. We will briefly give an intuition for why this is the case: Ramalingam [54] reduces aliasing analysis to Post's correspondence problem [50], which is known to be undecidable[4]. The reduction works by observing a program that uses two pointers p, q to traverse a binary tree. In a loop, the program chooses a random integer i, and uses it as index to access to sequences of integers w and z. The integers are interpreted as relative directions in the tree, each `0` in the integer's binary representation corresponds to a left-step, each `1` to a right-step. The p pointer is updated using the value drawn from w, the q pointer is updated using a value drawn from z.

At the end of that loop, the pointers may alias iff post's correspondence problem has a solution for the sequences $w_1, \ldots, w_r$ and $z_1, \ldots, z_r$.

Obtaining that "engineering discipline" that programmers are supposedly should have is not easy: we want to conserve the useful features of pointers (cheap passing of data, dynamically sized data structures, intended sharing), but we want to prevent the bugs caused by unintended sharing.

To balance the quote from before, we can give another quote, one that is as harsh as the previous quote is forgiving:

“ *References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high level languages has been a step backward from which we may never recover.*

— C.A.B Hoare [32].

[4]Post's Correspondence Problem: given two sequences of strings $w_1, \ldots, w_N$ and $z_1, \ldots, z_N$ over some alphabet, decide whether or not there is non-empty sequence of indices $i_1, \ldots, \ldots, i_r$ s.t. the concatenations $w_{i_1} \cdots w_{i_r} = z_{i_1} \cdots z_{i_r}$ are equal.

While unintended sharing of mutable data is clearly problematic, we sometimes intend to; then, we often want to avoid changing the data to not inadvertently change other parts of memory indirectly. The next two sections give a birds-eye view in the form of classification of features of systems that restrict aliasing and mutation. As a birds-eye view usually does, the classification in these two sections is painting in broad strokes: it will compare type-safe abstractions with programming idioms and libraries used in low level programming languages (which usually have features to disable all safety guarantees). The reason to introduce this category is to give the reader a basic understanding of many of the concepts we will talk about.

1.4 Restricting Aliasing: A Taxonomy

We will break up tools and techniques to avoid or highlight the sharing of mutable data into orthogonal categories in order to later on use those categories to systematically compare existing research and implementations of both programming language, and library abstractions. Our categorisation is informal (we know of no formal system to capture the whole range that we will cover here, a situation that others have also remarked on [46]) but covers a wide range of the research presented in the rest of this chapter, ranging from almost no restriction (reference variables in imperative programming languages), all the way to unique references (Definition 7) and alias control methods based on encapsulation like islands, and ownership types (*c.f.* Section 2.1.2, [16, 33, 47]).
The categories that we will use are:

The mode of alias restrictions. An object can be *shared* (when more than one references to it may exist), it can be *temporarily shared* (when the program's run-time is divided into phases where more than one reference to an object exist, alternating with phases where there is only one alias), or it can be *unique* (when only one reference to it exists). Temporary sharing is a technique that is commonly used to permit users to call methods on uniquely referenced objects: calling the method creates an alias reference, available through the method's `this` parameter. But when `this` is borrowed (in other words, it will never be stored into the heap [5, 11]) this alias only exists for the duration of the method call. Calling methods on a uniquely referenced object would require the callee reference to be made unavailable at call site (for instance, by setting the referring variable to `null` in the process) and passing the `this` reference back to the caller as part of the return value.

The reach of alias restrictions. While the mode of alias restrictions is about the reference(s) to an object, the reach of alias restrictions is about what is known about the references read, transitively, from the object. This is a mechanism that is used to protect the inner state of an object by preventing uncontrolled outside access. The reach of alias restrictions can be *shallow*, meaning that at least some references read from the object are globally shared (they can be passed anywhere); or it can be *deep*, meaning that the access to the object's inner state is restricted.

The scope of alias restrictions. Alias restrictions can apply to a whole program, meaning that there really is only one reference to a certain object in all the program. This kind of aliasing restriction, which we call *global alias restrictions* (*c.f.* global uniqueness in Paper I), is desirable – but many situations are not that simple. *E.g.,* a tree data structure might be guaranteed to contain only one reference to a certain object, but references outside of the data structure may also refer to the object. We use the term *local* alias restrictions when a reference is guaranteed to be the only one in a certain set of fields or variables strictly smaller than the set of fields and variables referring to objects of the same object type.

We are not the first ones to come up with a categorisation of aliasing like aliasing modes: Mycroft and Voigt [46] categorise aliasing into linearity (*c.f.* Definition 8, p. 39), spatial aliasing, and temporal aliasing, which roughly correspond to our aliasing modes[5].

The idea of a scope of alias restrictions will be important in Paper I, which introduces a static type systems to express a wide range of fine aliasing, ranging from global deep kinds of aliasing restrictions (the most restrictive – but safest) to globally shared (the most permissive – but least safe), see Chapter 4 and Paper I.

A large number of systems deals with systems that use alias restrictions. Clarke et al. [15] give a comprehensive overview of ownership types, a family of type systems that are used to guarantee object encapsulation in various versions. Rust uses deep alias control as a default (references into an object may never outlive the object) but comes with smart pointer types that can explicitly disable this behaviour where needed. We categorise a number of type systems, code patterns, modes of uses of existing languages, etc. in Table 1.1 and Table 1.2.

1.5 Forms of Immutability: A Taxonomy

At a first glance, it might seem like immutability has a simple, and single definition. However, this is not so: the term "immutability" can refer to a wide range of semantics. This section groups semantics of immutability – informally – for the purpose of understanding differences and similarities of this wide range. These features are:

[5]Both their and our definitions are informal. Our aliasing modes can be combined further with scope and reach to yield a classification that subsumes more diverse techniques.

Table 1.1. *Examples for forms of shallow alias restrictions. We will introduce those systems in Chapter 2.*

	Local	Global
Shared	Encapsulated objects are local to objects reachable from the encapsulating object.	C pointer/C++ pointer/Java reference type fields/variables.
Temp. Shared	C's `restrict` pointers force a programmer to use only the one restricted alias to an object as long as this pointer is alive, *c.f.* Section 2.1.3.	A data structure's spine and the iterator's field that references the current node (temporary when iterators have a bounded lifetime, as is common). "Standard" unique references with borrowing (*c.f.* Section 2.1.3).
Unique	The set of element-fields of a set data structure may never contain the same reference twice.	Linear types (*c.f.* Section 2.1.3, immutable data reachable from them is shared [59]). Global uniqueness (*c.f.* Chapter 4, Paper I), if explicit aliasing is avoided and the referred-to object contains shared data.

The reach of immutability. Immutability can be shallow (apply to the object itself, but not to all references stored by the object) or deep (the object itself is immutable, and also all objects reachable from it).

The granularity of immutability. Immutability can mean that an object may not be mutated through a *single reference* to an object (but through other copies of that reference), or it can mean that *an object* may not be mutated (in other words to all references to the object are reference immutable), or it can apply to *all instances of a type*. We speak of reference-, object-, or class immutability.

The mode of immutability. Immutability can be permanent (*e.g.*,, a permanently immutable object may never be mutated once it's constructed, like in purely functional programming) or temporary (*e.g.*,, an object that undergoes an initialisation phase and at some point "freezes", like objects using memoisation or lazy initialisation; or an object that has clearly delineated phases in its lifetime during which it won't be mutated, like passing a C string to a function that will write to it, *e.g.,* for making a copy: `strcpy(a)`).

Considering the list above, we see that the term "immutability" can refer to a great variety of meanings, some of which are very restrictive (but maintain

Table 1.2. *Examples for forms of deep alias restrictions. We will introduce those systems in Chapter 2.*

	Local	Global
Shared	From Paper I: Object with no/`global` domain parameters, referenced through a shared domain (*c.f.* Chapter 4, Paper I).	Pointer to Java/C/C++ object using encapsulation. Bridge Objects in Islands [33].
Temp. Shared	Rust values enforce that their aliases never outlive the object, and that any references taken from inside an object have a lifetime shorter than the object (deep). From Paper I: a unique reference that is temporarily aliased and later recovered to unique (*c.f.* Chapter 4, Paper I).	Balloon objects [2, 55], see Section 2.1.2. Dynamic borrowing (*e.g.,* Rust's `std::cell` [18], *c.f.* Section 2.2.1) ensures that the object is accessed through only one of the aliases at a time. Mutex-locking accesses to an object that encapsulates its representation ensures that only one of the references may access the object at each time.
Unique	From Paper I: local uniqueness of an object that uses only globally unique type parameters (*c.f.* Chapter 4, Paper I). The `next` fields in a list's spine form a set of fields that contains each reference only once, but an external iterator will contain aliases of the `next` fields[6].	From Paper I: a globally unique reference (of type `T#unique`), where `T`'s type parameters are all globally unique. (*c.f.* Chapter 4, Paper I) and explicit sharing is avoided.

strong invariants for reasoning about programs), and others are very permissive (but maintain only weak invariants).

To illustrate how we can use these different kinds of uniqueness, we'll conduct a small litmus test: whether reordering two statements will preserve the program's meaning, or not. In other words, we ask whether there is interference between the two statements:

```
a.foo();
x = b.f;
```

Definition 2 (Interference between statements) *Two statements interfere if one of them writes data that the other reads or writes.*

		immutability of b						
		none	shallow reference	shallow object	shallow class	deep reference	deep object	deep class
immutability of a	none			✓	✓		✓	✓
	shallow reference			✓	✓		✓	✓
	shallow object			✓	✓		✓	✓
	shallow class			✓	✓		✓	✓
	deep reference	✓	✓	✓	✓	✓	✓	✓
	deep object	✓	✓	✓	✓	✓	✓	✓
	deep class	✓	✓	✓	✓	✓	✓	✓

Figure 1.2. Assuming the references are immutable at least temporarily for the scope of the code snippet, the answer depends on which kinds of immutabilities the variables `a` and `b` have. ✓: reordering preserves program semantics.

Figure 1.2 shows the results. For example, if variable `a` is deeply reference immutable during the code snippet, that is *sufficient* to conclude that semantics will not change. This is, because line 1 can not change `b.f`, even though the reference in the variable `b` might be included somewhere in `a`'s transitive closure. However, that the variable `b` is deeply reference immutable, is *not sufficient*: a simple counter example would be if `a` and `b` are aliases, and the method call in line 1 changes the reference stored in `b`.

These kinds of immutability are used in practice; be it in the form of type abstractions, abstractions provided by libraries, or simply coding idioms. *E.g.,* on the restrictive side, we have permanent deep class immutability, like in purely functional data structures, *c.f.* Section 2.2.3. On the least restrictive side, we have temporary shallow reference immutability, like C's `const` pointers [38]. The overviews in Table 1.3 and Table 1.4 show some common use cases and implementations of the combinations of the kinds of immutability we can express in the classification from this section. We will, starting with Chapter 2, go through these systems and explain them one by one. In Chapter 3, we will talk about how commonly used some of these kinds of immutability are in practise.

Now that we have demonstrated what aliasing is, how it is problematic when mutable state is aliased, and what ways to mitigate the problems are, we are ready to survey some means of alias control that have been designed by practitioners as well as researchers.

Table 1.3. *Different forms of shallow immutability, and examples (the examples are not exhaustive) for where they are used. We will introduce those systems in Chapter 2.*

	Reference	Object	Class
Permanent	The references a collection has to its elements are usually not used for mutation. C/C++ `const` pointer when the referent may be `const`. Casting const-ness away and modifying the object would then be illegal [38] (*c.f.* Section 2.2.1).	Const values (`const T`, Section 2.2.1). A pointer that is both unique and a read-only reference (*c.f.* Section 2.2.1). C++ `unique_ptr<const T>` (“unique pointer to `const`”, Section 2.2.1).	C/C++ struct with only `const` fields or Java class with only `final` fields. Immutable collections or option types, when they contain mutable data (*c.f.* “conditional deep immutability”, Section 3.2.1).
Temporary	C/C++ `const` pointers to a non-const object (then const-ness can be typecasted away, Section 2.2.1). Java’s unmodifiable collections, when there are references to the inner collection left (Section 2.2.1).	Freezing a Java collection by moving (Definition 7) it into an unmodifiable collection (*c.f.* Section 2.2.1).	—

Table 1.4. *Different forms of deep immutability, and examples (not exhaustive) for where they are used. We will introduce those systems in Chapter 2.*

	Reference	Object	Class
Permanent	Javari read-only references (see Section 2.2.1).	`const T` (non-pointer type, see Section 2.2.1) in C++, if `T` encapsulates its reachable state. A permanent read-only reference that is known to be unique, *c.f.* Section 2.2.1. C++ `propagate_const <unique_ptr<const T >>` pointer (see Section 2.2.1) if `T` encapsulates the reachable state or uses `propagate_const` as well.	Immutable data structures (*e.g.,* [48]). Javari `readonly` class (see Section 2.2.1).
Temporary	C++ `T const *` (see Section 2.2.1), if the type `T` encapsulates its reachable state. If the referent is not actually `const`, it can be typecasted to `T *`.	Rust objects are immutable during periods of aliasing. Fractional permissions (Section 2.1.1).	Lazily initialised classes. Classes with caching (*e.g.,* java strings that cache the result of the `hashCode` method).

2. Language Abstractions for Alias Control

The big lie of object-oriented programming is that objects provide encapsulation.

J. Hogg [33]

Aliasing of mutable state is a double-edged sword. On the one hand, it can support useful programming idioms (Section 1.3). On the other hand, it can lead to bugs because how far an object is shared is not always clear.

Alias control (Definition 3), roughly, is the attempt to express in a program how, and whether, objects can be aliased and often includes protocols that may include additional constraints on the use of aliased data, related to mutation of aliased data.

Definition 3 (Alias control) *The term alias control was coined by Hogg et al. [34]. Alias control groups techniques that permit aliasing to happen, but do so in a way that permits a programmer to tell when this is the case. Aliasing control aids programmers to ensure that unexpected aliasing will never occur.*

In the next section, Section 2.1, we will survey techniques that let a programmer avoid or make them explicit in code in a way that is suitable to avoid bugs caused by unintentional aliasing. After, in Section 2.2, we will cover some techniques that are available to restrict instead the modification of data in order to safely share data.

2.1 Restricting Aliasing

There are two obvious solutions to avoid problems caused by aliasing of mutable state: to restrict aliasing of mutable data, or to restrict mutation of aliased data by using forms of immutability. This section covers the first: to restrict aliasing, while Section 2.2 covers the latter.

We will start with encapsulation, a pattern that is widely used in object oriented programming, and its limitations. We will continue with type systems that are designed to control precisely how references can be aliased in order to give more reliable means of understanding code than today's mainstream programming languages.

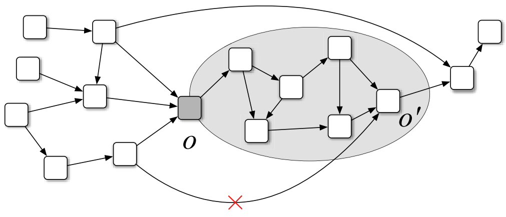


Figure 2.1. Object o encapsulates (Definition 4) the objects drawn within the bubble, including object o'. As o' is encapsulated, references from the outside (like the crossed reference) are prevented. The object reachable directly from o' is *not* encapsulated inside o, as there is a path of references to it that does not go through o.

2.1.1 Encapsulation of Mutable State

One the most well known techniques to restrict aliasing in programming practice is encapsulation (Definition 4). Figure 2.1 shows a partial object graph where an object o encapsulates several objects that are transitively reachable.

