UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1319

Language Constructs for Safe Parallel Programming on Multi-Cores

JOHAN ÖSTLUND

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

ISSN 1651-6214
ISBN 978-91-554-9413-1
urn:nbn:se:uu:diva-266795

Dissertation presented at Uppsala University to be publicly examined in 2446, ITC, Lägerhyddsvägen 2, hus 2, Uppsala, Monday, 18 January 2016 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: CR1 Ludovic Henrio (INRIA Sophia Antipolis).

Abstract

Östlund, J. 2016. Language Constructs for Safe Parallel Programming on Multi-Cores. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1319. 105 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9413-1.

The last decade has seen the transition from single-core processors to multi-cores and many-cores. This move has by and large shifted the responsibility from chip manufacturers to programmers to keep up with ever-increasing expectations on performance. In the single-core era, improvements in hardware capacity could immediately be leveraged by an application: faster machine - faster program. In the age of the multi-cores, this is no longer the case. Programs must be written in specific ways to utilize available parallel hardware resources.

Programming language support for concurrent and parallel programming is poor in most popular object-oriented programming languages. Shared memory, threads and locks is the most common concurrency model provided. Threads and locks are hard to understand, error-prone and inflexible; they break encapsulation - the very foundation of the object-oriented approach. This makes it hard to break large complex problems into smaller pieces which can be solved independently and composed to make a whole. Ubiquitous parallelism and object-orientation, seemingly, do not match.

Actors, or active objects, have been proposed as a concurrency model better fit for object-oriented programming than threads and locks. Asynchronous message passing between actors each with a logical thread of control preserves encapsulation as objects themselves decide when messages are executed. Unfortunately most implementations of active objects do not prevent sharing of mutable objects across actors. Sharing, whether on purpose or by accident, exposes objects to multiple threads of control, destroying object encapsulation.

In this thesis we show techniques for compiler-enforced isolation of active objects, while allowing sharing and zero-copy communication of mutable data in the cases where it is safe to do so. We also show how the same techniques that enforce isolation can be utilized internal to an active object to allow data race-free parallel message processing and data race-free structured parallel computations. This overcomes the coarse-grained nature of active object parallelism without compromising safety.

Keywords: Programming Languages, Type Systems, Ownership Types, Concurrency, Parallelism, Actors, Active Objects, Structured Parallelism, Data Race-Freedom, Immutability, Uniqueness

Johan Östlund, Department of Information Technology, Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-9413-1
urn:nbn:se:uu:diva-266795 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-266795)

till Olle och Hedda

Uppsala Progra
Multicore Arch
Research Cent

This work was carried out at the UPMARC Linnaeus Centre of Excellence.

www.upmarc.se

This work was also partially funded by the Swedish Research Council project Structured Aliasing and the EU project FP7-612985 Upscale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations.

Acknowledgments

This has been a long journey. A great many people have joined me on the way in one capacity or another (some more than one), collaborators, co-workers, and friends. I'd like to thank all of you.

A very special thanks goes out to my advisor, Tobias Wrigstad, who has been along for the entire trip, from start to end, with its twists and turns. Tobias has always been a great support, no matter what. Thanks for believing in me.

Dave Clarke for collaborations, but frankly, I appreciate our lunches the most. Thanks for talking about anything but work, most of the time. Oh, and thanks for Ownership Types, appreciate it.

I'd like to send a big thanks to Jan Vitek for taking me on as a PhD student at Purdue. I learned so much during my time there. I'm very happy I went, and I'm very happy I left. I had to. I know you understand. You have a part in this thesis becoming reality.

On a personal level I'd like to mention my mom and dad who have stood by me always, in rain or shine, thank you! My sisters with families, for always making me feel welcome. Special thanks to Anna for letting me stay with your family during my visits home, and for visiting me in the US.

Åsa, you have been so very supportive and patient with this work, especially during the sprint to the finish line. This thesis could not be if it weren't for you.

Finally, for not not caring the least bit about any of this rubbish, Olle and Hedda, you deserve the biggest thanks of all.

Uppsala, Nov 17, 2015

List of Papers

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

I *Ownership, Uniqueness, and Immutability* [108]

II *Minimal Ownership for Active Objects* [39]

III *Welterweight Java* [105]

IV *Multiple Aggregate Entry Points for Ownership Types* [107]

V *Refined Ownership:* Fine-Grained Controlled Internal Sharing [31]

Reprints were made with permission from the publishers.

Contents

List of Figures

1

Introduction

The last decade has seen the transition from single-core processors to multi-cores and many-cores. This move is prompted by the laws of physics—increasing clock speeds, using more transistors and ever thinner connections between transistors give an exponential increase in power consumption and heat dissipation. Around 2005, the point was reached where a signal would have trouble reaching the far end of the chip in one clock cycle. These reasons led chip manufacturers to switch direction and increase compute power by adding more computing elements (cores) to each chip, rather than increasing the performance of a single core; hence multi-core.

In the single-core era, moving a program to a faster machine would immediately make use of the increased performance. In the multi-core era, this is no longer the case since programs must be written in specific ways to be able to make use of available parallel computing elements. Thus the multi-core era has by and large shifted the responsibility from the hardware manufacturers to software developers to keep up with the never ending thirst for increased speed. In short: we have to write parallel programs.

Parallel programming has long been a task reserved for a select few experts writing specialized software in, for instance scientific computing, or for server applications, which are often intrinsically highly concurrent [123] (sometimes called "embarrassingly parallel".) With the advent of multi-core, parallel programming suddenly became reality for all programmers, expert or otherwise, and now in the client application realm, in which applications are typically not so intrinsically concurrent [123]. This stresses the fact that support for parallel programming in most mainstream programming languages is poor at best.

Although the trend is shifting, most popular programming languages still use explicit parallelism in the form of threads. The reason for this is likely that these languages were designed for sequential execution, not with parallelism as a main concern. Threads are easily added to a language as a library, but employing them puts the burden on the software developer, not the programming language designer.

Threads are hard to understand, error-prone and inflexible [80, 123]. Threads are difficult because they are cross-cutting and low-level. Threads move from object to object in a way that is impossible to analyze locally for most interesting programs. Thus complexity quickly increases with program size. Avoiding data races, deadlocks, atomicity violations and other parallelism and/or concurrency defects relies on testing, which is hard to get right in a multi-threaded environment because run-time behavior is dependent on the interleaving of threads, and the particular timing of events in a particular run, things which are typically outside the control of the program. Thus an error might show up only under certain circumstances, and worse, testing and debugging may affect programs in ways that suppress bugs because of logging or other kinds of added code that changes the timing, so-called "heisenbugs"[1].

Our work is carried out in an object-oriented setting. While there are other popular paradigms, some of which have seen an increased interest in recent years (some due to the increasing importance of parallel programming), object-oriented programming languages continue to be the de-facto standard in application development. The object-oriented paradigm, and in particular its implementation in different languages, presents many appealing features, but the most important one is encapsulation.

[1] `https://en.wikipedia.org/wiki/Heisenbug`

> *The fundamental concept of the object-oriented approach is that objects are encapsulations [...] (The notion of class inheritance, though crucial to object-oriented programming, is secondary to the idea of encapsulation).* [98]
>
> – Oscar Nierstrasz, recipient of the Dahl-Nygaard Award 2013

As stated in the quote above from Oscar Nierstrasz, encapsulation is a cornerstone of object-oriented programming. Encapsulation brings compositionality and local reasoning, but is often sacrificed by the addition of multiple threads for aforementioned reasons. Thus, object-oriented parallel and concurrent programming loses one of its cornerstone properties. Concurrency control is not a modular concern and hence cannot be implemented locally [123]. Therefore, *compositionality of objects*, and subsequently much of the power of the object-oriented approach is lost. *This motivates our quest for better ways to express and constrain concurrency and parallelism within the realm of object-orientation.*

Rather than patching existing programming language technology, which has evolved in a predominantly sequential setting, this thesis takes the position that a more radical shift is needed to help programmers transition into ubiquitous parallelism. There is a need for other abstractions, and other language constructs than threads and locks, abstractions and constructs which do not mandate parallelism but unlock the possibility of the run-time scheduler to make use of available parallelism at any given time. The key problem lies in finding the right balance between abstraction and performance – or what performance is possible to squeeze out of the abstraction.

In this thesis we present language constructs aimed to take popular object-oriented languages to the age of the multi-cores. We focus on front-end aspects of programming such as abstractions for concurrency and parallelism, how to express in a statically checkable way what computations may safely run in parallel, and how to fortify the object abstraction in the presence of multiple threads of control. We base our work on the actor/active object abstraction, and show how we may perform parallel computations inside the active object to better utilize available resources on multi-cores, all the while retaining the single thread of control abstraction.

1.1 Contributions

The contributions of this thesis are a set of programming language features to support data race-free parallel programming. We add isolation, safe sharing and internal parallelism to active objects. We also present extensions to Ownership Types–the underlying foundation of our work–to remedy some restrictions inherent therein. As a third part, this thesis presents an easily extendable core Java-like language with mutable state, multi-threading and locks which has been a basis for formalizing our other work.

In Chapter 5 we introduce our programming language Joelle, a unification and extension of several discrete bodies of work. The publications that this thesis encapsulate focus mostly on formal aspects – formalizing and stating and proving properties. In Chapter 5, we take a different route and describe the resulting language and its implementation, and how it incorporates our published results to form a whole, in a "by example" style. Most notably, we show how we extend the work on data race-free active objects in Paper II to intra-parallel active objects (both parallel message processing and structured parallelism during the processing of a single message), regions, and effect yielding.

We now briefly summarize the publications upon which this thesis is based.

Paper I: Ownership, Uniqueness, and Immutability

This paper presents a novel type system, building on Ownership Types, where owner parameters are decorated with modes which govern not only whether an object can be referenced (which is what ownership types does) but also how it may be used (with respect to modification, observational exposure) in a particular context. Ownership modes can express several aspects of use, including read-only references, immutability, and uniqueness. The combination of modes and ownership is particularly powerful as it allows expressing e.g., an immutable list of mutable objects, or present a client with a mutable list whose elements cannot be mutated. The system extends prior work on external uniqueness to allow the construction phase of immutable objects to be arbitrarily long and not tied to a particular scope.

Paper II: Minimal Ownership for Active Objects

In this paper we explore the ideas in Paper I further. There are two main contributions in this paper: First, it shows how the previously explored ideas can be applied to solve the problem of encapsulating active objects to protect their single thread of control abstraction. In doing so, the paper also shows how to safely do zero-copy message passing through unique references. The second contribution of this paper is the reduction of the complexity of the ownership annotations, and the use of defaults to lower annotation burden substantially. The simplified ownership types system divides the heap into isolates, one for each active object, and only allows aliasing for parts of objects which are guaranteed to never change. Consequently, the system achieves data race-freedom, with a static guarantee.

Paper III: Welterweight Java

In an effort to understand how to better describe, formally, the differences and similarities between the different "toy programming languages" we were building, we sought a sufficient core calculus. The de-facto standard formalism for Java-like languages, Featherweight Java [72], does not include aliasing which makes it difficult to describe alias control systems convincingly.

To this end, Welterweight Java is a core Java-like formalization with mutable state, multi-threading and locks, incorporating several tricks such as using a "named form" for expressions, that had been developed over time to simplify formalizing imperative languages. Welterweight Java is specifically designed to be an easily extendable basis for other work, which we show by extending the formalism with Ownership Types and a non-null type system.

We prove type soundness and that our locks correspond to the semantics of Java. In the ownership extension we prove *owners-as-dominators* and in the non-null system we prove that non-null variables never contain a null value.

Subsequent formalization in papers IV and V builds on Welterweight Java.

Paper IV: Multiple Aggregate Entry Points for Ownership Types

This paper partially mitigates one major point of criticism of ownership types: imposing too strong restrictions on the shape of a program and thereby preventing the use of many common object-oriented patterns. The paper presents a principled relaxation of ownership types that allows two or more objects to share a common representation in a controlled fashion without exposing the representation to the outside. (In all other work on ownership types, there is a single point of entry dominating the entire representation of an aggregate object.) The relaxation is "pluggable" in that it only applies to objects in a shared representation – all other objects enjoy the usual encapsulation of *owners-as-dominators*.

We prove type soundness as well as our *ombudsmen-as-dominators* theorem.

Paper V: Refined Ownership

The last paper explores the coarse-grained nature of ownership types when applied to access-control for concurrent and parallel programming.

Classical ownership types divide the heap in a hierarchical fashion. This allows statically checked disjointness of parallel execution only between different aggregates, and between different nesting depths in a system. As we argue in [106], the necessary conditions for nesting-based disjointness rarely occur in practice, leaving only relatively coarse-grained parallelism, excluding parallel operations on objects in a collection.

This paper presents an extension to ownership types that allows the heap to be partitioned, not only in a hierarchical fashion, but each partition may also be further subdivided into smaller parts, down to individual objects. This extends the power of ownership-based effect systems to express parallelism on different elements in collections without imposing global constraints on the elements of the data structure or external objects pointed to by the elements.

In addition to the standard meta-theory of type soundness, we formalize a disjointness invariant that asserts that the accessible mutable state of two threads do not overlap in a well-formed heap and stack, and state deterministic parallelism as a corollary.

The papers included in the printed version of this thesis are reprints with minor fixes, and superficial changes due to different format. Also, the printed version of Paper II in this thesis includes the extended technical report [40], not the version published at APLAS [39].

The Author's Contributions

I Main author.
II The ideas of the author from Paper I led to the work in this paper. Developed in collaboration with co-authors. Implementation of prototype type-checker by the author.
III Main author.
IV Main author.
V Originator of original idea and implementer. Otherwise, contributions were equal among the authors.

Other Contributions

Although not explicitly discussed and included in this comprehensive summary, the following work has contributed in one way or another to the work in this thesis.

1. Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 117–136, New York, NY, USA, 2009. ACM.

2. Johan Östlund and Tobias Wrigstad. Regions as Owners – A Discussion on Ownership-based Effects in Practice. In *IWACO '11, International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, 2011.

3. Johan Östlund, Stephan Brandauer, and Tobias Wrigstad. The Joelle Programming Language. In *International Workshop on Languages for the Multi-core Era (LAME)*, 2012.

4. Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias

Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer Berlin Heidelberg, 2013.

5. Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 233–269. Springer Berlin Heidelberg, 2013.

Artifacts

Much of the work in this thesis has been implemented in our Joelle compiler. The compiler has been an important exploratory tool in working out ideas. Often an idea has first been implemented and tried out, and then further developed and published.

A common criticism of Ownership Types is the lack of compilers and real-life results. The author has implemented a compiler which, to the best of our knowledge, is the first to implement ownership based effects á la Joe_1 [41], safe references, ownership types-isolated active objects, owner inference for owner-polymorphic methods, as well as our own work presented in this thesis.

Other publications

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 377–388, New York, NY, USA, 2010. ACM.

1.2 Thesis Organization

This thesis is organized as follows:

Chapters 2 and 3 cover the necessary background and motivates our work, including related work on concurrency control and alias management. They survey the current state-of-the-art, focusing on the most closely related works.

Chapter 4 further develops the background, highlighting the touch points of Chapters 2 and 3 and how alias management can be applied in concurrent setting, with a focus on ownership-based systems.

Chapter 5 introduces Joelle, the programming language ultimately culminating from combining elements from Papers I–V. Chapter 6 concludes.

The rest of the thesis are Papers I–V.

1.3 Summary in Swedish

Under det senaste decenniet har vi rört oss från en värld där enkärniga processorer är normen, till en värld där fler- och mångkärniga processorer är gängse. Anledningen till detta skifte är att fysiken sätter gränser för hur mycket vi kan öka klockfrekvensen i processorerna, vilket styr hur många operationer en processor kan utföra per tidsenhet. En ökad klockfrekvens leder till ökat effektuttag vilket i sin tur leder till ökad värmeutveckling. Detta ledde i sin tur till en strategiomläggning – istället för att öka beräkningskapaciteten genom att göra en processors kärna snabbare, började processortillverkare öka beräkningskapaciteten genom att bygga processorer mer många kärnor, som därigenom hade möjligheten att utföra flera operationer parallellt.

Innan detta skifte kunde ett befintligt program flyttas från en maskin till en nyare, snabbare maskin och omedelbart dra nytta av den ökade hastigheten. Idag, då flerkärniga processorer är standard, gäller inte detta längre. Program skrivna för att köra på en specifik flerkärnig arkitektur kan inte nödvändigtvis utnyttja den befintliga parallellism som en annan arkitektur erbjuder. Detta medför att programmerare måste skriva program som skalar parallellt, och som dessutom fungerar på datorer med väsentligt olika parallell kapacitet.

De flesta populära objektorienterade programspråken erbjuder idag undermåligt stöd för parallellprogrammering. Den förhärskande modellen bygger på trådar och lås. Trådar och lås är oflexibla, svåra att förstå och svåra att programmera korrekt. Låsstrategier implementeras nödvändigtvis på ett sätt så att olika objekt inte är losskopplade från varann. Detta betyder att inkapsling – den viktigaste komponenten i objektorientering – bryts, och får till följd att ett programs komplexitet blir svårhanterlig.

Actors, aktiva objekt, har föreslagits som en alternativ modell för samtidighet och parallellism i objektorientering. Aktiva objekt är autonoma agenter som kan exekvera parallellt och kommunicerar asynkront med varandra. Inkapslingen bygger på att föränderligt data inte delas mellan aktiva objekt. Om föränderligt data delas så fungerar inte modellen, utan situationer kan uppstå då flera aktiva objekt samtidigt gör åtkomst (där minst en försöker skriva) till en viss del av minnet, ett så-kallat kapplöpningsproblem. Aktiva objekt, precis som trådar och lås, möjliggörs i många populära objektorienterade programspråk genom återanvändbara bibliotek. Tyvärr finns sällan något stöd inbyggt för att verifiera att programmen använder dessa bibliotek korrekt – i synnerhet förhindrar delning av föränderligt data, varför kapplöpningsproblem fortfarande kan uppstå. Programmerare uppmanas att inte dela föränderligt data, men det är inte tillräckligt, inte minst med tanke på moderna systems omfattning och komplexitet – miljoner rader kod och miljarder objekt i komplexa grafer. Om föränderligt data delas fungerar inte längre modellen som avsett, och alla garantier upphör att gälla.

