UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1611

Capability-Based Type Systems for Concurrency Control

ELIAS CASTEGREN

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018

ISSN 1651-6214
ISBN 978-91-513-0187-7
urn:nbn:se:uu:diva-336021

Dissertation presented at Uppsala University to be publicly examined in sal 2446, ITC, Lägerhyddsvägen 2, hus 2, Uppsala, Friday, 9 February 2018 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Alan Mycroft (Cambridge University).

Abstract
Castegren, E. 2018. Capability-Based Type Systems for Concurrency Control. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1611. 106 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0187-7.

Since the early 2000s, in order to keep up with the performance predictions of Moore's law, hardware vendors have had to turn to multi-core computers. Today, parallel hardware is everywhere, from massive server halls to the phones in our pockets. However, this parallelism does not come for free. Programs must explicitly be written to allow for concurrent execution, which adds complexity that is not present in sequential programs. In particular, if two concurrent processes share the same memory, care must be taken so that they do not overwrite each other's data. This issue of data-races is exacerbated in object-oriented languages, where shared memory in the form of aliasing is ubiquitous. Unfortunately, most mainstream programming languages were designed with sequential programming in mind, and therefore provide little or no support for handling this complexity. Even though programming abstractions like locks can be used to synchronise accesses to shared memory, the burden of using these abstractions correctly and efficiently is left to the programmer.

The contribution of this thesis is programming language technology for controlling concurrency in the presence of shared memory. It is based on the concept of reference capabilities, which facilitate safe concurrent programming by restricting how memory may be accessed and shared. Reference capabilities can be used to enforce correct synchronisation when accessing shared memory, as well as to prevent unsafe sharing when using more fine-grained concurrency control, such as lock-free programming. This thesis presents the design of a capability-based type system with low annotation overhead, that can statically guarantee the absence of data-races without giving up object-oriented features like aliasing, subtyping and code reuse. The type system is formally proven safe, and has been implemented for the highly concurrent object-oriented programming language Encore.

Keywords: Programming languages, Type Systems, Capabilities, Concurrency, Parallelism, Data-Race Freedom, Lock-Free Data Structures, Object-Oriented Programming, Actors, Active Objects, Object Calculi, Semantics

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

ISSN 1651-6214
ISBN 978-91-513-0187-7
urn:nbn:se:uu:diva-336021 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-336021)

Dedicated to my sister, Sara

List of papers

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

I E. Castegren and T. Wrigstad:
Reference Capabilities for Trait Based Reuse and Concurrency Control. Technical Report 2016-007, 2016. Uppsala University. [40]
This is an extended version of "Reference Capabilities for Concurrency Control", published at *European Conference on Object-Oriented Programming*, 2016 [39]. The extended version contains an appendix with omitted rules, additional examples and full proofs.

II E. Castegren and T. Wrigstad:
Kappa: Insights, Status and Future Work. *International Workshop on Aliasing, Capabilities and Ownership*, 2016. [38]

III E. Castegren and T. Wrigstad:
Types for CAS: Relaxed Linearity with Ownership Transfer. *In submission*, 2017. [43]
This is an extended version of "Relaxed Linear References for Lock-Free Data Structures", published at *European Conference on Object-Oriented Programming*, 2017 [42]. The extended version contains additional examples, expands on future work, and presents the full proofs.

IV E. Castegren, J. Wallin, and T. Wrigstad:
Bestow and Atomic: Concurrent Programming using Isolation, Delegation and Grouping. *In submission*, 2017. [45] This is an extended version of "Actors without Borders: Amnesty for Imprisoned State", published at *Programming Language Approaches to Concurrency- and Communication-cEntric Software*, 2017 [44]. The extended version presents two variants of the original system with full proofs, and discusses case studies and a prototype implementation.

V E. Castegren and T. Wrigstad:
OOlong: an Extensible Concurrent Object Calculus.
To appear at Symposium on Applied Computing, 2018 [46]

Reprints were made with permission from the publishers.

Sammanfattning på svenska

De senaste 40 åren har utvecklingen av datorhårdvara följt Moores lag, som säger att antalet transistorer som får plats på ett datorchip dubbleras vartannat år. I praktiken betyder det att även datorers prestanda mer eller mindre har dubblerats vartannat år sedan 1970-talet. Fysikens lagar sätter dock en gräns för hur många transistorer som kan få plats på ett chip innan energiåtgången blir för hög, och i början av 2000-talet började hårdvarutillverkare känna av den gränsen.

För att kunna fortsätta följa den utvecklingskurva som Moores lag förutspår har de flesta hårdvarutillverkare istället vänt sig till flerkärniga processorer. Idag finns parallell hårdvara överallt, från företagens stora serverhallar till telefonerna i våra fickor. Teoretiskt kan en processors prestanda fördubblas genom att låta ytterligare en processor arbeta parallellt med den första, men i praktiken är detta inte alltid sant. För att kunna utnyttja en flerkärnig processor till fullo så måste program skrivas så att de tillåter parallell exekvering.

Att skriva parallell mjukvara introducerar en komplexitet som inte finns i sekventiella program, där kontrollflödet kan följas genom att helt enkelt läsa programmet från början till slut. I ett parallellt program, med flera kontrolltrådar, måste särskild hänsyn tas till minne som delas mellan trådar. Om två exekverande trådar samtidigt arbetar med samma minne kan de råka skriva över varand-

ras resultat, vilket kan leda till att programmet beter sig på oförutsedda sätt. Sådana problem kallas för *kapplöpningsproblem* (eng. *data-races*).

De flesta programspråk har utvecklats med fokus på sekventiell programmering, vilket betyder att programmeraren i allmänhet inte får någon hjälp från kompilatorn att skriva korrekta parallella program. När det gäller kapplöpningsproblem så måste programmeraren själv lista ut vilket minne som delas mellan trådar, om detta minne någonsin används samtidigt, och i så fall hur trådarna ska samarbeta för att inte störa varandra.

Dessa problem är frekvent förekommande i objektorienterade programmeringsspråk, där *aliasering* (eng. *aliasing*) – alltså när samma minne är åtkomligt via flera olika namn – är en del av programmeringsstilen. En viktig observation är att aliasering är nödvändigt för att två trådar ska kunna dela på minne. Det går att ha aliasering utan att ha delad åtkomst till minne (alltså att ha aliasering inom en och samma tråd), men det går inte att ha delad åtkomst till minne utan att ha aliasering. Många av de mest använda språken idag är objektorienterade, och för att de ska kunna fortsätta användas på ett effektivt sätt med modern hårdvara behövs programmeringsstöd för att hantera parallellism.

Den här avhandlingen introducerar ett antal nya programspråkstekniker för att hjälpa programmerare att skriva korrekta och effektiva parallella program. Centralt för arbetet är tesen att nyckeln till att hantera parallellism är hur aliasering hanteras. Genom att noggrant kontrollera hur referenser till minne får skapas och spridas i programmet kan situationer där kapplöpningsproblem skulle kunna inträffa helt och hållet uteslutas. Avhandlingen presenterar ett typsystem som med låg syntaktisk kostnad garanterar att ett kompilerande program aldrig råkar ut för kapplöpningsproblem, utan att för den sakens skull överge de principer som objektorienterade programspråk bygger på. Programspråksteknikerna har formaliserats och bevisats korrekta, och har även implementerats i det objektorienterade språket Encore.

Ett vanligt sätt att förhindra kapplöpningsproblem är genom att upprätthålla *ömsesidig uteslutning* (eng. *mutual exclusion*), vilket innebär att en tråd som arbetar inom en viss sektion av minnet är den enda tråden som just då har tillgång till detta minne. En teknik för detta är att skydda det delade minnet med ett lås. Om en tråd försöker få tillgång till minne som redan används tvingar låset tråden att vänta tills den andra tråden har arbetat klart. Ett annat alternativ är att låta minnessektioner permanent tillhöra en viss tråd, och istället låta andra trådar delegera sitt arbete till tråden som äger minnet. Typsystemet i denna avhandling garanterar att båda dessa tekniker används på ett korrekt sätt.

Ömsesidig uteslutning är en kraftfull egenskap, men när många trådar arbetar med samma minne kan det leda till att de tvingas lägga majoriteten av sin exekveringstid på att vänta på att minnet ska bli ledigt. I dessa situationer används ofta så kallade *låsfria algoritmer*, där trådar koordinerar sitt arbete med delat minne på ett sätt så att ingen tråd behöver vänta på någon annan. Typsystemet i denna avhandling hjälper programmerare att implementera sådana algoritmer genom att garantera att felaktig koordinering som skulle ha lett till kapplöpningsproblem inte kan inträffa.

Vi har sedan länge lämnat den sekventiella programmeringens tidsålder, och idag är nästan all hårdvara mer eller mindre parallell. Programspråksteknikerna i denna avhandling tillhandahåller en mångsidig verktygslåda för nya språk som utvecklas för den parallella värld vi lever i, utan att för den sakens skull tvinga programmerare att lämna sina gamla, objektorienterade verktyg bakom sig.

Acknowledgements

Even though I think the metaphor of "having made a journey" is a bit worn, I can't help but feeling like it has been a good ride! There are many people to thank for this.

First, I want to thank my advisor Tobias Wrigstad, without whom my doctoral studies would never even have started. Your constant positivity, creativity and encouragement is an inspiration, and I could write another thesis about everything that you have taught me that isn't about programming languages. Having had Dave Clarke as a co-advisor means that there has never been a boring meeting, and that all my papers have gotten feedback that saved them at least one full round-trip to the reviewers before being accepted. Thank you both!

During these years I have had the pleasure of working in the same group as Stephan Brandauer, Kiko Fernandez, Albert Yang and Huu-Phuc Vo. Thank you for all the interesting discussions and all the fun we have had! Stephan, we started our PhDs at almost the same time, and I am very happy for your company over the years. One day, clapital letters will be *the* standard for non-ambiguous pronunciation of variable names.

In addition to Stephan, I have also been working in the same office as Stavros Aronis, David Klaftenegger, Andreas Löscher, Kjell Winblad, Magnus Norgren and Magnus Lång. Some of you moved to offices of your own, while the rest of you stuck it out till the end (my end, that is). Either way, I am glad to have had

such pleasant company for coffee, tea and lunch breaks, and I always enjoyed our off-(and sometimes on-)topic discussions. To all my colleagues: good luck with your theses!

While not technically a colleague, I am also happy to have spent many a lunch break taking walks with Kristoffer Franzén, looking at birds, sharing chocolate wafers, and talking about everything from poetry and music theory to physics and philosophy. It is no exaggeration to say that you are one of my best friends!

To everyone who have ever sung in Kalmar Nation's Choir, thank you for trusting me to be your conductor for nine years! You are literally to many to fit in this space, but I am incredibly grateful to have had the opportunity to learn and develop together with you. Who would have thought that "the little choir with the big heart" would grow up to become this beautiful, fifty headed instrument? Wednesdays without you will never be the same! Karin Bengtsson deserves a special mention, being the only person who has been in the choir since from before I started. If it wasn't for your support, chances are I wouldn't have made it past the first year.