Definition 4 (Encapsulation) *An object o encapsulates another object o′ as long as o′ can only be accessed during method calls to object o. This definition of encapsulation is about reachability of data, unlike some definitions of the same term in the context of object oriented programming.*

Encapsulation is Difficult to Implement

A problem with encapsulation is that it is hard to enforce consistently. Consider the following class of financial portfolios that we have already used as an example in Section 1.3.4. A reasonable invariant is that a portfolio must always encapsulate the checking account within it, such that no outside code can access the checking account directly. This makes encapsulation a *deep* form of alias restriction.

```
1 public × class Portfolio {
2   protected Account acc;
3   public Account getAcc() { return this.acc; × }
4   public void setAcc(Account a) { this.acc = a; × }
5 }
```

This snippet contains several violations of encapsulation: passing data to the outside, passing data to the inside, and privileged access through inheritance. We now go through these in order.

Passing Data to the Outside

First, the getter in line 3 violates encapsulation: the `getAcc` method must not return the object referenced by the `acc` field—or otherwise, outside code could call `portfolio.getAcc().withdraw()` without going through the portfolio implementation. The getter could instead return a copy of the account instead: `return this.acc.copy()`. Other options would be to, if the languages has that feature, return a read-only reference to the account (Section 1.5), or move the account out of the portfolio by setting the account field to `null` before returning the account.

Passing Data to the Inside
Second, the setter in line 4 is broken: client code could install a checking account, retain a reference to it and use it later to withdraw money:

```
Account acc = new Account();
somePortfolio.setCheckingAccount(acc);
// ... after portfolio owner has put money in ...
acc.withdraw();
```

Instead, the account could use a *copy* of the `newAcc` that is passed in: `this.acc = newAcc.copy()`.

Privileged Access Through Inheritance
Third, the class is not marked `final` in line 1, meaning it can be inherited from: even after the `Portfolio` class is fixed by copying the account every time it crosses the encapsulation boundary, careless usage of inheritance can break encapsulation once again:

```
public class BrokenPortfolio extends Portfolio {
  @Override public Account getAcc {
    return this.acc; ×
  }
  // similar for setAcc
}
```

Note that not even preventing the subclass from accessing the `acc` field (by making it `private`) would prevent inheritance from causing a problem, as we could simply add a new account field in the subclass and override the setter, as well as the getter to use that new field instead.

This list of problems is not exhaustive[1], but shows that even for a single and simple class, enforcing aliasing intentions is surprisingly hard and – when cloning is used – costly. Also, importantly, *reading* the code is hard as well: the fact that encapsulation is present is never said explicitly. Alias control in mainstream languages is often painted in the negative space—by carefully omitting things, not by explicitly stating. In the next section, we will see systems that can help programmers enforce properties such as encapsulation in a more reliable manner by stating the properties positively, rather than by omission.

[1]we could, for instance, also use Java's reflection to make the `account` field public

2.1.2 Alias Control by Subdivisions of the Heap

One way to let programmers denote explicitly how much aliasing is possible is to divide the heap into disjoint regions of data by using a type system. Each object lives inside exactly one region, and usually, this is an object property (all references to the object agree). Most commonly, these regions are defined by objects like the portfolio in Section 2.1.1: an object could define that – for instance – every object reachable from it is considered to be *inside* it and therefore an alias can not be passed to the outside; similarly, no outside reference can be simply copied to the inside without making sure that that there are no outside aliases left.

By looking at the region a reference refers into, we can often rule out aliasing: if two references refer to objects in different regions, they may not be aliases; but two references referring to objects in the same region may be aliases. In such a system, a type usually carries information about which regions of memory the objects that are indirectly referred to from it (Definition 5) are in.

Definition 5 (Indirect Reference) *A reference r is an indirect reference to an object o if there is a non-empty chain of accesses $r.f_1 \dots f_n$ that evaluates to o. If two variables contain these references, we also call the variables indirect references to o.*

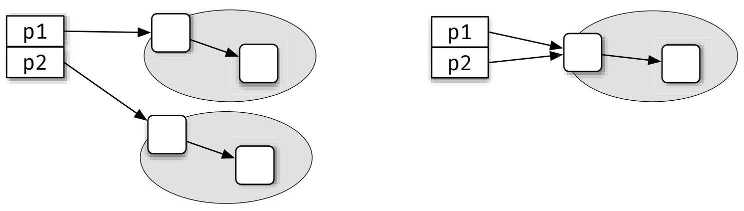
By reasoning about the regions of memory that are transitively reachable from two objects, we can often infer there can be no indirect aliasing (Definition 6).

Definition 6 (Indirect Aliasing) *Two references r, s, are indirectly aliased if there exists at least one object o such that both r and s are indirect references to o, in other words: if the transitive closures overlap. If two variables* `x` *and* `y` *contain these references, we also call the variables indirect aliases. The variables* `portfolio1` *and* `portfolio2` *in Figure 1.1a are indirect aliases because they are both indirect references to the objects † and ‡.*

We will provide an overview of this field and make an attempt to bring out some important similarities and differences of the work we present.

Islands and Balloons: Reliable Encapsulation

Islands, a proposal by Hogg [33], introduce the idea to enforce encapsulation (Definition 4) at a language level. An *island* is a region of the heap that contains all objects reachable from a *bridge object*. In order to modify the objects within an island, one must call methods on the bridge object.



(a) Valid islands and balloons. *(b)* Valid islands, not balloons.

Figure 2.2. Bridge objects in islands can be aliased, but not in balloons.

From a freshly allocated object, no other objects can be reached—a freshly allocated object therefore is a bridge object representing an (empty) island as it was just described. In order to maintain encapsulation, Hogg's work additionally demands some properties of bridge objects. As bridge objects are the entry points into a disjoint region of memory, by ensuring that

a) only unaliased references get passed inside a bridge object (including during the object constructor), and that
b) no references to the internal representation are copied outside the bridge object.

If we follow these rules, we get a guarantee that islands stay, in fact, disjoint from each other[2]. A nice property of the system is that bridge objects can be modularly type checked: an object is a bridge object if all methods accept only unaliased arguments and return only aliased data. Figure 2.2a and Figure 2.2b both depict valid islands: islands have a single bridge object that connects an encapsulated part of the heap (drawn shaded) with the surrounding memory, and bridge objects themselves *may* be referred to from multiple aliases.

Coming back to the portfolio class we talked about in Section 1.3.4 and Section 2.1.1, we can now show how this class could be implemented using islands (we use Java-like syntax for consistent presentation).

```
public bridge class IslandsPortfolio {
  protected Account acc;
  public @Unique Account getAcc() { return this.acc.clone(); }
  public void setAcc(@Unique Account a) { this.acc = move a; }
}
```

The island class may return only unique references, and accept only unique references as parameters to its methods (both made explicit in the getter by using an `@Unique` annotation that denotes a globally deeply unique reference). Assuming the account has a `clone` method that returns a unique account reference (as it returns a freshly allocated object), we can enforce that the getter

[2]References to deeply object immutable data can also cross through bridge objects even when aliased, but we will focus on the aliasing part of the system here.

returns an actual clone to the outside. Similarly, the fact that we only accept a unique reference to an account in the `setAcc` method forces a client to give up its own reference to the account. The `move` keyword is required to read a unique reference, it expresses that as a side effect of reading the reference in `a`, the variable is being set to `null` in order to maintain uniqueness (see Section 2.1.3).

The islands work features two important concepts as language abstractions:

– The notion of nested regions: an island can contain references to bridge objects, describing a deeply structured program memory,
– Unique references: in order to enforce rules a) and b) above, islands require consistent use of unique references for all parameter types and method return types.

The islands system requires annotations on variables, arguments, and fields that denote whether or not objects reachable through them are read-only, or whether the references are unique (as read-only data is exempt from having to be unique).

The islands abstraction is easy to understand, but also coarse grained. Consider a collection class that is a bridge object, and that we want to iterate over in a loop. In order to achieve that, the collection needs to remove every single element from itself, only to re-insert it once the loop is done with that item, as we can not have the item be referred from the outside (where the iteration loop is) and the inside at the same time. Figure 2.3a shows an example of such a situation, where a list exists, and a variable `current` should refer to one element of the list. Implementing this iteration with islands is possible but needs to deconstruct and reconstruct the list in the process.

Balloon types by Almeida [2] provide a type abstraction that has similar semantics to islands, with certain differences: the entry into a sub-region of the heap (what in islands is called bridge objects), the *balloon object*, can not be aliased from fields of objects. There can, however, be temporary aliases from stack variables. The fact that balloon objects can not be aliased is less expressive, but provides a stronger guarantee than islands, the guarantee that two variables referring to balloon references are always disjoint (with islands, the variables could be aliases).

Compared to islands, balloons do not require all references passed in or out of them to be unique. This works by using, instead of uniqueness at the boundary, static analysis for each balloon class. This analysis step ensures that methods of balloon types can not introduce illegal sharing between balloons—which, in the islands proposal is ruled out by the uniqueness requirement. Instead the static analysis tracks which balloon a reference is coming from and ensures that the reference does not get passed into another balloon. To track the flow of references accurately, the balloons system requires all balloon references to be unique. Compared to islands: balloons can store a reference to an element that is inside a balloon-collection in a stack variable, where islands need to elaborately shift references outside and back inside a collection to achieve the same.

Balloons have been successful in research: they have been applied to parallelism (*e.g.,* by Gordon et al. [26], Servetto et al. [55]).

Flexible Alias Protection and Ownership Types: Fine-Grained Encapsulation

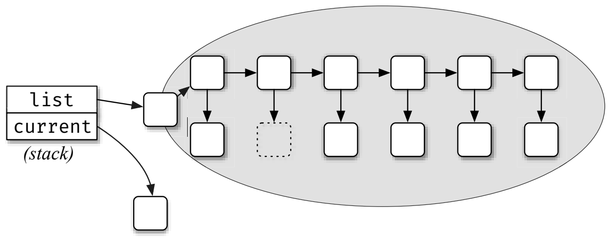
The "Flexible Alias Protection" proposal (short: FAP) and ownership types [16, 47] add an important idea to the abstractions used in islands and balloons: where islands and balloons insist on encapsulation of *all* the mutable state reachable from a bridge/balloon object, FAP and ownership types permit more fine-grained description of aliasing. Whereas islands and balloons only accept references from objects within a balloon to go to objects within the same balloon (modulo exceptions like references to immutable data), FAP and ownership types remove this limitation and permit references from one group of objects to another *as long as the references are annotated accordingly*. Figure 2.3b presents an example using ownership types: the list encapsulates all of the nodes, but the data are stored in a different region.

FAP achieves the increased granularity through inventing *aliasing modes*. Aliasing modes are reference annotations[3] that annotate each field or variable with the role they play. The most important modes are `rep` and `arg`: a field that is annotated with `rep` contains a reference to an object that is part of the field-owner's mutable inner side, here called the "representation": references to this object can never leak outside the object that holds the field—the invariant maintained for `rep` is similar to islands. In addition, however, flexible alias protection adds the `arg` mode: an object referred to from an `arg` field is a reference to a value stored inside a collection (called an "argument"). The collection can not modify the object through this field, only use read-only operations on the value.

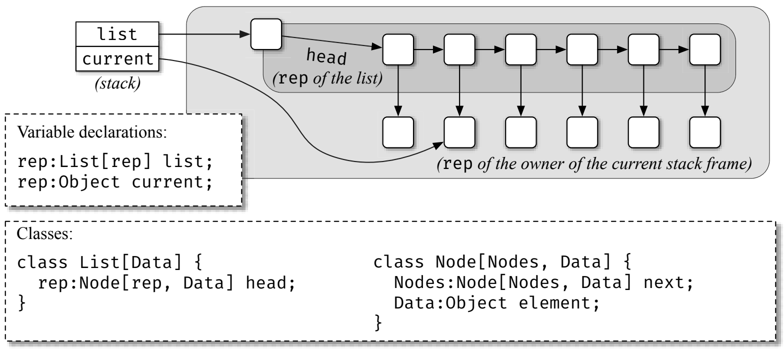
Collections can be parameterised by modes: the list can take a type parameter representing the aliasing mode of its data. A client can now bind *e.g.,* its own `rep` to the list's aliasing mode for the data. The ability to parameterise types is quite powerful, considering its relative simplicity. A list with iteration in FAP could be implemented easily: it would be parameterised with an `arg` mode to declare the access mode it should use to store its contained elements; the rest of its structure (the nodes), it would store using the `rep` access mode, as the list needs full write access. The object whose `rep` is bound to the list's `arg` (the "owner" of the data in the list) is then free to iterate over the data.

Ownership types Clarke, Potter, and Noble [16] were built on the ideas of flexible alias protection. Ownership types provide similar abstractions, but differ in minor ways; for instance, they do not enforce that objects may only

[3]An object itself does not have a mode, but all variables referring to the objects do; and the variables do not necessarily agree.



(a) In order to iterate over a list in the islands system, we have to move the elements out of the list.



Variable declarations:

```
rep:List[rep] list;
rep:Object current;
```

Classes:

```
class List[Data] {
  rep:Node[rep, Data] head;
}
```

```
class Node[Nodes, Data] {
  Nodes:Node[Nodes, Data] next;
  Data:Object element;
}
```

(b) While a list may always encapsulate its nodes, it may not encapsulate its data, flexible alias protection and ownership types (drawn) permit this use case. Syntax: a type `o:C[p,q, .. ]` denotes an object of class `C` that is in the ownership context `o` and binds ownership contexts `p`,`q`. . . to its domain parameters, meaning it may hold references into those domains.

Figure 2.3. Islands are semantically simple, but too coarse-grained for many situations. Flexible alias protection and ownership types are more complex, but also more expressive.

mutate data they own directly. In ownership types, objects are grouped in "ownership contexts". In Figure 2.3b, we can see different ownership contexts in use, the representation of the object *o* owning the current top stack frame in light grey, which contains the list; the list itself has its own representation (dark grey), which it uses to hide away the nodes it contains. It is parameterised over the light grey ownership context in order to make the elements of the list–unlike the nodes–accessible from *o* directly. Ownership contexts are nested regions of the heap, just like islands and balloons are. Every class instance in ownership types defines manages an ownership context of its own, called "rep"—its mutable representation, like in flexible alias protection. An object can, like in FAP, also be parameterised over ownership contexts. One difference between ownership types and flexible alias protection is that the former do *not* conflate aliasing with limitations on updates: modifications through an `arg`-moded variable are not permitted in flexible alias protection. Ownership types have been extended and implemented in various systems. For instance, they have been applied to parallelism [14, 49], and side-effects tracking using an effects system [12] that expresses, for each method which ownership contexts it reads from and which it writes to for more fine-grained expression of non-interference. A method that only writes to an ownership context effectively turns all references that refer into this ownership context into deeply read-only references, augmenting ownership types with a form of temporary reference-immutability.

Ownership contexts can be useful to describe mutation using effects: for example, Clarke and Drossopoulou [12] add an effect system to ownership types where methods annotate inside which ownership contexts they read from or write to objects. These annotations allow reasoning about non-interference of statements: two statements that only write to non-overlapping ownership contexts and/or that only read overlapping ownership contexts can not interfere with each other and can therefore be safely reordered, or executed in parallel. A class that takes an ownership context as a parameter that its methods only ever read from is similar to an `arg` parameter in FAP–but `arg` expresses this property directly, while in ownership type with effects, this property is less declaratively expressed by omission.

The access modes and ownership contexts we have covered are abstractions that permit aliasing *within* them, but avoid aliasing *between* them. They group the data into (ideally) small bubbles of unrelated state. As they provide forms of encapsulation, these systems are inherently forms of deep alias control – a very useful property for reasoning about programs. They do not readily lend themselves to express absence of aliasing *within* such a bubble, like expressing a list with set semantics (that contains no object twice). The other dominant way of alias control that we will cover next is a dual of sorts: by starting with references that have no aliases, they easily express the list with set semantics, but it is comparably harder for these systems to deliver deep alias control.