I den här avhandlingen presenterar vi ett nytt programspråk som erbjuder isolerade (inkapslade) aktiva objekt, där delning av icke-föränderligt data är tillåtet, och föränderligt data kan skickas mellan aktiva objekt, men endast finnas hos ett i taget. Program som bryter mot dessa regler identifieras innan de körs och platserna där felen uppstår pekas ut. Vi visar också hur aktiva objekt kan tillåtas exekvera flera meddelanden parallellt utan att introducera kapplöpningsproblem, samt hur strukturerade parallella kapplöpningsproblemsfria beräkningar kan tillåtas inom aktiva objekt.

Arbetet utvecklar ägarskapstyper, ett teoretiskt väl utforskat område. Vi visar hur ägarskapstyper kan användas för att isolera aktiva objekt, och således garantera kapplöpningsproblemsfria program. Ägarskapstyper ger varje objekt en ägare och garanterar att objektet inte kan kommas åt utanför sin ägare. Varje aktivt objekt utgör en ägare och alla objekt inom det aktiva objektet åtnjuter således en garanti att inte kunna kommas åt av andra aktiva objekt. Aktiva objekt måste dock kunna kommunicera med varann för att intressanta program ska kunna byggas, och ofta involverar kommunikation data som skickas mellan aktiva objekt. Objekt måste följaktligen tillåtas som argument i ett meddelande. Vi tillåter icke-föränderliga objekt att delas fritt mellan aktiva objekt. Föränderliga objekt kan tillåtas som argument i ett meddelande om det kan garanteras att det aktiva objekt som skickar meddelandet inte kan komma åt objektet

efter att meddelandet har skickats. Detta kan i sin tur garanteras om vi vet att det endast finns en referens till objektet som ska skickas, vi kallar sådana objekt unika. Då kan referensen förstöras och objektet är inte längre åtkomligt. På så vis flyttas objektet från ett aktivt objekt till ett annat, utan att för den sakens skull introducera ett kapplöpningsproblem. Föränderliga objekt som inte är unika måste kopieras när de skickas som argument. På så sätt sker ingen delning.

Ägarskapstyper inför restriktioner på hur ett programs data kan byggas upp. I många fall är dessa restriktioner för starka, vilket får till följd att många vanliga objektorienterade idiom inte tillåts. Vi presenterar en lösning på detta problem där två eller flera objekt tillåts dela på en area där objekt kan leva. Denna delning är explicit i det att bara sådana objekt som givits tillåtelse kan delta. Vi kan på detta sätt tillåta flera objektorienterade idiom utan att bryta mot inkapslingsprincipen. Detta tänjer på den strikta regim som ägarskapstyper upprätthåller, men det gäller enbart objekt som lever i den delade arean, och alla andra objekt åtnjuter samma starka garantier som ägarskapstyper normalt ger.

Ett annat problem med ägarskapstyper är att det är svårt att skilja på objekt som tillhör samma ägare. Ett typexempel är datasamlingar, som exempelvis träd. Normalt med ägarskapstyper är alla noder i trädet ägda av trädet, och på så sätt inkapslade. Ett problem uppstår när man vill använda den information ägarskapstyper ger för att garantera kapplöpningsproblemsfria parallella beräkningar. Ägarskapstyper garanterar att objekt med olika ägare inte delar data, och således kan parallella operationer på sådana objekt tillåtas utan att introducera kapplöpningsproblem. Men i trädet vi tidigare beskrev så ägdes samtliga noder av samma ägare, och ägarskapstyper är alltså inte tillfredsställande i detta avseende. Vi löser detta problem genom en utökning av ägarskapstyper där en ägare tillåts delas upp i mindre delar, ner till att varje objekt kan ses som det har en egen ägare, skild från alla andra objekt. På så vis kan olika objekt skiljas från varandra och parallella beräkningar över trädformade datastrukturer tillåtas med garanti mot kapplöpningsproblem.

Vi har implementerat en kompilator för vårt programspråk och visar att parallellism inom aktiva objekt medger att den parallella beräkningskapaciteten i flerkärniga processorer kan utnyttjas bättre än med traditionella aktiva objekt.

2

Concurrency Control

The related work of this thesis can be crudely divided into two major parts; concurrency control and alias management. Alias management aficionados will kindly have to sit through this chapter about concurrency control (or skip directly to Chapter 3), as this is where we begin.

In concurrent and parallel programming concurrency must be controlled in some way, and there are several options how to do this. We will review these options but focus mainly on those related closely to our own work.

Concurrency and parallelism are two distinct concepts – concurrency deals with asynchronous events which may or may not happen in parallel while parallelism is a form of program optimization to utilize parallel computational resources. The goal of this thesis is to make possible compile-time guarantees about effect disjointness which allows interference-free parallel execution of concurrent tasks. The work of scheduling tasks efficiently is therefore largely orthogonal to the work in this thesis and an interesting direction for future work.

2.1 The Problem with Threads and Locks

Parallel programming in almost any of the most popular object-oriented programming languages today is synonymous with using threads. Threads are easily added to a language as a library because they are "already there", provided by the operating system. However, multi-threading is difficult: we want parallel performance to be portable (what is the right number of threads for a particular deployment), efficient (e.g., false optimizations – when does it pay to start/stop a thread to perform something), memory-aware (what is in the cache of the core running this thread), correct (e.g., free from data races or multi-object atomicity violations) and these are not easily obtained using threads. While libraries can be built on threads to provide more high-level concurrency abstractions, such implementations typically rely on programmers "getting it right" for correctness. Providing a concurrency library that guarantees race-freedom statically is generally very hard, if at all possible, because the necessary restrictions cannot be implemented in the base language, and providing guarantees based on dynamic checking requires expensive run-time checks. As an example, the popular Akka framework for actors in Scala advises its uses not to share mutable data across threads, but does not provide any tools to support this. Haller and Odersky proposed a way to support safe efficient sharing across actors in Scala [61] and the system has been implemented as an optional compiler plug-in, though it has not been included in the language, in part because it requires a huge effort to annotate existing libraries.

Threads are predominantly used with shared memory. Shared memory makes communication between threads efficient, but it also requires that accesses to shared memory be managed to avoid simultaneous access. To this end various kinds of concurrency control mechanisms may be employed, such as semaphores, monitors and locks. These all have in common that their correct use can prevent threads from entering a piece of code under certain circumstances. Using a concurrency control mechanism, such as a lock, a shared location can be protected in the sense that before the access a thread can acquire (or lock) the lock and prevent another thread from accessing the same location. When done accessing the shared location the thread can release (or unlock) the lock and allow other threads to acquire it. While the principle is simple, in practice, this turns out to be very hard to get right due to the complexity of modern software. One underlying reason for this is that code which does not require con-

currency control is syntactically identical to code that does, generally requiring non-local reasoning and checking available documentation. Acquiring a lock to protect a shared location does not prevent simultaneous access if not all other accesses to the same location acquire the same lock (protocol violation). This can be somewhat helped by forcing lock-taking at definition-site, e.g., Java's synchronized methods, but this forces locks to be reentrant (allowing the same thread to acquire the same lock more than once) and increases lock complexity. Disregarding this aspect, a variable is often dependent on other variables and operations may require multiple locks to be acquired to sufficiently protect program invariants. Furthermore the introduction of locks and other concurrency control mechanisms can easily lead to a situation where two threads wait for each other to release their respective locks. In this situation execution comes to a halt and will never continue–this is called a *deadlock*. Programming with locks is thus a delicate balance: too many locks (fine-grained locking) allows for increased level of parallelism but can lead to insufficient protection and delicate deadlocks, too few locks (coarse-grained locking) might cause excessive waiting for locks to be released, but are generally easier to get right.

When two or more threads are allowed to simultaneously access the same memory location, and at least one of them is a write action we say that it is a *race condition* or a *data race*. If two or more accesses are reading the location, while no writes occur, there is usually no problem for program correctness and if it is intentional we can say that it is a *benign race*. In the presence of data races, not only is it possible that a reader may read corrupt data, but also whether it does so may depend on the particular timing of the particular run of the program. In one run there may be no apparent problem, while in the next, the problem surfaces. Finding bugs of this kind can be very difficult because they may be intermittent and adding or removing code for testing purposes or running the program in a debugger may change the timing and make the bug disappear.

Finally, in the object-oriented setting, threads and locks break encapsulation, causing objects to lose compositionality [123]. Object-oriented design tackles complexity by breaking large complex problems up into smaller pieces (classes/objects) and solve each piece in isolation. The pieces may then be composed to solve the larger problem. Without composition complexity is difficult to reduce, and without encapsulation composition is lost. For example, objects that work well in isolation can either cause deadlocks when composed because of how they interact with shared state or cause thread-counts to exceed the suit-

able limit for a platform. Thus, concurrency (and parallelism) cross-cut application logic in a way that causes threads to cross object boundaries. Ultimately, this calls for other programming abstractions for concurrent and parallel programming that enable concurrency rather than mandate it, and avoid complicated protocols which are hard to get right without non-local reasoning.

2.2 Fork/Join-style Structured Parallelism

Fork/join[1] parallelism is a way of splitting (fork) execution into two or more parallel executions which later merge (join). At the join site the forked tasks wait for all other tasks to finish and the result of the computation can be computed or combined. Fork/join parallelism is at the core of the OpenMP framework [104] as well as the Cilk language [14] (and later derivatives). In recent years fork/join libraries have been added to both C# [81] and Java [79]. HabaneroJava supports this style of programming through a construct called async/finish [33], inspired by X10 [34]. The async statement is used to create a new asynchronous task which may run in parallel with the current thread. If asyncs are used inside a finish block, execution of the current method is halted until all asyncs inside the finish block have terminated. This design has some nice properties such as deadlock-freedom [115] and as long as there are no data races, parallel execution is deterministic [16, 111]. Furthermore, finish blocks clearly scope asynchronous behavior syntactically, which may improve program understanding, and provide an obvious place to deliver exceptions thrown in the tasks [34].

The code in Figure 2.1 shows HabaneroJava syntax for an async/finish block. The for loop will create `n` parallel tasks which all call `f()`. The enclosing finish block performs the join and makes sure that execution will not continue after the finish until all spawned asyncs are done.

Fork/join parallelism is particularly handy with computations on recursive data structures such as trees. The forked tasks can themselves fork new tasks to handle computations in subtrees. The code in Figure 2.2 shows a simple fork/join computation in Java[2], summing numbers in an array. Each task is given a range

[1] The first mention of fork/join in the literature appears to be Conway's A Multiprocessor System Design, 1963 [44].

[2] The example is taken from Dan Grossman's lecture notes: `http://homes.cs.washington.edu/~djg/teachingMaterials/spac/grossmanSPAC_forkJoinFramework.html`

```
1    finish {
2      for (int i=0; i<n; i++) {
3        async { f(); }
4      }
5    }
```

Figure 2.1. Using the HabaneroJava async/finish construct to spawn n asynchronous tasks and wait for them to finish.

of the array to work on. As long as the range is big enough (greater than `SEQUENTIAL_THRESHOLD`) new tasks are forked and the range split accordingly. The actual summing of a particular range of values happens when the range is small enough, and is done sequentially. Once the summing is done the task is done and the parent (forking) task can retrieve the result and combine (sum in this case) it with its own result. Once all tasks are done the complete result is computed. Of course this example is very simple, but the technique is powerful and many problems can be massaged to fit this form.

For fork/join parallelism to work properly, i.e., be deterministic, there can be no data races between tasks. In the example we use primitive values, so race-freedom comes down to splitting the ranges correctly (granted, the array is only read so even overlapping ranges would not break determinism in this case.) In a more complex setting where the elements are objects which may point to shared state the situation becomes more involved. Getting this right is completely up to the programmer in most languages and frameworks, including Java [79], C# [81], OpenMP [104] and Cilk [14]. A notable exception is Deterministic Parallel Java which uses effect annotations and regions to guarantee that parallel computations are race free in certain cases [16].

Our work on Refined Ownership (see Section 5.3.1, Paper V) overcomes this problem and allows deterministic parallel computations over structures with pointers to external elements.

2.3 Actors in Theory and Practice

The actor model was conceived in the seventies by Carl Hewitt and his group at MIT's Artificial Intelligence Laboratory [68]. Further work on the theoretical framework was done by Baker [67] and later Agha [4]. The actor model is a computational model, where the actor is the fundamental unit of computation and

```
1   import java.util.concurrent.ForkJoinPool;
2   import java.util.concurrent.RecursiveTask;
3
4   class Globals {
5     static ForkJoinPool fjPool = new ForkJoinPool();
6   }
7
8   class Sum extends RecursiveTask<Long> {
9     static final int SEQUENTIAL_THRESHOLD = 5000;
10    int low;
11    int high;
12    int[] array;
13
14    Sum(int[] arr, int lo, int hi) {
15      array = arr;
16      low   = lo;
17      high  = hi;
18    }
19
20    protected Long compute() {
21      if(high - low <= SEQUENTIAL_THRESHOLD) {
22        long sum = 0;
23        for(int i=low; i < high; ++i)
24        sum += array[i];
25        return sum;
26      } else {
27        int mid = low + (high - low) / 2;
28        Sum left  = new Sum(array, low, mid);
29        Sum right = new Sum(array, mid, high);
30        left.fork();
31        long rightAns = right.compute();
32        long leftAns  = left.join();
33        return leftAns + rightAns;
34      }
35    }
36
37    static long sumArray(int[] array) {
38      return Globals.fjPool.invoke(
39             new Sum(array,0,array.length));
40    }
41  }
```

Figure 2.2. Simple example using Java's fork/join library to sum an array.

is made up of three constituent parts: processing, storage and communication. Actors have addresses, meaning they have an identity, and if one actor knows the address of another actor it can send a message to it. Messages received by an actor are processed sequentially to completion. During the processing of a message an actor can perform a finite combination of three actions: create more actors, send messages to other actors to whom it knows the address, and designate how it will handle the next message it receives. The last point is important because this is what makes actors stateful. In the theoretical actor model the actor replaces itself with a new actor containing the new state (while keeping its address), thus while the actor itself is a functional entity the replacement action makes it stateful in the view of others[3]. Practical implementations typically let the actor overwrite the old state by a new one in an imperative fashion. Actors have some sort of buffering and arbitration mechanism to order incoming messages which arrive close in time, usually referred to as the mailbox, mail queue, message queue or just queue.

Actors need to represent their internal state somehow. Practical realizations of the actor model do this in different ways. Many languages use regular objects as internal representation which is powerful, but leads to problems if such objects are shared between actors. The ABS language [36], as a contrast, employs functional programming within actors, thus avoiding the sharing issue altogether.

In the last decade, with the success of languages such as Erlang and Scala, the actor model has had a revival. The multi-core revolution and the poor tools for concurrent programming have had people turn to actors for help. Indeed actors seem to be a much better fit for concurrent object-oriented programming than do threads and locks [94, 75, 118]. Actors, in theory, enforce encapsulation, thus retaining compositionality and the ability to reason locally about correctness, even in a concurrent setting.

There are two main categories of actors in the wild:

Active Objects are an implementation of the *Active Object* pattern [78] and specify an external interface, like a plain old class. Messages are usually called *asynchronous method calls*. If statically typed, a type system can ensure that the receiver of a message understands the message (as in typed

[3] This is similar to how Erlang becomes stateful in the presence of actors. In Erlang, processes (actors) typically model state by recursion and the passing of arguments.

object-oriented languages.) Akka [5] and ProActive [30] are examples in this category.

Actors do not specify an external interface. A type system can in general say nothing as to whether a message is acceptable by a receiver, even in an otherwise statically typed language. Scala actors [60] and Erlang [9, 10] are examples in this category.

The actor model is intentionally abstract and says nothing about its implementation. However, because of hardware and operating system constraints, there are a bounded number of feasible ways to implement actors in reality. Most implementations are very similar, and in particular they impose discipline on the programmer in order for programs to work correctly (Erlang [10] is a notable exception among widely used actor languages.) Actors are assumed to not share state, or at least not in a way that would cause data races. Hewitt et al. write:

> *"Sending a message to an actor is entirely free of side effects [...]. Being free of side effects allows a maximum of parallelism and allows an actor to be involved in several conversations at the same time without being confused."* [68].

Most actor languages or frameworks provide no guarantee that mutable state is not shared between actors. This is non-satisfactory because in the presence of sharing, the actor model does not work. However, providing such guarantees in a framework built on an existing language may be hard to do statically and requires expensive run-time checks if done dynamically.

2.4 Dealing with Asynchrony

In active object systems with asynchronous method calls *futures* [12] are often employed. A future is a placeholder for the return value of an asynchronous method call, as this value may not be available right away. So the result of an asynchronous method call is an object, a future, which may or may not contain the result of the request. Depending on the particular implementation of future different operations on it may exist. Often the receiver of a future can check whether the value has arrived, and otherwise proceed to do other useful work. The receiver of the future may also choose to block on the future and wait until the value becomes available. Without futures a programmer must manually

implement some protocol to keep track of which answers are due to which requests. Futures encode such a protocol automatically by connecting the answer to the request. This makes asynchronous method calls resemble synchronous method calls. Futures exist in different varieties and have been called different things over the years.

Futures first appeared in the late 1970's, where Baker and Hewitt [12] use futures to evaluate sub-expressions in parallel. According to Baker and Hewitt, Friedman and Wise call the same concept *promises*, while Hibbard calls them *eventuals*. Halstead's MultiLISP [62] in the early 1980's appears to be the first implementation of futures in a language. Being a variant of LISP these futures were untyped and not distinguishable from other values. When trying to access a future which did not have its value the current thread would block and wait for the value. Liskov et al. [85] take the future concept and bring it to a typed setting, now called *promises*. In the typed world one can distinguish futures[4] from other values and thus do away with dynamic checks in run-time. Liskov's promises also had the ability to deliver exceptions thrown in the other thread.

We should note that there are other variations of futures as well, as for instance used in X10 [34], where futures can be used to actively spawn parallel tasks. In the context of this thesis, however, we are mostly interested in futures as placeholders for eventual return values from asynchronous method calls. There is plenty to read about futures in the literature, please refer to e.g., de Boer et al. [46] for discussion and formalization of the semantics of futures.