I have also had the honour of playing in the world music band Morfis together with Staffan Björklund, Hannah Sundkvist, Christofer Bäcklin, David McVicker, Murat Yalçin, Sofie Renemar, Mattias Zetterberg and Alexander Larsson. You are my favourite people to play music with, and with a sample size of over a hundred gigs (and countless hours on the road), I can safely say "you rock!".

My parents, Karin and Staffan, made me who I am today, and so far it has been working out well. Thanks for that! Mattis, I can't imagine growing up with a better brother. Also, if it wasn't for you I wouldn't even have started studying computer science. My sister Amanda, you are an inspiration. You have taught me that life never stagnates, and that it is always possible to change course. To Kajsa Yngve, Sara Eklöf and Kajsa Mayrhofer: you have always been around, and I hope that this will never change!

Finally, Ida, I love all the things we do together, whether its travelling to distant lands or just having ice cream in front of the TV. Thank you for constantly reminding of all the wonderful things in the world that are more important than type systems!

To everyone above, and to anyone I may have forgotten:

Thank you!

Contents

1. Introduction

For the last 40 years, the evolution of computer hardware has followed Moore's law [107], which states that the number of transistors that fit on a chip doubles approximately every two years. In practice, when factoring in the increased performance of single transistors, this means that the performance of computers has roughly doubled every two years since the 1970s [90]. However, due to energy requirements and heat dissipation, there is a physical limitation to how many transistors can be added to a chip. In the early 2000s, manufacturers of computer chips started to approach this limit [70].

In order to keep up with the performance predictions of Moore's law, hardware vendors instead turned to multi-core processors, and today parallel hardware is everywhere, from server halls to phones. In theory, one can double the performance of a processor by adding another processor running in parallel. In practice, however, this is not always true; in order to fully utilise the potential of a multi-core processor, programs must be written in such a way that they allow for concurrent execution [133]. This is captured by Amdahl's law, which states that the maximum performance improvement of a parallel program is proportional to the amount of code that can be run concurrently [9].

Writing concurrent software introduces complexity that is not present in sequential programs, where the control flow can be followed by simply reading the code from the beginning to the end. In a concurrent program with multiple threads of control, special care must be taken with the memory shared between threads. For example, if two threads access the same memory concurrently, they may overwrite each others' results, leading to unexpected behaviour. Such situations are known as *data-races*.

Most mainstream languages were designed with sequential programming in mind, meaning that the programmer is left with little or no support from the compiler to write correct concurrent programs. When it comes to data-races, it is up to the programmer to figure out which memory is shared between threads, which shared memory is potentially subject to data-races, and how to properly synchronise concurrent accesses to this memory.

These problems are prevalent in object-oriented programming languages where mutable objects and *aliasing*, *i.e.*, multiple references to the same object or memory address, are central features [51]. Today, four out of the five most used languages are object-oriented [135], and with the ubiquity of parallel hardware, developing language technology for handling concurrency in an object-oriented context is imperative for allowing programmers to write efficient software. An important observation about aliasing is that it is a prerequisite for sharing; one can have aliasing without sharing (*i.e.*, aliasing from within a single thread), but can never have sharing without aliasing.

This thesis defends the statement that controlling aliasing is key to controlling sharing between threads.

1.1 Contributions

The main contribution of this thesis is a number of static and dynamic language features for controlling aliasing in a concurrent setting. At the core of these features is the idea of a *reference capability*, an abstract token attached to each reference which defines what operations are available on both the underlying object (*e.g.*, which methods may be called) and the reference itself (*e.g.*, if the reference may be copied). By controlling the creation and propagation of reference capabilities, situations where data-races could occur can be completely avoided.

In this thesis, the tracking of reference capabilities is implemented in the form of a type system called Kappa. The type system supports object-oriented features like subtyping, code reuse and encapsulation, and a program written using Kappa is guaranteed to be free from harmful[1] data-races. Kappa ensures that a reference may always be used to the full extent allowed by its type without fear of data-races, regardless of which objects are reachable through the reference. In addition to the object-oriented paradigm, the contributions of this thesis extend to both procedural programming and programming with actors using mutable state. Kappa incorporates ideas from many existing systems for alias control and expresses them in a unified system.

[1]In certain cases, concurrent mutation can safely be explicitly allowed by the programmer.

This thesis consists of a collection of papers that cover different aspects of the Kappa type system. This section continues with a summary of the contribution of each paper.

PAPER I
Reference Capabilities for Trait Based Reuse and Concurrency Control

---

This paper presents the original formulation of Kappa in the shape of a type system for concurrent, object-oriented programming. Objects are guarded by reference capabilities, and an object may always be accessed without fear of data-races. The type system uses traits for subtyping and code reuse. When implementing a trait, programmers can assume mutual exclusion and encapsulation of private data, which simplifies and localises reasoning.

How mutual exclusion is achieved is specified when the trait is used. This advances the state-of-the-art by allowing traits to be reused across different concurrency scenarios. Traits are also used to reason about the possible effects caused by calling a method, and allows safely accessing an object concurrently when the effect footprints of two methods are disjoint. All of this is done without explicit ownership types or effect annotations.

PAPER II
Kappa: Insights, Status and Future Work

This paper expands Paper I by providing more details on the connections between Kappa and related work. It discusses the implementation of Kappa in Encore, a programming language based on active objects, and how it facilitates safe sharing between active objects. It also outlines some directions for future work.

PAPER III
Types for CAS: Relaxed Linearity with Ownership Transfer

The type system presented in Paper I ensures data-race freedom by guaranteeing mutual exclusion. This is a powerful property, but also too strict for many fine-grained concurrency patterns, where threads cooperatively access shared mutable state, following some protocol to ensure that their interaction is safe. This paper presents a type system for capturing patterns in lock-free data structures, centered around the atomic compare-and-swap (`CAS`) primitive. It extends Kappa by allowing shared mutable state in a controlled fashion. It is flexible enough to allow the implementation of several fundamental lock-free data structures, while still guaranteeing the absence of uncontrolled data-races. The paper formalises the type system and proves it sound, and also reports on a prototype implementation.

At the core of the system is the observation that aliasing is only harmful if more than one alias is used to access mutable state. The type system tracks the ownership associated with each reference, using the `CAS` primitive to transfer ownership between aliases, and ensures that there is never more than one owning reference to an object. The guarantee given by the system is that access to owned data is always exclusive, and therefore free from data-races.

PAPER IV
Bestow and Atomic:
Concurrent Programming using Isolation, Delegation and Grouping

The type system presented in Paper I relies on encapsulation—keeping the internal state of an object truly private—to ensure that exclusive access to an object also implies exclusive access to the objects that make up its internal state. This property is useful, but also restrictive as it forces all interaction with an object aggregate to go via the "owner" of these objects. Paper IV extends Kappa with a construct which allows references to an object's private state, enforcing that all operations through such references are implicitly delegated to the owner, and ensuring that concurrent accesses are properly synchronised. This facilitates switching between synchronisation based on isolation and synchronisation based on delegation, which is useful for programming with both actors and locks.

Additionally, the paper introduces a construct for grouping several operations so that they are performed back to back without any interleaving of concurrent operations. This allows programmers to introduce new atomic operations by composing existing operations, without having to worry about the effects of concurrent accesses to the same object. The paper formalises both constructs in three different variations and proves them all sound. The paper also reports on a prototype implementation.

PAPER V
OOlong: an Extensible Concurrent Object Calculus

This paper introduces OOlong, a small object calculus with support for concurrency and locking. It was first used in the formalisation of the type system in Paper I, but has since been stripped down to its essentials, making it suitable for extension. In contrast to commonly used Java-based calculi, OOlong does not aim to model any specific language, but rather object-oriented languages in general. For this reason, it uses a simple subtyping mechanism based on interfaces rather than relying on class inheritance. To facilitate extension and formal reasoning, the semantics have been mechanised and proven sound in the theorem prover Coq, and the source code is publicly available. OOlong serves as a starting point for researchers who want to develop and reason about new language features in concurrent object-oriented languages.

The Author's Contributions

I Manuscript written together with second author. Sole implementor. Semantics and proofs written in collaboration with second author.

II Main author.

III Main author. Semantics written in collaboration with second author. Sole contributor of proofs and implementation.

IV Main author. Semantics written in collaboration with third author. Sole contributor of proofs. Implementation written in collaboration with second author.

V Main author. Sole contributor of semantics and proofs.

Related Publications

VI **Capable: Capabilities for Scalability** [37], Elias Castegren and Tobias Wrigstad. *International Workshop on Aliasing, Capabilities and Ownership*, 2014
Sketches of the ideas in Papers I–III were first presented in this workshop paper.

VII **Reference Capabilities for Concurrency Control** [39], Elias Castegren and Tobias Wrigstad. *European Conference on Object-Oriented Programming*, 2016
Paper I is the extended version of this paper.

VIII **Relaxed Linear References for Lock-Free Data Structures** [42], Elias Castegren and Tobias Wrigstad. *European Conference on Object-Oriented Programming*, 2017
Paper III is the extended version of this paper.

IX **Types for CAS: Relaxed Linearity with Ownership Transfer** [41], Elias Castegren and Tobias Wrigstad. *Nordic Workshop on Programming Theory*, 2016
The ideas of Paper III were presented in this extended abstract.

X **Actors without Borders: Amnesty for Imprisoned State** [44], Elias Castegren and Tobias Wrigstad. *Programming Language Approaches to Concurrency- and Communication-cEntric Software*, 2017
Paper IV is an extended version of this article.

XI **Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore** [29], Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. *Formal Methods for Multicore Programming*, 2015
This paper introduces the programming language Encore, in which all implementations have taken place.

Artefacts

All the implementation in papers I–XI was carried out in the Encore compiler [29, 62], which was written from scratch in parallel with this thesis as part of the UPSCALE project [138]. At the time of writing, the author is the number one contributor to the compiler's development. The compiler is open source software and can be obtained from the following URL:

`https://github.com/parapluu/encore`

Funding

This work was partially funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the Swedish Research Council project Structured Aliasing, and the EU project FP7-612985 Upscale.

1.2 Outline

The rest of this extensive summary is structured as follows:

- Chapters 2 and 3 review the necessary background on concurrent and object-oriented programming, setting the stage for the contributions of this thesis.
- Chapter 4 overviews existing techniques for alias control and data-race prevention, and briefly discusses verification techniques based on program logics and model checking.
- Chapter 5 introduces Kappa, explains the basics of the Kappa type system, and discusses extensions and variations thereof.
- Chapter 6 discusses the implementation of Kappa in the active object language Encore, focusing on the features that are available in the implementation but not in the formal treatise.
- Chapter 7 concludes and discusses some directions for future work.

The overarching goal of the work in this thesis is to prevent concurrency errors caused by data-races. We therefore begin by explaining some background on how concurrency is commonly achieved and controlled.

2. Achieving and Controlling Concurrency

This thesis introduces language features which help programmers write correct concurrent programs, notably by guaranteeing the absence of data-races. In order to explain the context of these contributions, this chapter reviews how concurrency is commonly achieved and controlled. Section 2.1 discusses concurrency based on threads, what kind of errors may occur when memory is shared between threads, and how these errors are avoided by using locks (Section 2.1.1) or other more fine-grained techniques (Section 2.1.2). Providing support for this kind of programming is the focus of the Kappa type system, and Papers I–III.