There are many more papers that could be mentioned here, like work on universes [20, 44, 52, 60, 61] that also subdivide the heap into nested regions, but these are not presenting new ideas that we will be necessary in order to explain the background of our contributions.

Disjointness domains, that we contribute in Paper I (also *c.f.* Chapter 4) relate to the systems in this section: they have the advantage that they can quite naturally express encapsulation as well as a list with set semantics in a unified system; however, they lack a true `rep` that confines an object's internals within it. While we think that this could be added to the system by adding an existential domain to each object that it may not expose to a client, the work as presented permits only a statically fixed number of shared domains that most closely resemble the semantics of `rep`. An advantage that disjointness domains have is that using unique references is the syntactic default (it requires the least amount of annotations), while in ownership types and FAP, strong invariants are achieved by putting data into many disjoint contexts which requires annotations.

2.1.3 Uniqueness, Linear Types, Permissions

In this section, we will talk about systems that avoid aliases of single references, for instance by enforcing that the reference is never copied, only moved (like in islands, that we have covered in Section 2.1.1).

We will show how this basic idea that is as useful as it is limiting, more expressive systems can be created.

Unique References

The systems that subdivide the heap into nested regions that we have just covered would avoid the problem in Figure 1.1a by enforcing that the two data structures may never share any data, while expressing the aliasing in Figure 1.1b by putting all the nodes inside the same ownership context/access mode.

Unique references are a simple language abstraction that can effectively avoid unintended sharing by avoiding sharing altogether. They are a means of conserving two of the listed core features of references (dynamically sized data structures in Section 1.3.1, and cheap passing of data in Section 1.3.2) but do not permit the last listed core feature, sharing of data, at all:

Definition 7 (Unique references) *are guaranteed to be unaliased [33]. A common way to implement unique references is to* move *references from variable to variable, rather than copying. For example, setting the source variable* `y` *to* `null` *as a side effect of the assignment operation* `x := y` *maintains uniqueness of the reference now stored in* `x`.

Another way that avoids using `null` and the resulting run time errors is to make `y` inaccessible for further reads as part of a type check (the Rust programming language does that) or a static analysis [4], or to use a `swap` operator that replaces the reference being read with a new reference atomically [30].

As *intentional sharing* of mutable data is a feature, prevention of all sharing is a limitation – in fact, a significant part of the work presented on the following pages will be dealing with carefully allowing limited sharing of data. Despite or due to their simplicity though, unique references are used relatively widely in programming practice, perhaps also due to the fact that many references in actual programs happen to be unique anyway (Chapter 3).

Unique references are often conflated with linear types (and we will generally use the term uniqueness to refer to both as well), a related concept, and while they have similarities, linear types still differ in important ways. These are described next.

Linear Types

Linear type systems are type systems where *linear values, are to be used exactly one time*—not zero or more times. Linear types integrate well with functional programming (*c.f.* work by Wadler [59]), where they have the advantage that values can be cheaply mutated in-place, without breaking equational reasoning.

Definition 8 (Linear and Affine Types) *A value of a* linear type *must be used* exactly *once [59]. In contrast,* affine types *enforce that a value of affine type is used* at most *once. These terms are sometimes used interchangeably, with the term "linear types" being used in place of "affine types".*

A major difference to unique references is that linear values must be used exactly once; this means that holding a value of linear type can be considered an *obligation* to consume it; the program will not pass type checking otherwise. This limitation can be very useful as a basis for certain advanced type systems that enforce usage protocols of a data type. Using type systems with support for linear types, it becomes possible to implement small examples like a class representing a file handle that enforces that the handle is closed after usage (because closing is the only way to consume the obligation [19, 23]); or to enforce that communication with a web server follows a certain protocol, like in so-called behavioural type systems like session types [35].

A related consequence of using each value only once is that a linear value can not exist in a shared value. Imagine that `s` is a shared variable, and `t` is an alias; if the referred object has a field `f` of linear type, we could access this field as `s.f` and `t.f`, breaking the use-once rule. This is a non-issue with unique-references, as they can be used many times, just not aliased. There is work addressing this issue, but it adds significant complexity to the type system. *E.g.,* work by Fähndrich and DeLine [23] introduces the `adoption` and `focus` keywords. In their system, all values are linear by default. But a linear value o_1 can *adopt* another linear value o_2. In turn, the user gets a non-linear reference to the adoptee. This non-linear reference now prevents accessing any linear components that the adoptee might contain in order to avoid breaking the use-once rule. To temporarily allow access to the linear components, they design a `focus` expression that lets a user temporarily recover the linearity of the adopted value at linear type; in order to make sure that multiple focus operations can never lead to several linear-typed values that are actually aliases, the type check removes the *adopter* from the type environment for the duration of focusing. This means that the adopter (that remains linear after adoption) can be used as a token that permits temporarily recovering linearity of *one of the adoptee's aliases at a time*. We classify the resulting system as globally temporarily shared.

Another difference is that where unique references *have a bottom value like* `null` *or* $\perp$ [4], linear values differ by not having `null`. While this difference might seem inconsequential, it is not: it means that when reading a field of linear type, the whole object containing the field it must be "consumed" (made inaccessible for later use in the program) at the same time; after all, the field can only be used one time, and the object containing the field can not exist without a value for that field. With unique references, the reference can be moved out of the object, setting the object's field to `null`. This may leave the object in a broken state, but a programmer could now choose to insert a fresh reference into that field, thereby restoring any invariants that may have been broken intermittently.

Unique References and Linear Types in Mainstream Programming Languages

Unique references are a widely used abstraction in the contemporary use of the C++ systems programming language. We are not aware of research that gives us empirical support for that statement, but the fact that the class is recommended commonly by text books [27, 57] and coding standards [25] is some evidence. The class `std::unique_ptr<T>`[5] represents an unaliased pointer to an object of type `T`. These pointers can not be copied using the usual assignment operation `x = y`, but instead, pointers have to be moved explicitly: `x = std::move(y)`[6]. This move operation will leave `y` in an empty state, dereferencing it is forbidden. As C++ is a language without garbage collection, unique pointers are useful for automatic reclamation of heap memory: when a unique pointer goes out of scope, the reference it contains can never be used again; when a

[4]Usually, but there are exceptions *e.g.,* Harms and Weide [30] suggest a `swap` operator to avoid `null`.

[5]`std::unique_ptr<T>` documentation: `https://en.cppreference.com/w/cpp/memory/unique_ptr`.

[6]The specific implementation details, although interesting, are not important for this document.

unique pointer is overwritten, the old reference can never be used again. Therefore, the pointer may automatically deallocate the object in both these cases.

The C programming language does not know of (globally) unique references, but it provides the `restrict` keyword to express a form of *local shallow uniqueness* (Section 1.4). The `restrict` keyword is used to notify the compiler of optimisation possibilities when certain pointers are unaliased locally, but programmers have to respect the constraints aliasing of aliasing imposed – or the program behaviour is undefined. A pointer that is annotated with the keyword, `* T restrict`, points to an object that (or parts of which) may only be accessed through that restrict pointer, never others. This restriction holds only as long as the pointer is used.

In programming languages research, unique references have been used by the islands proposal, and balloon types (Section 2.1.2) to support a "transfer of ownership": moving a unique reference from one object to another means that the other object becomes responsible for managing that object. In type systems that divide the heap into non-overlapping regions, it becomes possible to transfer a uniquely referenced object from one heap-region into another soundly, as it is clear that no reference to the object remains in the old region.

The Clean [56] programming languages uses unique references to be able to mutate data, while keeping referential transparency. One advantage of that is that, for instance, arrays in clean can be changed in place without needing to use monads (like Haskell's state monad).

The Rust programming language comes with the `std::Box<T>` type, that represents a unique reference to a heap-allocated object of type `T`. Compared to C++'s `unique_ptr`, an assignment operation of boxes `x = y` in Rust makes the source variable `y` inaccessible *by employing a type check*, it is therefore not possible have run-time crashes due to a dereference of an empty box pointer. We can model semantics closer to C++'s `unique_ptr` *e.g.,* by wrapping the box in an option type (where the `None` variant represents a pointer that has been consumed), using the type `std::option<std::Box<T>>` instead, should such behaviour be intended. One reason to do that would be to avoid having to consume the whole object holding a unique reference when the unique reference needs to be moved out of an object—the option type can be set to `None` instead (*c.f.* Section 2.1.3).

Unique references and linear types, as we have just presented, are means of preventing aliasing; fractional permissions, the next system, is an extension of linear types that adds a means of temporarily aliasing references; but aliased references my not be used to cause updates.

Fractional Permissions

In work by Boyland [5], *fractional permissions* quantify variables with a "degree of ownership": a variable can fully own the data it references. But data can also be aliased, in which case the permission of all aliases will express that they *only represent ownership of a fraction of the data.* Boyland uses this system to guarantee non-interference by only allowing mutations of fully owned data. References may always be aliased, but in the process, we must accept that modifying the data they refer to is no longer possible. An object may therefore be mutable during some program phases, and deeply immutable during others, we classify fractional permissions as a form of temporary deep immutability.

The syntax for permissions in this system is as follows (ρ denotes an abstract location that is used to reason about aliasing of data *c.f.* [**alias-types**, 1]):

base permission	β	::=	$v : \mathrm{ptr}(\rho) \mid \rho$
fraction	ξ	::=	$1 \mid \varepsilon$
partial fraction	ε	::=	$z \mid 1 - \varepsilon \mid \varepsilon \cdot \varepsilon'$
fractional permission	π	::=	$\xi\beta$

A fractional permission can be either a full permission (1β is a full permission to access the abstract location ρ in memory, permitting mutation of the object; and $1v : \mathrm{ptr}(\rho)$ is a full permission to read and re-assign reference variable v that currently refers to location ρ') or a partial permission $\varepsilon\beta$. Two aliases x and y can have the partial permissions $z\beta$ and $(1-z)\beta$ respectively. If we alias the permission $z\beta$ further, the resulting aliases would have the permissions $z \cdot z'\beta$ and $z \cdot (1 - z')\beta$ for a fresh z'.

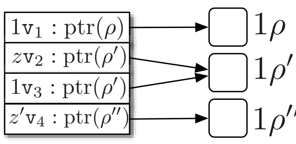
The *permission environment* Π is a set of permissions π of the form $\xi\beta$. Type checking guarantees that a heap location ρ can only be modified if ρ is fully owned (in other words, if $1\rho \in \Pi$). Additionally, a variable can only be assigned if the variable is fully owned (in other words, if $1v : \mathrm{ptr}(\rho) \in \Pi$). Variables carry the abstract location ρ they refer to at a given time in their type (the ptr(ρ) part), and this permits tracking what abstract location a reference refers to at a given time. For example, Figure 2.4 lists a parallel expression involving four variables. The context to the left (Figure 2.4a), this expression is well-typed. In the context to the right (Figure 2.4b), it is not: The left sub-expression of the parallel composition || requires at least partial ownership of ρ' (in order to dereference *v2). At the same time, the right sub-expression needs *full ownership* of ρ' (in order to execute *v3 := ... , after storing ρ' into v3).

To summarise, type checking fractional permissions differs from "standard" linear types in two crucial details:

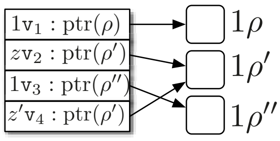
1. Where linear types are unaliased, here aliasing is tracked in terms of abstract locations β, and
2. a linear reference can be split into read-only aliases with smaller fractions during type checking with the constraint that a partial permission may not be mutated.

That means that fractional permissions constitute a means of bringing aliasing back, even for linear values, without losing the freedom of side effects that is required in pure functional programming.

Fractional permissions relate to our work on disjointness domains in Paper I: there, we also build a system that permits temporarily aliasing objects. Dis-


$1\mathrm{v}_1 : \mathrm{ptr}(\rho)$
$z\mathrm{v}_2 : \mathrm{ptr}(\rho')$
$1\mathrm{v}_3 : \mathrm{ptr}(\rho')$
$z'\mathrm{v}_4 : \mathrm{ptr}(\rho'')$
1ρ
$1\rho'$
$1\rho''$

(a) An environment in which the expression is *well-typed.* The permission $1\rho'$ is duplicated to both subexpressions (the first subexpression receives the permission $z''\rho'$, the second receives $(1 - z'')\rho'$). Type checking with partial permissions to ρ' works because they only need read access to the abstract location ρ'.


$1\mathrm{v}_1 : \mathrm{ptr}(\rho)$
$z\mathrm{v}_2 : \mathrm{ptr}(\rho')$
$1\mathrm{v}_3 : \mathrm{ptr}(\rho'')$
$z'\mathrm{v}_4 : \mathrm{ptr}(\rho')$
1ρ
$1\rho'$
$1\rho''$

(b) An environment in which the expression is *ill-typed.* After the assignment `v3 := v4`, the `v3` variable refers to the ρ' locations. But to type check the `*v3 := ..` assignment, a full permission to ρ' would be needed; this full permission is not available, due to the fact that the left parallel subexpression requires at least a partial permission in order to execute `*v2`.

Figure 2.4. Fractional permissions example. Type-checking the parallel expression `join { fork { *v1:=*v2; v1:=v4 }; fork { v3:=v4; *v3:=3+*v2 } }`. The `*` operator denotes dereferencing, `:=` is an assignment, and `join { fork {` e_1`}; fork {`e_2`} }` executes the two expressions e_1 and e_2 in parallel, then joins the two threads. Example program from Boyland [5].

jointness domains differ from fractional permissions in that they do not prevent aliasing or mutation at all (they are descriptive). Also, they use unique references as a starting point, together with a `move` operator.

2.1.4 Summary

We have now covered several means of restrictions of aliasing: systems that subdivide the program's heap, and systems that finely control the aliasing of individual references, all with various levels of expressive power.

In several of these systems, we have already mentioned that they achieve greater expressive power by adding flavours of immutability to have aliasing that would otherwise be deemed unsafe. The next section will cover how we can use forms of mutability to mitigate the negative effects of aliasing, instead of constraining the flow of references through our programs.

2.2 Preventing Modification

Interference requires that shared data is modified and read. The last section showed how interference can be avoided by limiting sharing, but limiting sharing can be difficult, especially in mainstream languages. This section shows how to avoid interference by limiting the modification of shared data instead.

This approach is commonly used in programming (functional programming languages being a prominent example), yet it has disadvantages. This section will survey systems, libraries, and languages that aid programmers in reliably implementing immutability. In order to understand better the situation that the field is in today, we will again pay extra attention to some mainstream languages, and the support for different kinds of immutability that they provide.

2.2.1 Immutability through Read-Only References

A way to prevent modifications of † and ‡ in Figure 1.1 is to classify all methods of † according to whether they mutate the object itself or objects reachable from it—or not. Then, the fields `f` and `g` can be marked as deeply read-only (a version of reference immutability), meaning that via these fields, only methods that do not mutate data may be called. We can also implement shallow read only references that way by only tracking mutations of the object itself, not reachable state. In contrast to object immutability and class immutability like in functional data structures (which we will cover in Section 2.2.3), read-only references are a form of reference immutability: they do not necessarily prevent objects from changing! Read-only references only prevent modifications going through that one reference, or copies of it but permit the existence of other writable aliases. Once the object is shared between the aggregates, we can limit the effects of sharing by converting, for instance, both references to deep read-only references for as long as sharing continues: in Figure 2.10a, we have drawn the same data and overlayed a graph as dashed arrows that denotes the direction in which data can be read and written: a field that can be used to *read and write* lets values travel in both directions (aligned with the pointer direction in the case of a write, against the pointer direction in the case of a read), but a field containing a read-only reference can only let data travel against its reference direction. It becomes clear that no values can flow from the aggregate after `x` to that after `y`, as `y` is not reachable from `x` (or vice-versa). While this property does not exclude modifications coming from other possible aliases, it can still be useful (for instance, for running operations on the two aggregates in parallel). But we could, depending on the invariants required, choose to convert only one of the references. Depending on which of the references (if any) we make deeply immutable, the possible interferences between statements involving the variables `x` and `y` in Figure 2.10a change:

1. Making reference `f` deeply read-only, we can prevent value flow from statements using `x` to statements using `y`;

2. making reference `g` deeply read-only, we can prevent value flow from statements using `y` to statements using `x`.
3. making both references `f` and `g` deeply read-only, we can implement non-interference between `x` and `y`,
4. if we know that `f` and `g` store the only references to object † and that object † encapsulates object ‡, making `f` and `g` read-only references is sufficient to conclude that the immutability is object immutability (Section 2.2.1).