Please note that the meanings of the terms future and promise are not consistent in the literature. For example, in some work promise is used to mean a future which is not tied to a particular piece of code (e.g., the return value of a method) but as an entity that can be passed around and fulfilled by anyone who has a reference to it, e.g., as in [2].

2.5 Actors in the Wild

Many incarnations of the actor model exist, some more faithful than others. In this section we cover a small subset of these, focusing on those most well-

[4] Note that some languages prefer MultiLISP's "transparent" futures even in a typed setting. ProActive [30] is an example using this design, calling it *wait-by-necessity* [28].

known as well as the most interesting in the context of our own work. In particular actor and active object languages with unusual features and different strategies for isolation. The presentation order of these has no significance in any way, although the most widely used languages break the ice, and where there are dependencies the presentation begins with the original.

2.5.1 Erlang

Erlang was created by Joe Armstrong and his team at Ericsson in the later part of the eighties [9]. Erlang is a mostly functional language designed to build concurrent and distributed, fault-tolerant, soft real-time and high-availability applications [9, 10].

Actors (processes in Erlang parlance) are easily created in Erlang by supplying a function value to a `spawn` expression, which returns a handle to the new actor. Actors communicate by sending messages asynchronously. Pattern matching is a central ingredient in Erlang, and in particular when inspecting incoming messages. There is no explicit interface specifying what messages an actor accepts. In Erlang values are immutable, so sharing is not an issue for data race-freedom. A defining feature of Erlang is its fault-tolerance. Actors are related and a parent actor may decide on how to handle a signal from a crashed child actor, e.g., by creating a new actor and let it continue the execution.

Erlang has seen a huge increase in popularity in recent years. Many well-known projects and companies ranging from telecom nodes to gaming sites to e-commerce use Erlang to implement high-availability services. Erlang is probably the most widely used actor-based language out there.

2.5.2 Scala

Scala [103] is an object-oriented programming language with functional features, created by Martin Odersky. One of the famous features of Scala is its actors library [60], the primary concurrency model. While implemented as a library the lax and extensible syntax of Scala makes the integration seamless. Actors in Scala do not provide an interface with asynchronous methods. Instead messages are received and identified with pattern matching, similar to

Erlang [9, 10]. Scala does not prevent sharing of mutable state between actors, although an extension has been proposed to support safe and efficient sharing [61]. The extension has been implemented as a compiler plug-in, but has not been included in the language because it requires a huge effort to annotate existing libraries.

2.5.3 Akka

Akka [5] is an additional actor library for Scala (and Java) which provides distribution and fault-tolerance similar to Erlang [9, 10]. Akka supports active objects (called *typed actors*) as well as Scala-like actors. Typed actors, like regular classes, export a public interface, signaling which messages (asynchronous method calls) it accepts. Akka implements a fault-tolerance strategy similar to Erlang, where actors are hierarchically related and parents supervise their child actors. In Akka supervision is mandatory whereas in Erlang it is optional. Akka does not prevent sharing of mutable state across actors, programmers are just advised not to.

2.5.4 ProActive/ASP

ProActive is an active object framework for Java. It is a realization of the ASP object calculus [29]. ProActive supports active and passive classes. Active classes communicate via asynchronous method calls and futures. ProActive is statically typed, but futures are *transparent*, meaning that it does not show in the type whether a variable points to a future or a regular object. Accessing a future which does not yet contain the computed result blocks the current thread until the future has been resolved. Caromel et al. call this *wait-by-necessity* [28]. To preserve a non-sharing discipline, passive objects are deep copied when sent between active objects. Henrio et al. [66] propose an extension to ProActive, adding parallel message processing. We discuss this extension in Section 5.3.2.

2.5.5 E

E [92] is an object-based (or prototype-based, similar to Self [127]) dynamically typed language sporting an actor-like concurrency model in which groups of

objects (such a group is called a vat) share an event loop (thread of control), a stack (for sequential internal calls) and a pending delivery queue (mailbox.) E handles sharing of mutable state by distinguishing between *near references* and *far references*. A near reference is a regular reference from one object to another within the same vat. A far reference is a reference from an object in one vat to an object in another vat. Such references may only be used for asynchronous communication. All asynchronous communication sent to objects of the same vat ends up in the same pending delivery queue, and messages are picked one at a time and processed to completion. Thus execution within a vat is synchronous, despite multiple entry-points. The result of an asynchronous call is a *promise*, a non-blocking future which itself accepts asynchronous calls, delivered once the promise is resolved (fulfilled).

2.5.6 AmbientTalk

AmbientTalk[5] [128] is a language, although general purpose, designed for the volatile nature of mobile networks, where nodes and resources may come and go at any time and without prior warning. AmbientTalk is very similar to the E language [92] both in its object and concurrency models.

2.5.7 Creol

The active object semantics of Creol [74, 46] sports an unusual feature. In Creol, like most actor/active object systems only one thread may be active inside the active object at a time. However, Creol allows for cooperative scheduling through a `yield` keyword that suspends the execution of a message and allows it to be resumed later. A yield stops the execution of a running method, saves the local state of the method, and allows the active object's scheduler to start executing another pending method invocation. Later the stopped method can be scheduled again and continue where it left off. Thus several method executions can be active at the same time, but not running at the same time. The programmer is responsible for making sure that no interleaved operation interferes with the

[5] AmbientTalk was first published in [47] but a revised version of the language was subsequently published [128] subsuming the first. We describe the revised version.

suspended method. Consequently in Creol there is a potential issue with atomicity violations although the programmer must explicitly allow such behavior.

2.5.8 CoBoxes

The JCoBox language [118] features CoBoxes[6], which are very similar to vats in the E language [92] (and actors in AmbientTalk [128]), albeit in a statically typed Java setting. Far references and asynchronous method calls are used for inter-CoBox communication. JCobox borrows cooperative scheduling from Creol [74, 46] to allow several method executions to be active (all but one suspended) at the same time. CoBoxes have been integrated into the ABS programming language (there called cogs.) ABS does not support passive objects [36].

2.5.9 SCOOP

SCOOP [97, 94] is a concurrency extension to the Eiffel programming language [91]. In SCOOP the heap is divided into regions and each object belongs to one region. A region can have a processor (thread of control) and then becomes similar to an active object. Sharing of state is allowed in SCOOP but a reference from an object in one region to an object in another region is annotated *separate* (similar to far references in the E language [92].) Communication between regions happens via *separate calls* which can be either synchronous or asynchronous, depending on a rather intricate set of rules. Notably there are no futures, and consequently a separate call to a *query* (non-void method) is always synchronous. Although references are allowed across regions, objects of a region are only exercised by a single processor, making regions sequential.

2.5.10 Kilim

Srinivasan and Mycroft's Kilim [122] is a low-level actor framework for Java combining a number of techniques to support concurrency by means of green threads and a no-sharing discipline. Sharing is restricted by treating messages

[6] A previous version of the language model was first published [117] but subsumed by the JCoBox language.

as a special category of objects, and enforcing that mutable messages are tree-shaped and uniquely pointed to and thus easily and safely transferable between actors. Immutable messages may be shared without restriction.

2.6 Software Transactional Memory

Software Transactional Memory [121] (STM) is an optimistic concurrency control mechanism in which accesses to shared memory are treated as a single atomic transaction. During the transaction a log is kept on all reads and writes to (logical) memory. Once the transaction is finished the log is compared with the actual memory to see if the values match up. As long as no conflicting writes by other threads happen during the transaction, all writes of the transaction can be made public by committing them to memory as a single atomic operation. If there are conflicts, however, the transaction is rolled back and restarted, hoping that it will run without conflicts the next time. The caveat in the STM approach is that operations with side-effects, such as I/O, cannot be rolled back, in the general case. STM has some benefits over lock-based concurrency control in that it can be made deadlock-free and relieves the programmer of having to find the right locking granularity, though similar compositionality issues as are found in lock-based concurrency control apply in the STM setting. Harris et al. propose a STM system for Haskell where side-effects are caught by the type system (this is easily done in Haskell as side-effects are always visible in the types of functions) and compositionality is helped by making transactions nestable [64]. Although performance has been improved significantly over the years, there will always be a cost incurred by the book-keeping in STM systems.

❧

Having surveyed the need for concurrent and parallel programming, the state-of-the-art of concurrency control, and actor-based systems, we move on to the second background theme: alias management. Aliasing is a prerequisite of data races, and proper alias management is key to isolating an actor's state, to enable sequential reasoning and compositionality. Later, in Chapter 4, we will discuss how alias management has been applied to concurrent and parallel programming in the past.

3

Alias Management

Aliasing in programs occurs when there are two or more paths which lead to a particular piece of data. To put it simply (but not precisely) we can say that aliasing means that two or more variables in the program point to the same object. Aliasing is powerful because it allows efficient implementation of many common object-oriented patterns, but it also complicates reasoning. For example, it is not possible to know what is printed by the following program without also knowing what objects are "pointed to" by `x` and `y`.

```
x.f = 5; y.f = 4; print x.f;
```

If `x` and `y` are *not* aliases, the program will always print 5. If `x` and `y` *are* aliases, the program will always print 4.

In a concurrent setting, the problem is even more complicated which can be explained by the following program:

Thread 1	*Thread 2*
`x.f = 5; print x.f;`	`y.f = 4;`

Not only do we need to know whether `x` and `y` alias, we need to know how Thread 1 and 2 are scheduled. If `x` and `y` are *not* aliases, the program will always print 5. However, if `x` and `y` *are* aliases, whether `x.f = 5` is scheduled before or after `y.f = 4` controls the program's output. This illustrates how the addition and combination of concurrent or parallel programming exacerbates the aliasing problem. (The possibilities of compiler reordering for optimization and weak memory models make matters even more complicated, see e.g., [17, 116].)

Thus, aliasing is key to data races and atomicity violation[1].

In object-oriented programming encapsulation is one of the pillar concepts. Encapsulation is what makes it possible to reason about individual objects in isolation and what makes them compositional. A common way to enforce encapsulation is by so-called *name-based encapsulation*. Name-based encapsulation is usually implemented as a modifier on class members, and for this discussion particularly on fields (e.g., a `private` field in Java), which makes them accessible only from other members in the same class (there is usually a version that grants access from subclasses, making the problem even worse.) Name-based encapsulation thus protects, as the name suggests, the name of the member, i.e., it makes the name illegal to mention outside the class. Name-based encapsulation, however, does not protect the object referred to by the name (field in our discussion.)

The code example in Figure 3.1 shows a simple Java class `Leaky` and a companion class `Data`. Examining this code, field `d` is declared `private` and the method `setDataString(String)` checks its incoming argument to see that it is not null, surely `Leaky` does all it can to ensure that its `toString()` method will never throw a `NullPointerException`, and in isolation is clearly correct! But as soon as someone calls `peekData` `Leaky` is no longer in control over its representation and an outsider, possibly unaware of the expectations of class `Leaky`, could very well set the string to `null` in the `Data` object. This very simple example is enough to show that local reasoning–the ability to inspect a piece of code in isolation–can be very difficult in the presence of aliasing.

In 1991, John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux and Richard Holt set out in *The Geneva Convention on the Treatment of Object Aliasing* to define what alias management is and classify different flavors of alias manage-

[1]Global variables are also a source of these problems, but if viewed as each thread having its own variable pointing to a shared variable, then this too is aliasing.

```
1   class Leaky {
2     private Data d = new Data();
3     public void setDataString(String s) {
4       if (null == s) {
5         throw new NullPointerException();
6       }
7       d.setS(s);
8     }
9     public String toString() {
10      return d.toString();
11    }
12    public Data peekData() {
13      return d;
14    }
15  }
16
17  class Data {
18    private String s = "";
19    public void setS(String s) {
20      this.s = s;
21    }
22    public String toString() {
23      return s.toString();
24    }
25  }
```

Figure 3.1. Example showing the problem with name-based encapsulation.

ment [70]. A huge body of work on alias management has since been produced. We will discuss only a subset of this work, in particular work that is related to *ownership types*, as this is a foundation of our own work presented in this thesis.

3.1 Flexible Alias Protection

Object aliasing and stable object identity are at the core of object-oriented programming. They allow efficient implementation of many common object-oriented patterns. At the same time, as we have demonstrated, aliasing can be a source of many hard to track down bugs, where values may change under foot and invariants break. In 1998 Noble, Vitek and Potter introduced Flexible Alias Protection (FAP) [101], a conceptual model of inter-object relationships. The goal of FAP was to manage the effects of objects being updated through aliasing. This is done by limiting how and where aliasing is allowed. To this end

FAP provides a set of programmer annotations, or modes, which control how a reference may be used. The modes are `rep`, `free`, `var`, `arg` and `val`. The `rep` mode means that an object belongs to the *representation* of another object and cannot be *leaked* outside of its *owner* object (this would have been appropriate for field `d` in the leaky class example in Figure 3.1.) The `free` mode denotes a reference that is uncaptured by any other variable in the system; it is unaliased. The `var` mode means the reference can be shared freely and used to modify the object; this is basically the normal reference of most popular object-oriented languages, except that a `var` reference cannot be assigned to a variable with any other of the modes, as this would be unsound. The `arg` mode is perhaps the most interesting and novel of the modes: it denotes a reference that can only be used to access immutable state, i.e., neither modify nor observe concurrent modifications through other, less restricted references. This allows sharing of immutable subparts of an object via an `arg` reference. The `val` mode is used for value types, meaning objects that cannot change at all; we call these *immutable* objects.

3.2 Ownership Types

Attempts to further understand and formalize FAP lead to the invention of *Ownership Types* [42]. In ownership types every object belongs to an *ownership context*, or simply an *owner*. Whenever a new object is created a new ownership context is created with it. Thus we may also say that every object is owned by another object[2]. The ownership context created with an object is usually called its *representation*, `rep` in FAP terminology, though in original ownership types it is called `this`. With ownership types each object can specify which other objects belong to its representation and these are then protected from being *leaked*, or aliased, outside the owning object. To emphasize that objects are usually composed of a number of representation objects, we sometimes speak of *aggregates* instead of objects [101].

The fact that a representation object cannot be leaked to the outside of its owning object has the consequence that in order to exercise a representation object from the outside one must ask the owner to do so. Thus any access to a repre-

[2] This is not strictly true because there is also a context `world` which encloses all other contexts and does not correspond to an object.

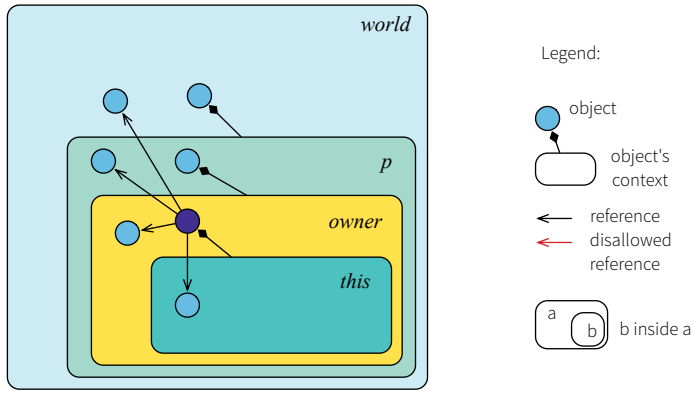

Figure 3.2. Permissible references in ownership types from the perspective of the dark blue object. Boxes with rounded corners denote ownership contexts, and nesting denotes ownership. The representation context is linked to its owning object by a diamond arrow.

sentation object will have the owner of that object in its access path. The rules of ownership types put constraints on the topology of object graphs. Namely, the object graph forms a tree, and the owner of an object is always a parent (dominator) node in this tree. This fact was formalized by Clarke et al. [42] and subsequently named *owners-as-dominators* [35].

Figure 3.2 shows allowable references in ownership types. Apart from accessing its own representation (this), an object has the right to access its owner's representation (owner) as well as any object in the outermost ownership context `world` (world). An object can also be given permission to access other outside ownership contexts (p). This is done by parameterizing the type with the names of these contexts. We call such parameters *owner parameters* and the names used to instantiate the type we call *actual owners*.

Figure 3.3 a partial linked list implementation. The `List` class is parameterized over `data`, which is a local name for the owner of the elements added to the list. The field `first` points to the first link and links are part of the representation (owned by the list instance), hence `this` prepended to its type. The permission to access objects in `data` is passed on to the links which actually hold the elements. The `Link` class again is parameterized over `data` (the name is local and its resemblance to the parameter in class `List` is merely a coincidence), which is used to type the element stored in the link. The `next` field is interesting as it makes use of `owner`. Every object has access to its owner context, and `owner` (as well as `this`) is therefore usually an implicit owner parameter. The owning

```
1   class List<data> {
2     this:Link<data> first;
3
4     void add(data:Object el) {
5       ...
6     }
7   }
8
9   class Link<data> {
10    data:Object element;
11    owner:Link next;
12  }
```

Figure 3.3. Linked List example in ownership types.

context of a `Link` instance is the `List` instance, so `owner` in `Link` corresponds to `this` in `List`. In the `Link` class we make use of the `owner` parameter to type the `next` field, which thus points to an object in the same ownership context as the current link instance. Figure 3.4 shows the resulting object graph.

We have already mentioned that objects and thus owners can be outside (and inside) other owners. An object's representation context is inside its owner's representation context. All ownership contexts are inside **`world`**. The nesting structure of owners can be made explicit in the owner parameters of a class. There it is possible to put constraints on the actual owners used to instantiate the type by saying for instance that a particular owner must be outside another owner. Thus information about the relation of owners may be transferred from

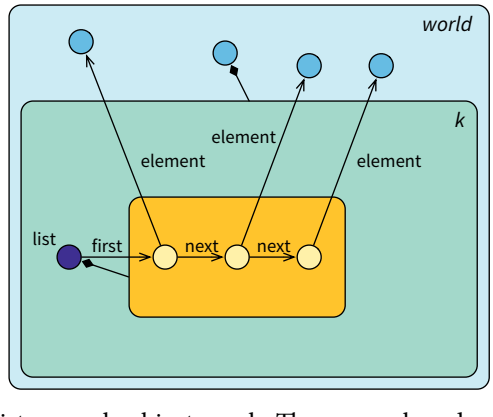

Figure 3.4. Linked list example object graph. The orange box denotes the lists' (dark) representation which nests the links (yellow). Elements objects are owned by world.

one context to another. The default relation on owner parameters is outside `owner`, which is a requirement for soundness.