This chapter also overviews how conflicting accesses to shared memory can be resolved dynamically by using software transactional memory (Section 2.1.3) and how channels and message passing allow threads to communicate without using shared memory (Section 2.1.4). Finally, Section 2.2 discusses concurrency based on actors instead of threads, and overviews some existing actor systems. Actors are relevant for the work presented in Paper IV, and for the implementation of Kappa in Encore (*cf.*, Chapter 6).

2.1 Threads and Locks

The most common concurrency model is one where a program has one or more threads of execution that are running concurrently. In this model a thread can spawn (*fork*) new threads, and wait for one or more threads to finish (also called a *join*). Commonly, threads do not necessarily map to the same number of processor cores; a single core can run many software threads, meaning that the software threads are scheduled to run one at a time, sharing the same core.

In general, one discerns between concurrency, which is any situation where more than one thread is making progress, and parallelism, which is when several threads are actually executing simultaneously [132]. Concurrency is about

```
 1  void calculate(int *result) {
 2    int x = ... // Perform some work
 3    *result = x;
 4  }
 5
 6  int main() {
 7    int result = 0;
 8    thread_t t = fork(calculate(&result));
 9    join(t);
10    printf("%d", result);
11    return 0;
12  }
```

Figure 2.1. A simple example of thread communication. Removing the **join** on Line 9 introduces a data-race between the reading of `result` on Line 10 and the write to `result` on Line 3.

operations being logically simultaneous, while parallelism is generally about performance optimisation. For example, the window manager of an operating system is a concurrent program; there may be one window showing a video while the user is entering text in another window, and a third window is displaying the current time through an animated clock. All of these software threads could be run without parallelism (*i.e.,* on a single core) and still appear to be running at the same time to the user. It is the scheduling of concurrent threads that decide if the program is actually parallel.

A web server is also a concurrent program which could be implemented by spawning a new thread for each incoming connection[1]. When running on a single core, each connection reduces the amount of processing time given to each software thread, possibly introducing latency for the connecting clients. In order to handle a larger number of connections, a web server may turn to parallelism and distribute the load of incoming connections over several cores. These cores may be run on the same machine, or be distributed across different machines.

The most basic way for two threads to communicate with each other when running on the same machine is via shared memory. For example, one thread may spawn another thread and give it access to some known memory location where the spawned thread writes the result of its computations before finishing. Figure 2.1 shows this interaction in C-like pseudo code (using **fork** and **join** as

[1]In reality, it would probably use a more lightweight solution, like a thread pool.

```
1  void send_to_printer(document_t *d, int *count) {
2    int pages = ... // Send d to printer
3    if (pages > 0) {
4      *count = *count + pages;
5    }
6  }
7
8  int main() {
9    int count = 0;
10   thread_t t1 = fork(send_to_printer(new_document(), &count));
11   thread_t t2 = fork(send_to_printer(new_document(), &count));
12   join(t1);
13   join(t2);
14   printf("%d", count);
15   return 0;
16 }
```

Figure 2.2. An example of two threads racing to write to the same memory.

primitives). The **fork** on Line 8 spawns a new thread which runs the `calculate` function. The **join** on Line 9 waits for the spawned thread to finish.

In this scenario, there are two points of communication: the sending of the shared memory location `result` to the new thread (a form of direct communication), and the passing of the spawned thread's result via this shared memory (a form of indirect communication). The first form of communication is the most simple of the two, as it will never fail and will always result in the spawned thread getting access to the address of `result`. In contrast, if the **join** on Line 9 is removed the second point of communication may fail. Depending on whether the spawned thread manages to write to `result` before it is read by the spawning thread, the program may have different results for different runs.

This is a simple example of a *data-race*. Two threads are racing to read from and write to the same memory, and the order of these operations depend on how the threads are scheduled (which may be different for different runs of the program). The **join** operation gets rid of the data-race by introducing *synchronisation*, which prevents certain orderings of the operations (in this case that the read happens before the write).

2.1.1 Synchronisation using Locks

Figure 2.2 shows an example with a total of three threads running concurrently. The main thread spawns two threads running the `send_to_printer` function, which tries send a document to some printer and then conditionally increments the shared integer `count` with the number of pages that were printed. Once both spawned threads have finished, the original thread prints the value of `count` to the screen.

This program has a data-race, as both spawned threads attempt to update `count` concurrently. This data-race is more complicated than the read-write race in Figure 2.1 for two reasons. First, neither of the racing threads is "aware of" the existence of the other thread, so there is no way to explicitly wait for the other thread as in Figure 2.1. Second, it is generally not possible to detect if a data-race occurred or not. The increment on Line 4 will be performed in three atomic steps: reading the value of `count` into a register, incrementing this value, and finally writing the new value back to memory. Between any of these steps, the other thread may access the same memory. For example, the following interleaving of operations for threads `t1` and `t2` is possible (assuming the value of `pages` is `1` for both threads):

Time	**t1**	**t2**
1	read `*count` into register `r1`	
2	increment `r1` by 1	
3		read `*count` into register `r2`
4		increment `r2` by 1
5		write contents of `r2` to `*count`
6	write contents of `r1` to `*count`	

In this scenario, both threads read the initial value `0`, increment it locally to `1` and then write it to memory, leaving the final value of `count` at `1` instead of the expected value `2`. This is known as a *lost update*; even though both threads locally appear to successfully increment the counter, the resulting state is as if one of the operations was never performed.

A simple way to prevent this data-race from occurring would be to spawn the first thread, wait for it to finish, and then spawn the second thread. This would however destroy any potential performance gained by running the threads in parallel. A better solution would be to have the two threads only synchronise

```
 1  void send_to_printer(document_t *d, int *count, mutex_t *mutex) {
 2    int pages = ... // Send d to printer
 3    if (pages > 0) {
 4      lock(mutex);
 5      *count = *count + pages;
 6      unlock(mutex);
 7    }
 8  }
 9
10  int main() {
11    int count = 0;
12    mutex_t *m = mutex_init();
13    thread_t t1 = fork(send_to_printer(new_document(), &count, m));
14    thread_t t2 = fork(send_to_printer(new_document(), &count, m));
15    join(t1);
16    join(t2);
17    printf("%d", count);
18    return 0;
19  }
```

Figure 2.3. An example of synchronising memory accesses by using locks.

when accessing the shared memory. This is commonly achieved by using locks. A lock provides a way for threads to communicate that they are requesting exclusive access to some resource, for example a memory region. A thread may *acquire* a lock and later *release* it. If a thread attempts to acquire a lock that has already been acquired by another thread, it will stop executing until the lock is released.

Figure 2.3 shows a modified version of the program from Figure 2.2. Here, the spawning thread first creates a lock on Line 12 (abstracted into the type `mutex_t`) and passes it to the spawned threads together with the shared integer `count`. When one of the spawned threads is about to update the counter, it acquires the lock (Line 4), performs the update and then releases the lock (Line 6). This way, the lock will serialise all accesses on the counter, but allow the rest of the function to run concurrently.

Even though these examples are simplified, they show some of the complexity introduced when memory is shared across threads. Notably, there is no way to distinguish operations that need synchronisation from operations that do not, without inspecting the full program. In Figure 2.1, no locks were needed when writing to shared memory, whereas in Figure 2.2, omitting synchronisation lead to a data-race. Sometimes the same piece of data may need synchronisation to be safely accessed in one place, but not in another. In Figure 2.3, operations on `count` must be wrapped in a lock in `send_to_printer`, but not after the **join**s in `main`. Readers-writer locks allow a lock to be acquired for reading or writing, allowing several concurrent readers but only a single writer at a time [120], but this still relies on the programmer to ensure that readers do not perform writes.

In all of these examples, the mechanisms for achieving synchronisation is disjoint from the data that is being protected, and it is up to the programmer to keep track of which locks are used for which data, and when they are needed to prevent incorrect thread interleavings. Apart from avoiding obvious mistakes, like forgetting to acquire or release a lock, the programmer must also ensure that a locally correct refactoring of one part of a program does not lead to synchronisation bugs in another part of the program. Mainstream languages generally provide little or no support for any of these things.

For all these reasons, getting locking right in a program can be very tricky. Locking too little leads to bugs caused by data-races, while locking too much degrades performance by removing parallelism. Additionally, for programs with more than one lock, programmers must be careful with the order in which locks are taken to avoid having two threads both waiting to acquire a lock already being held by the other thread. This is known as a *deadlock*, and while it is an important class of bugs this thesis does not explore them further. Deadlocks caused by threads acquiring the same lock twice can be avoided by using *reentrant locks* [121].

One of the contributions of this thesis is type system support for tracking which operations need synchronisation to avoid data-races and which operations do not. Kappa enforces that shared data is never accessed without proper synchronisation. This guarantees *mutual exclusion* whenever a thread accesses mutable state, meaning that no other thread may access the same memory at the same time. This part of the system is outlined in Chapter 5 and detailed in Paper I.

While mutual exclusion is a powerful and useful property, it is sometimes too strong a restriction. Certain algorithms and data structures require several threads to have shared access to memory which is updated concurrently. The following section overviews some of the techniques for coordinating threads without using locks.

```
void send_to_printer(document_t *d, int *count) {
  int pages = ... // Send d to printer
  if (pages > 0) {
    fetch_and_add(count, pages);
  }
}

int main() {
  int count = 0;
  thread_t t1 = fork(send_to_printer(new_document(), &count));
  thread_t t2 = fork(send_to_printer(new_document(), &count));
  join(t1);
  join(t2);
  printf("%d", count);
  return 0;
}
```

Figure 2.4. An example of using atomic operations to allow concurrent updates without data-races.

2.1.2 Fine-Grained Concurrency without Locks

The previous section showed how to use locks to achieve mutual exclusion and avoid data-races. According to Amdahl's law [9], the parallel speedup of a program is restricted by the amount of code that cannot be executed concurrently. In a program where many threads share the same data, or where there is a lot of contention on some shared memory, even a minimal amount of locking may degrade performance, since a lot of time will be spent waiting to acquire locks guarding shared data. This section overviews some of the techniques for handling concurrency in a more fine-grained manner, without using locks.

The data-race in Figure 2.2 stems from the fact that the increment on Line 4 is not an atomic operation, meaning that two increments can be interleaved in such a way that one of the updates is lost. The program in Figure 2.3 uses locks to guarantee atomicity. Since incrementing values in memory is such a common operation, there are special instructions for reading and updating a memory location in an atomic fashion[2]. For example, a `fetch_and_add` operation will atomically read a value from memory, increment it by some amount, and then write the new value back to memory [13].

[2] Locks are commonly implemented using these operations to guarantee atomicity of *e.g.*, aquiring a lock.

```
struct node {
  void *elem;
  struct node *next;
};

struct stack {
  struct node *node;
};

void *pop(struct stack *s) {
  struct node *tmp = s->top;
  if (tmp == NULL) return NULL;
  s->top = tmp->next;
  return tmp->elem;
}
```

Figure 2.5. A partial implementation of a stack data structure. Using this stack concurrently without synchronisation would lead to data-races.