Constructing Object Immutability from Reference Immutability with Knowledge of Aliasing

Even though read-only references may help implementing non-interference, their power is limited – read-only references can profit from some kind of reasoning addressing aliasing. Consider the example in Figure 2.10b: here, we have again marked the fields `f` and `g` as read-only (again, drawn as single dashed arrows). But we have added a writable alias, via field `e` (marked ×) from an external object to object †. This means that we can use this field to modify object † or object ‡. If we can guarantee that *all references* to † are deeply read-only, we can conclude that † and ‡ are, in fact, *object immutable*.

This has practical consequences: if all we have done is make `f` and `g` deeply reference immutable, this gives us the ability to execute statements in parallel:

```
// the finish block waits for its inner
// forks to terminate before continuing
join {
  fork { x.foo(); } // fork runs its inner block in a separate thread
  fork { y.bar(); }
}
```

That join is necessary, as later statements might modify †, ‡ using the external writeable alias; this would constitute a data race.

But if we are able to rule out any further writeable aliases to † or ‡, we require less synchronisation:

```
// we can immediately continue execution without joining,
// as no other statement may interfere with any of the
// spawned threads
fork { x.foo(); }
fork { y.bar(); }
```

Read-only references are available—with differing semantics—in commonly used programming languages and research proposals. The technique to be able to guarantee object immutability using only knowledge of the absence of aliasing and reference immutability is indeed very useful: it can be used by a static analysis and type systems to gain the ability to express object immutability. Object immutability would be hard to track otherwise, as it would require a static alias analysis. Versions of this technique have been used Gordon et al. [26] and Milanova and Dong [42] (*c.f.* Section 3.1).

The rest of this section will survey selected implementations of read-only references ordered in (roughly) increasing order of abstraction.

Pointers to Const in C

The `const` keyword in the C programming language allows defining a pointer that may only be used for reading a value, but not writing it. A variable with the type `T const *` is a pointer to a value of type `T` that may not be changed *using that pointer*, but a pointer to `const` does not preclude the existence of writable aliases, making it a form of reference immutability. In addition, `const` is shallow, as data reachable from the object being protected is still freely mutable.

Const can *also* be applied to values directly. The type `const T` (where `T` is not a pointer) denotes a `T` value that is object immutable. But unlike in stronger type systems, object immutability is not something the compiler enforces for the programmer, it's rather something the compiler expects of the programmer to enforce themselves: whether or not a reference to const is a form of permanent immutability or not is potentially very hard to find out in a C program: if the value referred to by the pointer was not defined to be object immutable (*e.g.,* `int x`, rather than `const int x`), then it is legal to type-cast `const` away, making it a temporary form of immutability. But if the value *was* declared to be object immutable (*e.g.,* `const int x`), then casting it away is illegal. A safe default is to treat pointers to `const` as permanently immutable.

The listing in Figure 2.5a shows an example of shallow semantics: a `struct node const *` pointer only protects the node it is referring to from modification, but not the other nodes reachable after it. This means that pointers to `const` are not sufficient to guarantee non-interference. We can, even though this takes a lot of work and is error-prone, implement deep `const`: consider the code in Figure 2.5b. There, we have added accessor functions `next_const( .. )` and `next( .. )`. The idea is that if we only have a pointer to a `const` node, we can only call `next_const`, which gives us yet another pointer to `const`; but using a non-const pointer, we can also call `next` for mutable traversal of the data structure. However, for nodes that have more than one pointer field, we would need to replicate these accessor functions for each field, and implement deep const for each data type they refer to, making this approach laborious in practice.

Table 2.1. *C pointer syntax: reading the type backwards explains the meaning.*

Type	Read as..
`T*`	Pointer to `T`.
`T* const`	Constant pointer to `T` (will always store same reference, but supports mutating the object).
`T const *`	Pointer to constant `T` (reference immutability).
`T const * const`	Constant pointer to constant `T` (reference immutability).

```
struct node {
  struct node *next;
  int elt;
};

// ...

struct node const * n =  ... ; // allocate new node
(n->elt)++; // × prevented modification
(n->next)->elt++; // ?? permitted modification
```

(a) A pointer to `const node` does not convert pointers read through it to pointers to `const node`. Therefore, only the first node is protected from writes, but not the second, nor any subsequent node.

```
struct node const *next_const(struct node const *n) {
  return n->next;
}

struct node *next(struct node *n) {
  return n->next;
}

// ...

struct node * const n = init(); // allocate node
next(n)->elt++; // × prevented calling next
next_const(n)->elt++; // × prevented modification
```

(b) Using explicit accessors that preserve the `const` qualifier of the pointer, we can enforce that a non-const pointer can only be obtained from a non-const node, thereby implementing a deep version of `const`. However, this requires care and needs to be done for each field (like `next` in this example) individually.

Figure 2.5. Expressing deep immutability using const-pointers is possible, but laborious.

```
struct node {
  unique_ptr<node> _next;
  int _elt;

  node(int elt) { this->_elt = elt; } // constructor

  void insert(int x) {
    unique_ptr<node> oldNext = std::move(this->_next
        );
    this->_next = std::make_unique<node>(x);
    this->_next->_next = std::move(oldNext);
  }
};
```

(a) A node in C++ can express that its next node is an unaliased reference; the insert method then needs to *move* rather than copy references.

```
node writable = node(1);
writable.insert(2);

const node& read_only = writable;

read_only._elt++; // × modification prevented
read_only._next->_elt++; // ⁇ modification permitted
```

(b) Even using the `unique_ptr` class in C++, immutability remains shallow.

Figure 2.6. A `const` unique pointer does not apply the const-ness to its referent, just like a `const` pointer.

C++

Like in C, pointers and references[7] in C++ implement a shallow version of `const`. As Figure 2.6 shows, we can construct a node class using a unique pointer (unique pointers do not permit their referent to be aliased from other unique pointers). C++ adds several relevant differences to C: const methods and smart pointers.

[7]Although C++ has const pointers like C, we shall use references in our examples in order to match idiomatic C++. The difference between pointers and references is not important for these examples.

```
  struct node& next() { return *_next; } //null checks omitted
  const struct node& next() const { return *_next; }
```

(a) Using a similar trick to the one for C `const` pointers in Figure 2.5b, we can create two overloaded getter methods for nodes to implement deep immutability for the `node` type.

```
  writable.insert(2);
  const node& read_only = writable;

  read_only._elt++; // × prevented modification
  read_only.next()._elt++; // × prevented modification
}
```

(b) As `read_only` is a const reference, only the const overloading of the `next` getter can apply; therefore, the result of `next` is also protected from writes.

Figure 2.7. Implementing two getters for the `next` field, we can achieve deep immutability semantics.

Using Const Methods to Implement Deep Const

C++ adds the possibility to have `const` annotations on methods. A method that is annotated as `const`, intuitively[8], is a method that will not modify the object it is called on. If an object in C++ is object immutable (`const T`, like in C), or referred from a pointer-to-`const` (*e.g.,* `T * const`, `unique_ptr<const T>`), or a C++ `const` reference (`const T&`), only methods annotated as `const` can be called on it. Programmers can use this mechanism for implementation of deep `const`.

By using encapsulation, we can implement deep const semantics. By making the `_next` field private and instead adding two overloaded getters like in Figure 2.7, we can use C++'s type system: the getter that returns a writable reference (type `node&`) is not marked as `const` method. From the type system's view, obtaining a writeable reference into the node now changes the node and is there not permitted on a `const` node. The other getter returns a read-only reference (type `const node&`), and that getter is marked using the `const` keyword—the type system will allow calling this getter even on a `const` node. This way, we ensure that a user can only obtain `const` references to the `next` node of `const` nodes. We still need to take great care to not accidentally return any mutable references (direct references or indirect references as part of a return value) to the outside.

Using Smart Pointers to Reduce the Complexity of Deep Const

Using C++'s smart pointers, the solution using overloaded getters can be made easier and more reliable. Consider the node declaration in Figure 2.8a. Instead

[8]The precise semantics vary with the version of C++ [31].

of declaring two overloaded getters for the field, a user can instead use a wrapper for a pointer that gives the wrapped pointer `const`-propagating semantics. This means that if the pointer itself is `const`, then the pointed-to value will also be `const`. This means that the safety feature that we had to implement manually by carefully implementing appropriate getters in Figure 2.7a can be implemented once in the standard library in the form of a wrapper class for pointers, and then just re-used. It also makes the code easier to read, as this property is now documented in the field declaration, rather than implicitly in the method implementations. Figure 2.8a shows usage of the class `propagate_const` that is being evaluated for addition to the C++ standard at the time of this writing. The authors of the proposal to add `propagate_const` to C++'s standard library, Jonathan Coe and Robert Mill [39], include a comment that is especially remarkable in the context of this thesis[9]:

> “ *Given absolute freedom we would propose changing the* `const` *keyword to propagate const-ness. That would be impractical, however, as it would break existing code and change behaviour in potentially undesirable ways. A second approach would be the introduction of a new keyword to modify* `const`*, for instance,* `deep const`*, which enforces* `const`*-propagation. Although this change would maintain backward-compatibility, it would require enhancements to the C++ compiler.*
>
> — Jonathan Coe and Robert Mill [39]

Reference Immutability in the Rust Programming Language

Unlike C and C++, the Rust programming language syntactically defaults to immutable semantics: mutable data has to be annotated using the `mut` qualifier that has semantics that are different from C/C++. In particular, `mut` is a deep property. Declaring a node like in Figure 2.9, the fact that reference `r0` is a read only reference prevents us from modifying any node of the list. Rust's type system enforces that, at any one time, there can be either at most one mutable reference or any number of read-only references to a datum [3]. This constitutes a big difference between Rust and C++: Rust propagates immutability by default, like Coe and Mill [39] in the last section suggested. Not giving `mut` deep semantics, would violate the reference semantics, as two writable references to a single node could then be obtained.

In practice, this means that Rust programmers have to do extra work to achieve interference when it is really needed: where the `Box<T>` type consti-

[9]This stance is further backed by the fact that `const` methods, starting with C++11 but unlike C++98 and earlier, need to be thread safe. This is because the standard library is not obligated to explicitly synchronise `const` method calls [31].

```
struct node {
  std::experimental::propagate_const<unique_ptr<node>> _next;
  int _elt;

  node(int elt) : _elt(elt) {} // constructor

  void insert(int x) {
    auto oldNext = std::move(this->_next);
    this->_next = std::make_unique<node>(x);
    this->_next->_next = std::move(oldNext);
  }
};
```

(a) Using the `propagate_const` pointer wrapper for the `_next` field in the node gives the node deep immutability semantics.

```
  node writable(1);
  writable.insert(2);
  const node& read_only = writable;

  read_only._elt++; // × prevented modification
  read_only._next->_elt++; // × prevented modification
```

(b) Even without explicitly implementing the two overloaded getters, a `const` node reference is now deeply protected from writes.

Figure 2.8. Although this feature is not widely used, C++ supports safe means of deep immutability.

```
struct Node {
   elt: i32,
   // Box<T> can not be 'null', therefore use Option<..>:
   next: Option<Box<Node>>,
}

// ...

let mut n0 = ... ;

let r0: &Node = &n0; // read only reference to n0
r0.elt += 1; // × compiler prevents modification
let r1: &Node = r0.next.as_mut().unwrap(); // × compiler prevents
     taking mut reference
r1.elt += 1;
```

Figure 2.9. Unlike C and C++, Rust uses deep immutability by default.

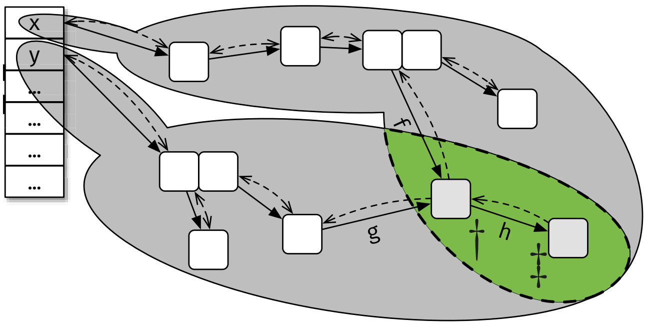
tutes an unaliased pointer[10], a Rust programmer may use a variety of reference-like (similar to smart pointers in C++) classes with weaker guarantees. For instance, the type `RefCell<T>`[11] to share a `T` value between different locations. A `RefCell<T>` permits modifications from different locations, but not concurrently: before changing the value contained in a `RefCell`, the programmer needs to explicitly borrow the value for writing, but that borrowing must end before the value can be borrowed again. This linearizes the accesses to a shared value. The `RefCell` ensures that, if two borrows happen at the same time, an error is raised. By picking from a rich set of pointer semantics, a Rust programmer can gradually decrease the power of reasoning, but increase the power of expressivity. The difference to the other mainstream languages here is that the language uses safe defaults, and requires extra work for things that are potentially dangerous.

Reference Immutability in Java

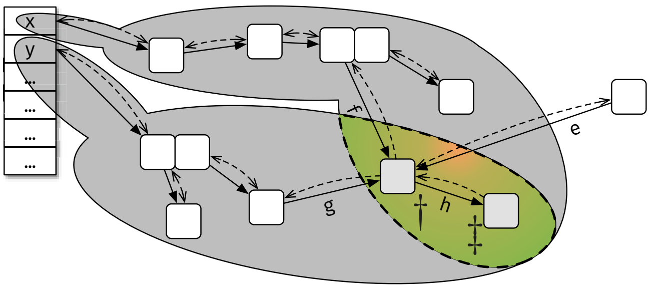
Java `final`

Java comes with the `final` qualifier (already mentioned in Section 2.1) that, when used on a field or variable declaration of reference type, states[12] that this field or variable will always refer to the same object. That all final fields are initialised during object construction is checked during compilation, and the compiler prevents re-assigning final fields. This is similar to a constant pointer in C/C++, rather than the pointers-to-const we have mostly talked about above

[10] `std::boxed::Box` documentation: `https://doc.rust-lang.org/std/boxed/index.html`.
[11] `std::cell` documentation: `https://doc.rust-lang.org/std/cell/index.html`.
[12] This protection can be circumvented using reflection.



(a) When all references to object † are deeply read-only, object † is deeply object immutable.



(b) Deep read-only references can be used to construct non-interference between aggregates (like the data reachable from x and y. In order to get deep immutability, writable aliases have to be prevented.)

Figure 2.10. The example from Figure 1.1 with deep read-only references f and g. Value-flow is drawn as dashed lines. Fields that can be used for reading and writing their referent can be used to move a value in direction of the reference and back (drawn ←−−−→). Fields that can only be used for reading can be used to move a value *against the reference direction* (drawn ←−−−−).