Owner-polymorphic methods[3] [35, 38, 130, 18] (OPMs) are methods parameterized over owners. These owners are different from the ones in the class header, thus temporarily extending what the method is allowed access. Any owner introduced as a parameter on a method is valid only during the execution of the method, thus making arguments whose types contain an owner parameter safe from being captured by the method (any heap alias created is unreachable by the rest of the system.) OPMs are even allowed to break owners-as-dominators as it is only for a limited time and scope and once the method exits owners-as-dominators is guaranteed to hold.

Ownership types provide a strong notion of encapsulation. An object is put in an ownership context at creation time and stays there for its entire lifetime (external uniqueness allows movement [38], explained in Section 3.4.) This can be very helpful in a variety of applications, such as verification, memory management, security, visualization and understanding of systems and software [37] and perhaps most related to our studies, concurrency and parallelism, which we will discuss later.

The benefits of ownership types, however, come at the cost. There are common object-oriented patterns that are difficult or even impossible to encode with ownership types [6, 99]. Iterators are one such example. An iterator needs to be accessible outside the data structure it iterates over[4], but at the same time it needs to access the representation of the data structure. This is not compatible with owners-as-dominators, and goes against everything we have previously said about ownership types. Figure 3.5 shows the object graph with the illegal reference from the external iterator to the list representation. Several attempts at solving the issue have been presented which all require relaxing owners-as-dominators to allowing an object to leak its representation voluntarily, either both to the heap and the stack [18, 86, 20, 132, 35] or just to the stack [41].

We will later introduce our work on *ombudsmen* (Section 5.6, Paper IV), a principled relaxation of owners-as-dominators which allows iterators and other patterns where more than one entry-point to a single representation is required.

[3] Called *context-polymorphic methods* in the original formalization by Clarke [35].
[4] Arguably, there are many ways to implement an iterator, see e.g., Noble's review of encapsulation in eight different iterator designs [99].

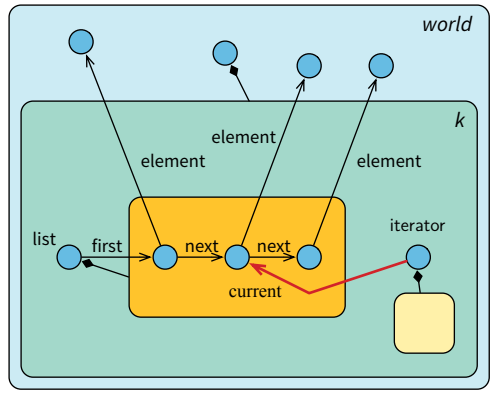

Figure 3.5. Linked list object graph with iterator.

The main advantage over the previously mentioned work is the clear separation between which objects are shared and which are not.

Over the years several flavors of ownership have been proposed, as well as a wealth of work applying ownership for different applications. We will only cover a subset of this work here (and in Chapter 4.) Please refer to our survey paper for a more comprehensive study of ownership systems and their applications [37].

The Universes system [95] introduces an aliasing discipline called *owners-as-modifiers* [49]. Owners-as-modifiers allows objects to escape their owning context but such references become *read-only*. Via a read-only reference fields may only be read, and only calls to *pure* methods are allowed. A pure method may not modify any object that existed prior to its invocation, including its receiver. Universes thus allows for more flexible aliasing, but at the cost of observational exposure [24] (see Section 3.3.) Universes has been extended with generics [48, 50] and ownership transfer [96].

Aldrich and Chambers take a different approach with Ownership Domains [6], decoupling ownership mechanism and aliasing policy. Ownership domains allow the programmer to define whatever aliasing policy fits her purposes, enforced by the system. In contrast to ownership types, where a program design may need to be massaged into a shape that fits the strictness of owners-as-dominators, in ownership domains the massaging is of the aliasing policy to fit the program design. A problem with this strategy is that it can be difficult to understand what an ownership domains policy gives in terms of guarantees, and therefore unclear exactly what it means to have a program statically checked to

conform to an ownership domains policy. Abi-Antoun et al. present a tool for extracting conservative approximations of run-time object graphs from static ownership domains annotations [1]. Such a tool might help in this respect.

3.3 Immutability and Read-Only

Objects whose state cannot change are often referred to as *immutable* objects. Immutability is a powerful and stable property, useful in many situations, and particularly in a concurrent setting as it allows unrestricted sharing without data races. Immutability may be deep or shallow, meaning it can apply to all objects reachable from the immutable object, or only to the fields of the immutable object itself. Deep immutability is required for data race-free sharing. Construction of immutable objects is sometimes non-trivial especially in the presence of cyclic object structures. Complex structures may change state (during initialization) and finally "freeze" to never change again. In particular, making sure that initialization of an immutable object does not leak a reference to the object is crucial for soundness.

A clever way to solve the initialization problems of immutable objects is to use uniqueness (see Section 3.4.) A unique object has only a single reference to it and thus any mutation of the object is guaranteed to be unobservable via any other reference is the system. This is precisely the situation needed to initialize immutable objects. Thus uniques can be used to make the initialization phase arbitrarily long and not tied to a particular scope. Once the initialization phase is over the unique reference can be destroyed and turned into an immutable reference, now safely sharable. To the best of our knowledge, this trick was first used in SafeJava [18].

Creating intricate immutable object structures can be difficult, in particular circular structures. In order to create a circular immutable structure at least two objects must be in their initialization phase at the same time. Note that using uniques does not solve this problem, without some additional machinery. A common solution to this issue is to allow arbitrary mutation under some restricted conditions which guarantee that no mutable references can be leaked to the rest of the system. This way, once the initialization is done all but one reference can be made unreachable and the structure made immutable. Gordon et al. [56] do precisely this by allowing for instance a unique reference to be

converted to a regular mutable reference at which point arbitrary mutation is allowed. Once done, the reference can be turned into an immutable reference. The intuition is that if a context receives as input (e.g., formal function parameters) only immutable and unique references then any reference going out from the context (e.g., is returned from the function) must be either one of the arguments (unique or immutable) or a newly created object which is unaliased outside the context. Thus the reference can be safely turned into an immutable reference. Zibin et al. allow a similar situation, employing ownership to enforce that mutations are not visible to the outside [132]. Servetto et al. propose a novel strategy based on *placeholders*, a system for dealing with circular dependencies (not just in the context of immutable objects) [120]. Placeholders are used where circular dependencies exist and a three-phase evaluation scheme creates the objects and substitutes all the placeholders for references to the created objects.

Read-only references are references which cannot be used to change the state of an object. The difference between read-only and immutable may seem subtle at first, but immutability is a property of an object, or rather of all references to an object, whereas read-only is a property of a reference. A read-only reference does not necessarily preclude the coexistence of mutable references to the same object. Thus changes to an object may be observed via the read-only reference; Boyland calls this *observational exposure* [24], and hence to data races. An alternative name for read-only in the literature is *reference immutability*, e.g., in [56, 71]. This is really a misnomer. There is nothing immutable about reference immutability. Mutating an object via such a reference is prohibited, but nothing prevents mutations from being observed.

In Paper I we give a fairly extensive overview of work on read-only references and related concepts, and our recent survey on immutability [110] discusses the subject in more depth, also covering more recent proposals.

3.4 Uniqueness

Uniqueness is a powerful concept and intuitively an easy concept to understand: a variable annotated as *unique* is the only variable in the system pointing to a particular object, or it points to nothing. In other words, at any moment in

time, a unique object is pointed to by only one variable in the entire system[5]. A simple property indeed. However, uniqueness in an object-oriented setting requires careful design. Maintaining uniqueness calls for some kind of discipline on assignment. Destructive reads [69, 93, 38] nullify the right-hand side of an assignment, thereby moving a reference from one variable to another, as opposed to copying it as is the usual semantics. Another strategy is to swap the contents of the left- and right-hand sides, which also preserves uniqueness [63, 61]. Boyland proposes *alias burying* which allows multiple variables to point to a unique object as long as all but one are untouched.

The creation of unique objects must be also be considered. Intuitively creating a new object produces the only reference in the system to this object. However, this is true only as long as the object does not share itself during the initialization phase.

Another issue is the treatment of uniques during method calls. A method call on a unique receiver will break the uniqueness invariant (because of the `this` variable.) This is acceptable if the method does not consume `this`, meaning it does not store `this` in a field or send it as argument to another method. The rational being that the only other alias is on the calling stack frame and cannot be reached until the current method exits (this assumes that the method was called on a unique local variable.) For this approach to allow modular checking, it requires some sort of annotations in a class' interface (either class-wide or per method) signaling how a method treats `this` [93, 69, 22]. Such annotations reveal purely internal implementation details to the outside, breaking abstraction, and causing internal implementation changes to propagate to the outside.

Clarke and Wrigstad solved these problems with *external uniqueness* [38, 130]. External uniqueness exploits the ability of ownership types to determine the boundary of entire object aggregates and distinguish the inside of an aggregate from the outside. This enables arbitrary aliasing inside an aggregate while there may be only one unique reference to the aggregate from the outside. The intuition here is that since the aggregate is uniquely referenced any alias inside the aggregate can only be reached via the unique reference to the aggregate. The situation is depicted in Figure 3.6, and is for the most part exactly the same as in traditional ownership types; objects inside an ownership context may not be ref-

[5] As we shall see this is not actually the property maintained by most systems with uniqueness, it would be too strict, but effectively the property holds.

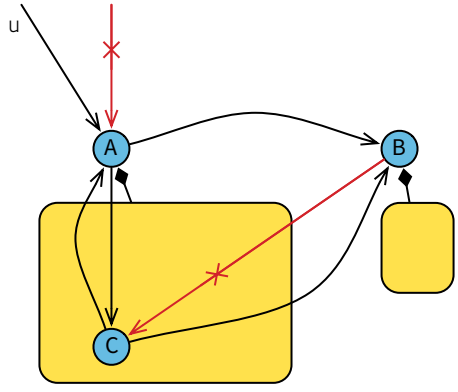

Figure 3.6. Permissible references in external uniqueness. Red, crossed-out references are not permitted. The in-degree of `A` is two, but only one incoming reference may be from outside of `A`. As usual in ownership types, references from outside a context to within is only permissible form the context's owner (`A` to `C` is allowed, but not `B` to `C`).

erenced from outside, but out-going references are allowed. The only difference is the reference marked "u" which is an externally unique reference to object `A` (which consequently forms an externally unique aggregate, including object `C`), disallowing any other external reference to `A`. Notably, the reference from object `C` to object `A` is allowed. External uniqueness makes it possible to move objects from one ownership context to another, which is not possible otherwise. Movement is only allowed inward in the ownership hierarchy, as moving outward could lead to a situation where owner parameters are ordered inside the owner of the object, which would break owner-as-dominators.

Method calls on unique receivers are dealt with by employing a borrowing construct, similar to many other systems (even though some call it lent, non-consumable, "!", etc., the concept is the same in all) [22, 11, 7, 21, 25, 69, 83, 93]. Borrowing is also used to send unique objects as method arguments without having to go through the trouble of linearizing their use by giving up the reference when calling the method and having the method return the reference when it is done. Intuitively borrowing means "for a limited time". A common restriction in all cited proposals is that "for a limited time" is maintained by confining borrowed references to formal parameters of methods and sometimes also to local variables on the stack, as allowing assignment to fields might result in residual aliasing. Borrowing in external uniqueness relies on ownership and a *fresh* owner to make sure that any aliases, regardless of whether they are

on the stack or on the heap, are destroyed (unreachable) once the scope of the borrowing ends.

Gordon et al. [56] propose a system with uniqueness similar to Haller and Odersky's *separate uniqueness* [61] (see Chapter 4), but allow references from inside a unique aggregate to point to immutable objects outside, which also makes it a restricted variant of external uniqueness. In this system unique references are allowed to be cast to regular unrestricted mutable references and then converted back to unique references, without explicit borrowing, in situations where residual aliasing cannot happen.

3.5 Effect Systems

Computations have effects such as reading and writing of certain data[6]. An effect system can typically infer the effects of a computation and given a set of allowable effects check whether the inferred effects are described by, or included in, the declared effects, and otherwise reject the program. In practice effect declarations are usually given on method headers and thus describe what data is manipulated when the method is called. So just by looking at the method headers of a class it may be possible to determine for instance whether two methods can be called in parallel without the risk of a data race. Effect systems are usually straightforward to understand, however, their soundness can be difficult to prove in a non-trivial language where aliasing is not controlled in some way.

The FX system [55] first introduced effects [54, 89]. FX is a higher-order functional language with reference cells (mutable state.) Later work introduced effect inference to lessen the syntactic burden on the programmer [76, 124].

Effects in an object-oriented setting were studied first by Greenhouse and Boyland [57] with a system often referred to OOFX. The motivation for this work was to help semantics preserving program transformations of Java code. In OOFX fields are put into regions, which are named abstract entities and the basis for the effect declarations. Regions are ordered in a hierarchical fashion, and therefore inclusion is straightforward when checking inferred effects against de-

[6] Other effects such as allocation, I/O and even domain-specific user-defined effects have been considered in the literature [55, 89, 126, 114]. Checked exceptions in Java is another example.

clared effects. OOFX was later extended and used to aid evolution of concurrent programs [58].

Clarke and Drossopoulou proposed a novel effect system based on ownership types in their Joe_1 language [41]. The strong encapsulation guarantees offered by ownership types apply not only to single objects, but to entire groups of objects. Thus using owners as basis for the effect system it is straightforward to identify disjoint aggregates which can safely be operated on in parallel. Effect inclusion is also simple in this setting as it is based on the ownership hierarchy which already provides clear relations on the owners.

Ownership and effects have been used together by other researchers as well. Yu, Potter and Xie [87] use effects to relax the aliasing constraints in ownership types. Objects may leak outside their owner, however how such references can be used is limited by an effect system. This is similar to Universes *owners-as-modifiers* discipline [49].

3.6 Regions

Regions have been used in a number of systems to give an abstract name to a group of fields and to hide implementation details as well as to deal with overriding of methods in subclasses. Leino identifies the problem of method overriding in his Data Groups paper [82]. Using field names in an effect clause has a huge impact on method overriding in subclasses. It is easy to realize that extending an effect clause in a subclass is unsound because subsumption can hide what data is actually touched by the method bound to in run-time. However, one of the benefits of subclasses is the ability not only to alter the behavior in terms of overriding methods, but also to add new state that these new methods can use. To tackle this problem Leino introduces Data groups (regions) to which fields belong. Data groups are then used in the effect clause of methods. Subclasses are allowed to extend data groups from a superclass with new field members, and so can touch new data without extending the effect clause. Thus the soundness issue is gone. Leino's proposal is different from many other region systems in that a field may belong to several data groups. Leino's data groups can be nested hierarchically by actively declaring one data group to be a member of another data group. This defines inclusion semantics on data groups, and thus effects.

Greenhouse and Boyland's OOFX system [57] uses regions in a similar way to Leino but it does not allow fields to belong to more than one region. The region hierarchy, and thus effect inclusion, is programmer-defined by declaring one region to be the parent of another.

Deterministic Parallel Java [16] (DPJ) introduces a novel way of nesting regions. In DPJ the main concern is computation on recursive data structures, and nesting of regions is interesting in particular when such computations happen. DPJ uses *region path lists* and region parameters on classes to specify the nesting of regions. This nesting forms a tree on the object graph and is what guarantees disjointness of effects and therefore deterministic parallelism. We describe DPJ in slightly more detail in Chapter 4.

❧

This chapter has surveyed techniques for alias management, including techniques for preventing aliasing, controlling the scope of aliasing, or controlling the effects of aliasing through restricted capabilities. The next chapter discusses how alias management has been applied to concurrent and parallel programming in the past, setting the stage for the in-depth discussion of our own contributions, described in Chapter 5.

4

Ownership and Concurrency

Ownership has been used in several concurrency-related applications, ranging from enabling locking of entire object aggregates to supporting thread-local data and data race-free parallel computations. In this chapter we overview applications of ownership to concurrent and parallel programming.

4.1 Ownership Systems

Parameterized Race Free Java [21] (PRFJ) is a type system based on ownership which guarantees race-freedom in concurrent Java programs. The basic idea is to let ownership boundaries, or contexts, be cut-off points where locks can be used to acquire an entire aggregate, i.e., everything within this ownership context. The discipline is called *owners-as-locks*, and is similar in some ways to work on *lock types* as in e.g., Flanagan and Abadi's Types for Safe Locking [53]. Later work added deadlock-freedom to PRFJ [19], but this requires the programmer to specify an ordering of the locks, which may be impractical.

Lu et al. propose a system for *structural lock correlation* [88]. Structural lock correlation provides a way to abstract locks to their depth in the ownership hierarchy, promoting modularity. Previous approaches would either be non-modular or require locking information in the interface of class, thus breaking abstraction. The system is similar to PRFJ, but uses effects along with ownership to track memory accesses as well as which locks are held while accessing memory. The effect system builds on that in Clarke and Drossopoulou's Joe_1 [41], which can describe effects in terms of depth in the ownership hierarchy. Lu et al. exploit this for lock abstraction by the simple fact that two effects on the same depth are either aliased, in which case their system guarantees mutual exclusion (by locking) or they are not aliased, in which case no action is necessary.

Universe Types for Race Safety [45] builds on the Universes ownership model to achieve a similar system to PRFJ, i.e., use a lock to protect all objects in a particular ownership context (though not transitively as in PRFJ.) Having Universes as the ownership model gives a much simpler type system which puts less burden on the user, but there is also a loss of expressiveness which is somewhat recovered by the introduction of an effect system.

Loci [131] uses a simple ownership-like type system to support thread local data. Based on the notion of *owners-as-threads* the heap is conceptually divided so as to give each thread its own memory area, as well as one heap shared between all threads. The type system guarantees that each thread has unique access rights to its own memory area, thus ensuring data race-freedom when accessing thread local data. Accesses to the shared heap must be managed.