Figure 2.4 shows the program from Figure 2.2, modified to avoid data-races by using an atomic `fetch_and_add` instruction. While the two threads are still reading from and writing to the same memory location concurrently, this interaction is no longer considered a data-race. This is because all operations on shared state are performed using atomic instructions, meaning there can be no lost updates. Another way to motivate why this interaction should not be considered harmful is that the program has the same outcome as if all function calls were run sequentially by a single thread. This property is known as *serialisability* [85].

`fetch_and_add` and similar instructions work well for programs where the memory shared between threads contains integers, but they can not be used for more advanced operations, such as modifying references in a data structure. Figure 2.5 shows a partial implementation of a stack data structure which would be correct if it was only used in a sequential setting. However, using it concurrently would lead to unwanted behaviour, due to data-races. The cause of the error is analogous to the lost update in Figure 2.2: if two threads are executing the function concurrently the program might see the following interleaving of operations (`NULL` checks omitted):

Time	t1	t2
1	read s->top into tmp	
2	read tmp->next into register r1	
3		read s->top into tmp
4		read tmp->next into register r2
5		write contents of r2 to s->top
6	write contents of r1 to s->top	

In this scenario both threads read the top node into a local variable tmp, and then replace s->top by the successor of tmp. In other words, both threads "successfully" pop the same top node and return its element, which is most likely unexpected behaviour. One might be tempted to fix this by reading tmp->next into a local variable and adding an if-statement to check if tmp is still an alias of s->top before performing the assignment (and retrying the whole operation if it is not), but this will not work as the value of s->top may be updated concurrently after the check but before the assignment.

The solution to this problem without resorting to locking is another atomic operation called compare_and_swap (or CAS for short) [13]. It has the same behaviour as the following function

```
1 bool CAS(void **p, void *old, void *new) {
2   if (*p == old) {
3     *p = new;
4     return true;
5   } else {
6     return false;
7   }
8 }
```

with the important property that the comparison on Line 2 and the (potential) assignment on Line 3 happens atomically; if the comparison evaluates to true, the assignment will happen before any other operation of any other thread.

Figure 2.6 shows the partial implementation of a stack that uses CAS to avoid data-races (originally developed by Treiber [136]). The assignment on Line 12 speculatively reads s->top into a local variable. The CAS on Line 14 checks if the speculation is still valid, and if it is, overwrites s->top with its successor node. If the CAS is successful, the element of the popped node is returned. If the CAS fails, the pop is retried by starting the loop over from the beginning. The buggy interleaving seen in the program of Figure 2.5 is no longer possible;

```
 1  struct node {
 2    void *elem;
 3    struct node *next;
 4  };
 5
 6  struct stack {
 7    struct node *node;
 8  };
 9
10  void *pop(struct stack *s) {
11    while (true) {
12      struct node *tmp = s->top;
13      if (tmp == NULL) return NULL;
14      if (CAS(&s->top, tmp, tmp->next)) {
15        return tmp->elem;
16      }
17    }
18  }
```

Figure 2.6. A partial implementation of a Treiber stack. Using this stack concurrently is safe from data-races.

if two threads are just about to pop the same node, only one of them can succeed with the `CAS`. When the first thread has successfully updated the top node, the next `CAS` will fail, as the speculation from Line 12 is no longer valid.

In order to reason about the correctness of the code in Figure 2.6, the programmer needs to look at *all* the code that accesses the data structure, and assume that all this code may be executed concurrently at any point in time. Using atomic operations like `CAS` to update shared memory does not automatically mean that concurrency errors go away.

As an example of the kind of subtle bugs that can occur when implementing concurrent data structures, Figure 2.7 shows a function for extracting the second node from the stack seen in the previous examples. In a sequential setting, this implementation is correct, and several threads can even run `pop_snd` concurrently without problems. However, introducing this function breaks the internal consistency of the data structure when run concurrently with `pop` from Figure 2.6. Consider the following interleaving of threads `t1` running `pop` and `t2` running `pop_snd`:

```
void *pop_snd(struct stack *s) {
  while (true) {
    struct node *top = s->top;
    struct node *tmp = top->next;
    if (tmp == NULL) return NULL;
    if (CAS(&top->next, tmp, tmp->next)) {
      return tmp->elem;
    }
  }
}
```

Figure 2.7. A function for extracting the second node from the stack seen in Figure 2.6. Concurrently running pop_snd and pop may result in the same node being popped twice.

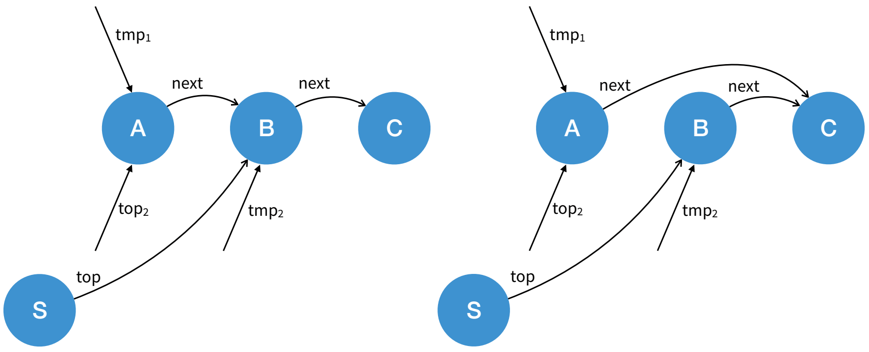

Figure 2.8. The state of the stack after running the `CAS` in pop, and after running the `CAS` in `pop_snd`.

Time	t1	t2
1	read `s->top` into `tmp`$_1$	
2		read `s->top` into `top`$_2$
3		read `top->next` into `tmp`$_2$
4	`CAS(s->top, tmp`$_1$`, tmp`$_1$`->next)`	
5		`CAS(top`$_2$`->next, tmp`$_2$`, tmp`$_2$`->next)`

The state of the stack after time step 4 is shown in the left side of Figure 2.8. The stack's `top` node is the same node as `tmp`$_2$, the node just about to be popped by the thread running `pop_snd`. When the `CAS` in this function is run, the state of the stack is not actually changed as the node being updated (A) has already been popped from the stack by the thread running `pop` (right side of Figure 2.8). Even though thread `t2` just successfully popped node *B* and may read its element, the next thread that runs `pop` will successfully pop the same node!

An interesting observation is that the bug in this case is not caused by data-races; all operations on shared memory are atomic operations. Instead, this is an instance of a more general class of concurrency errors known as *race conditions*, where an error is caused by an unforeseen interleaving of operations. Another example of a race condition not caused by a data-race is when an error is caused by two threads acquiring locks in an unexpected order. In the bug introduced by `pop_snd`, the race condition leads to a potential data-race as the two threads popping the same node gets access to the same memory (the `elem` reference of the node), which they may subsequently try to update without synchronisation.

Just as when programming with locks, the programmer is left with little or no support from the compiler or programming language to get fine-grained concurrency algorithms like the Treiber stack in Figure 2.6 correct. Such algorithms are arguably even harder to reason about than locks, as there is no mutual exclusion; the programmer must always keep all possible interleavings of other threads in mind. Another contribution of this thesis is a type system that prevents bugs like having two threads believing that they successfully popped the same node from a data-structure. The type system tracks the permissions of each memory address, and makes sure that at most one alias of the same memory may be used to update that memory in a non-atomic fashion. While it cannot guarantee correctness of an implementation, it lifts the burden of certain classes of concurrency errors from the shoulders of programmer. This type system is outlined in Chapter 5 and detailed in Paper III.

Non-Blocking Algorithms

This section overviews some other important correctness properties of fine-grained concurrent algorithms. The motivation for using fine-grained concurrency algorithms without using locks is to reduce the time that threads spend waiting for a resource to become available. In general, an algorithm is *non-blocking* if any thread can be suspended mid-execution without hindering the progress of the other threads [69]. An algorithm that uses locks is generally blocking, since suspending a thread that is holding a lock will force other threads to wait indefinitely for the lock to be released.

More specifically, an algorithm is *lock-free* if the algorithm makes global (meaningful) progress in a finite number of steps, regardless of the progress of individual threads [69]. Unfortunate scheduling may hinder the progress of a single thread, but there will always be some thread that makes progress. An algorithm is *wait-free* if each thread is guaranteed to locally make progress in a finite number of steps. The Treiber stack from Figure 2.6 is an example of a lock-free, but not wait-free, data structure: if two threads are continuously popping from the stack, scheduling may in theory cause one of the threads to always fail its `CAS` operation (meaning that the algorithm is not wait-free), but this will mean that the other thread has succeeded and the algorithm has made global progress (meaning that the algorithm is lock-free).

It is important to note that, although the terminology suggests it, a program without locks is not automatically lock-free. An algorithm without locks can still be designed so that global progress depends on the progress of a single thread. A simple example would be a version of `pop` from Figure 2.6 which replaces the top reference with `NULL` before replacing it with the successor node. If the thread was to be suspended between these operations, it would prevent all other threads from making meaningful progress, meaning that the algorithm would no longer be lock-free.

Other than never having threads wait for each other, a non-blocking algorithm needs to be correct with respect to some specification, even when run concurrently. On a high level, the expected property is that the concurrent version of an algorithm does not allow behaviours that are not present in the sequential version of the algorithm. For example, the stack implementation in Figure 2.5 allows two threads to pop the same node from the stack, which could never happen if the two operations were run sequentially.

In Section 2.1.2, Figure 2.4 showed an example of an algorithm fulfilling the property of *serialisability*, where the concurrent version of a program is the same as if all operations were executed sequentially by a single thread. Another correctness condition that is often used for reasoning about concurrent systems is *linearisability*, which states that each operation appears to take global effect at a single point in time, known as the *linearisation point* of the operation [85]. For the stack in Figure 2.6, the linearisation point of the `pop` operations is where the `CAS` happens. In contrast, the stack in Figure 2.5 is not linearisable as it is possible for two threads to successfully pop the same node, meaning it is not possible to find a linearisation point for this operation.

Another example of a subtle bug that can appear in `CAS` based algorithms is what is known as the *ABA problem*. This occurs when a thread speculatively reads some value, which is then updated by another thread and subsequently restored to its original value. When the first thread continues, it appears as if the value has not changed, allowing *e.g.,* a `CAS` to succeed when it should not. For example, in the stack of Figure 2.6, a thread T performing a `pop` may read the `top` field (the address of a node N_1) and its successor (the address of a node N_2) right before another thread successfully pops N_1, followed by popping its successor N_2. If these nodes are deallocated, chances are that the same memory addresses will be reused when allocating other nodes for the stack. If the memory for N_1 is reused for a new node pushed to the stack, the first thread T may incorrectly believe that its original speculation was correct, and replace the `top` field by the address of the (now deallocated!) node N_2. This kind of problem can be avoided by deferring the deallocation of nodes until no aliases remain, for example by using hazard pointers [103], or by running the algorithm in a system with automatic garbage collection.