(`T * const`, rather than `T const *`): the fact that the variable can not be re-assigned does not protect the referred-to object from modification.

```
final Node n = new Node();
n = new Node(); // compiler prevents re−assignment
n.next = new Node(); // compiler does not prevent modification
```

Unmodifiable Collections
Even in languages that do not support static typing for read-only references, we can sometimes write code that gives us similar functionality. What this demonstrates is that static types are not the only way to reach this feature. An example is Java: it does not have read-only references as part of its type system, limited support for read-only references can however be implemented (and is commonly used) dynamically. So-called "unmodifiable collections"[13], are wrappers for a collection data structure that ensure that the wrapper can not cause any update to the contained data structure.

An unmodifiable collection, in Java, is a collection that does not support any update operations. It will throw an exception when code tries to modify it, rather than prevent a modification at compilation time. Unmodifiable collections protect the collection itself from modification, but not the elements contained in the collection, which may still be modified and are therefore a shallow form of immutability.

A disadvantage of unmodifiable collections is that, effectively, a part of the interface is disabled at run-time: all methods that are supposed to update the collection are overridden to throw an exception instead, and the type system does not track help us to avoid them. In Java, unmodifiable collections are a subtype of their mutable counterparts, meaning that the type system can not prevent users to attempt to call these methods. This means that erroneous modifications will only be found at runtime, rather than during compilation.

Java Type System Extensions for Deep Read-Only References
Even though Java does not have reference immutable types, there is research on adding those to Java. We will describe one such type system here, Javari by Tschantz and Ernst [58], but Javari is not the only system that adds read-only references to Java-like languages [36, 45, 60, 61].

Javari [58] is a Java dialect that adds read-only type annotations to Java types and classes. These annotations have deep semantics, meaning that neither the object, nor any object reachable via a read-only reference may be mutated, but explicitly permitting other variables that are not annotated as read-only to make such modifications.

[13]Documentation for `unmodifiableList`: `https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableList(java.util.List)`; other collections are supported using similar methods.

Java's `final` variables and fields and Javari's `readonly` are orthogonal: for example, a variable may be assignable (non-final), yet still read-only, meaning that the object referred to from this variable may change, but the variable may not be used to cause modifications on any of the objects it references, and that it may not be used to obtain writable references to any of its internal state.

When accessing a field of an object, that field's mutability mirrors the mutability of the object hosting it—a field of an object accessed through a read-only reference is `final` and `readonly`[14].

Javari treats mutable references as being of a subtype of a read-only reference. Therefore, an assignment `readonly Node x = mutableNode` is permitted, but not `Node y = readonlyNode`, as this would permit to obtain a writable reference from an originally read-only reference. One may think of a type `readonly T` as an interface containing only non-mutating methods, whereas `T` has the full set of methods.

In order to protect state reachable from an object, the programmer is required to split the interface of a class into those two parts explicitly, by annotating methods with information on whether or not they require read-only or full mutable access to the method's implicit `this` parameter, and type checking is extended with support for only allowing methods requiring mutable access to be called on mutable objects.

Javari offers similar behaviour as const-propagating pointers when reading fields from objects: as explained above, the reference read from a read-only referenced-object are in turn going to be read-only. However, Javari adds a further qualifier for ergonomics (this qualifier serves to avoid code duplication), called `romaybe`. A method that is marked as, or uses a parameter or return type marked as `romaybe`, is considered to be templatized: during type checking calls to that method may be type-checked in two ways: either with all occurences of `romaybe` replaced by `mutable` or with all occurences of `romaybe` replaced by `readonly`. Figure 2.11 shows an example of a class using this feature: the method `getVal` can change its annotation of the implicit `this` parameter, depending on whether the result will be used in its read-only or in its mutable form. Should the result reference be mutable, then *obtaining the reference already is considered a side effect by the type system, similar to calling the non-const getter in our above section on C++*. This feature permits soundly tracking aliasing of the object's internal state by registering a mutation before it happens, making complicated tracking of aliasing of the result reference unnecessary.

In Figure 2.5 and Figure 2.7, we have seen that—for example—getter functions or methods need special treatment when using const-pointers to implement deep immutability. The typical way in C/C++ was to provide accessors `next` to get a mutable pointer to the next element of a mutable node and

[14] An exception holds for fields that are annotated as `mutable` which serve to bypass `readonly` protection, for use cases such as result caching.

```
class Node {
 private Object value;
 Node next;

 // If the result gets stored into
 // a readonly variable, romaybe
 // becomes readonly, otherwise
 // mutable:
 romaybe Object getVal() romaybe {
  return this.value;
 }
}
```

(a) The `romaybe` qualifier behaves as if there were two overloaded methods, one where all occurences of `romaybe` are replaced by `readonly` and one where they become `mutable`.

```
readonly Node n = ... ;

// does type check:
readonly Object roval = n.getVal();

// does not type check:
mutable Object mutval = n.getVal();
```

(b) Type checking picks the overloading that is applicable.

Figure 2.11. The `romaybe` qualifier permits templatisation wrt. mutability, thereby avoiding code duplication.

`next_const` to get only a `const` pointer to the next element of a `const` node. This led to duplication in both languages. We showed, in Figure 2.8, that in C++ the duplication can be avoided by using const-propagating smart pointers. However, to our knowledge, C++ does not offer a reasonably easy way to implement a feature like Javari's `romaybe`.

In addition, as we will cover in Section 3.1, Javari has been extended with support for inference of its annotations.

2.2.2 Limitations of Reference-Immutability

Reference immutability and read-only references as an implementation thereof are not without problems; their semantics are not strong enough to express many desirable properties.

Consider again our example of portfolios. We have made our class secure earlier by returning a copy of the account to our client, and by copying all accounts that the user of the class passed inside. But copying bank accounts wastes memory[15].

We can therefore attempt to make the class secure by using read-only references, rather than cloning:

```
public final class Portfolio {
  private Account acc;

  // return only a read-only reference to the client
  public readonly Account getAcc() { return this.acc; }

  public void setAcc(× readonly Account a) { this.acc = a; }
}
```

What we quickly discover is that reference immutability lets us protect ourselves from returning inner data to the outside writably, but it does not let us prevent the user from retaining a writable alias. The user can still withdraw from our portfolio's account. This problem is also identified by Boyland [6], who suggests a solution: using a form of uniqueness for the setter (that uniqueness needs to come in a deep variety that prevents the user from retaining references *into* the account as well [13]).

Additionally, the read-only references we have presented have prevent us from using much of the code we write, namely the mutable part of the interface of a class becomes unusable when using a read-only reference. This particular problem is addressed by functional data structures in the next section.

[15]It also amounts to printing money, which might raise some eyebrows in the accounting department and break all sorts of invariants in the larger program.

2.2.3 Immutability Through Functional Data Structures

The read-only references we have described in Section 2.2.1 have a crucial disadvantage: in order to prevent problems caused by sharing of mutable state, we are sacrificing parts of the functionality of our data structures: all code that mutates them is simply made unavailable.

Functional data structures are data structures that are designed to never be mutated, no matter under which circumstances, a form of class immutability; instead of *changing in place*, a functional data structure will instead return a new instance that has the change applied to it.

Looking again at Figure 1.1, instead of making fields `f` and `g` mutable, we can implement the objects † and ‡ as functional data structures. If any operation from the aggregate `x` then changes one of the shared objects, it will not mutate the data in place, but rather receive an updated, new reference to a freshly allocated object back.

This fresh object is sometimes referred to as a "version" of the data structure:

Definition 9 (Versions) *We use the term "version" to denote a snapshot of a functional data structure at one point in time. Mutating a version is not possible, but obtaining a version that represents a altered data structure (*e.g.*, containing the same elements as the previous version, but with one added element, etc.) is.*

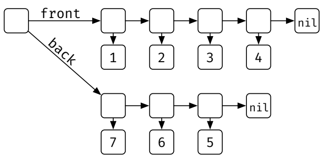
In this section, we will do two things: first, we will show why functional data structures are no silver bullet; second, we will show how even functional data structures heavily depend on restriction of aliasing and therefore, even though they are more commonly seen as an *alternative* to alias control, can themselves profit from tightly controlling aliasing: they depend on the absence of aliasing for performance.

Functional Data Structures Can be Slower Than Their Mutable Counterparts

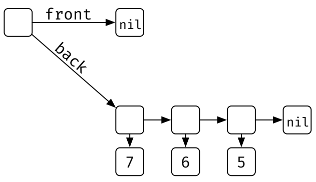
A special power of shared mutable state is that a single update can change the meaning of all objects indirectly referencing (Definition 5, p. 32) it. If all data is immutable, we give up this power. In Section 1.3.3, we promised to introduce an example of the linked list that maintains a `last` pointer in order to achieve $\mathcal{O}(1)$ complexity for append new data at the list's end.

Changing a mutable list to support constant-time append is simple: we add a reference to the last node in a list, and update this reference for all insert and remove operations; appending to the list then is to simply dereference the reference in the `last` field, modify that node by inserting a new node after it, and updating the `last` reference to refer to the newly inserted node.

Supporting constant-time append in immutable lists is significantly more involved than for the mutable list. Additionally, we have to work harder in this instance, and – again, in this instance – have to accept that some operations only have good *amortized* complexity, rather than worst case. A well-known functional list-like data structure that supports a constant-time append is the the "deque" (for double-ended queue) by Okasaki [48]: Okasaki's deque splits the elements it contains into two lists, a "front" and a "back" list. Each time a datum is inserted at the beginning of the deque, that datum is placed at the beginning of the front list in constant time; and each time a datum is inserted at the end deque, that datum is placed at the *front* (!!) of the back list in constant time. This means that the back list is going to contain elements in reverse order.



(a) An instance of Okasaki's deque data structure, containing the values $\langle 1, 2, 3, 4, 5, 6, 7 \rangle$. The values in the `back` part of the data structures are in reverse order.



(b) A version of the data structure from the previous subfigure where the first four values have been removed. Before we can remove from this list's beginning, we need to re-balance.

Figure 2.12. Okasaki's data structure improves on the simple linked list for appending, but removing from its front is not worst-case constant time, only amortized constant-time now.

For removing values from the front and back, we similarly access the front of the respective list. Figure 2.12a depicts an instance of this data structure that contains the values $\langle 1, 2, 3, 4, 5, 6, 7 \rangle$, where the values $1 - -4$ happen to be in the front part of the list in standard order, and the values $5 - -7$ happen to be in the back part of the list in reverse order. We could achieve this configuration by prepending the values $4, 3, 2, 1$ to an empty deque in this order, then appending the values $5, 6, 7$ to the resulting version of the deque. Now consider a later version of the data structure, after the values 1, 2, 3, and 4 have been removed from it. This is depicted in Figure 2.12b: the front part of the deque is now empty. If we now want to remove yet another value from its front, we find an empty list that we can not get our value from. The value we are looking for, in the case of an empty front, is the last value of the back part. The case where we want to remove from an empty part of the deque is handled by splitting the other part (the back part in the example) into two roughly-equal length parts and further using them (in the respectively correct order) as the new front and back parts. This step has $\mathcal{O}(N)$ complexity, and represents the worst-case cost of the removal-from-front operation. Removal from the back mirrors this operation and comes at the same asymptotic cost.

Functional data structures, therefore, can lead to worse asymptotic performance. This is *not to say that it always will lead to higher asymptotic complexity*, even data structure designers that have mutability available as a tool are well advised to look at functional data structures as an implementation technique. For example, it is possible to revert Okasaki's deque in constant time $\mathcal{O}(1)$ by simply switching the front and the back of the list, compared to $\mathcal{O}(N)$ needed to naively revert the list representation, functional data structures can be used by several threads without synchronisation, etc.

To understand better *why* we can not use the `last` reference to update the data structure, we will explain structural sharing, a technique that most functional data structures use.

Structural Sharing

We have explained that "updating" an object in a purely functional data structure does not mutate the object, but produces a new version – a slightly modified copy – instead.

For performance reasons, it is crucial to understand that not the whole data structure needs to be copied, just a part of it that needs to change in the new version. The newly copied part can share memory with the old version—this is called *structural sharing*.

Structural sharing, however, is affected by aliasing itself—the more aliasing there is in a data structure, the less structural sharing can be used.

First, to give an example of structural sharing, consider the immutable linked list data structure in Figure 2.13: version n of the data structure contains the values $\langle 1, 7, 2, 9, 13, 4, 8\rangle$. Version $n+1$, however, has been updated to negate the sign of the 4th element: $\langle 1,\ 7,\ 2,\ -9,\ 13,\ 4,\ 8\rangle$. To make this change from version n to version $n+1$ without changing version n, it is not sufficient to only copy the marked node †, because the node reaching it now needs to be updated to refer to the copy; but updating this node triggers copying it as well, meaning its predecessor also needs to be updated, and so on. The copying process propagates back all the way to the root object of the data structure. The "tail" (all nodes after the marked nodes) can be simply shared by versions n and $n+1$. Crucially, the tail of the list is *not* copied, versions n and $n+1$ share part of their structure.

Structural sharing works for more data structures than just lists: consider the binary search tree in Figure 2.14. After inserting the value 13 into the tree, a large part (the shaded part) of the old tree can be reused by the new version.

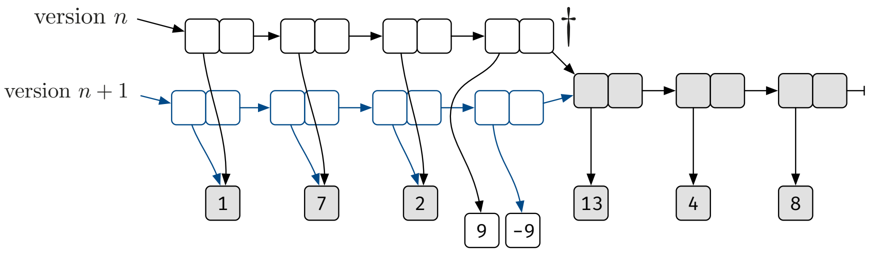


Figure 2.13. Structural sharing in a linked list. Version n contains the values $\langle 1, 7, 2, 9, 13, 4, 8 \rangle$, version $n+1$ has inserted the value -9 instead of 9. Both versions can share a substantial part of their structure (shaded grey).

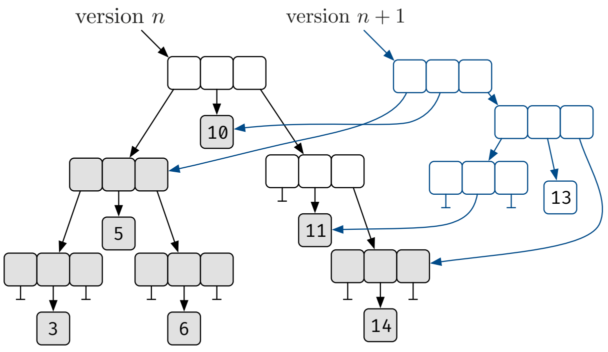


Figure 2.14. Structural sharing in a binary search tree. Version n contains the values $\langle 3, 5, 6, 10, 11, 14 \rangle$, version $n+1$ additionally contains the value 13. Both versions can share a substantial part of their structure (shaded grey).

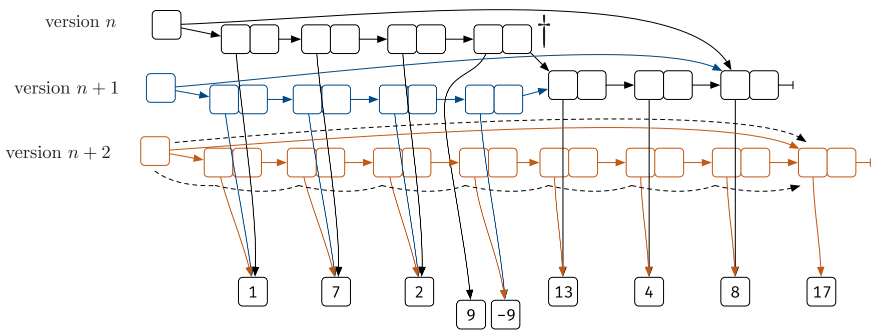


Figure 2.15. When we modify/replace the `last` node in version $n+2$ to refer to the newly inserted node containing the value 17, we need to modify/replace version $n+1$'s `last` node to refer to the new node. From there, we have to propagate the updates/replacements back to all of the N list nodes. This means that the `last` field does not improve the complexity of `append` from $\mathcal{O}(N)$ to $\mathcal{O}(1)$ like it would have in a mutable data structure.