Gruber and Boyer [59] propose a run-time approach to achieve isolation of actors. The approach is a run-time ownership model where objects may be free (not owned), owned by a message or owned by an actor. Ownership is tracked by adding an extra field to each object as well as read and write barriers to control isolation. In order to achieve reasonable run-time performance the system uses an unusual semantics for message sending, called *tail migration*, in which all messages sent during the processing of a message are buffered and sent when the processing has finished.

Deterministic Parallel Java [16] (DPJ) uses regions to express disjointness in parallel computations. Regions are declared in classes and fields of classes each live in one region. Classes may be parameterized over regions, and disjointness of region parameters may be required. A field in a class may have a type which

is a class which may declare its own regions and fields, defining nesting of regions. Thus the region hierarchy forms a tree and the object graph of a running program will as well (similar to ownership types.) Methods are annotated with effect clauses revealing which regions are touched by the method when called. A novel feature of DPJ is region path lists (RPLs) which are used to name regions. An RPL is a list of region names, expressing the nesting structure of regions, or wild cards, representing an unknown list of region names. RPLs are used to compute disjointness of effects in parallel computations. RPLs which share a common prefix and then deviate from each other denote disjoint regions, and thus effects on such RPLs can be allowed in parallel. While DPJ guarantees deterministic parallelism in certain cases (i.e., with primitive data elements) it does not support complex data elements with references to external objects. A solution to this problem was proposed [15] which relies on a (trusted) correctly implemented library which then can perform deterministic parallel computations even in the presence of references to external objects.

4.2 Capabilities and Permissions

Capabilities and permissions are ways to encode, track and describe or limit how a resource is, or may, be used in a particular context. Boyland's Fractional Permissions [23] builds on a simple idea that a permission can be either whole or a fraction. Fractions are created by splitting a whole in two, or by further splitting a fraction into smaller fractions ad infinitum. A whole permission means exclusive rights (uniqueness) and thus mutation can be safely allowed, whereas a split permission means that someone else also has a split permission and mutation cannot be allowed. Split permissions may later be returned to form the whole at which point mutation can again be allowed. A variant of fractional permissions is used by Westbrook et al. [129] who formalize an extension to HabaneroJava called HJp. In HJp data race-freedom of async/finish and array parallelism is established. In contrast to fractional permissions HJp allows writable fractions but they cannot be passed between tasks (a task is sequential, so thread-local aliasing is allowed.) HJp does not enforce strong encapsulation, but it can be achieved by storing exclusive permissions in fields making the object structure tree-shaped, similar to message objects in Kilim [122].

Reference capabilities [25, 32, 43, 56] equate or associate references with *capabilities* – an unforgeable token that governs the access to some resource. Reference capability systems are more ad hoc than ownership types systems in that they do not impose a specific structure on a program, and therefore do not provide any clear guarantees, like ownership types (owner-as-dominators) or Universes (owners-as-modifiers). Instead, they offer an increased level of flexibility which does not restrict programs in the same way.

Haller and Odersky [61] introduce a notion called *separate uniqueness* to allow efficient and safe sending of mutable state between actors in Scala. Separate uniqueness shares similarities with Hogg's Islands [69] and other proposals with full encapsulation [101]. References from inside a unique aggregate to the outside are not allowed. The rational for this design is that when sending a unique aggregate between actors, some additional constraints or machinery is required to ensure race-freedom if outgoing references are allowed. Separate uniqueness is maintained by well-formed construction; a separately unique object may be built out of other separately unique objects.

The work on Kappa by Castegren and Wrigstad [32] defines a complex hierarchy of reference capabilities where objects can be thread-local, unique or shared under some protection which may be internal (data race-freedom is guaranteed from measures taken at declaration-site, e.g., locks, the object is an actor, use of transactional memory or lock-free programming) or external (some form of synchronization employed at call-site.) Kappa capabilities include a *subordinate* capability which gives a shallow form of nesting, similar to that of Loci [131] or [39] which is used to guarantee that all accesses to certain objects are protected by some other capability in the system, e.g., in order to name some object, a lock must already have been taken somewhere, which excludes races on this object. With the exception of the subordinate capability, Kappa does not restrict the visibility of aliases except across multiple threads of control. Kappa allows splitting a capability either into smaller, non-overlapping, mutable sub-capabilities or into overlapping read-only sub-capabilities.

Clebsch et al. [43] use *deny capabilities* to, as opposed to describe what a particular reference may be used for, describe what all other aliases cannot be used for. Deny capabilities are employed in the context of Pony [109], an active object language, to achieve safe efficient sharing. References are equipped with local and global capabilities signifying what local and global aliases may do to the object

(global meaning shared, as in shared between active objects.) A reference that is mutable has the global capability that no read and no write alias may exist, thus banning sharing altogether. In order to to send a mutable object between active objects a reference with global capabilities no read and no write and local capabilities no read and no write (it is unique) must be used. To be concrete, deny capabilities in Pony ensure data race-freedom by only allowing references to immutable, unique and active objects to be sent between active objects. A write-by-one/read-by-many capability is also available but not sharable across active objects.

Mozilla's Rust language [113] uses capabilities to avoid garbage collection but is mostly interesting for its inclusion of many concepts from the area of ownership, uniqueness, and borrowing which results in a language with statically guaranteed data race-freedom built in. In Rust, values mediate between linear–mutable and sharable–immutable, similar to Fractional Permissions [23]. Linear values use a scheme similar to Boyland's alias burying [22], and Rust uses borrowing to simplify programming. Rust does not have strong encapsulation. This means that there is no notion of inside/outside of an object as with ownership types, there is no single entry point and an object's internals can leak. Avoiding the latter is possible only at the cost of preventing aliasing inside the data structure, similar to the restrictions on Kilim messages [122].

❧

This chapter has overviewed ownership- and capability-based applications for concurrency control, including data race-free parallel computations and safe sending of mutable data. In the next chapter we discuss our own contributions, starting with a short prehistory and an in-a-nutshell introduction of Joelle, our programming language. Following that, we present the defining features of Joelle in more detail.

5

Joelle: Isolated Safe Parallel Actors

Our initial foray into the area of alias management was not motivated by facilitating concurrent and parallel programming, but for safety and program understanding in a sequential setting. The advent of multi-core, however, along with some key discoveries in our work on the research language Joe_3 [108], led us to apply our techniques in a concurrent setting. It largely inspired the work on Joëlle [39] and in abandoning the thread-model, several simplifications were made possible that caused many of Joe_3's features – most notably its modes, which will be explained shortly – to be streamlined. Less syntactically heavy, at the cost of expressivity.

Joe_3 introduced a novel ownership system where owner parameters are decorated with *modes* governing how objects owned by a particular owner may be used in the current scope. This allows very flexible patterns, e.g., a data structure could be writable in some contexts and read-only in others. A mutable reference, provided that is is unique, can be turned into an arbitrary number of immutable references and then reinstated as a mutable reference at a later time, encoding something akin to Boyland's fractional permissions [23]. Joe_3

also supported read-only lists containing mutable elements, which at the time no other system could do.

Unfortunately read-only references suffer from observational exposure [24]: While a read-only reference does not allow mutation, it does not exclude the coexistence of references that may mutate the object, and subsequently may be used to witness such mutation. This problem is exacerbated in a parallel or concurrent setting where observational exposure may lead to a data race. This prompted the replacement of read-only references with a variant where mutable objects may be shared but only immutable parts of such objects may be accessed concurrently.

Other parts of this work have proven useful in the concurrent setting. Immutable objects can be shared freely, but creating them may be difficult if they have an involved construction phase. Joe_3 solved this problem by allowing unique mutable objects to be turned into immutable objects. This approach follows prior work by Boyapati [18], but Joe_3 allows programmers to express that something is immutable and not just not read-only through the current reference.

5.1 Joelle in a Nutshell

The goal of this thesis work has been to develop an active object system with isolation and safe sharing. The rational for this design is the apparent compatibility between object-orientation and active objects [78, 98]. With active objects, transforming an existing sequential application into a parallel equivalent becomes an exercise in finding suitable cutoff points where different parts of the system can be packaged into discrete entities that do not share mutable state, communicating via asynchronous method calls. The graph-like nature of object-oriented programming, and the tendency to share mutable state may cause the partitioning of an object-oriented application in such a fashion to only expose a very limited parallelism, e.g., only create a handful of parallel parts. Not only does this not scale but active objects that are on the critical path of many others risk becoming a bottle neck of an application if their messages must not only be handled sequentially, but also each message processed in a single thread of control. Indeed, Henrio et al. suggest that programming with sequential active objects efficiently on multi-cores is impossible [66]. To this end we have extended the features that provide isolation (see Paper II) following earlier work

by Clarke and Drossopoulou [41] to allow data race-free parallelism *within* active objects. In addition to providing support for executing multiple messages in parallel, we also construct ownership-supported means to implement the processing of a single message using parallel programming, also in a data race-free manner (see also Paper V.) Section 5.3 covers these things in detail.

Much of the language features described in this thesis have been developed in an exploratory way where an idea has been implemented and tried out in our compiler before being refined and published. The focus of our work has been on the front-end and even though the compiler outputs runnable code, no serious attempt at optimizing the code and get good performance has been made. Compiling our language to an existing active object framework should be straightforward as most of our features reject unacceptable programs statically and would require little if any change to the base implementation.

This chapter presents applications or extensions of the programming constructs found in Papers I, II, IV, V in a unified non-toy (Joelle is a clean extension of Java) programming language called Joelle[1]. Joelle is an active object-based object-oriented programming language[2]. An active object encapsulates its state and methods but also the thread that executes its methods. In normal object-oriented programs the execution of a method is performed by the calling thread, entering the target object, executing the statements of the method and then returning back to where the call was made. With active objects the calling thread instead asks the receiving object to execute the method using its own thread, this is what we call asynchronous method calls. Thus a thread never leaves its active object to enter another one. The borders of active objects are respected by threads and reasoning about an active object in isolation becomes possible.

Inside active objects there can be passive objects. Method calls on passive objects are synchronous, i.e., the calling thread performs the execution of the called method. Therefore the active object model depends on the non-sharing of (mutable) passive objects between active objects. With sharing of mutable state the borders of active objects are breached, allowing threads to enter and cause data races. This is recognized in the actor model but most actor implemen-

[1] Historical note: the observant reader will notice that the language in Paper II is called Joëlle. This was the unfortunate effect of a joke played by one of the co-authors (I'm sure some readers can figure out which one) who forgot to remove it before the paper was submitted. Now, one might argue that it was a good thing because while the language we now call Joelle builds on Joëlle they are not the samë.

[2] The implementation is based on Java, but in principle it could be C# or any other similar language.

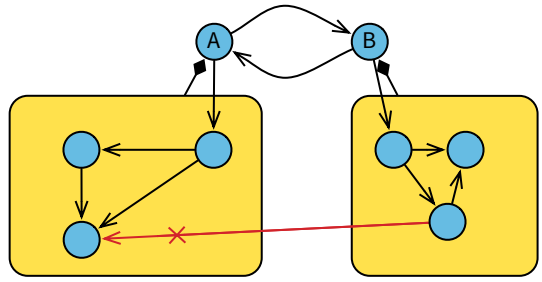

Figure 5.1. Active objects (`A` and `B`) encapsulating passive objects. Each active object has a discrete subheap which does not allow incoming alias from outside, modulo from its owner.

tations only ask the programmer to avoid such sharing, they do not enforce it. We overcome this problem by employing ownership types to build a system that statically guarantees that mutable data is not shared between active objects in a way that could cause data races.

In Joelle, each active object forms an isolated aggregate where passive objects live (Paper II.) In ownership parlance we can say that passive objects are owned by active objects. Thus, by virtue of ownership types, a passive object cannot leak outside the active object so there can be no sharing of mutable state. Figure 5.1 shows two active objects `A` and `B` and their respective contexts (the rounded rectangles) hosting passive objects (unlabeled circles) pointing to each other (arrows).

To share objects through messages in a data race-free manner, there are a number of options to consider. We could copy any data that is sent from one active object to another. This is certainly a viable option and is used in ProActive [30], for instance. If each active object has its own copy of the data there is no sharing and therefore no data races. The problem with copying data is that it is potentially very expensive [77]. There is a cost incurred by traversal of the object graph, allocation and initialization of objects. If big data structures are sent back and forth performance is going to suffer, and even with small objects, if there are a lot of them being sent, a similar situation occurs.

Another option is to only allow immutable data to be sent. Erlang does this and indeed it solves the problem[3]. Immutable data, by definition, cannot change so data races cannot occur. The problem with immutable data is precisely that it cannot change. A computation involving immutable data will create new data,

[3] Erlang also copies the data, mainly for distribution and fault-tolerance reasons [8].

or at least convert immutable data to a mutable version. Suffice it to say that in many cases the situation is effectively not very different from copying mutable data when it is sent.

As concluded earlier the reason not to allow sending of mutable data between active objects is that it may cause data races. But if the sending active object gives up the right to access the data when it is sent, data from one object can be transferred to another and then there would be no issue. This is realizable if one can guarantee that after the data has been sent there is no alias to that data from the sending object, which in turn can be achieved if one can guarantee that there is only one alias to the data we want to send. When the data is sent the sender's alias can be destroyed and the data is not shared between active objects. A variant of this approach is taken in Kilim, using a special kind of message data that is only allowed to point to other message objects in a tree-shaped fashion. Data structures internal to an active object which are not tree-shaped cannot be passed around, or must be transformed into a tree-shaped equivalent, and then transformed again by the receiver.

Joelle supports all means of data race-free transferring above (and a little more.) The programmer is free to choose whichever fits the best in any given situation.

5.2 Safe Sharing

Joelle's type system prevents passing references which may lead to data races as arguments to an asynchronous method call.

Immutable objects cannot change and thus cannot be involved in a data race. Therefore, there is no restriction on the sharing of immutable objects. In fact, from an ownership types perspective, immutable objects can be considered to live in a common part of the heap to which all objects have access (Paper II.) Our implementation of Joelle allows adding a *movement bound* to the type of an immutable object to prevent its exposure. The movement bound specifies the outermost context into which the object can escape, making them subject to the same sharing restrictions as regular objects. Figure 5.2 depicts the situation. Green circles denote immutable objects, and the nesting of an immutable object inside an active object's context denotes its isolation. Movement bounds are also used on unique references [38] where they are required for the soundness of

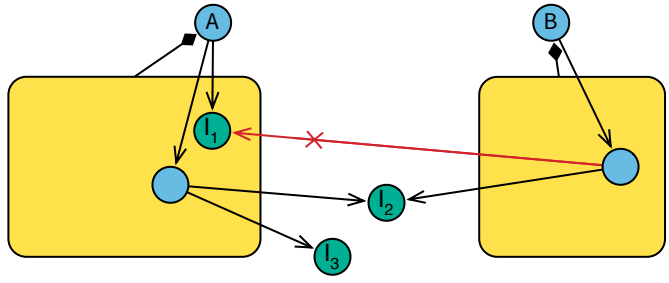

Figure 5.2. Two active objects (`A` and `B`) sharing an immutable object I_2. The immutable object I_1 is encapsulated inside `A` and cannot be pointed to externally. The immutable object I_3 is currently only referenced by `A`, but is free to share, just like I_2.

the type system. With immutable objects, movement bounds instead become an optional way to express and enforce programmer intent[4]. The theory is the same as for External Uniqueness [38].

Joelle also supports a variant on immutable data, called *safe* (Paper II.) A safe reference can only be used to access immutable data of its referenced object. So an object may very well contain mutable state, but this cannot be observed via a safe reference. This allows sharing of objects containing mutable state without copying them and without imposing on the programmer to split one object into several objects just to avoid copying.

Unique objects, by definition, can not be shared, as they may not be aliased. This property is exactly what makes them useful with active objects. Sending mutable data between active objects is only a problem if it can cause a data race, and a data race can only occur if the sender keeps an alias to the data it sends[5]. If the sender gives up the only existing reference as it sends the data then there is no residual aliasing after the send. We employ uniqueness in Joelle to achieve zero-copy sends of mutable data (Paper II.)

A passive object which is not immutable, not unique and which it does not make sense to share as safe, must be copied when sent as argument to an asynchronous method. Deep cloning an entire object graph is not always required though. Any fields pointing to immutable or safe data or to an active object can be aliased, so in these cases only the reference must be copied. This is an

[4] Controlling sharing of immutable objects may have other benefits as well. For instance, the standard Erlang implementation copies immutable message arguments to support parallel garbage collection.

[5] We do not consider pointer arithmetic and forging of references as we assume a type safe language.

instance of *sheep cloning* [100] which has recently been studied and formalized by Li [84].

The alias management-savvy reader may recognize the similarities between these sharing restrictions and Flexible Alias Protection [101]. Indeed they are similar. In Flexible Alias Protection *rep* is used to denote representation objects which cannot escape their owner, in Joelle these are all passive objects which do not fall into one of the other categories. The *arg* mode is the same as safe in our terminology, what we call unique is similar to *free*, what we call immutable is called *val* in Flexible Alias Protection nomenclature. There is a fifth mode in Flexible Alias Protection called *var* which is unrestricted in where it can escape and how it may change, except it cannot be used in a way that breaks the invariant of any other mode, e.g., be assigned to a val variable. Joelle does not support *var* references because it would break isolation of active objects and introduce data races. Paper I and II together are the first to offer a *complete formalization* of the flexible alias protection reference modes, modulo its unsafe *var* reference. To the best of our knowledge, and with the same caveat, Joelle is also the first language to offer a *complete implementation* of the same reference modes.

5.3 Hyper-Active Objects

In most actor/active object systems, messages are picked one at a time from the mailbox queue and processed to completion before a new message is picked and processed. This gives sequential execution within each active object and concurrent execution between active objects. A potential problem with this style of concurrency on multi-cores is that the amount of parallelism one can achieve is a function of the number of actors in the system. This may lead to awkward designs where distributing a problem over several actors becomes necessary in order to utilize the available hardware. Such a solution either must introduce some form of sharing (and take care not to introduce data races in the process), or pass state around which can be very inefficient, even without copying, and lead to high latency, and possibly dwarf any performance gains in the computation [66, 125]. Moreover, asynchronous communication may require complex communication protocols to be implemented for correctness [125]. In such cases allowing parallelism inside active objects would be highly beneficial. Imam and Sarkar make a similar argument concerning the pipeline pattern:

> *[…] the pipeline pattern is a natural fit with the [actor model] since each stage can be represented as an actor. The single-message-processing rule ensures that each stage/actor processes one message at a time before handing it off to the next actor in the pipeline. However, the amount of concurrency (parallelism) in a full pipeline is limited by the number of stages. One way to increase the available parallelism, apart from creating more stages, is to introduce parallelism within the stage […]. The slowest stage is usually the bottleneck in a pipeline, increasing the parallelism can help speed up the slowest stage in the pipeline and improve performance.* [73]

The problem with allowing parallelism within an active object with mutable state is that it threatens to reintroduce all the problems of concurrency that the actor model sets out to do away with. However, if one can guarantee that parallel computations inside the active object will not touch the same data (modulo read–read accesses), then no data races can occur, and parallelism can be allowed without any observable difference, except a possible speedup.