The type system of Paper III does not attempt to guarantee serialisability nor linearisability, but these properties are non the less important in order to understand the background and the related work presented in Chapter 4. The type system is no more susceptible to the ABA problem than other languages, and it can be avoided using the same approaches.

2.1.3 Transactional Memory

The concurrency control offered by locks is inherently *pessimistic*. Taking a lock forces all other threads to wait, regardless of whether these concurrent accesses would be harmful (*i.e.,* cause data-races) or not. For example, it would be safe to concurrently append and prepend elements to a (non-empty) linked list, but if the whole list is protected by a single lock, threads will be forced to synchronise their operations. In contrast, fine-grained concurrent algorithms like the stack in Figure 2.6 are typically *optimistic*, allowing threads to compete for completing their operations without waiting, and retrying when an operation fails (in the case of Figure 2.6 when the `CAS` fails).

```
1  void send_to_printer(document_t *d, int *count) {
2    int pages = ... // Send d to printer
3    if (pages > 0) {
4      atomic {
5        *count = *count + pages;
6      }
7    }
8  }
9
10 int main() {
11   int count = 0;
12   thread_t t1 = fork(send_to_printer(new_document(), &count));
13   thread_t t2 = fork(send_to_printer(new_document(), &count));
14   join(t1);
15   join(t2);
16   printf("%d", count);
17   return 0;
18 }
```

Figure 2.9. An example of two threads using software transactional memory to synchronise access to shared memory.

Another form of optimistic concurrency control is offered by *transactional memory* [84, 129]. The idea of transactions stems from databases [76], where they capture the notion of an operation which either finishes completely or has no visible effect whatsoever. With transactional memory, when a thread performs an operation on some shared memory it records all its reads and writes in a log. At the end of the operation, the thread can verify if the locations in the log were updated concurrently by some other thread, and if so, roll the changes back and restart the operation. If there were no conflicting accesses, the changes in the log are *committed* to memory, atomically making the changes visible to other threads. In this way, transactional memory gives the illusion of having all transactional operations be atomic; a thread can never observe the intermediate result of another thread's operations. Of course, this comes at the price of the overhead of logging reads and write.

Figure 2.9 shows the program from Figure 2.2, in an imagined language with support for transactions in the form of **atomic** blocks. When a thread executes the **atomic** block on Line 4 it starts a transaction, and on the next Line records that it has read from and written to the memory pointed to by `count`. At the end of the block, the new value of `count` will only be committed to memory

```go
package main

import "fmt"

func sendToPrinter(d *Document, c chan int) {
  pages := ... // Send d to printer
  c <- pages
}

func main() {
  c := make(chan int)
  go sendToPrinter(new(Document), c)
  go sendToPrinter(new(Document), c)
  success1 := <-c
  success2 := <-c
  count := success1 + success2
  fmt.Println(count)
}
```

Figure 2.10. A Go program implementing the program from Figure 2.3, but with channels instead of locks.

if there was no concurrent updates to the value, otherwise the transaction will roll back and retry from Line 4.

2.1.4 Alternatives to Shared Memory

While this thesis focuses on alias control when programming with shared memory in a concurrent setting, it is important to note that shared memory is not a requirement for concurrent programs. In systems where the processing units are distributed across different physical machines, it may not even be possible for threads to share memory in any meaningful way. This section briefly overviews two alternatives for how threads may communicate without using shared memory: channels and message passing using MPI.

A *channel* is a construct that allows point-to-point communication between threads [118]. A channel has two ends, and a value written to one end of a channel by one thread may be read from the other end by another thread. Channels thus provide a more abstract way for threads to share values than directly sharing memory addresses. In a distributed setting, the channel may be implemented by using sockets connected over the network.

Figure 2.10 shows a version of the program from Figure 2.3 using channels instead of locks. The example is written in Go [72], where channels are primitives of the language. Reading from or writing to a channel blocks the thread until there is a thread performing the opposite operation in the other end[3]. Go additionally has built-in support for dynamic data-race detection (concurrent writes to channels are not considered races). The `main` function starts by creating a channel and sending it to the two "goroutines" (Go's lightweight threads) spawned on Lines 16 and 17. The goroutines run the `sendToPrinter` function which ends with writing the number of printed pages to the channel (Lines 8 and 10). The spawning goroutine waits for two values to arrive in the channel (Lines 18 and 19) and then sums them up. The channel thus serves both as a mechanism for communication of values and synchronisation between threads.

A similar approach to communication, but without explicit channels, is found in *MPI* (short for *Message Passing Interface*) [108]. MPI is a standard for communication via message passing. It has been implemented, fully or partially, for several languages, including C, Java, C# and Python. In MPI, processes (threads) are organised in a virtual topology which is either a Cartesian grid or a graph, and messages can be sent to individual processes or to several processes based on the current topology (*e.g.*, to all adjacent processes in the grid). This makes MPI suitable for parallel processing of structured data, for example matrix operations. There is some overhead when copying and distributing data across the topology, but this is unavoidable when the underlying hardware is not a single shared memory machine.

The concept of communicating via message passing is something that also appears when programming with actors. The message queue of an actor also is similar to the channels of *e.g.*, Go. Further similarities and differences between actors and channels is discussed in work by Fowler *et al.* [68].

2.2 The Actor Model

Another popular concurrency model is the actor model [4, 16, 86]. An actor can be thought of as an independent process, logically similar to a thread, running concurrently with other actors. Each actor has an address, and knowing

[3]Go also features buffered channels where writes do not block.

```
-module(m).

send_to_printer(Document, Pid) ->
    Pages = ..., % Send d to printer
    Pid ! {self(), Pages}.

main() ->
    spawn(m, send_to_printer, [#document{}, self()]),
    spawn(m, send_to_printer, [#document{}, self()]),
    receive
        {_, Count1} ->
            receive
                {_, Count2} ->
                    io:format("~w~n", [Count1 + Count2])
            end
    end.
```

Figure 2.11. An Erlang program implementing the program from Figure 2.3, using selective message receives to synchronise the actors.

the address of an actor allows sending messages to it. Message sends are asynchronous, and messages are buffered in the message queue of the receiving actor until they are read. In addition to sending and receiving messages, an actor may spawn new actors.

The concurrency model based on threads is an extension to sequential programming, as each thread can be understood as a smaller sequential program. Any sequential programming language can be turned into a concurrent one by adding support for spawning new threads. In contrast, the actor model is a programming model that is naturally concurrent. The actor model is designed to express computation by spawning actors and having them communicate via asynchronous message passing.

Figure 2.11 shows an Erlang version of the program from Figure 2.3. The `main` function spawns two actors running the `send_to_printer` function, and then waits to receive two replies (in any order). The address of the spawning actor is passed to the two `send_to_printer` actors (the argument `Pid`) so that they know where to send the replies.

Actor systems are often classified in two different ways based on how messages are sent and received: actors and *active objects* [21]. With traditional actors there is typically no limitation on what messages can be sent to an actor, and an actor can choose to selectively receive messages at any point during execution. Selective receive means that an actor can choose to receive any message in its message queue, regardless of the order that they arrived (like in the `main` function in Figure 2.11). Active objects on the other hand can be understood as Java-style objects[4] with a fixed interface, but with their own thread of control. Calling a method on the object enqueues a message in a message queue, where it is eventually picked up by the object's thread, which calls the corresponding method. An active object has no way of selectively receiving messages or receiving messages in the middle of a method call.

In practice, these terms are not used strictly: languages that use active objects as defined above are sometimes also referred to as "actor languages". For this thesis, the distinction is not very important as the focus is not on how messages are sent and received between actors/active objects. In order to not add to the confusion, a language that calls itself an actor language will be referred to as an actor language in the remainder of this thesis (and vice versa for active objects), regardless of how messages are handled.

A more interesting topic for this thesis is how data is shared between actors. The actor model simplifies concurrency by adopting message passing, where synchronisation happens naturally, and allows reasoning about each actor in isolation. However, if two actors can share and update the same data concurrently, values may change underfoot, and sequential reasoning for individual actors is lost. Just like when programming with threads, avoiding data-races is paramount when writing correct concurrent programs.

Some languages (*e.g.,* Erlang [12]) enforce that message payloads are copied on send, avoiding any sharing. Other languages, especially those implemented as libraries on top of existing languages (*e.g.,* Akka [5]) do not prevent unsafe sharing and thus leave it up to the programmer to ensure the absence of data-races (such languages may also mix actors with other concurrency primitives, *e.g.,* threads [134]). Some languages employ a type system which prevents unsafe sharing, for example by only allowing sharing of data that is safe for the receiver to access (*e.g.,* Kilim [130]). Section 2.2.2 gives an overview of some existing actor languages.

Regardless of whether an actor language allows data-races or not, programs written in it may still have concurrency errors due to race conditions, for ex-

[4]The original formulation of active objects is as a design pattern for object-oriented programming [97].

```
active class Worker
  def sendToPrinter(d : Document) : int
    val pages = ... // Send d to printer
    return pages
  end
end

active class Main
  def main() : unit
    val w1 = new Worker()
    val w2 = new Worker()
    val f1 = w1 ! send_to_printer(new Document())
    val f2 = w2 ! send_to_printer(new Document())
    val count = get(f1) + get(f2)
    println(count)
  end
end
```

Figure 2.12. An Encore program implementing the program from Figure 2.3, using futures to synchronise the active objects. A method in an **active** class implicitly returns a future when called asynchronously.

ample when a program behaves differently depending on in which order two messages are delivered. This thesis does not aim to prevent such race conditions, but only to ensure that data shared between actors is not subject to data-races. Chapter 6 details the implementation of the Kappa type system in the active object language Encore, where it facilitates safe sharing and transfer of data between active objects.

2.2.1 Structured Actor Programming with Futures

Actors communicate by passing messages. This is a one-way, asynchronous communication channel, where the sender is the writer and the receiver is the reader. Sometimes, an actor sending a message may want to wait for a reply from the receiving actor before continuing its execution, similar to how function calls work in procedural languages. In a language where actors can selectively receive messages, the sender can pass its own address with the message and then wait for the receiver to reply with another message. In languages without selective receives, this kind of synchronisation needs additional mechanisms. This section presents one such mechanism, called a *future* (or sometimes a *promise*) [15, 100].

A future acts as a placeholder for a value that will be available at some later time. An actor may block on a future, waiting until the future is *fulfilled* and the value is made available. When a reply is required immediately, a message send can directly return a future which will later be fulfilled by the receiving actor. The caller can choose to block on the future immediately, perform some computation of its own before waiting for the result of the message send, or pass the future on to another actor without first waiting for the result. Just as with locks, having two actors blocking on futures which each are supposed to be fulfilled by the other actor will result in a deadlock. Similarly, an actor will deadlock if it blocks on a future that it is expected to fulfill itself.

Figure 2.12 shows an Encore version of the program from Figure 2.3, which uses futures to get the return values from asynchronous calls. The `main` method creates two active objects of type `Worker`, and then calls the method `sendToPrinter` on them asynchronously. In Encore, asynchronous calls implicitly return future values, which can be read (blocking if needed) by using the `get` operation. Here, futures are used both for synchronisation and passing values.