Aliasing, Cycles, and Structural Sharing

When using structural sharing to limit the overhead of copying, aliasing is highly relevant. Aliasing can make structural sharing ineffective, in the worst case, the whole data structure has to be copied for any new version that is being produced.

To give an example of the cost of aliasing in immutable data, let's extend our list. The list data structure we have shown is simplistic—it does not have a constant time `append(List, int)` operation that inserts a given integer at the end of a given list. In an imperative language, we would improve the complexity of `append` by adding a `last` pointer from the list head to the last node, as we have showed before. If we would do the same for an functional data structure, it would not fix our problem of asymptotic append complexity: from the viewpoint of structural sharing, the crucial change is that the data structure now uses aliasing (the last node is reachable both via iteration through the list, as well as by following that new field). This affects what data we need to copy. In particular, we need to copy not only the objects on the path we took to reach the element which we modify, but also the objects on *other paths between the version root and the modified object*. In the case of appending at the end, we modify an aliased object and therefore have to copy more data. In Figure 2.15, we accordingly copy the whole spine of the data structure to ensure that from the version $n+2$ we can find the change both by dereferencing the `last` field as well as by iteration through the spine. The immutable version of the data structure will not see the same asymptotic improvement—in fact, its append operation still has the same complexity $\mathcal{O}(N)$ as without the `last` reference.

Cyclic data structures are rare in pure functional programming because to close a cycle, one generally needs to modify a previously existing object, but there are ways around that (*E.g.,* lazy evaluation permits a technique called "tying the knot" that can construct cyclic structures, or work by [26] lets a programmer `freeze` a mutable cyclic data structure to make it object immutable).

A reason why they are uncommonly used is that they exacerbate the problem that we get when using aliasing even further. Consider the tree in Figure 2.16. It is the same tree data structure as in Figure 2.14, but with added parent pointers that refer from each node to its parent. Structural sharing in this data structure does no longer work at all: for every new version, we must copy the whole spine of the tree. If we would not do that, it would always be possible to find a path from the new version root that lets us read back outdated data. Figure 2.16a shows such a path: starting at version root $n+1$, we read the path `root.right.right.parent.value` that gives us the value `11`. But reading the path `root.right.value` gives us `13`. The invariant that for any node `n`, the equality `n.right.parent.value = n.value` holds is broken.

Figure 2.16b shows the correct implementation that copies the whole spine which is prohibitively expensive at an $\mathcal{O}(N)$ cost to insert something into a sorted binary tree.

To conclude, we have shown that functional data structures can be an effective solutions for aliasing problems. However, by removing mutability from the design space of data structures, they can incur a performance cost. Additionally, we showed how – perhaps ironically – the implementation of functional data structures themselves must treat aliasing with great care in order to not sacrifice performance.

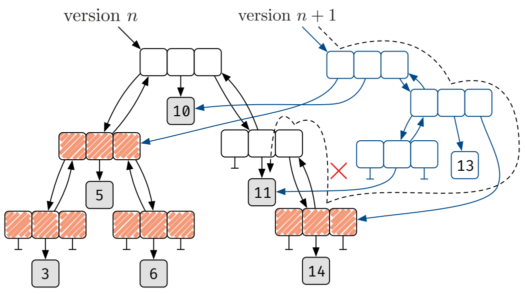
Automatically Making Data Structures Immutable

Functional data structures solve an important problem: we can have side-effect free sharing without giving up part of the interface of our data structure. To give an example, one practical advantage over read-only references that simply make the mutable part of data inaccessible is that we do not have to rewrite our code when we start to use multiple threads (which otherwise would require *e.g.,* locking or not using the mutable part of our data's interface).

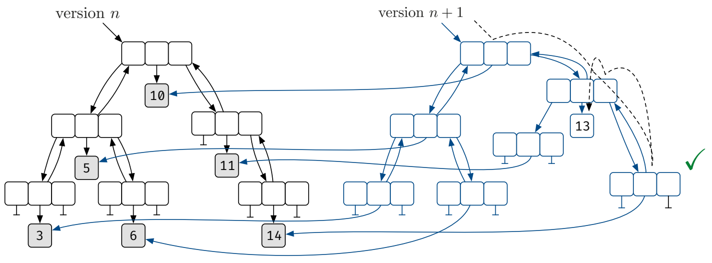
But using mutable data structures in situations where there is no aliasing is permissible even in pure functional programming (Section 2.1.3), and is useful to avoid the performance overhead that functional data structures can incur.

This section covers a set of techniques that produce data structures that can be configured[16] to be either mutable or immutable, giving us the attractive ability to implement a data structure once, but instantiate it mutably or immutably, whichever the situation requires. Then, rather than implementing mutable and immutable versions of data structures and picking the appropriate one for each use case, it would be useful to implement the data structure once, and be able to obtain a mutable and an immutable version from the same source.

[16]Whether this is done during compile- or run-time is not relevant for our purposes.



(a) The binary search tree with parent pointers that copies as much data as the one without parent pointers is broken: in version $n+1$, accessing the path `right.right.parent.value` yields the result `11`, when we expected `13`.



(b) In order to solve this problem, we have to copy the *whole spine* of the data structure, at a $\mathcal{O}(N)$ cost.

Figure 2.16. Structural sharing becomes ineffective with aliasing.

This means we can switch between mutability and permanent, deep object immutability in our taxonomy of mutability restrictions[17].

As a side-product, these techniques address the problems with structural sharing with aliasing that we have highlighted in the previous section on structural sharing.

Figure 2.17a shows a correct functional doubly linked list. Since the data structure is cyclic, the whole spine lies on paths (highlighted by the dashed path) from the version root to the modified object—this means, like in the previous tree example that used parent references, that the whole spine must be copied. Therefore, producing any new version of an immutable double linked list has $\mathcal{O}(N)$ complexity.

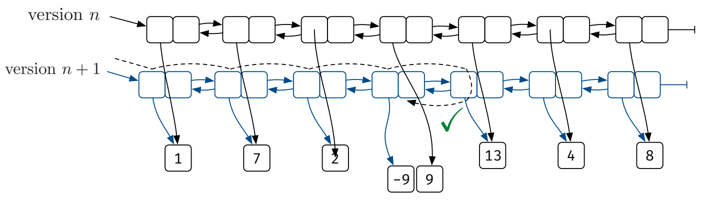
A solution that copies less data is depicted in Figure 2.17b: the so-called "fat node method" by Driscoll et al. [22] handles this case without copying an excessive amount of memory. This method generates a so-called "persistent" data structure[18] that returns new versions on every update. The basic idea of the fat node method is to, instead of overwriting a field that is being assigned in an update, *record the new value, but also the old value* that the field held. In order to achieve that, each object in the fat-node instantiation of a data structure contains the same fields as in the mutable instantiation, but fields are able to store an unbounded number of values–each value they ever had, and each value is tagged with a "version stamp" that specifies the version at which that particular value was assigned. For each update operation, in the fat-node instantiation, we generate a new version ID, and for each field assignment during that operation, we add that value, tagged with the operation's version ID to the target field. If the field already has a value with that same ID, we overwrite it. For each read, we read the "newest" value: the value with the highest ID smaller or equal than the version ID. Figure 2.17b shows a linked list that is implemented using the fat node method, and – for comparison – Figure 2.17a shows the corresponding functional data structure using structural sharing that needs to copy the whole spine for each update due to aliasing.

The fat node method has the advantage that it only needs extra space in the object that is actually changed in the new version, but does not need to copy all paths that lead to that object. But it has two disadvantages:

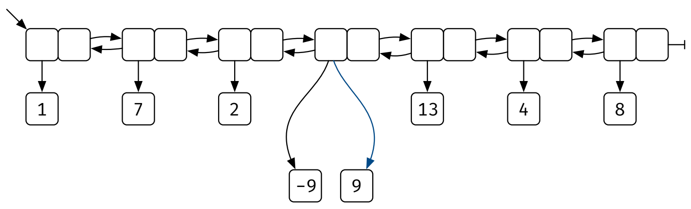
1. Reading a field value has an asymptotic overhead because it has to select newest value for the version being read from from a collection of field values.
2. We can not update any version of the data structure other than the newest one. This comes from the fact that versions (as presented) are linearly ordered; this can not model a *version tree* that we would expect to get

[17]Assuming we apply the same techniques to the data contained in our data structures, if we would leave the data mutable, the immutability would be classified as shallow instead.

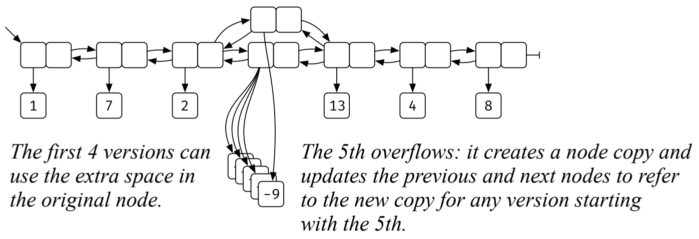
[18]Persistence in data structures is a generalisation of immutability; in short, a persistent data structure *may* use mutation internally, but it still has to produce versions that protect from interference.



(a) To correctly update the doubly linked list, we must copy the whole list spine, not just the part leading up to it. This means that even a modification at the head of the list, which would be $\mathcal{O}(1)$ for a mutable doubly linked list, would cost $\mathcal{O}(N)$ for the immutable version.



(b) A doubly linked list using fat nodes can avoid copying the whole structure. The node we have modified has now not one, but two element references that are each tagged with their respective version ID (IDs not drawn). Inserting at the end of this list is $\mathcal{O}(1)$, but accessing the end is $\mathcal{O}(\lg m)$ (m is the size of the number of versions of this node), as each field is implemented as a collection containing all references the field ever stored.


1
7
2
13
4
8
-9
The first 4 versions can use the extra space in the original node.
The 5th overflows: it creates a node copy and updates the previous and next nodes to refer to the new copy for any version starting with the 5th.

(c) A doubly linked list using node copying may need to copy data on the path from the root, but only when a node overflows. Here, one node has overflowed, and the nodes referring to it (previous and next node of the list) have added references to the new version of the node. This requires all nodes to maintain references to their predecessors (not drawn, but this is done automatically; they are parallel to the forward- and back-references of the list structure in this example).

Figure 2.17. When aliasing occurs in a data structure, fat nodes can be cheaper than traditional immutable data structures.

when we mutate older versions of data structures, which are only partially ordered.

In order to solve the first of these problems, they design a more sophisticated technique to represent nodes altogether, the “node-copying” technique.

Instead of storing the whole history of a field in the original object, we only make space in the object for a fixed number of version ID/value pairs (we will arbitrarily choose 4 for our examples). When we assign a field value for the 5th time, an object overflows. In this case, we create a new object of the same layout that contains for each field the newest value in the now-full version and then space for 3 more values for each of its fields. Since we have created a new object for that newest version, we must be able to find all objects in the data structure that referred to the old version (the “predecessors”), and update their reference to our new version with available space at the current version ID. In order to be able to find these objects, each version needs to maintain a list of inverse references that contains references to all versions of objects that refer to itself. Updating the predecessors to refer to the new version of our node may cause the predecessors to be full, requiring the copying to propagate backwards through the data structure. This looks much like in a functional data structures, but the difference is that a back-propagated copy only happens when a node overflows, which is rare. Figure 2.17c shows our list after a node has overflowed due to it’s element field being assigned 5 times. Note how the new version of that node is reachable from the predecessor in the list, as well as the successor. The authors prove how operations on objects in these data structures, *provided each node in the mutable instantiation has a bounded in-degree of references*, have an amortized bound of $\mathcal{O}(1)$ on the nodes copied and time per update.

The second problem can be solved using using a significantly more complex technique, called “node splitting”, which is derived from the node copying methods we just explained, it retains the same asymptotic complexity as node copying.

In Chapter 4, Paper I, and Paper IV, we will show how two of our contributions, relate to this body of work:

1. Disjointness domains in Paper I relate by giving programmers a tool to statically limit the number of references that can refer to an object.
2. C♭ in Paper IV relates by being able to declaratively configure objects to be immutable; the technique C♭ uses is different (it can also produce many other configurations for performance reasons). This different technique is made possible by C♭ not only limiting the number of aliases of an object to some fixed number, but to only one unique reference.

2.3 Summary

We have shown means of alias control as they are used in practice today, as well as the research community has proposed them. We have categorised those means into a small set of differences, both in this chapter, as well as in Section 1.4 and Section 1.5. We think that even though these means vary widely on first sight, many can still be understood as ultimately being founded on a comparably small set of seminal concepts, and we have attempted to make those concepts explicit by placing these means of alias control in our taxonomies of restrictions of aliasing and mutability.

It is easy to think of immutability as a technique that replaces techniques like encapsulation and uniqueness (or the other way around). But the chapter demonstrated in many places how systems that constrain the flow of references can benefit from immutability (for example, islands permit deeply immutable objects to be aliased freely; ownership types are extended with effects systems, etc), as well as how immutability can profit from a lack of aliasing (functional programming languages that use linear types for performance; aliasing makes structural sharing difficult; and automatic conversions of mutable to immutable data structures require the number of references to an object to be bounded to achieve good asymptotic complexity). Therefore, we claim, that these approaches should be seen as best being used in concert, rather than as mutually exclusive choices. Another reason to support both kinds of technique is that this kind of knowledge can be very useful to expose to a compiler or language run-time for performance reasons. Apart from micro-optimisations, this knowledge can be (perhaps surprisingly) useful: one example are techniques that use aliasing and immutability in order to optimise concurrent garbage collection [17, 24].

In the next chapter, we will survey techniques for understanding how common alias control is in practical programs.

3. Mining for Alias Control

In Chapter 1, we showed that aliasing is hard to analyse precisely. Yet, in order to design mechanisms to mitigate the risks inherent in aliasing of mutable state, we should ask ourselves: have programmers found that "engineering discipline required to produce high quality software" (Section 1.3)? We will not be able to conclusively answer this question in full. But we will, in this section, present certain evidence that programmers are using less aliasing and mutation than their languages would permit them to. To this end, we survey work that looks for evidence of uniqueness, encapsulation, and immutability in its various forms in programs.

In Section 3.1, we survey *techniques* for finding such evidence. Many of these systems analyse several properties, covering versions of immutability as well as uniqueness and encapsulation.

In Section 3.2, we will show selected results obtained by using these techniques, that show, amongst other things, that much of aliasing comes from using a relatively small number of code idioms, and shared data is often immutable. These results are relevant input for programming language design: safety properties that are often used may be supported by new type systems or may be applied for optimisations.

3.1 Mining Techniques

The techniques to mining for uniqueness and encapsulation that we present here are drawn from the fields of *static analysis*, *type inference*, *dynamic analysis*, and *snapshot analysis*. These different approaches have different relative strengths and weaknesses, and no one approach is universally preferable. Static analysis has the advantage that a result is usually valid for all program inputs—after all, all code paths in the program have been subject to analysis. Dynamic analysis, on the other hand, has the advantage that results only include the code that is actually being executed (and also how often each instruction was executed), ignoring unreachable code and emphasising code that is often used. Snapshot analysis has the advantage that it can easily achieve a very complete picture of the running program, making it comparably straightforward to investigate the deep structure of the program's memory.

Some of the techniques that we present are sound – they will not classify an object as immutable, unless it really is, and some are not. But while soundness is sometimes a strict requirement, we should not ignore unsound results: for the kinds of information that we are looking for in this chapter, soundness is not strictly required.