Luckily, borrowing from work by Clarke and Drossopoulou [41], we can capitalize on the ownership and effect information already present in Joelle to determine whether two expressions interfere [112] to enable parallelism that is statically checked for data race-freedom. Since every type of a field or variable carries ownership information, it describes conservatively the locations on the heap that can be affected by operating on it. Thus, every expression can be statically associated with a similar "footprint" which can be turned into abstract sets of objects. If the intersection between the sets of objects that two expressions can modify is empty, the expressions do not overlap. If the intersection only includes read effects, there is overlap, but in a benign way. In Section 5.4, we will discuss effects and disjointness in more detail. Until then, we will work with the intuition behind these concepts.

In the code snippet below we show Joelle's `async` statement, in the spirit of X10 [34]. The `async` statement may execute the statements in its body in parallel with other `async`s (and the current thread) and the `finish` block will wait until all parallel `async`s inside its body have terminated before allowing execution of the current method to continue. The effect checker will reject the program if any of the parallel `async`s touch overlapping data in a way that could lead to a data race. The exact meaning of names `A` and `B` is not crucial for this example, suffice it to say they denote disjoint partitions of memory, the details are explained in Section 5.4.

```
1  void m1() writes(A) { ... }
2  void m2() writes(B) { ... }
3
4  void m3() writes(A, B) {
5    finish {
6      async { m1(); }
7      async { m2(); }
8    }
9  }
```

From this code, method `m1` writes data in memory partition `A` and method `m2` writes data in memory partition `B`. If `A` and `B` do not overlap, it is clear that these messages can run in parallel. The theory behind ownership-based effects is straightforward, but there are two problems with classical ownership types which complicate matters when the example is a bit more involved. The first problem is due to the inherent coarse-grained nature of ownership types and is discussed in Section 5.3.1. The second problem is due to the reliance of nesting depth to express disjointness, and is discussed in Section 5.4. In Section 5.3.2 we finally show how parallel processing of incoming messages is supported.

5.3.1 Internal Fork/Join Parallelism

The code example in the previous section is very simple and often this style of expressing parallelism is used to perform parallel computations on recursive data structures, such as trees. Unfortunately ownership falls short here as it cannot express disjointness in such a data structure in a satisfactory way. A tree-shaped data structure is inherently safe to traverse in parallel provided that the computation actually traverses it as a tree, i.e., there are no accesses of references between the nodes that would violate the tree shape. Additionally, for safe parallel computation on the elements of the tree, each element in the tree must be different, and this is also true for anything that the computation touches in the elements' transitive state. Figure 5.3 shows the necessary structure of the traversal. It is clear that all elements are disjoint from each other and so are objects reachable from the elements.

Ownership types inherently guarantees disjointness in tree-shaped structures, but requires that each node in the tree be the owner of its subtrees. The down-

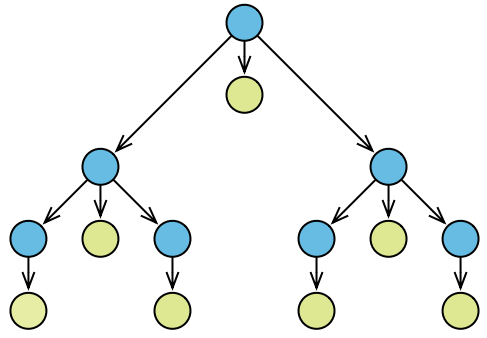

Figure 5.3. Tree-shaped data structure. Blue nodes are data structure nodes – the spine. The green nodes are elements in the data structure. Extending the data structure with the green nodes should preserve the tree-shape for deterministic parallelism.

side of this requirement is that nodes in the tree cannot be moved around for balancing or removal, as this would require a node to change type, which would not be sound. An alternative solution is to have the nodes all be owned by the same owner but make each node unique. This would allow moving nodes around, however it would preclude auxiliary references between nodes, like a "parent" reference (Paper V.) The story for elements of the tree is similarly disappointing. To be able to guarantee race-freedom, each element in the tree must be different. This is easily achieved by making the node the owner of the element. However, that makes elements part of the data structure's representation and a user of the data structure cannot add or retrieve elements from it; a pretty useless collection. The uniqueness solution can be applied to elements as well, but again uniqueness is too strong because it does not allow, for instance, putting the same element into two collections with different orderings.

It turns out that restricting the shape of the object graph in the data structure is not necessary to guarantee race-freedom. It is the shape of the access path of the computation that matters. If the computation accesses the data structure in a tree-shaped way, then race-freedom can be guaranteed, provided that the elements of the structure are all different.

Our work on *refined ownership* (see Paper V) addresses precisely these issues. Traditionally in ownership types parameterizing a class over an owner `p` gives permission to access all objects owned by `p`. In refined ownership we view an owner as a set of access permissions to access individual objects owned by that owner. To continue our tree example from before, let `p` be the parameter that grants permission to our tree to access the elements we wish to store in it. The

tree can then split p into three disjoint sets – *refined owners* – each allowing unique access to a disjoint group of objects owned by p. The tree would pass one set to its left subtree, one to its right subtree and keep one to type the element to store in the current node. Each node of the tree carves off a piece of the set given to it and passes the rest on. Because permissions to access objects are propagated through the spine in a tree-like fashion, all the access paths to these permissions are tree-shaped, but this does not preclude other arbitrary object structures to coexist. The fact that a data structure is tree-shaped guarantees that no two paths through the data structure leads to the same element pointer. However, we must also make sure that no two element pointers point to the same element. There are a few ways to achieve this where the easiest (and most efficient) is to only allow extending the tree with unique elements, which may subsequently be aliased freely outside the data structure.

We have now sketched (see Paper V for details) how tree-shaped structures can be built and elements can be stored in them. However, common operations such as sorting requires the ability to manipulate structures. At first glance the problem appears to be just as difficult as in classical ownership: each node is typed using a different owner and moving a node would require it to change type. To support movement we allow a program to mediate between different views of a data structure. Introducing a "supercharged version" of Deline and Fähndrich's *focus* construct [52] we allow accessing the tree in such a way that only the tree-shaped spine is accessible. This yields a much simpler type which is temporarily linear. Linear manipulation of a tree-shape yields another tree-shape, meaning that after a modification, we can suspend the focus operation and regain access to the tree's original, more expressive types.

5.3.2 Parallel Message Processing

We have seen how ownership and effects can be used to safely split computations up into parallel tasks inside an active object. We will now explore asynchronous method calls and processing them in parallel using similar techniques.

Asynchronous method calls can be seen as requests by one part to another to run a particular method at a convenient time and then report back with the result when the computation is finished (if indeed the method returns a result.) As mentioned earlier most actor implementations perform each request in se-

quence and only start a new one once the previous has finished. Again, this is because running parallel requests risks causing data races; the problem is essentially the same problem as previously discussed regarding `async`s. Conceptually only one message may be processed at a time, but in reality any number of messages can be processed in parallel as long as it is not observable in the result of the computation[6]. The effect system built on ownership, again, makes it straightforward to keep track of what data a message uses (reads and writes); any two messages that do not conflict by touching overlapping data may be run in parallel. If `A` and `B` below do not overlap, `m1` and `m2` could run in parallel.

```
1  active class C {
2    ...
3    void m1() writes(A) { ... }
4    void m2() writes(B) { ... }
5  }
```

There are two kinds of internal parallelism: speeding up the execution of a single message (e.g., `async/finish`) and supporting parallel processing of incoming requests. There is a difference in nature between the two. An async/finish block appears within an active object and thus in a (observably) sequential setting where any data touched is "owned" (for the time being) by the currently executing method. This means that splitting the computation with an async/finish block is an atomic operation which cannot be interfered with by anyone else.

On the contrary calling two asynchronous methods on an active object is not an atomic action. Active objects may be shared with any number of other active objects who could be calling methods concurrently. Therefore two asynchronous method calls performed right after each other in a sequential piece of code cannot be parallelized at the call site. The decision whether asynchronous methods can be run in parallel is a concern of the receiving active object.

The analysis of which messages may be run in parallel can be done in compile time. However, the organization of messages in the mailbox and selection of which actual messages can be run at any given moment is likely to impact performance. Run-time performance is not our main focus in this work, rather the goal is to find suitable abstractions to enable the scheduler to use as much of the available resources as possible. That being said, some effort has been made

[6] A common misunderstanding, it seems, regarding the actor model is that it would demand sequential processing within actors. This is clearly not the case. See e.g., [3] for a discussion (on replacement.)

to implement a reasonable run-time. We discuss our back-end implementation in Section 5.7.

Imam and Sarkar propose a unification of the actor model and async/finish model in the context of HabaneroJava (and HabaneroScala) [73]. The system is similar to ours but does not guarantee data race-freedom and therefore no determinism guarantee for fork/join computations (although this has been addressed in other work [129].) Parallel message processing is supported by letting asyncs outlive the scope where they are created. This way the actor can pick a new message from its queue and start processing it in parallel with the previous one. However, this is only allowable for stateless actors. The following example taken from their paper shows the concept:

```
1  / * ** Habanero-Scala code ** * /
2  class StatelessActor() extends ParallelActor {
3    override def behavior() = {
4      case msg : SomeMessage =>
5      async { processMessage(msg) }
6      if (enoughMessagesProcessed) { exit() }
7      // return immediately to process next message
8  } }
```

Henrio et al. propose a *multi-active object* system where active objects may process messages in parallel [66]. A programmer may define groups to which methods belong. Two groups may be declared *compatible* (a group can also be *self-compatible)*, signifying that their respective methods can be run safely in parallel. In order to increase the parallelism possible in the system decisions to schedule requests in parallel may be done in run-time depending on the current state, e.g., it might make sense to run two requests for the same method in parallel if their arguments are not equal, according to some definition of equality. An operational semantics for the multi-active object system is presented and it has been implemented on top of ProActive. Henrio et al. present benchmark results showing that parallelizing active objects is indeed worthwhile [65]. The parallel active object system proposed by Henrio et al. is very similar to ours, but does not guarantee freedom of data races, although the annotation scheme seems intuitive and likely helpful to programmers. Also, in contrast to our system conflicting methods can be allowed to run in parallel if shared state is protected by e.g., a lock. Our system would need additional machinery to allow this, e.g.,

incorporating (parts of) Castegren and Wrigstad's capabilities [32], though the situation is somewhat improved by the introduction of *yields* (see Section 5.5.)

Parallel actor monitors (PAMs) by Scholliers et al. [119] is a system for parallel message processing built on top of AmbientTalk [128]. PAMs are strategies which can be plugged into an active object to achieve different schedulings of incoming messages. Messages are categorized as being readers or writers and this can be used by the scheduler to e.g., allow many read messages to be processed in parallel but only single write messages. Basing parallel scheduling on whether a message is a reader or a writer is fairly coarse-grained as it does not discriminate between different parts of the active object state, however it allows for generic scheduling strategies to be implemented in a library.

5.4 Disjointness, Regions, and Effects

Clarke and Drossopoulou studied how ownership types and effects go together and how disjointness of effects is described by the types of objects [41]. In Java and other languages, the type system will ensure that variables with types without subtype relation cannot be aliases. Ownership types give the same guarantee, but also that variables with types that are subtypes (in terms of inheritance) cannot alias if they contain provably different owners. This fact about aliasing along with the strong encapsulation of ownership types means that it is possible to identify entire disjoint aggregates of objects. If two variables are not aliases any access to the aggregates pointed to by these variables will be safe to do in parallel. However, we have previously shown [106] that while this is theoretically sound, the necessary conditions rarely occur in practice.

Consider for example the following two variables, `v1` and `v2`:

```
1    p:C v1;
2    q:C v2;
```

In the code above, the types[7] `p:C` and `q:C` tell us that variables `v1` and `v2` cannot alias if we can prove that owners `p` and `q` are not aliases for the same owner.

[7] We write the owner before the class type, separated by a colon. Any additional actual owners are written in angle brackets e.g., `p:Foo<a,b>`, where `p` is the owner and `a` and `b` are additional parameters required to form the type.

Owners p and q usually come from owner parameters in the class header as demonstrated by the following code example:

```
1  class D<p, q> {
2    p:C v1;
3    q:C v2;
4  }
```

What this tells us is that p and q are both nested outside **owner**; this is required by ownership types for soundness reasons. Unfortunately this information is not enough to establish whether v1 and v2 can alias, as p and q could be aliases for the same owner. What is known in the situation above is the following:

1. p and q are outside **owner**
2. **world** is outside **owner**
3. **owner** is outside **rep**

From these facts (and that rep is disjoint from all other owners by definition) we can establish the following:

1. **rep** is disjoint from p, q, **world** and **owner**
2. p, q, **world** and **owner** can be aliases

Code analysis of ownership types programs suggest this situation is very common – it is not possible to establish whether two variables can alias given the available information. To improve on this situation one could place requirements on the relations of p and q. One could say for instance:

```
1  class D<p, q> where p disjoint q {
2    p:C v1;
3    q:C v2;
4  }
```

With the disjointness[8] requirement we say that p and q are not only outside **owner** but also that they are not aliases for the same owner. With this information it is possible to prove that variables v1 and v2 are not aliases.

[8] In the literature a *strictly outside* (non-reflexive relation) requirement has also been used to convey disjointness [41, 108].

```
1   class C {
2     // r1 and r2 are partitions of rep
3     region r1, r2;
4     r1:Object v1 in r1;
5     r2:Object v2 in r2;
6   }
```

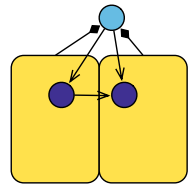

Figure 5.4. Regions partition an object's representation. Instances of `C` have their representation split into two discrete "subcontexts". Aliasing across regions is allowed.

5.4.1 Regions

A significant downside of introducing disjointness requirements on owner parameters is that proving disjointness becomes a concern at each use-site of the owners involved, regardless of whether or not such disjointness is relevant, or even desirable. This *local* concern, hence propagates *non-locally*, breaking abstraction and complicating refactoring. Furthermore, relying on external owners to express disjointness sacrifices encapsulation. It requires at least one object to be owned by an external owner, which in turn allows that object to leak. Consequently the local information is not strong enough to safely allow parallelism. However, if one could introduce disjoint owners where they are needed both the problem of propagation and the problem of encapsulation would be avoided. To this end, Joelle allows an object's representation to be partitioned into multiple disjoint *regions*, which are interpreted in ownership types as *subcontexts* of the enclosing object's representation. Figure 5.4 shows syntax and an interpretation in the form of an ownership diagram.

Regions are named entities which can be declared by the programmer. A region declaration introduces a new owner with the same name into the scope. The region owner can be used like any other owner to form types. Fields can also be declared to belong to a specific region, using the `in` keyword, otherwise they belong to the default region `rep`. Putting fields into regions can be a great way to convey design intent, clearly showing which fields belong together. Regions also make overriding methods in subclasses sound, which otherwise is a tricky matter [82]. Whenever a field is accessed or updated the effect is respectively a read or a write on the region to which the field belongs. Region owners are disjoint by definition, and thus regions allow the introduction of disjoint owners without creating new objects. We call this horizontal disjointness, as a comple-

ment to vertical[9] disjointness in classical ownership types. Revisiting the example from before we can now use disjointness restrictions on owner parameters but avoid propagation:

```
1  class D<p, q> where p disjoint q { // any disjointness
2    p:C v1;
3    q:C v2;
4  }
5
6  class E {
7    region r1, r2; // horizontally disjoint
8    r1:D<r1, r2> f; // bind r1 to p and r2 to q
9  }
```

5.4.2 Concrete and Abstract Effects

Joelle's effect system, like Clarke and Drossopoulou's, uses owners to express effects. Using the underlying ownership system has several benefits. One benefit is that polymorphic effects [89, 114, 126] are supported by default using owner-polymorphic methods:

```
1  class UniversalMutator {
2    <x> void mutateArg(x:Arg a) writes(x) {
3      a.mutate(); // mutate() reports effect on owner
4    }
5  }
```

Method `mutateArg` is parameterized over owner `x` which is used in the **`writes`** clause to express the effect of the method body. This makes methods effect-polymorphic in one dimension. Supporting polymorphism in the read/write dimension would require additional machinery.

With regions added to the ownership system disjointness becomes much easier to exploit, however, there are issues with having regions as effects if region information is private to an object because disjointness information is not available externally – at the *use-sites* of the object. The example on the following page illustrates this with a class `DoubleCounter`, with region-based disjointness, used by a class `Client`.

[9] Vertical, because of the ownership hierarchy.

```
1   class DoubleCounter {
2     region r1, r2;
3     int c1 in r1;
4     int c2 in r2;
5
6     void increment1() writes(r1) { c1 = ...; }
7     void increment2() writes(r2) { c2 = ...; }
8     void incrementBoth() writes(r1, r2) {
9       finish {
10        async { increment1(); } // writes(r1)
11        async { increment2(); } // writes(r2)
12      }
13    }
14  }
15
16  class Client<p> {
17    void m() writes(p) {
18      p:DoubleCounter v = new p:DoubleCounter();
19      finish { // Compiler will reject
20        async { v.increment1(); } // writes(p)
21        async { v.increment2(); } // writes(p)
22      }
23    }
24  }
```

Methods `increment1` and `increment2` clearly operate on disjoint data and can be run in parallel, which method `incrementBoth` does. However, owners `r1` and `r2` are only visible in the scope of class `DoubleCounter`. An outside user of this class only knows about the owner of the enclosing object. Therefore effects need to be subsumed to something that is known to the outside. In this case `r1` and `r2` can be subsumed to **`rep`**, and then **`rep`** can be subsumed to **`owner`** which in turn is called `p` in the scope of class `Client`. Therefore a user cannot see that `increment1` and `increment2` have disjoint effects, and therefore is not allowed to call these methods in parallel. Somehow a class needs to export to the outside which methods interfere and which do not.