Like "actor" and "active object", the terms "future" and "promise" are not used consistently in the literature. Sometimes they are used interchangeably, and sometimes the two are seen as different views of the same structure, where one is the part that is read and the other is the part that is written (*i.e.,* fulfilled). Again, this thesis uses the terminology of the referenced work.

2.2.2 Actor Languages

This section overviews some existing languages and frameworks which use actors or active objects and discusses how they deal with shared mutable state. One of these languages is Encore, whose type system is part of the contributions of this thesis (*cf.,* Chapter 6). As mentioned earlier, the terminology varies slightly between languages, and this section uses the terminology from the referenced language, clarifying when necessary, to avoid adding to the confusion.

Erlang

Erlang [12] is an actor language that was originally developed at Ericsson for programming telephone switches. It has since been used for other highly concurrent systems [11]. In Erlang, each actor is a stateless functional process (although state can be emulated using recursive function calls) which does not share any data with other actors[5] . This is ensured by copying data when sending messages. Giving each actor its own private heap which is never shared also allows garbage collection to happen concurrently, without having to stop more than one actor at a time. Actors may selectively receive messages at any time, and there are no restrictions on the type of messages an actor may receive.

Elixir [63] is another actor language which runs on the Erlang virtual machine, and which therefore shares many of the properties of Erlang.

Akka and Scala

Akka [5] is an actor library for the Java virtual machine. It is written in Scala, but has bindings for both Scala and Java. Actors communicate via message passing, but may also use the futures from Scala's standard library. Akka supports both traditional actors with an untyped interface and active objects (called *typed actors*) which define which messages they may receive. As the code implementing the behaviour of actors is written in Scala or Java, sharing mutable state is possible and is subject to data-races. In practice, Scala programmers also mix the actor model with other means of concurrency, *e.g.,* threads [134].

Scala itself also provides an actor library of its own [81]. While Scala is not data-race free, there have been efforts to implement type systems based on capabilities which can give guarantees about aliasing and data-races [80, 82]. These type systems are discussed further in Chapter 4.

Kilim

Kilim [130] is an actor framework for Java. In contrast to Akka and Scala actors, Kilim can guarantee that mutable data is never shared between actors. It does this by utilising Java's annotations framework and performing additional static analysis after Java's regular compilation. In Kilim, messages are treated as a special kind of data which is restricted to be tree-shaped, *i.e.,* have no internal aliasing. A `cut` operation destructively extracts branches from such a tree. Additionally, a message may only have a single heap alias at any point in time

[5]There are exceptions, such as the `ets` (Erlang term storage) module, which implements a dynamic table that may shared between actors. Its implementation however ensures that operations are atomic, so it is not subject to data-races.

(*cf.,* Section 4.1.1). This allows messages to be efficiently and safely transferred between actors, without introducing any sharing. A special marker interface `Sharable` acts as an escape hatch from safety checking, allowing programmers to identify classes of objects that may be safely shared.

A later extension of Kilim, called Ownership-Kilim (or O-Kilim) [78], allows arbitrary aliasing inside a message by dynamically tracking the owner, a message or an actor, of each object. Like in Kilim, a special operation is used to extract an object (and all the objects reachable from it) to or from a message, dynamically changing the owner of the object(s). Using a reference to an object whose ownership has been transferred to another actor or message results in an exception being thrown.

Creol

Creol [57, 91] is an object-oriented language with active objects. Method calls are asynchronous and return futures. A distinguishing feature of Creol is that an active object may have several executing method calls (with at most one running at a time). A special **await** operation allows suspending the execution of the current message and resuming it at a later stage (similar to cooperative multitasking in operating systems [64]). For example, when waiting for a future to be fulfilled, the programmer can chose to block, suspending the active object until the future is fulfilled, or **await**, allowing other messages to be handled while waiting. Reducing the time spent blocking opens up for more parallelism, but also requires the programmer to think about when it is safe to delay finishing an operation.

Joelle

Joelle [115] is an object-oriented programming language with active objects. Method calls to active objects are asynchronous and return futures. In addition to parallelism stemming from active objects running concurrently, Joelle supports *intra-object* parallelism, both by allowing messages to be processed in parallel, and by having the internal thread of control spawn new threads for doing parallel computations. This parallelism, as well as the sharing of objects between active objects running concurrently, is safe from data-races thanks to Joelle's type system, which is discussed further in Chapter 4.

E, Vats and Far References

E [106] is an object-oriented language implementing the *object-capability model* [60]. In this model, an object capability is an *unforgeable* reference to an object, coupled with the permission to send messages to (call methods on) this object. The only way for an object o_1 to gain access to another object o_2 is if o_2's object capability is passed to o_1 in a message (and not *e.g.*, by reading a reference from a field or a global variable). This restricted version of object-oriented programming facilitates the so called *principle of least authority*. An object can only ever access the objects that it is explicitly sent.

The similarity to actor systems comes from the fact that each object belongs to a *vat*, where a single thread of control handles messages sent to the objects belonging to that vat. A *near reference* is a reference between two objects in the same vat, allowing message sends to be handled synchronously (like a normal method call). A *far reference* is a reference that crosses the boundary of a vat, and which only allows asynchronous message sends (similar to message sends in an actor system). A message send through a far reference returns a promise (*cf.*, Section 2.2.1).

Vats are similar to CoBoxes [126], which also permit far references. CoBoxes additionally support processing several messages concurrently (with at most one running at a time), similar to how Creol allows yielding control to the scheduler in the middle of a method call. Far references also appear in AmbientTalk [140], but to objects owned by actors.

While actors typically rely on *isolation* to avoid races, vats in E rely on *delegation*. Far references refer directly to objects inside other vats, but do not access these objects directly. Instead operations are implicitly delegated to the owning vat, which eventually handles the message. One of the contributions of this thesis is a construct that allows actor programs to safely switch between relying on isolation and relying on delegation. Actors may send private data to other actors, but any operations on these objects will be delegated to the owning actor. This construct is outlined in Chapter 6 and detailed in Paper IV

Pony

Pony [119, 124] is an object-oriented language which uses actors (active objects, by the definition of this thesis) to achieve concurrency. Pony uses a capability-based type system to ensure that there are no data-races. Data is either uniquely owned by a single actor at a time, or is immutable and therefore safe to access concurrently. The type system of Pony is co-designed with a concurrent garbage collection scheme [54], which utilises the guarantee of data-race freedom to allow actors to collect garbage in their own local heaps without having to stop other actors, or copy data on message sends.

Pony's type system is discussed further in Chapter 4. In addition to the guarantee of data-race freedom, there is no concept of **null** and exceptions must always be handled. There are no futures, but since actors cannot synchronise, there is also no way for actors to deadlock.

Encore

Encore [29] is an object-oriented programming language which uses active objects as a means to achieve concurrency. It uses futures to support returning values from asynchronous method calls. Part of this thesis' contributions is the implementation of Kappa as the type system for Encore, which allows active objects to share data without data-races. The integration of Kappa and Encore is detailed in Chapter 6.

With the necessary background on concurrency and concurrency control covered, we continue by discussing object-oriented programming, which is the programming paradigm that Kappa was designed for.

3. Object-Oriented Programming

The Kappa type system ensures that concurrent programs are free from data-races, but also aims to facilitate a familiar object-oriented programming style. This chapter therefore reviews the distinguishing features of object-oriented programming, focusing on subtyping and code reuse (Section 3.1) and encapsulation (Section 3.2). Section 3.3 covers the topic of object calculi, a useful tool for formally reasoning about the behaviour of object-oriented programming languages. In addition to the mentioned language features for concurrency control, the development of Kappa gave rise to an object calculus called OOlong, which is specialised for concurrent object-oriented programming. OOlong is covered in detail in Paper V.

Although not a necessity for object-oriented programming, the focus of this chapter (and this thesis) is on statically typed, class-based languages. As the contributions of this thesis revolve around type systems for alias control, dynamic languages like Python, JavaScript and Smalltalk mostly fall out of scope.

3.1 Subtyping and Code Reuse

The most important feature of object-oriented programming is arguably the idea of subtype polymorphism: a piece of code that handles objects of type `A` can also be used with objects of type `B` if `B` is a subtype of `A` (this is know as the *Liskov substitution principle* [99]). For example, a Java method that takes an argument of type `List` can be called with an instance of type `LinkedList` or an instance of type `ArrayList` since they are both subtypes of `List`. Subtype polymorphism fits nicely together with the object-oriented philosophy that objects carry their own behaviour. The method in the example above only needs to know that its argument knows how to behave like a `List`, but does not need to know exactly how this behaviour is implemented. Method calls are dispatched dynamically to the corresponding implementation, based on the runtime type of the argument.

In many mainstream languages, such as Java, C# and C++, subtyping can be introduced via *inheritance*. A class B inheriting from another class A will automatically include all the methods and fields of A. Methods may be overridden to implement new behaviour, but all method names in A are guaranteed to be present in B. The methods that are not overridden can reuse the implementation of the original methods in the superclass. This way, inheritance enables code reuse in subclasses, in addition to the code reuse enabled by subtype polymorphism. Coupling inheritance and subtyping is not a requirement however. For example, Go [72] uses inheritance for code reuse, but does not equate subclassing with subtyping. Similarly, C++ supports private inheritance, which gives code reuse without subtyping. Dynamic languages such as Smalltalk, Python and Ruby support inheritance, but implement subtyping structurally rather than nominally: an instance of class A may be substituted for an instance of class B in context c if all of A's attributes accessed in c are also in B, regardless of whether or not B inherits from A[1].

Coupling inheritance and subtyping introduces design issues, as classes related through subtyping need to be arranged in a hierarchy. The higher up in the hierarchy a class C is, the more subclasses there are which have a dependency on C, meaning that any changes to C will need to be compatible with all of its different subclasses (this is known as the *fragile base class problem*). Implementing a hierarchy where A is a subtype of both B and C is also problematic. Either B needs to be made a subclass of C (or vice versa), which may not make sense for other uses of the classes, or A needs to inherit from both B and C, which introduces complexity and ambiguity when the same attributes are inherited from both classes (for example, the infamous *diamond problem* [102]). Many class-based languages disallow multiple inheritance to avoid these issues; C++ being a notable exception.

In order to get around some of the issues of inheritance-based subtyping, some languages complement classes with *interfaces*. An interface is a collection of method signatures, and a class that *implements* an interface must provide implementations for each of these methods. If a class C implements an interface I, the class type C is a subtype of the interface type I, and can thus be used wherever an I is expected. In the Java example in the beginning of this section, List is an interface that is implemented by both LinkedList and ArrayList.

[1]Structural typing can also lead to the famous example where an Artist object ends up in a duel with a Cowboy object, because they both support the method draw.

In contrast to class inheritance, having a class implement many interfaces is unproblematic. Since there are no implementations in the interfaces, including two interfaces with overlapping method signatures simply means that the class has "promised twice" to implement a method with that signature; there is no ambiguity as to which implementation should be used. Interfaces let programmers define subtyping relations through composition instead of tying subtyping to the class hierarchy.