The systems presented are made for a wide range of purposes: they range from summarising program snapshots for programmers to quickly understand a program they might have never seen before [43] to understanding and categorising the specific intentions programmer have when using aliasing [28] and to type inference of different versions of immutability. Due to this wide range of purposes, the selection of work is not complete – our intention is to give the reader of the variety of available techniques.

3.1.1 Snapshot- and Trace-Analysis

Query-Based Snapshot Analysis

The "Fox" tool by Potanin, Noble, and Biddle [51] analyses snapshots of programs to search for, amongst other properties, object uniqueness, and evidence of nested heap structures similar to the ones that ownership types enforce. They define ownership as graph-domination[1]. But that definition, they note, is potentially not yielding the result that one would expect: the authors note that the "average-depth" metric, that measures the average depth of all objects in the ownership tree varies greatly between applications. When investigating this circumstance, they found that their definition had classified many of linked data structures as deep ownership hierarchies; to give an example, just looking at graph domination, a singly linked list would be seen as having the ownership structure in Figure 3.1. The programmer's mental view of ownership in this case, though, is most likely that the head of the list owns all the nodes, which are at a single level.

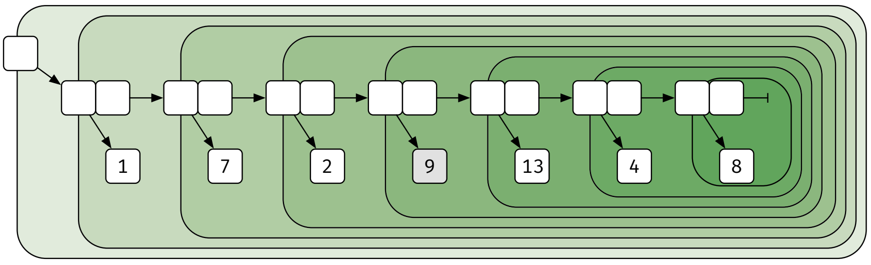


Figure 3.1. Using the dominator relation to mine for ownership gives us overly deeply nested results. The programmer's mental model of the data structure was, most likely, to only draw the outermost ownership context.

[1]A node n in a graph dominates another node n' if all paths to n' pass through n; this is similar to our definition of encapsulation (Definition 4, p. 30).

They mitigate this problem by counting successive chains of objects of the same type to count as chains of length 1, thereby putting all nodes of the list at the same depth.

Fox comes with a query language that lets users formulate their own queries; we will quote some of the results obtained in Section 3.2.

Query-Based Trace Analysis
This thesis contributes, in Paper II and Paper III, Spencer. Spencer is a tool that instruments running Java programs to emit comprehensive traces: these traces contain, amongst other information, details on ever variable and field-read, or -update. Spencer hosts these traces in a data base. Spencer lets users explore the behaviour of programs, in order to understand a wide range of safety properties discussed in this thesis. Spencer is a web service that users can interact with that hosts a selection of large data sets of dynamic program traces. A user of Spencer can query that data set by formulating query expressions using a small domain specific language that is accessible via a web browser (for interactively exploring a data set), or a web API (for building further analysis on top of the of the capabilities of the domain specific language).

Chapter 4, Paper II, and Paper III treat this contribution in more detail.

Summaries of Snapshots
Work by Mitchell [43] attempts to *summarise large snapshots* of program heaps in order to make the program more easy to understand by developers. To summarise means to simplify the program's memory graph by clustering connected objects into representative nodes. The resulting graph ideally hides irrelevant detail, but preserves the essential, birds-eye view of a program's memory. A common question in software development is which part of the program is responsible for high memory usage in order to understand or optimise an application. Unfortunately, it is rarely sufficient to look at the class of objects responsible for most of the used memory: *e.g.,*, in a Java program, much of the memory is usually used by primitive arrays [43]. Many of those primitive arrays are arrays of characters contained in strings, which are commonly used by a lot of code everywhere in the system under analysis. In general, the problem is that some basic types are used in many places, and finding a program feature that is responsible for high memory use is hard. In their work, an algorithm takes that snapshot and simplifies it by applying a series of *graph edits* (a graph edit maps many objects to one "representative object" and changing edges to originate/refer to the respective representative object). These graph edits recognise a variety of common structures that appear in programs and picks, for a group of objects, a representative object that it keeps in the graph instead of the group. For example: objects that are dominated by another are removed from the graph, and the dominator is kept as their representative. The algorithm produces overviews of program memory that contain only a small

set of remaining nodes (much less than 100 in their examples of real-world programs) with computation times that are in the seconds to minutes range.

Static Analysis and Type Inference

Hackett and Aiken [28] build a static analysis of aliasing which they use to analyse more than one million lines of C code. In addition to implementing a scalable static analysis, their contribution is manually classify all occurrences of aliasing, finding that "just nine idioms of aliasing account for nearly all aliasing in [the] study". Such a finding is important: it suggests that language constructs can be built comparably easily that can express aliasing at a higher level of abstraction than simply using references for all applications of aliasing. We will describe their results in closer detail in Section 3.2.

Work by Quinonez, Tschantz, and Ernst [53] introduces the *Javarifier* tool that statically analyses Java source code and annotates it to use the reference immutability of the previously existing Javari Java dialect that we have presented in Section 2.2. The system infers type annotations by emitting constraints; by default a variable is read-only. Type constraints are used to record deviations from this default: the type constraint `x` states that variable `x` is mutable. This constraint would be generated from analysing a statement like `x.f = y`. There are also guarded constraints that are generated by analysing dereferences of fields and assignments. For instance, the statement `x = y` would generate the constraint `x->y`, meaning that `x` being a mutable variable implies that `y` must also be mutable. Similarly, a method call `y.m(p..)` generates (amongst other constraints) a guarded constraint that states that if the `this` reference inside the `m` method is mutable, so must be `y`. Solving these constraints works by essentially using unguarded constraints `x` to trigger guarded constraints `x->y` and adding `y` to the constraint pool, executing this process until no further guarded constraints can be matched.

The *ReIm type system and ReImInfer tool* by Huang et al. [36], similarly to Javarifier, infers reference immutability type annotations in an extended version of Java. Work by Milanova and Dong [42] extends ReIm by Huang et al. [36] by elegantly constructing support for deep object immutability on top of support for reference immmutability, as we have mentioned in Section 2.2.1: the core idea is to split an object's life time into two phases, one phase where the object is initialised and one phase where it is used without further mutation. If the system can show that at the end of the initialisation phase, the following properties hold, it can soundly conclude that the object itself is object immutable:

– There is only one canonical reference to the object.
– The object only contains references to other immutable objects.
– The canonical reference is, from here on, *reference immutable*.

Work by Haller and Axelsson [29] contributes a Scala compiler plugin that conducts a static analysis of Scala programs in order to empirically study how common (shallow and deep class-) immutability in Scala programs is. They also categorise types that are *not immutable* by reasons for why they are.

3.2 How Common Uniqueness, Encapsulation, and Immutability are in Practice

The kind of results that the systems in Section 3.1 produce are extremely varied, and even the *purposes they were built for are.* This sometimes makes it difficult to compare the results obtained from these works, especially when mining for encapsulation.

It is still possible to summarise the results that they obtain that are relevant to this thesis. We will summarise the most relevant results, preferring to give numbers for the strongest properties that the research investigates.

3.2.1 Immutability

Immutability has been investigated by a comparably large number of proposals; Table 3.1 contains an overview of the systems whose results we mention here and, what kind of results those system deliver, and what technique they use.

Haller and Axelsson [29] report that of the Scala programs they analyse, between 45% and 61.3% of the types they analyse are deeply permanently class immutable or "conditionally immutable". A conditionally immutable type is one that is conventionally deeply immutable but has *e.g.,* a type parameter that may be bound to type with unknown (im)mutability.

Quinonez, Tschantz, and Ernst [53] find that between 35%–59% of all reference type variables and fields can be annotated as deeply read-only.

Spencer in Paper III shows that on average, 47.3% of all objects in Java programs are permanently deeply object immutable, and 21% of the classes that produce at least 10 instances produce *only* immutable objects (these types might have deep class immutability). Spencer is an unsound dynamic analy-

Table 3.1. *The analysis tools referred in Section 3.2.1 and the granularities they analyse. Legend:* d) *analyses deep immutability;* s) *analyses shallow immutability;* $d+s$) *analyses both, deep and shallow immutability.*

Tool ↓ Granularity →	Ref.-	Obj.-	Class Imm.	Technique
Haller and Axelsson [29]			$\checkmark^{d+s}$	Static analysis.
Hackett and Aiken [28]	$\checkmark^{s}$	$\checkmark^{s}$	$\checkmark^{s}$	Static analysis.
Quinonez, Tschantz, and Ernst [53]	$\checkmark^{d}$			Type inference.
Milanova and Dong [42]	$\checkmark^{d}$	$\checkmark^{d}$		Type inference.
Paper II, Paper III		$\checkmark^{d+s}$	$\checkmark^{d+s}$	Trace analysis.

sis, it might judge that an object is immutable only because it was not mutated for a particular execution, even though it would have been with other executions; we can contrast that by a sound analysis by Milanova and Dong [42] that finds that 36.5% of all objects are permanently deeply object immutable or temporarily deeply immutable (their analysis also finds late initialisations of deeply immutable objects).

Summary
Even though the presented techniques vary a lot, the amount of immutability in Java programs is high; Scala programs have even higher numbers. This might be explained by Scala being strongly influenced by functional programming.

Looking at the data reported by all these systems, it seems to us that support for deep immutability in programming languages is an addition that lets programmers more reliably implement the forms of immutability that they already heavily use in their programs.

3.2.2 Uniqueness and Encapsulation

Fox by Potanin, Noble, and Biddle [51] find that, on average, 86.4% of objects were unaliased, but Spencer Paper III finds "only" 45.5% unaliased objects. The difference might be because Spencer is analysing comprehensive program traces, while Fox looks at snapshots of program memory and might therefore miss aliasing that happens before or after the snapshot is taken. Additionally, Spencer finds that 46% of all fields and 27% of all classes with 10 or more instances only ever contain references to/produce instances that are uniquely referenced.

Fox measures the average depth of objects in the ownership hierarchy. This depth is 5.3 accross the programs they analyse, avoiding the problem shown in Figure 3.1. This suggests that the hierarchy of ownership when looking at the program memory is quite deep (for our expectations).

Finally, Mitchell [43] does not investigate encapsulation directly, but abstracts the shape of the heap using sophisticated patterns they define to simplify the heap into successively smaller and smaller representations. Their algorithm does simplify the heap where it finds dominator nodes, but it also does other simplifications; this means that we can not easily compare their results with systems that only look at dominators. Nonetheless, the fact that they are able to summarise a graph that contains between $\approx$ 29.000.000 objects by replacing it by 14 nodes (and similar numbers for the other programs they analyse) that represent a high-level view of the graph is strongly suggestive of a structure being present in the programs we write, and that we can understand and talk about the shape of these graphs. How to convert these insights into programming language abstractions (and whether that is a worthy goal) is, however, neither in the scope of their work, nor obvious.

Hackett and Aiken [28] conclude that the vast majority of the aliasing found in the C programs they analyse is due to aliasing invariants that apply to all instances of a type and global variables–*e.g.*,, a `previous` pointer of a doubly-linked list node should always maintain the invariant `node->prev == NULL || (node->prev->next == node)`. Outside of user-defined data structures and aliasing of global values, aliasing is rare. They find at only 3.5% of functions make use of aliasing that exceeds aliasing implied by such global and type invariants. This is a very important insight for us, as Paper IV is a way to implement data structures without relying on aliasing, but executing them with a data storage that may use aliasing transparently. This way, we can have much of the performance benefit of aliasing, but ignore the aliasing in our understanding of the program, as the high level language is free of aliasing.

Summary

Uniqueness is very common in Java programs: the numbers we have presented vary between analysis techniques, but are consistently high. Much of the aliasing in C programs happens in or around data structures, "regular" data is often un-aliased.

To mine for encapsulation is less common, and we can not easily compare results from system to system, as the form of the results varies.

4. Conclusion

We have now explained what aliasing is, where it comes from historically, and what dangers come with aliasing of mutable state. We have shown how the programming languages community has tried and is trying to provide to solutions that tame aliasing. To be able to evaluate trade-offs, we have shown several techniques for how to analyse the use of aliasing in real world programs. The contributions of this thesis cover a wide range of topics, but they have the goal in common to advance the field towards a more unified view of alias control.

The contributions are threefold:

- Paper I: Disjointness Domains, a type system for alias control that forms a bridge between type systems based on uniqueness and type systems subdividing the heap.
- Paper II and Paper III: Spencer, a tool to analyse program traces that comes with a query language. Spencer is an attempt to aid researchers in the field to focus their efforts on alias control that has practical impact. We use Spencer, in Paper III to mine for safety properties in Java programs, *e.g.,* different versions of uniqueness, immutability.
- Paper IV: C♭, a domain-specific language for building data-structures that exploits the fact that aliasing in the spine of data-structures is comparably rare and that notably permits automatically deriving mutable *and* immutable implementations of data structures from the same data structure implementation.

These contributions are related to the work described in this introduction in several ways:

A Type-System for Alias Control

Paper I relates to the work that is presented in Section 2. Disjointness domains are a type abstraction that enables a programmer to express fine-grained aliasing invariants. Disjointness domains are able to express aliasing invariants similar to those found in type systems based on unique references, as well as aliasing invariants similar to those found in type systems coming from an object-oriented tradition (ownership types [12, 15, 16, 46], balloons [2, 55], islands [33]). We believe that disjointness domains are a step forward in the field, as they can express strong invariants even about aliased state; for instance, a doubly linked list can express that all of its back-references are unique amongst the back-references, and all of the forward-references are unique amongst the forward references, even though nodes are aliased. Knowledge like this could be applied for data-race-free parallelism. Strong disjointness domains are closely related to fractional permissions, a core difference is that fractional permissions distinguish between whether or not a reference is aliased for purposes of controlling mutation, but our work permits to locally reason about aliasing even for references that are aliased elsewhere: for instance, several references in the same strong disjointness domain may never be aliases, even though all of these references might have aliases in other disjointness domains. Shared domains, then, behave in ways that are reminiscent to encapsulation-based proposals like islands, ownership types, balloons: they permit aliasing amongst a number of fields that are in the same shared disjointness domain. The objects that hold fields in that domain can be seen as a unit of encapsulation.[1] Additionally, strong domains provide a means of statically limiting the number of references that may refer to an object; this, we have shown in Section 2.2.3, is crucial to automatically convert data structures to immutable versions with good asymptotic complexity. While we do not do this conversion, disjointness domains are still related as a stepping stone for such techniques.

A Tool for Dynamic Trace analysis and an Empirical Study of Immutability and Uniqueness
Paper II and Paper III relate to analysis of immutability and aliasing that was covered in Section 3. Our goal was to build a system that could make it easy for researchers in programming language design to get quick feedback on design trade-offs very early in the design process. To this end, we think that it is important that this tool is easy to install, use, and results are easy to share. Contrary to the related work covered, Spencer is designed to be used as an online service. This represents an interesting trade-off: Spencer's query language is designed to be able to take advantage of caching of sub-expressions—this makes it feasible for us to routinely use algorithms that would be far too expensive were they run every single time. Using this approach, we are also able to store a large amount of data in our data sets without requiring users to have access to powerful computers. While Spencer's domain-specific language is not flexible enough to define every conceivable query (which was not the design goal), we have found it useful in practise and especially Paper III documents how we can use the tool to find, amongst other properties, deep immutability and uniqueness in Java programs.