Joelle introduces a novel notion of abstract effects. These are names which hide concrete effects–reads and writes to actual owners. Abstract effects (lines 3–6

below) are programmer defined and specify *public* effect names as well as the *private* meaning of the effects, as shown in the following example.

```
1  class C {
2    region r1, r2; // private
3    effect A is writes(rep); // names A--D are public,
4    effect B is writes(r1); // their respective concrete
5    effect C is writes(r2); // effects are not
6    effect D is B, C;
7
8    disjoint B, C; // public
9  }
```

The disjointness declaration on line 8 is public. It serves two purposes: first, it tells the compiler the programmer intends for these effects to be disjoint. The compiler will then check that this is indeed the case. Second, it allows disjointness information to be exported in the public interface of a class to be used at external call-sites. Thus a user of the class can be made aware of which methods may be called in parallel. Note, however, that the reason for disjointness is not exported. In the example above the outside will know that effects `B` and `C` are disjoint, but it will not know why they are disjoint. Consequently changing the reasons for disjointness is purely a local concern which does not propagate to users of the class.

Hiding the reasons for disjointness also has other benefits. In particular it presents an opportunity for implementing a kind of gradual effect system in which the compiler can be set to a mode where it allows ad hoc disjointness declarations to be made by the programmer without requiring proof of their correctness. Effect declarations may be *fully abstract* for this reason:

```
1  class C {
2    effect A; // fully abstract effects, not
3    effect B; // mapped to concrete effects
4    disjoint A, B; // programmer assertion
5  }
```

In a development stage it may be useful for a programmer to sketch the design without having all the facts and figures in order. The compiler will check what it can and report warnings or errors where the program is inconsistent. This avoids false positives, but also gives no guarantees that there will not be data

races at run-time. Later in the development process when the design has stabilized the compiler can be run in normal (no false negatives) mode in which it requires the programmer to back up any disjointness claims on abstract effects by mapping them to disjoint concrete effects. At this point race freedom is guaranteed for any program accepted by the compiler. Our intention with a "gradual" effect system is not to be sound unless all effects can be concretized and checked – indeed at such an early point in development, where parts of the system are likely a mock-up, or incomplete, it makes sense to focus on the parallelism the programmer intends to support, rather than being overly conservative about its correctness. Although we could, we do not use a dynamic race detector, where effect disjointness cannot be statically determined, or resort to other kinds of dynamic checks, like Bañados et al. [13]. We see gradual effects as a tool to express intent in the early development process. Later work on gradual effects by Toro and Tanter, however, introduces *customizable effects* where a domain-specific language of effects (and their relations) allows for arbitrary effects to be defined by the programmer (e.g., I/O) [126].

Abstract effects and checking are implemented in the compiler. The more speculative gradual effect checking remains to be implemented. We have also not yet completely formalized or proven the soundness of the gradual effect system.

5.5 Unlocking Effects

Using effects for data race-free parallelism is conceptually very similar to using locks. Annotating a method with an effect clause effectively locks the data described by the effects during the execution of that method, preventing any other conflicting method execution. There are two major differences, however, one favoring effects and one favoring locks. First, with an effect clause, all the necessary "locks" can be acquired by a single atomic operation, whereas acquiring several actual locks is done in sequence (non-atomic), which is slower and also may introduce deadlocks. Secondly, effects are held for an entire computation (a method typically), whereas locks can be released as soon as they are no longer needed; consequently locks may allow for more parallelism.

To remedy this shortcoming we introduce a *yield* command. A yield can be used inside a method body to inform the scheduler that an effect reported in the method's head will not be caused by the remainder of that method. This allows

the scheduler to more aggressively schedule pending methods. The compiler will of course reject a program which yields an effect prematurely, i.e., promises to not cause an effect after a certain point, and still causes it.

In fact, yielding (as well as parallel message processing as in Section 5.3.2) can be seen as being very true to the actor model. Agha and Hewitt write, regarding computation of the replacement actor (i.e., designating how the next message received will be handled):

> *Two important observations need to be made about replacement. First, replacement implements local state change while preserving referential transparency of the identifiers used in a program. An identifier for an object always denotes that object although the behavior associated with the object may be subject to change. In particular, the code for an actor does not contain spurious variables to which different values are assigned [...]. Second, since the computation of a replacement actor is an action which may be carried out concurrently with other actions performed by an actor, the replacement process is intrinsically concurrent. The replacement actor cannot affect the behavior of the replaced actor by changing the local state of that actor. The net result of these properties of replacement actors is that computation in actor systems can be speeded-up by pipelining the actions to be performed. As soon as the replacement actor has been computed, the next communication can be processed even as other actions implied by the current communication are still being carried out.* [3]

Actors are functional objects, but they become stateful by replacement (the address to the new actor is the same as for the old, though the old actor becomes unreachable as soon as the replacement has been computed.) The spurious variables mentioned would be imperative. Since we do have imperative variables, we use effects to achieve a similar situation, where we allow another message to be processed in parallel if it only touches the part of the state that is not touched by the current message processing (i.e., conceptually we compute a replacement actor that satisfies the needs of the next message.)

Figure 5.5 illustrates this. The class `Yielder` defines two regions `r1` and `r2` as well as abstract effects `A`, `B`, `C` and `D`. The method `computeR1Value(int)` does a short computation and then assigns its computed value to `i1`, which causes a write effect on region `r1`. Method `computeR2Value(int)` does a lengthy computation and then assigns its result to `i2`, which is a write effect on region `r2`. Method `computeValues(int)` (called asynchronously from the outside) requires effects

```
1   active class Yielder {
2
3     region r1, r2;
4     effect A is writes(r1);
5     effect B is writes(r2);
6     effect C is A,B;
7     effect D is reads(r1);
8
9     int i1 in r1;
10    int i2 in r2;
11
12    void computeR1Value(int n) A {
13      ... // Quick computation
14      i1 = ...;
15    }
16
17    void computeR2Value(int n) B {
18      ... // Lengthy computation
19      i2 = ...;
20    }
21
22    void computeValues(int n) C {
23      computeR1Value(n);
24      yield A;
25      computeR2Value(n);
26    }
27
28    int readR1Value() D {
29      return i1;
30    }
31  }
32
```

Figure 5.5. Code example showing yielding of effects.

`A` and `B` to be scheduled, because it calls `computeR1Value(int)` and then `computeR2Value(int)`, but in-between it yields the `A` effect required for calling `computeR1Value(int)`. This allows a pending request for method `readR1Value()` to be executed right away, and in parallel with the long-running computation of `computeR2Value(int)`. Thus modeling faithfully the words of Hewitt and Agha; a new actor is computed that can handle the next message.

Our yielding is similar in spirit with the compatibility functions of Henrio et al. where run-time decisions can be made as to whether two asynchronous methods calls can be run in parallel [66] (and the decision may be based on the use of

e.g., locks.) Compatibility functions can base such decisions on run-time information, such as (in)equality of message arguments. Our yields are less flexible, but can likely be implemented with less run-time overhead.

5.6 Shared Representations

Ownership types can be useful in many situations because it provides very clear and strong guarantees about aliasing and object movement. There are downsides too, of course, in such a restrictive system. Some find ownership types too cumbersome to use because of all the extra annotations. We showed in our original Joelle paper (Paper II) that this can be greatly reduced simply by using clever defaults. Perhaps a more serious critique is that the hierarchical heap decomposition prescribed by owners-as-dominators makes some common and useful object-oriented patterns hard or impossible to implement, at least in a natural and efficient manner. Iterators are one such example. An iterator traverses a data structure and delivers the elements in some sequence. In order to do so efficiently it typically needs access to the internals of the data structure. This goes against ownership types: each object owns its representation and there can be no references from outside the object to the representation. The issue was identified early on [99, 6] and a few attempts at remedying this situation have been put forth over the years. By allowing objects of inner classes to escape while having access to the outer object's representation [35, 20, 18, 132] a back door is created into the data structure, but in these proposals there is no distinguishing between the shared representation and the representation of the objects part taking in the sharing. Boyland proposes to let a data structure temporarily yield ownership of its representation to an external object, e.g., an iterator [26]. This scheme relies on uniqueness and therefore does not allow multiple entry-points into a data structure. Clarke and Drossopoulou relax owners-as-dominators for stack bound references, allowing iterators and other patterns, but their lifetime is limited to the current scope [41].

When investigating regions and how they could be married with ownership, it soon became clear that a region should be nested inside the current object. For completeness, we started thinking about the meaning of nesting a region outside the current object. This lead to the discovery of *ombudsmen-as-dominators*. Ombudsmen provide a way for a group of objects that are not nested to share

data between themselves with "equal rights". Classical ownership allows sharing between objects with a common owner. The problem is that all sibling objects have the same access rights. Thus, it is not possible to share a common object between just some siblings, but not all. With ombudsmen this becomes possible. By nesting a region technically outside `rep`, objects can be placed in that region and shared among siblings, but only with siblings who are given explicit access rights. A collection can now share its internal structure with an iterator, and the iterator can be used outside the collection. This means that an iterator can be efficiently implemented while retaining much of the encapsulation provided by ownership types. Any object that is not part of such sharing enjoys the usual encapsulation guarantees provided by owners-as-dominators. Ombudsmen are described in detail in Paper IV.

While this work, in the way we have described it, has no apparent benefits for parallel programming it does constitute a novel and crucial ingredient in making ownership types in general, and our system in particular, usable in practice. Moreover, we believe that ombudsmen can be used in active objects to efficiently share mutable data between select active peers. The shared context between ombudsmen can represent a "channel" (or a shared data area) through which mutable data can be shared without copying. Reading such shared data could be allowed without any concurrency control. Writing shared data, naturally, would require a concurrency control mechanism of some sort to prevent data races, but as sharing would be possible only among a limited set of peers and accesses to the shared data are clearly identifiable in the source code, locking could even be inferred and transparent to the programmer. What kind of concurrency control is necessary is likely dependent on what actual data structure is shared, and might be left as a choice to the programmer. We view this as an interesting direction for future work.

5.7 Implementation

The Joelle compiler is implemented using the Polyglot Extensible Compiler Framework by Nystrom et al. [102]. Polyglot comes as a Java to Java source compiler built to be easily extended with new grammar, checking phases and code generation. Joelle programs are translated into Java source which can be compiled and run with an off-the-shelf Java compiler and JVM.

For our implementation of our ombudsmen extension to ownership types for our Paper IV publication we used JastAdd [51] as compiler framework. We experimented using this framework for Joelle as well, but in the end settled on Polyglot, which we were more familiar with.

The Joelle front-end has been the major focus of our implementation effort, at the expense of the back-end and run-time. In principle our current back-end could be replaced by another active object implementation. A good candidate might be Henrio et al.'s [66] proposed extension to ProActive [30], which supports internal parallelism and which would enjoy Joelle's ability to verify programmers' disjointness assertions at compile-time. We have made some effort to implement a reasonable message scheduler for active objects. Brandauer evaluated a number of possible scheduler designs for Joelle in his Masters Thesis [27]. All but one design were deemed too slow and memory consuming (and most not even mentioned in the thesis.) The chosen design was then compared to Erlang and Akka in a number of micro-benchmarks.

The current Joelle scheduler uses several mailbox queues per active object, in fact one queue per message type (method.) For large objects with a rich interface, this is not scalable, but the approach seems promising and it is possible that the number of queues can be reduced to a smaller number by considering discrete effects rather than discrete methods. When a message arrives it is added to the queue belonging to that message type. In addition to adding the incoming message to its queue, a barrier object is added to all other queues which conflict (in terms of their effects) with the incoming message, preventing later messages from these queues from being processed until the newly incoming message has been processed. This scheme leads to a simple algorithm for deciding which messages can be scheduled: any message that does not have a barrier in front of it in the queue is free to run, or alternatively every queue that does not start with a barrier can be popped until there is a barrier as top element. After the processing of a message is finished all associated barriers are removed from the other queues, thus "unlocking" them for scheduling.

In an attempt to confirm our intuition that internal parallelism can be beneficial and estimate roughly how our design would stand up to the state-of-the-art, some micro-benchmarks were conducted, comparing Joelle to Erlang and Akka. The benchmarks are variations on the pipeline pattern [90], building a chain of active objects, each processing a message and then passing it on to the

next active object in the chain. There are two kinds of active objects in the chain, one that sorts a list of numbers and one that reverses the list. The two kinds are alternated in the chain. The second version of the benchmark is the same, but with the added wrinkle that after an active object in the chain has processed a message it sends a count message to a common counter active object, thus creating high contention in this object (it cannot be parallelized.) The benchmarks are run by creating 500 lists which are then sent as 500 messages to the first active object in the chain, followed by a stop message. Once the stop message is received from the last active object in the chain the benchmark is finished.

The benchmarks were run on a regular desktop machine with four cores and on a server machine with 32 cores (64 hardware threads). Brandauer's thesis [27] contains a complete description of the benchmarks and machine setup, as well as how the benchmarks were conducted.

The results of the benchmarks look promising, but not really surprising. In a nutshell, for short chains Joelle outperforms both Erlang and Akka because of support for internal parallelism. For longer chains the added complexity of handling several queues etc. makes Joelle suffer a bit, though still comparable to Erlang and Akka. Figure 5.6 shows the benchmark without counter, run on the server machine. Joelle clearly benefits from internal parallelization on runs with short chain length (few actors). On longer chain lengths the speedups are comparable. In Figure 5.7 the results of the same benchmark on the desktop machine are presented. A similar situation occurs, where Joelle outperforms the competition on short chains. We do not know why Joelle and Akka show a "super-linear" speedup, possibly a cache-related effect. When running the second benchmark with the counter enabled, the results on the desktop machine are comparable to running without the counter. On the server, however, the results are different. The increased parallelism on the server, and consequently the increased contention in the counter, makes all three language implementations suffer, but Erlang shows the least loss in performance, possibly due to better support for parallel garbage collection.

Our micro-benchmarks confirm our intuition that parallelizing active objects has a big benefit when there are a small number of active objects running in parallel, but the gain is quickly diminished once the number of active objects running in parallel exceeds the number of cores.

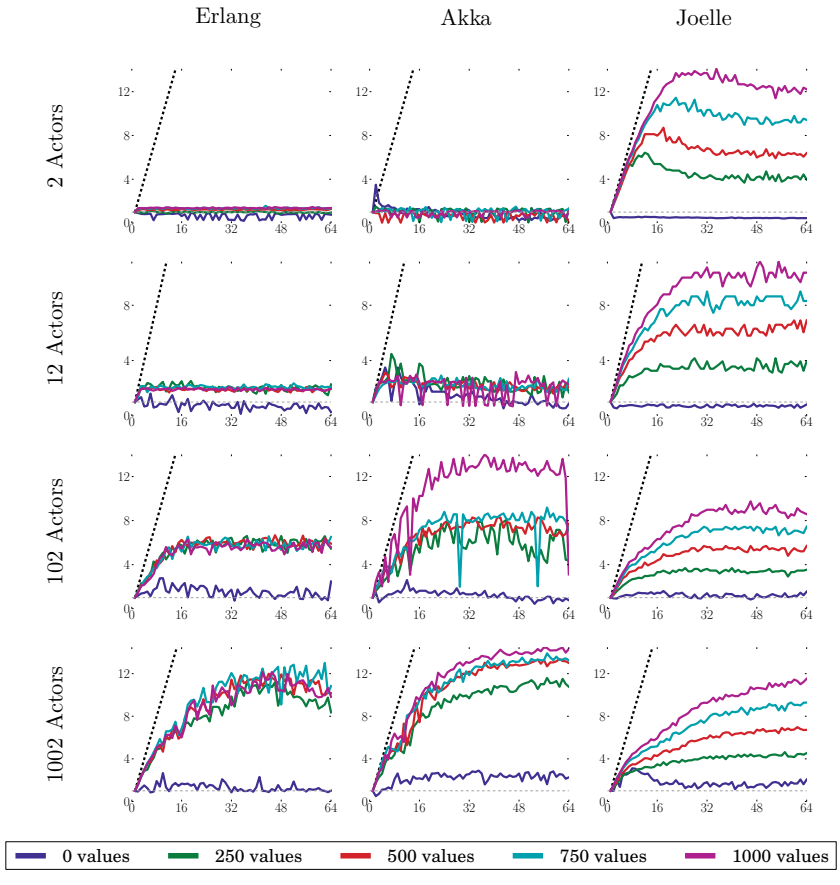

Figure 5.6. Benchmark on server, no counter. Speedup on y-axis, number of cores on x-axis. Higher is better. Values denote amount of work (list size.)

Figure 5.3: Speedups on the Server with counter=False. Higher is better Joelle shows good scalability on average but tends to be more sensitive to the task size than Erlang and Akka. The high scalability in the top row is due to successful internal parallelisation.