Interfaces allow programmers to decouple the type of an object (its available behaviour) from its implementation, but they do not provide code reuse like inheritance (Java 8 remedies this to some extent, see below). *Mixins* [27] have been proposed as an alternative to inheritance that also enjoys the compositional property of interfaces. A mixin is an abstract class that can be composed ("mixed in") with a class in order to create a new class that also includes the behaviour of the mixin. A class may include several mixins, and conflicts are resolved by including mixins one at a time and overriding any conflicting methods from earlier mixins.

Finding the correct order to include mixins may be hard, or in certain cases even impossible. To address this, *traits* have been proposed as a more compositional mechanism for code reuse [127]. A trait *provides* a set of methods and may *require* the presence of a set of fields and methods in the including class[2]. This allows the methods provided by the trait to use fields and methods that will be provided later by the including class. A class may include several traits, but contrary to mixins, the order of inclusion does not affect which methods are included (in fact, order does not matter at all). Instead, any ambiguities must be resolved by the programmer, either by specifying which method should be used, or by providing an overriding implementation in the class.

Figure 3.1 shows a simple example of a Scala program for building and evaluating syntax trees for simple arithmetic expressions. All classes implement the trait `Expr`, which requires a method `eval`. In the class `Constant` this method is provided by the class. In the classes `Addition` and `Subtraction`, the same method is provided by the included trait `Binary`, which in turn requires fields for the operands of the binary operation, as well as a function for evaluating the current expression. The same program can be written using classes and

[2]In the original formulation of traits, only methods could be required by traits [127]. Field requirements have been added later to other languages, *e.g.*, Scala [125].

```
trait Expr {
  def eval() : Int
}

class Constant(val value : Int) extends Expr {
  def eval() = value
}

trait Binary {
  val loper : Expr
  val roper : Expr
  val op : (Int, Int) => Int
  def eval() = op(loper.eval(), roper.eval())
}

class Addition(val loper : Expr, val roper : Expr)
      extends Expr with Binary {
  val op = _ + _
}

class Subtraction(val loper : Expr, val roper : Expr)
      extends Expr with Binary {
  val op = _ - _
}
```

Figure 3.1. A Scala program using traits for subtyping and code reuse. **class** `C` **extends** `A` **with** `B` is Scala syntax for having `C` implement traits `A` and `B`. The order of `A` and `B` does not matter. `_ + _` is Scala syntax for an anonymous function which adds its two arguments.

inheritance by making `Binary` a subclass of `Expr` (`Expr` would then be an abstract class to be able to defer the implementation of `eval`), but with traits this coupling is not necessary.

In practice, the line between interfaces and traits has become blurred over time, possibly due to the influence of traits and similar concepts. Since Java 8, interfaces may also supply default implementations of methods, which lets interfaces provide code reuse akin to what traits provide [143]. Interfaces in C# can include signatures of *properties*, which are fields with explicit getters and setters [33].

Kappa uses traits for both subtyping and code reuse, leaving out class inheritance altogether. A contribution of this type system is that it extends the code reuse offered by traits across different concurrency scenarios. The same traits can be reused to implement different versions of a data structure, depending on how and if the data structure is accessed concurrently (*cf.,* Section 5.1.1).

3.2 Object Encapsulation

Another cornerstone of object-oriented programming is the concept of *object encapsulation*; the idea that an object can be internally represented by an aggregate of other objects, and that these objects will never be referenced from outside the aggregate. The canonical example is a linked list that is represented by a chain of `Link` objects, which will only ever be referenced from within the list itself. Encapsulation is an important tool for abstraction and modularity as it lets programmers reason about the invariants of an object by looking at that object's implementation in isolation; if the links of a list could be accessed and changed from the outside, reasoning about *e.g.,* the shape of the list would be considerably harder.

In a concurrent setting, encapsulation comes into play when reasoning about what kind of concurrency control is necessary for accessing a piece of data. If a list is shared between threads, properly encapsulating its links implies that they will never be accessed in a racy fashion as long as accesses to the list object itself is synchronised. In a lock-free implementation (*cf.,* Section 2.1.2), proper encapsulation of the links means that only the methods in the `List` class needs to be considered when reasoning about the interaction of concurrent operations.

Unfortunately, in the mainstream languages of today, the burden of maintaining encapsulation rests on the shoulders of the programmer. Many object-oriented languages supports controlling the visibility of fields from outside of the class; declaring a field as private makes it inaccessible from outside the class. While this helps signal the *intent* of the programmer, it does not prevent writing a method which returns the contents of a private field. *In mainstream languages, encapsulation is a pattern and not a language feature.*

Section 4.1.2 discusses techniques for statically enforcing encapsulation, some of which have also been incorporated in Kappa. Encapsulation plays an important role in Kappa for simplifying the implementation of traits and allowing code reuse across concurrency scenarios (*cf.,* Section 5.1.1).

3.3 Proving Properties of Object-Oriented Languages

When developing type systems and other language features, it is desirable to be able to prove that they fulfill the properties for which they are designed. However, carrying out proofs about the semantics of full fledged languages executing on actual hardware is often too complex to be feasible[3]. Instead, a common approach is to develop a simplified formal model of a programming language and its execution environment, and prove the properties of the new language features in this context. The rationale behind this is that a correctness proof carries over to a real language implementation as long as all the relevant details are captured in the formal model[4].

By far the most common formal model of computation is the lambda calculus, which forms the foundation of functional programming languages [18]. For example, the semantics of the language features presented in Paper IV is explored using a version of the lambda calculus, extended with actors. The lambda calculus is purely functional, so a common extension when modelling imperative languages is to add reference cells which may be updated with side-effects. When modelling object-oriented languages, it is useful to also extend the type system with record types which can describe the interface and state of an object.

Even though it is *possible* to use the lambda calculus to model object-oriented languages, the indirection required means that there is a disconnect between the formal model and the actual language being modelled. To bridge this gap, object-oriented languages are often modelled using an *object calculus*, which have primitive support for the features common for object-oriented languages (*e.g.*, classes and subtyping). The purpose of an object calculus is to be to object-oriented languages what the lambda calculus is to functional languages [1].

Due to the popularity of Java, there have been several candidates for object calculi which model a core subset of Java, *e.g.*, `ClassicJava` [67], Featherweight Java [89], Lightweight Java [131], Middleweight Java [19] and Welterweight Java [116]. They differ in the choice of features modelled, but have all been used as a basis for reasoning about extensions for Java. Paper V further discusses the sim-

[3]WebAssembly is a recent example of a language which was designed with a formal semantics from the start [79]

[4]Recent work has showed that the type systems of both Scala and Java are unsound (they permit writing functions that changes the type of a value to any type), even though Java has been formalised and proven sound several times [10]. This is an example of where the formal models have failed to include all relevant details.

ilarities and differences between these Java calculi and presents an alternative object calculus called OOlong. OOlong does not aim to model Java in particular, but object-oriented programs in general. As argued in Section 3.1, subtyping is a fundamental concept in object-oriented programming but need not (and should not!) be coupled with class inheritance. In contrast with the Java calculi, OOlong uses interfaces for subtyping, omitting class inheritance completely. Additionally, OOlong is aimed at concurrent programming, natively supporting fork/join style parallelism and synchronisation using locks. OOlong was developed in the context of Kappa, whose formal semantics is based on (an extended version of) OOlong.

3.3.1 Mechanised Semantics and Proofs

Carrying out proofs by hand can be a tedious and error-prone process, and proofs about language semantics are no exception. Even when a proof has been completed and has been inspected closely enough to be trusted, any changes to the premise of the proof (*e.g.*, a change in the semantics of a language model) means that the whole proof needs to be inspected again in order to find where (and if) changes to the proof is needed. For large proofs, this can be a significant undertaking.

The burden of working with proofs can be alleviated somewhat with the help of an *interactive theorem prover* (or *proof assistant*). These are programs or programming languages that are based on the view of theorems as a special kind of data type, which can be combined in well-defined ways to form new theorems. A program that handles this data thus serves as the proof of the resulting theorem, and this proof is valid if and only if the program compiles (*i.e.*, type-checks). Such a program is known as a *mechanised proof.* If the premises of a proof changes, re-checking the proof can be done automatically, and only the parts that the proof assistant no longer accept need attention. Rather than trusting the person who writes the proof, one only has to trust the implementation of the proof assistant (whose core logic implementation is usually quite small, making them trustworthy).

There are several theorem provers in use by researchers today. Isabelle/HOL is a theorem prover based on Standard ML which supports proofs in higher-order logic [112]. It has been used to prove famous theorems such as Gödel's incompleteness theorem [117] and the prime number theorem [14]. Lightweight Java was mechanised and proven sound in Isabelle [131]. Coq is a language and theorem prover based instead on dependent types and the calculus of constructions [55]. Famous projects in Coq include a proof of the four-color theorem [73] and the CompCert compiler [98], which is a fully verified C compiler. Featherweight Java (extended with mutable state) has been mechanised and proven sound in Coq [101], and so has a later extension of Lightweight Java [58].

In order to facilitate reliability, reusability and extensibility, OOlong, the object calculus presented in Paper V, has been fully mechanised and proven sound in Coq. The formalism of the system presented in Paper IV has also been fully mechanised, including the two variations of the semantics.

This chapter, together with Chapter 2, has described the context in which Kappa was developed and the problems it aims to solve. We now continue by discussing existing approaches for solving these problems.

4. Related Work

Kappa stands on the shoulders of giants by incorporating and extending ideas from many existing proposals for alias control. This chapter overviews some of these techniques for alias control, focusing in Section 4.1 on programming language features which help programmers to write safe and correct code. As a contrast, Section 4.2 looks at techniques for doing program verification of existing programs.

4.1 Language Features for Alias Control

Programming languages should help programmers write correct code. They should provide tools for structuring code and managing data in an efficient way. While most mainstream programming languages provide support for data abstraction, procedural abstraction, code reuse *etc.*, support for controlling aliasing is only available in limited form, *e.g.,* the **const** annotations in C and C++ (which can be bypassed via casts, unless data is placed in ROM by the compiler) or the **restrict** keyword in C (which does not enforce correct usage). This section goes through some of the techniques for alias control that have come out of research, but have not yet made their way into mainstream languages. These are techniques that are related to the techniques used in this thesis, especially the type systems presented in Papers I–III.

4.1.1 Linear References

A *linear* (or *unique*) reference enjoys the powerful property of having no aliases; it is the *only* reference to the object it points to. The concept has its origin in linear logic [71], where propositions can be "consumed" (as opposed to classic logic, where using a proposition once does not prevent using it again). The terminology varies in the programming language literature, and sometimes a

```
(FREE ⊕ (BUSY ⊗ release))
 ⊗ *acquire(reply(release))
```

Figure 4.1. The behavioural type of a lock. Example taken from [56]

linear value is a value that *must* be used *exactly* once (a value that can be used at most once is then called an *affine* value). In this thesis this distinction is not important, and the terms “linear” and “unique” are used interchangeably to mean a reference without aliases (modulo borrowing and burying, see below). The alternative meaning of “linear” is referred to as “use-once linear”.