Completely banning aliasing, and having good performance anyway
Paper IV, finally, is the result of observations we learned from related work, as well as papers I–III: there is evidence that much of aliasing in data structures is

[1]To fully encode these systems encapsulation based systems, we would need to add existential disjointness domains that would permit expressing an object-local disjointness domain, much like the `rep` ownership domain. The system as presented in the paper supports only a statically bounded number of "`rep`" domains.

due to type-invariants (like every doubly-linked list node's predecessor's next-field refers to the node itself [28], Paper III) and we observed that the spine of a data structure can often be seen as a canonical tree with additional references added in for reasons of performance. Examples are:

- A doubly linked list can be seen as a degenerate tree with parent pointers; parent pointers are not necessary for correctness, but without them, operations like backwards iteration would be very slow, at $\mathcal{O}(N^2)$.
- A hash map can be seen as a degenerate tree of $(key, value)$ tuples were most elements are `null`.
- A 2D matrix can be seen as a list containing a list for each row (or for each column).

As we can look at all of these data structures as alias-free trees, C♭ is a design that lets a programmer implement the data structures in terms of their canonical tree representation, and simply choose a memory-layout strategy (by way of picking a so-called "storage back-end" implementation) that will make the tree representation execute similar to an optimised data structure implementation. Aliasing, therefore, exists in program memory, but not in the semantics of the DSL. For the use cases we test in the paper, C♭ leads to code that is subjectively very simple, and has good performance overall (often slightly behind normal Java implementations and sometimes far ahead; we believe that with extra engineering effort, we could match normal Java implementations in most cases). Since there is some evidence that a significant part of aliasing found in programs is the aliasing in data structure spines, we think that approaches like C♭ that essentially optimise programs by declaratively deciding how aliasing should be used to alter a data structure's performance but not its semantics can be a means of getting rid of many of the uses of aliasing, and we think that approaches that essentially side-step the issue of aliasing are under-explored in research today.

C♭ is also related to the work in Section 2.2.3: using C♭ we can generate functional as well mutable data structures without losing part of their interface, and without using their code.

Final words

The rest of the thesis consists of Papers I–IV, which we have already placed in the wide context of this introduction, but of course not completely covered. After having read this introduction, a reader has gained relevant background for reading the papers. Our papers approach the problem of aliasing from multiple angles, and so did this introduction. We hope that the presentation of the resulting wide background will prove illuminating for reading the individual, much more narrowly focused, papers.

Bibliography

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. "L^3: A Linear Language with Locations". In: *Fundam. Inform.* 77.4 (2007), pp. 397–449.

[2] Paulo Sérgio Almeida. "Balloon Types: Controlling Sharing of State in Data Types". In: *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*. Ed. by Mehmet Aksit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer, 1997, pp. 32–59. doi: `10.1007/BFb0053373`. url: `https://doi.org/10.1007/BFb0053373`.

[3] Alexis Beingessner. *The Rustonomicon: §3.1: References*. `https://doc.rust-lang.org/nomicon/references.html`. [Online; accessed 2018-08-02].

[4] John Boyland. "Alias Burying: Unique Variables without Destructive Reads". In: *Software—Practice and Experience* 31.6 (May 2001), pp. 533–553.

[5] John Boyland. "Checking Interference with Fractional Permissions". In: *International Symposium on Static Analysis (SAS)*. 2003, pp. 55–72. isbn: 978-3-540-40325-8. doi: `10.1007/3-540-44898-5_4`.

[6] John Boyland. "Why we should not add readonly to Java (yet)". In: *ECOOP 2005 Workshop on Formal Techniques for Java-like Programs*. Ed. by Francesco Logozzo and Jan Vitek. July 2005.

[7] Stephan Brandauer, Elias Castegren, and Tobias Wrigstad. "C♭: A New Modular Approach to Implementing Efficient and Tunable Collections". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018, Boston, MA, USA, November 7-8, 2018*. 2018, pp. 57–71. doi: `10.1145/3276954.3276956`. url: `https://doi.org/10.1145/3276954.3276956`.

[8] Stephan Brandauer, Dave Clarke, and Tobias Wrigstad. "Disjointness Domains for Fine-Grained Aliasing". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 2015, pp. 898–916. doi: `10.1145/2814270.2814280`. url: `https://doi.org/10.1145/2814270.2814280`.

[9] Stephan Brandauer and Tobias Wrigstad. “Mining for Safety using Interactive Trace Analysis”. In: *Fifteenth International Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)* 15 (2017), p. 14.

[10] Stephan Brandauer and Tobias Wrigstad. “Spencer: Interactive Heap Analysis for the Masses”. In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 113–123. doi: `10.1109/MSR.2017.35`. url: `https://doi.org/10.1109/MSR.2017.35`.

[11] Elias Castegren. “Capability-Based Type Systems for Concurrency Control”. PhD thesis. Uppsala University, Sweden, 2018. url: `http://nbn-resolving.de/urn:nbn:se:uu:diva-336021`.

[12] Dave Clarke and Sophia Drossopoulou. “Ownership, Encapsulation and the Disjointness of Type and Effect”. In: *ACM SIGPLAN Notices* 37.11 (2002), p. 292. issn: 03621340. doi: `10.1145/583854.582447`.

[13] Dave Clarke and Tobias Wrigstad. “External Uniqueness Is Unique Enough”. In: *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*. 2003, pp. 176–200. doi: `10.1007/978-3-540-45070-2_9`. url: `https://doi.org/10.1007/978-3-540-45070-2_9`.

[14] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. “Minimal Ownership for Active Objects”. In: *Asian Symposium on Programming Languages and Systems*. Springer, Berlin, Heidelberg. 2008, pp. 139–154.

[15] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. “Ownership Types: A Survey”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. 2013, pp. 15–58. doi: `10.1007/978-3-642-36946-9_3`. url: `https://doi.org/10.1007/978-3-642-36946-9_3`.

[16] David G. Clarke, John Potter, and James Noble. “Ownership Types for Flexible Alias Protection”. In: *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA ’98), Vancouver, British Columbia, Canada, October 18-22, 1998.* Ed. by Bjørn N. Freeman-Benson and Craig Chambers. ACM, 1998, pp. 48–64. doi: `10.1145/286936.286947`. url: `http://doi.acm.org/10.1145/286936.286947`.

[17] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. “Orca: GC and Type System Co-Design for Actor Languages”. In: *Proceedings of the ACM on Programming Languages* 1 (2017), p. 72.

[18] Rust community. *Rust Documentation of Module: std::cell*. [Online; accessed 2018-10-11]. 2018.

[19] Robert Deline and Manuel Fähndrich. "Enforcing High-Level Protocols in Low-Level Software". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York: ACM Press, 2001.

[20] Werner Dietl and Peter Müller. "Universes: Lightweight Ownership for JML." In: *Journal of Object Technology* 4.8 (), pp. 5–32.

[21] Edsger W. Dijkstra. "Structured Programming". In: *Software Engineering Techniques*. Ed. by B. Randell and J. N. Buxton. Proceedings of the NATO Software Engineering Conference. NATO Scientific Affairs Division, 1969, pp. 74–88.

[22] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. "Making Data Structures Persistent". In: *J. Comput. Syst. Sci.* 38.1 (1989), pp. 86–124. doi: `10.1016/0022-0000(89)90034-2`. url: `https://doi.org/10.1016/0022-0000(89)90034-2`.

[23] Manuel Fähndrich and Robert DeLine. "Adoption and Focus". In: *PLDI* 37.5 (2002), pp. 13–24. issn: 03621340. doi: `10.1145/543552.512532`.

[24] Juliana Franco, Sylvan Clebsch, Sophia Drossopoulou, Jan Vitek, and Tobias Wrigstad. "Correctness of a Concurrent Object Collector for Actor Languages". In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 885–911. isbn: 978-3-319-89884-1.

[25] Google, Inc. *Google C++ Style Guide*. `https://google.github.io/styleguide/cppguide.html`. [Online; accessed 2018-11-26]. 2018.

[26] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. "Uniqueness and Reference Immutability for Safe Parallelism". In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. Ed. by Gary T. Leavens and Matthew B. Dwyer. ACM, 2012, pp. 21–40. doi: `10.1145/2384616.2384619`. url: `http://doi.acm.org/10.1145/2384616.2384619`.

[27] Marc Gregoire. *Professional C++ (4th Edition)*. John Wiley & Sons, 2018. isbn: 978-1-119-42130-6.

[28] Brian Hackett and Alex Aiken. “How is Aliasing Used in Systems Software?” In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*. Ed. by Michal Young and Premkumar T. Devanbu. ACM, 2006, pp. 69–80. doi: `10.1145/1181775.1181785`. url: `http://doi.acm.org/10.1145/1181775.1181785`.

[29] Philipp Haller and Ludvig Axelsson. “Quantifying and Explaining Immutability in Scala”. In: *Proceedings Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017*. 2017, pp. 21–27. doi: `10.4204/EPTCS.246.5`. url: `https://doi.org/10.4204/EPTCS.246.5`.

[30] Douglas E. Harms and Bruce W. Weide. “Copying and Swapping: Influences on the Design of Reusable Software Components”. In: *IEEE Trans. Software Eng.* 17.5 (1991), pp. 424–435. doi: `10.1109/32.90445`. url: `https://doi.org/10.1109/32.90445`.

[31] Herb Sutter. *You don’t know const and mutable*. `https://channel9.msdn.com/posts/C-and-Beyond-2012-Herb-Sutter-You-dont-know-blank-and-blank`. [Online; accessed 2018-07-31]. 2012.

[32] C. A. R. Hoare. *Hints on Programming Language Design*. Tech. rep. STAN//CS-TR-73-403. Stanford University, Dept. of Computer Science, 1973.

[33] John Hogg. “Islands: Aliasing Protection in Object-Oriented Languages”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’91), Sixth Annual Conference, Phoenix, Arizona, USA, October 6-11, 1991, Proceedings*. Ed. by Andreas Paepcke. ACM, 1991, pp. 271–285. doi: `10.1145/117954.117975`. url: `http://doi.acm.org/10.1145/117954.117975`.

[34] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. “The Geneva Convention on the Treatment of Object Aliasing”. In: *SIGPLAN OOPS Messenger* 3.2 (Apr. 1992), pp. 11–16. issn: 1055-6400. doi: `10.1145/130943.130947`. url: `http://doi.acm.org/10.1145/130943.130947`.

[35] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty Asynchronous Session Types”. In: *J. ACM* 63.1 (2016), 9:1–9:67. doi: `10.1145/2827695`. url: `https://doi.org/10.1145/2827695`.

[36] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. “Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity”. In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. 2012, pp. 879–896. doi: `10.1145/2384616.2384680`. url: `http://doi.acm.org/10.1145/2384616.2384680`.

[37] IEEE Comp. Soc. *2000 Computer Pioneer Award*. `computer.org/web/awards/pioneer-harold-lawson`. [Online; accessed 2018-05-29]. 2000.

[38] ISO. *ISO International Standard ISO/IEC 9899:201x: Programming Languages — C. [Committee Draft]*. Apr. 2011, p. 683.

[39] Jonathan Coe and Robert Mill. *A Proposal to Add a Const-Propagating Wrapper to the Standard Library*. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4388.html`. [Online; accessed 2018-07-31]. 2015.

[40] William Landi and Barbara G. Ryder. “Pointer-Induced Aliasing: A Problem Classification”. In: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*. 1991, pp. 93–103. doi: `10.1145/99583.99599`. url: `http://doi.acm.org/10.1145/99583.99599`.

[41] William Alexander Landi. “Interprocedural Aliasing in the Presence of Pointers”. PhD thesis. Rutgers University, 1992.

[42] Ana Milanova and Yao Dong. “Inference and Checking of Object Immutability”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. 2016, 6:1–6:12. doi: `10.1145/2972206.2972208`. url: `http://doi.acm.org/10.1145/2972206.2972208`.

[43] Nick Mitchell. “The Runtime Structure of Object Ownership”. In: *Proceedings of 20th European Conference on Object-Oriented Programming (ECOOP), Nantes, France*. Ed. by Dave Thomas. Vol. 4067. Lecture Notes in Computer Science. Springer, 2006, pp. 74–98. doi: `10.1007/11785477_5`. url: `https://doi.org/10.1007/11785477_5`.

[44] Peter Möuller, Arnd Poetzsch-Heffter, and Fernuniversitöat Hagen. *Universes: A Type System for Alias and Dependency Control*. Tech. rep. 2001.

[45] Peter Müller and Arnd Poetzsch-Heffter. “A type system for controlling representation exposure in Java”. In: *ECOOP Workshop on Formal Techniques for Java Programs. Technical Report*. Vol. 269. Citeseer. 2000.

[46] Alan Mycroft and Janina Voigt. “Notions of Aliasing and Ownership”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. 2013, pp. 59–83. doi: `10.1007/978-3-642-36946-9_4`. url: `https://doi.org/10.1007/978-3-642-36946-9_4`.

[47] James Noble, Jan Vitek, and John Potter. “Flexible Alias Protection”. In: *ECOOP'98*. Springer-Verlag, 1998, pp. 158–185.

[48] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. isbn: 978-0-521-66350-2.

[49] Johan Östlund. “Language Constructs for Safe Parallel Programming on Multi-Cores”. PhD thesis. Uppsala University, Computing Science, 2016, p. 105. isbn: 978-91-554-9413-1.

[50] Emil Leon Post. “A variant of a recursively unsolvable problem”. In: *Bulletin of the American Mathematical Society* 52 (1946), pp. 264–269. doi: `10.1090/s0002-9904-1946-08555-9`.

[51] Alex Potanin, James Noble, and Robert Biddle. “Checking Ownership and Confinement”. In: *Concurrency - Practice and Experience* (2004), pp. 671–687. doi: `10.1002/cpe.799`. url: `https://doi.org/10.1002/cpe.799`.

[52] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. “Generic Ownership for Generic Java”. In: *ACM SIGPLAN Notices*. Vol. 41. 10. ACM. 2006, pp. 311–324.

[53] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. “Inference of reference immutability”. In: *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*. Paphos, Cyprus, July 2008, pp. 616–641.

[54] G. Ramalingam. “The Undecidability of Aliasing”. In: *TOPLAS* 16.5 (1994), pp. 1467–1471. issn: 01640925. doi: `10.1145/186025.186041`.

[55] Marco Servetto, David J Pearce, Lindsay Groves, and Alex Potanin. “Balloon Types for Safe Parallelisation Over Arbitrary Object Graphs”. In: *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*. Citeseer. 2013, p. 107.

[56] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. “Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs”. In: *Graph Transformations in Computer Science, International Workshop, Dagstuhl Castle, Germany, January 1993, Proceedings*. Ed. by

Hans Jürgen Schneider and Hartmut Ehrig. Vol. 776. Lecture Notes in Computer Science. Springer, 1993, pp. 358–379. doi: `10.1007/3-540-57787-4_23`. url: `https://doi.org/10.1007/3-540-57787-4_23`.

[57] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Pearson Education, 2014.

[58] Matthew S. Tschantz and Michael D. Ernst. "Javari: adding reference immutability to Java". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. 2005, pp. 211–230. doi: `10.1145/1094811.1094828`. url: `http://doi.acm.org/10.1145/1094811.1094828`.

[59] Philip Wadler. "Linear Types Can Change the World!" In: *Programming Concepts and Methods*. North, 1990.

[60] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. "Object and Reference Immutability Using Java Generics". In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. 2007, pp. 75–84. doi: `10.1145/1287624.1287637`. url: `https://doi.org/10.1145/1287624.1287637`.

[61] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. "Ownership and Immutability in Generic Java". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 2010, pp. 598–617. doi: `10.1145/1869459.1869509`. url: `http://doi.acm.org/10.1145/1869459.1869509`.

Distribution: publications.uu.se
urn:nbn:se:uu:diva-366932

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018