Henrio et al. [65] present benchmarks using their multi-active objects. One of the benchmarks is designed to show that parallel algorithms can be implemented efficiently using active objects with internal parallelism. The source code of the multi-active object implementation becomes considerably simpler and only about half of the lines of code are concerned with concurrency, compared to the original implementation (NAS Parallel Benchmarks.) The performance, notably, is on par with the original. Another benchmark is also presented, a CAN peer-to-peer network of active objects, where key/value pairs are stored in the nodes of the network and keys encode a route through the network to the node that holds the value. In a first scenario, two nodes at the opposite ends of the network issue requests to retrieve values from all other nodes. The benchmark shows speedups compared to regular active objects, but

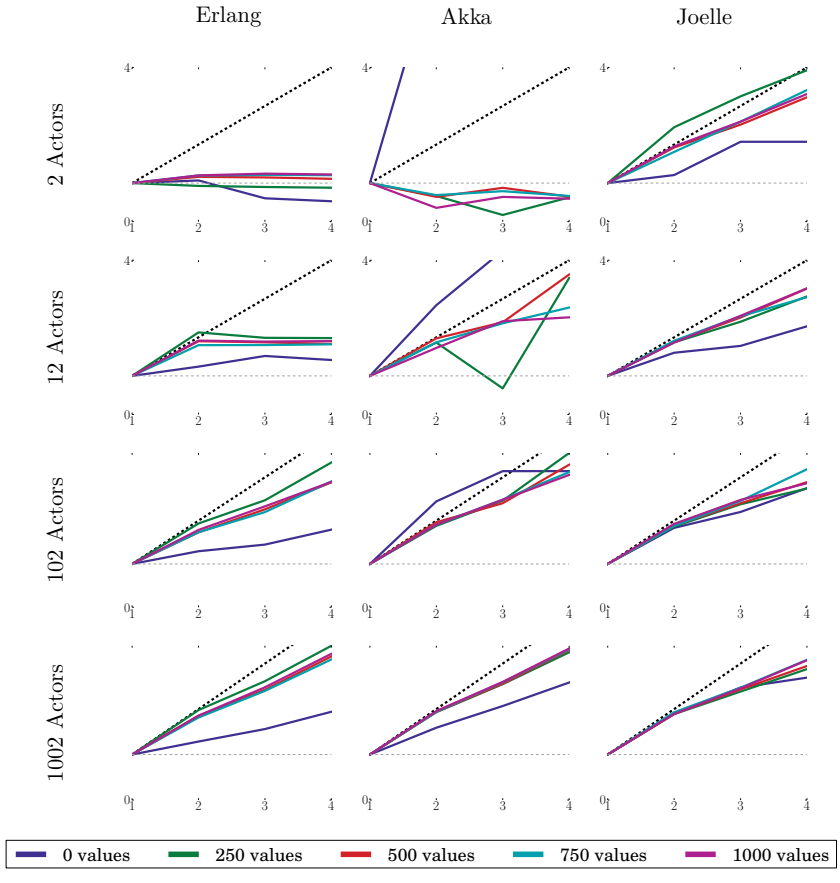

Figure 5.7. Benchmark on desktop, no counter. Speedup on y-axis, number of cores on x-axis. Higher is better. Values denote amount of work (list size.)

Figure 3.4: Speedups on the PC with counter=False. Higher is better All tools do a good job in terms of scalability. Contrary to the same test on the server, Joelle deals the best with small set sizes. Interesting are somewhat very high speedups for Akka and Joelle in the top row. We do not at present to explain those.

it is limited due to the serial nature of requests traveling through the network (there are not so many parallel requests going on.) The second scenario is where multi-active objects show a significant speedup. In this version all nodes in the system send requests to a single node in the center. With a regular active object all those requests must be processed in sequence, whereas with multi-active objects they can be processed in parallel using as many cores as there are available.

5.8 Welterweight Java

Paper III introduces Welterweight Java, a formalization of a core Java-like language with mutable state, multi-threading and concurrency management in the

form of locks. The rational for doing this work and publishing it was that we were finding ourselves reinventing the wheel every time we wanted to formalize some new feature. And the same seemed to be true for our fellow research community members, creating ad hoc formalizations with each paper. When formalizing new language features, it often makes sense to start with a formalization of a base language and then extend it. This does save a lot of work, but it can also save lots of pages, and staying within a page limit is usually non-trivial anyway. Additionally, basing work on a standard base language may help readers to focus on the novel parts instead of trying to understand the boilerplate.

At the time there were a number of alternative formalizations of core Java-like languages to choose from. Featherweight Java [72] (obviously slightly lighter than Welterweight Java at the weigh-in) is the de-facto standard formalism for Java, but others exist too. Featherweight Java formalizes a functional subset of Java, essentially it is typed lambda calculus with Java syntax. We found that while we would base or formalization on Featherweight Java we had to each time add in features that were missing, in particular stack frames and field updates; formalizing alias management constructs in a model that does not support aliasing makes it difficult to make a convincing argument.

The features of Welterweight Java have been developed over time and extracted from previous formalizations of ours. For instance we use a "named form" for expressions (only variables are allowed for instance as method arguments, not arbitrary expressions) which makes the formalism considerably simpler. We prove type soundness of Welterweight Java and that our locks correspond to the semantics of Java. Welterweight Java is specifically designed to be easy to extend and allow for easy removal of features which are not needed. To show the extensible nature of Welterweight Java we extend the formalism with Ownership Types and a non-null type system.

❧

In this chapter we have presented our language Joelle and its defining features. Isolation of active objects is essential to the concurrency model, and we have shown how asynchronous method calls can efficiently carry data between active objects without breaking isolation. Moreover, we have shown how internal parallelism can be supported without introducing data races. In the next chapter we end this thesis with a few concluding remarks.

6

Concluding Remarks

This thesis presents work on programming language constructs for data race-free concurrent and parallel programming. We have presented several discrete scholarly works on how to control sharing in object-oriented programs and how to apply those techniques to parallel programming. Finally, we have shown a unification and extension of our work in the form of the Joelle programming language, which is a clean extension of Java with several of our proposed constructs working in consort to go beyond the support described in our papers. Our goals with this language have been twofold: First, we wanted to provide strongly isolated active objects, but without sacrificing efficiency of message sends. Second, we wanted to provide additional levels of parallelism, both in terms of ability to use parallelism in the execution of a single message and the ability to process multiple messages in parallel. In both these cases, data race-freedom is guaranteed statically at compile-time.

Our work has been focused on the front-end, the source-level of the language. What guarantees do we provide? What can be expressed? Do syntactic choices strike a balance between precision and syntactic overhead? Do we overcome enough of the restrictions of classical ownership types?

The future of this work shifts focus to the back-end, to explore two distinct, but equally important aspects: capitalizing on Joelle's restrictions in the language run-time – including efficient scheduling of active objects, parallel message processing inside of them, potential for parallel garbage collection – as well as evaluating the language's "programmability". While we have written countless small programs with the intent of stressing certain design points at the type system level, larger case studies are important to truly understand the interplay between language features. This will require the implementation of a large selection of libraries with ownership and effect annotations, and possibly even rudimentary tool support for refactoring in the presence of annotations.

The popularity of Rust is anecdotal evidence to the fact that complicated type systems for manipulating ownership and effects is not beyond the capacity, or the stamina, of real-world programmers. As the world struggles to adapt to ubiquitous parallel programming, there is a window of opportunity for new languages and new constructs to get a foot in. Time will tell whether these will include ownership-based effects, intra-actor parallelism, and compile-time guaranteed strong isolation.

> *"it does seem likely that some combination of effect systems and/or ownership types will play an increasingly important role in concurrent programming languages"*
>
> *– Harris, Marlow, Peyton-Jones and Herlihy [64]*

EOF

7

References

[1] Marwan Abi-Antoun and Jonathan Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, pages 321–340, 2009.

[2] Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral Interface Description of an Object-Oriented Language with Futures and Promises. *The Journal of Logic and Algebraic Programming*, 78(7):491 – 518, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[3] Gul Agha and Carl Hewitt. Concurrent Programming using Actors: Exploiting Large-Scale Parallelism. In S.N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 19–41. Springer Berlin Heidelberg, 1985.

[4] Gul Abdulnabi Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT Artificial Intelligence Laboratory, June 1985.

[5] Akka Library. `http://akka.io`.

[6] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, volume 3086 of *LNCS*. Springer, June 2004.

[7] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, pages 311–330, 2002.

[8] Joe Armstrong. Concurrency Oriented Programming in Erlang. *Invited talk, FFG*, 2003.

[9] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[10] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent Programming in Erlang, 1993.

[11] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a Dialect of Java without Data Races. In *OOPSLA*, pages 382–400, 2000.

[12] Henry C. Baker, Jr. and Carl Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[13] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 283–295, New York, NY, USA, 2014. ACM.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[15] Jr. Bocchino, Robert L. and Vikram S. Adve. Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 306–332. Springer Berlin Heidelberg, 2011.

[16] Robert Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, pages 97–116, 2009.

[17] Hans-J. Boehm. Threads Cannot Be Implemented As a Library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM.

[18] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, February 2004.

[19] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[20] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *POPL*, pages 213–223, 2003.

[21] Chandrasekhar Boyapati and Martin C. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, pages 56–69, 2001.

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

[23] John Boyland. Checking Interference with Fractional Permissions. In *SAS*, pages 55–72, 2003.

[24] John Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, 2006.

[25] John Boyland, James Noble, and William Retert. Capabilities for Sharing. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2001.

[26] John Boyland, William Retert, and Yang Zhao. Iterators can be Independent "from" Their Collections. In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

[27] Stephan Brandauer. Task Scheduling using Effects in Joelle. Master's thesis, Uppsala University, 2013.

[28] Denis Caromel. Toward a Method of Object-Oriented Concurrent Programming. *Commun. ACM*, 36(9):90–102, September 1993.

[29] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous Sequential Processes. Technical Report 4753, INRIA Sophia Antipolis, March 2003.

[30] Denis Caromel and Mario Leyton. Proactive Parallel Suite: From Active Objects-Skeletons-Components to Environment and Deployment. In *Euro-Par*

2008 Workshops - Parallel Processing, volume 5415 of *Lecture Notes in Computer Science*, pages 423–437. Springer Berlin Heidelberg, 2009.

[31] Elias Castegren, Johan Östlund, and Tobias Wrigstad. Refined Ownership: Fine-Grained Controlled Internal Sharing. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *Lecture Notes in Computer Science*, pages 179–210. Springer International Publishing, 2015.

[32] Elias Castegren and Tobias Wrigstad. Capable: Capabilities for Scalability. In *IWACO '14, International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, 2014.

[33] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[34] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA*, pages 519–538, 2005.

[35] Dave Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia, 2002.

[36] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and Peter Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In *SFM*, pages 417–457, 2011.

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

[38] Dave Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, 2003.

[39] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal Ownership for Active Objects. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin / Heidelberg, 2008.

[40] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal Ownership for Active Objects. CWI Technical Report SEN-R0803, CWI, June 2008. This work was carried out under project SEN3 EU-FP6 CREDO.

[41] David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, pages 292–310, 2002.

[42] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[43] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[44] Melvin E. Conway. A Multiprocessor System Design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

[45] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, September 2007.

[46] FrankS. de Boer, Dave Clarke, and Einar Broch Johnsen. A Complete Guide to the Future. In Rocco De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin Heidelberg, 2007.

[47] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming in Ambienttalk. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer Berlin Heidelberg, 2006.

[48] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *ECOOP*, pages 28–53, 2007.

[49] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[50] Werner M. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. Ph.D., Department of Computer Science, ETH Zurich, December 2009. Doctoral Thesis ETH No. 18522.

[51] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[52] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, pages 13–24, 2002.

[53] Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *ESOP*, pages 91–108, 1999.

[54] David K. Gifford and John M. Lucassen. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. ACM.

[55] D.K. Gifford, P. Jouvelot, J.M. Lucassen, and M.A. Sheldon. FX-87 Reference Manual. Technical report, MIT, September 1987.

[56] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, pages 21–40, 2012.

[57] Aaron Greenhouse and John Boyland. An Object-Oriented Effects System. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 205–229, London, UK, 1999. Springer-Verlag.

[58] Aaron Greenhouse and William L. Scherlis. Assuring and Evolving Concurrent Programs: Annotations and Policy. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 453–463, New York, NY, USA, 2002. ACM.

[59] Olivier Gruber and Fabienne Boyer. Ownership-Based Isolation for Concurrent Actors on Multi-core Machines. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 281–301. Springer Berlin Heidelberg, 2013.

[60] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202 – 220, 2009. Distributed Computing Techniques.

[61] Philipp Haller and Martin Odersky. Capabilities for Uniqueness and Borrowing. In *ECOOP*, pages 354–378, 2010.

[62] Robert H. Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

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

[64] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[65] Ludovic Henrio, Fabrice Huet, and Zsolt István. A Language for Multi-threaded Active Objects. Technical Report RR-8021, INRIA Sophia Antipolis, 2012. hal-00720012v2.

[66] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded Active Objects. In Rocco De Nicola and Christine Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer Berlin Heidelberg, 2013.

[67] Carl Hewitt and Henry Baker. Actors and Continuous Functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, July 1977.

[68] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, August 1973*, pages 235–245, 1973.

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

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

[71] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and Checking of Object Ownership. In *ECOOP*, pages 181–206, 2012.

[72] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[73] Shams M. Imam and Vivek Sarkar. Integrating Task Parallelism with Actors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 753–772, New York, NY, USA, 2012. ACM.

[74] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):39–58, 2007.

[75] EinarBroch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In BernhardK. Aichernig, FrankS. de Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer Berlin Heidelberg, 2012.

[76] Pierre Jouvelot and David Gifford. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 303–310, New York, NY, USA, 1991. ACM.

[77] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, New York, NY, USA, 2009. ACM.

[78] R. Greg Lavender and Douglas C. Schmidt. Active Object: An Object Behavioral Pattern for Concurrent Programming. *Proc. Pattern Languages of Programs*, 1995.

[79] Doug Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[80] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[81] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[82] K. Rustan M. Leino. Data Groups: Specifying the Modification of Extended State. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 144–153, New York, NY, USA, 1998. ACM.

[83] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using Data Groups to Specify and Check Side Effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 246–257, New York, NY, USA, 2002. ACM.

[84] Paley Guangping Li. *Object Cloning for Ownership Systems*. Ph.D. Thesis, Victoria University of Wellington, 2015.

[85] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.

[86] Yi Lu and John Potter. A Type System for Reachability and Acyclicity. In *ECOOP*, pages 479–503, 2005.

[87] Yi Lu, John Potter, and Jingling Xue. Validity Invariants and Effects. In *ECOOP*, pages 202–226, 2007.

[88] Yi Lu, John Potter, and Jingling Xue. Structural Lock Correlation with Ownership Types. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 391–410. Springer Berlin Heidelberg, 2013.

[89] J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[90] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. *Patterns for Parallel Programming*. Pearson Education, 2004.

[91] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[92] Mark S. Miller, E.Dean Tribble, and Jonathan Shapiro. Concurrency Among Strangers. In Rocco De Nicola and Davide Sangiorgi, editors, *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer Berlin Heidelberg, 2005.

[93] Naftaly H. Minsky. Towards Alias-Free Pointers. In Pierre Cointe, editor, *ECOOP*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer Berlin Heidelberg, 1996.

[94] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. A Formal Reference for SCOOP. In Bertrand Meyer and Martin Nordio, editors, *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 89–157. Springer Berlin Heidelberg, 2012.

[95] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[96] Peter Müller and Arsenii Rudich. Ownership Transfer in Universe Types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 461–478, New York, NY, USA, 2007. ACM.

[97] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zürich, 2007.

[98] O. M. Nierstrasz. Active Objects in Hybrid. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 243–253, New York, NY, USA, 1987. ACM.

[99] James Noble. Iterators and Encapsulation. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*, pages 431–442, 2000.

[100] James Noble, David G. Clarke, and John Potter. Object Ownership for Dynamic Alias Protection. In *TOOLS (32)*, pages 176–187, 1999.

[101] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *ECOOP*, pages 158–185, 1998.

[102] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin Heidelberg, 2003.

[103] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.

[104] OpenMP. *OpenMP Application Program Interface - Version 4.0*, July 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[105] Johan Östlund and Tobias Wrigstad. Welterweight Java. In Jan Vitek, editor, *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 97–116. Springer Berlin Heidelberg, 2010.

[106] Johan Östlund and Tobias Wrigstad. Regions as Owners – A Discussion on Ownership-based Effects in Practice. In *IWACO '11, International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, 2011.

[107] Johan Östlund and Tobias Wrigstad. Multiple Aggregate Entry Points for Ownership Types. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 156–180. Springer Berlin / Heidelberg, 2012.

[108] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, Uniqueness, and Immutability. In *TOOLS (46)*, pages 178–197, 2008.

[109] Pony. `http://www.ponylang.org/`.

[110] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 233–269. Springer Berlin Heidelberg, 2013.

[111] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. Springer Berlin Heidelberg, 2010.

[112] John C. Reynolds. Syntactic Control of Interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 39–46, New York, NY, USA, 1978. ACM.

[113] Rust Programming Language. `https://doc.rust-lang.org/stable/book/`.

[114] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight Polymorphic Effects. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 258–282. Springer Berlin Heidelberg, 2012.

[115] Vijay Saraswat and Radha Jagadeesan. Concurrent Clustered Programming. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 353–367. Springer Berlin Heidelberg, 2005.

[116] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 379–391, New York, NY, USA, 2009. ACM.

[117] Jan Schäfer and Arnd Poetzsch-Heffter. CoBoxes: Unifying Active Objects and Structured Heaps. In *FMOODS*, pages 201–219, 2008.

[118] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *ECOOP*, pages 275–299, 2010.

[119] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel Actor Monitors: Disentangling Task-Level Parallelism from Data Partitioning in the Actor Model. *Science of Computer Programming*, 80, Part A:52 – 64, 2014. Special section on foundations of coordination languages and software architectures (selected papers from FOCLASA'10), Special section - Brazilian Symposium on Programming Languages (SBLP 2010) and Special section on formal methods for industrial critical systems (Selected papers from FMICS'11).

[120] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The Billion-Dollar Fix. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 205–229. Springer Berlin Heidelberg, 2013.

[121] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997.

[122] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[123] Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, September 2005.

[124] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2:245–271, 1992.

[125] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer Berlin Heidelberg, 2013.

[126] Matías Toro and Éric Tanter. Customizable Gradual Polymorphic Effects for Scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 935–953, New York, NY, USA, 2015. ACM.

[127] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[128] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented Event-driven Programming in Mobile Ad hoc

Networks. In *Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the*, pages 3–12, Nov 2007.

[129] Edwin Westbrook, Jisheng Zhao, Zoran Budimli, and Vivek Sarkar. Practical Permissions for Race-Free Parallelism. In James Noble, editor, *ECOOP 2012*, volume 7313 of *LNCS*, pages 614–639. Springer, 2012.

[130] Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Kista, Stockholm, May 2006.

[131] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple Thread-Locality for Java. In *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 445–469. Springer Berlin / Heidelberg, 2009.

[132] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and Immutability in Generic Java. In *OOPSLA '10: Proceedings of the 25th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 598–617, Reno/Tahoe Nevada, USA, 2010. ACM.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1319

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-266795

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016