An early use of linearity in programming languages was memory management; when a use-once linear value is used, its memory can be safely reclaimed [141]. Similarly, in a language where linear values may be used more than once, the memory of a linear value may be collected as soon as the reference to it goes out of scope [95]. Linearity also allows for *strong updates*, *i.e.,* dynamically changing the type of a value or an object (*e.g.,* [61]). In the same vein, linearity is often used when implementing *type state* (*e.g.,* [7, 105]) or *behavioural types* (*e.g.,* [34, 56]), where the state of an object is tracked through its type[1] – the canonical example being tracking whether a file stream is open or closed, thereby preventing errors like reading from a closed stream.

A more relevant example of type state for this thesis is the behavioural type of a lock [56], shown in Figure 4.1. This type states that a lock is either `FREE` or `BUSY` (*i.e.,* acquired). Through the use-once linearity of behavioural types, the specification states that in the latter case, the lock must (eventually) be released. Furthermore, a lock is interacted with through zero or more concurrent `acquire` operations, which must eventually be followed by a `release` of the lock. The details of this example can be found in [56].

In a concurrent setting, linear references give a trivial guarantee of data-race freedom. If a thread holds the only reference to an object, this object cannot be accessed by another thread concurrently; the mere existence of another reference to the object would violate the uniqueness of the original reference. Transferring a linear reference across threads implies a transfer of ownership of the

[1]Linearity greatly simplifies this tracking, but it can also be achieved in the presence of aliasing, for example by dynamically inspecting the type of the object.

object it refers to (*e.g.*, [22, 74]). In an actor system, linear references can be used to implement safe, zero-copy message passing (*e.g.*, [115, 124, 130]).

In any language with linear references, care must be taken to avoid accidentally creating aliases of linear values [24]. The simplest way to achieve this is to require that variables containing linear references are *destructively read* (*e.g.*, [74]), meaning that the original reference is set to **null** as a side-effect of performing the read. A similar technique is to require linear references to be swapped rather than copied (*e.g.*, [82]), so that `x =:= y` sets `x` to `y` and `y` to `x`. Some systems support both these techniques, for example the C++ library `unique_ptr`[2] [137]. Programming in this style quickly becomes tedious though, as linear values must be threaded through the program. For example, a function that takes a linear argument must return it again if the caller wants to use it after the call. Destructively reading variables introduces **null** pointers which the programmer needs to keep track of to avoid runtime exceptions or crashes. An operation as simple as calling a method introduces an implicit alias in the **this** variable, which must be controlled if the receiver is linear.

A static technique for maintaining linearity without destructive reads is *alias burying* [23]. The main idea of alias burying is that aliasing a linear reference is benign as long as the original reference is never used again (this reference is "buried"). After the assignment `x = y`, rather than setting `y` to **null**, the compiler can check that `y` is not used again and throw an error if it is. The problem of threading a linear value back and forth between function calls can be mitigated by using *borrowing* [23]. Borrowing allows temporarily creating an alias of a linear reference, as long as this alias is not reachable once it goes out of scope (*e.g.*, because it was stored on the heap) and the original reference is unreachable for the duration of the borrowing (it is "temporarily buried"). For example, a function that is called with a linear argument `x`, but does not retain it, can be called without using a destructive read; for the duration of the call, the corresponding parameter is the only reachable alias of `x`, and when the call ends and the parameter goes out of scope, linearity of `x` is restored.

A more fine-grained approach for relaxing linearity is found in the work on *disjointness domains* [30]. A disjointness domain is a set of references with the property that each reference is unique *within that set*. Having all references in a

[2] This library adds linear references to C++, but does not prevent the programmer from misusing it, *e.g.*, by initialising a `unique_ptr` with a pointer that is already aliased

single domain permits no aliasing, which corresponds to global uniqueness. A disjointness domain may further be extended with two or more child domains for a limited scope, and references can be moved from a parent domain into *both* child domains. A reference in such a child domain may thus have aliases in sibling domains, but not in the same domain or the parent. When the scopes of the child domains end, references may be moved back from one of the children to the parent domain. This way, borrowing of a variable `x` can be understood as creating two child domains, moving `x` into these domains as `x1` and `x2` respectively, using one of these aliases as the borrowed value (making sure not to touch references in the sibling domain), and finally using the other reference to restore the borrowed value into `x` at the end of the scope of the child domains. Disjointness domains also support patterns like implementing a doubly linked list, which is known to be a linear chain of references both when iterated from front to back and from back to front.

Although its not widespread enough to be called mainstream at the time of writing, Rust is an example of a programming language which uses linearity successfully [123]. In Rust, all references which support mutation of the underlying object are linear. By giving up mutation rights, a reference may be aliased. For concurrent programs, this rules out any data-races. There is support for borrowing, and the compiler tracks the lifetime of all references to completely avoid the need for destructive reads. When a reference reaches the end of its lifetime, the underlying memory is reclaimed. Thanks to linearity, this can be done without any need for a garbage collector or any other runtime machinery. For applications that need aliasing, Rust has an **unsafe** annotation which circumvents the type checker. This is used to implement trusted libraries which allow expressing more relaxed aliasing patterns, such as reference counted shared data, or data protected by locks which may be shared between threads. At the time of writing this thesis, a new version of the web browser Firefox was recently released, which had its C++-based CSS engine rewritten in Rust [65]. By using Rust, the developers could parallelise the code without having to worry about many of the errors that can occur in C++ programs, *e.g.*, data-races, dangling pointers and double frees.

4.1.2 Ownership Types

Section 3.2 discussed the importance of encapsulation in object-oriented programming, but also noted that mainstream languages rely on the programmer to maintain encapsulation. This section overviews the research into *ownership types* and related work, which provide language support for enforcing encapsulation. Before the invention of ownership types, earlier work (*e.g.,* [8, 88]) provided *full alias encapsulation* [114], allowing identifying borders over which no references could pass. The objects within these borders are encapsulated, but can also not refer to objects outside of the borders, directly or indirectly. This allows a list to encapsulate its links, but does not allow these links to refer to elements outside of the links. Ownership types instead provide *flexible alias protection* [114], allowing objects to be encapsulated within some aggregate (*i.e.,* not be aliased from outside of the aggregate), but still allowing outgoing references.

In traditional ownership types [52], each object defines an *ownership context* and also belongs to some context. The objects belonging to the context of another object is said to be *owned* by this object. The owner of an object is tracked via its type. At the start of the program, new objects belong to the root owner **world** (or **norep**). After an object has been created, new objects may be assigned to the ownership context of this object. Ownership types thus partition the heap into a nested structure of owners, and provide the guarantee that references whose types are annotated with different owners may not be aliases. Furthermore, traditional ownership types enforces a property known as *owners-as-dominators*; all paths from outside of an object o to an object owned by o must go via o, or put differently, all paths to an object must go via its owner. This means that the only way to have an effect on an owned object is via a method call on its owner. Objects with the same owner may refer to each other freely.

Figure 4.2 shows the partial implementation and the ownership hierarchy of a program with a linked list (defining the owner `L`). The links of the list are owned by the list and may not be accessed from outside the list. References from the links to elements in **world** are allowed. The special owners `rep` and `owner` refer to the owner defined by the current object and the owner of current object respectively. From the view of the list, the links are owned by `rep`, but from the view of the links they are owned by `owner` (in both cases the keywords refer to the owner `L`).

```
class Link {
  world T elem;
  owner Link next;
  ...
}

class List {
  rep Link first;
  ...
}
```

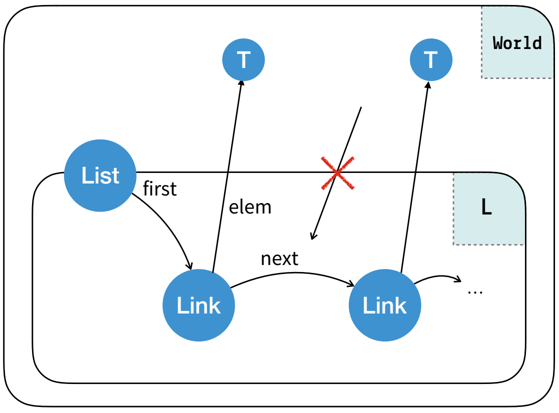

Figure 4.2. Partial implementation of a linked list using ownership types, and the resulting ownership hierarchy. The owners **rep** and **owner** both refer to the owner L in the figure.

The original treatise on ownership types [52] further supports parametric ownership, allowing for example a list which is polymorphic over the owner of its elements. Later extensions and variations of ownership types (*e.g.*, universe types [109] and ownership domains [6]) include supporting objects which have multiple owners [35], allowing objects to selectively share the objects in its representation [6, 144], allowing external read-only references into ownership contexts (enforcing *owners-as-modifiers*) [109], supporting reasoning about the internal structure of an ownership context [36], and allowing mutable references into ownership contexts as long as they are only accessed following a method call to the owner of the context (known as *owners-as-accessors*) [113]. The purpose of these different systems vary from giving strong aliasing guarantees to the programmer, to simplifying program verification [49].

Combining owners-as-dominators with linear references gives a strong property know as *external uniqueness* [50]: holding the only reference to an object o also means holding the only means of accessing the objects owned by o. By moving the reference to o, one also transfers ownership of all the objects (transitively) owned by o [50, 110]. Inside the externally unique aggregate, aliasing is unrestricted, including references to o. If the list in Figure 4.2 was externally unique, there could be at most one reference to it from within the owner **world**, but any number of references to it from the links in L. A variant of external uniqueness known as *separate uniqueness* disallows outgoing reference from the unique object aggregate, thus enforcing full alias encapsulation[3] [82].

In a concurrent setting, ownership types provide a way of excluding data-races by forbidding the existence of certain references. If a thread is guaranteed exclusive access to an object, it also gets exclusive access to the objects owned by that object. In a system with actors (or active objects), ownership types can be used to guarantee that actor-local data is not accidentally shared between actors, *e.g.*, Joelle (*cf.*, Section 2.2.2) [115]. Joelle also uses external uniqueness to transfer ownership of whole aggregates of objects between active objects, and further uses ownership information to infer which parts of an argument to a message which need to be copied and which can be transferred by reference. In a similar vein, Gordon's *isolated types* (*cf.*, Section 4.1.4) uses external uniqueness to transfer object aggregates between threads [74]. In this system the outgoing references from an externally unique aggregate may only refer to immutable data (which is safe to access concurrently).

4.1.3 Effects and Regions

Both linear references and ownership types are *prescriptive* in the sense that they forbid certain references from existing. Other systems take a *descriptive* approach to alias control and allow "dangerous" references to exist as long as the programmer does not use them in an incorrect way. One way to achieve this is by using an *effect system* [111]. An effect system complements a type system by providing an abstract description of the (possible) effects of running an expression. Effects have been used to describe I/O effects (such as opening and closing file streams) [28], tracking memory liveness to avoid dangling pointers [77], and reasoning about control flow. An example of the latter is Java's checked exceptions. When a method declares that it **throws** some exception, this describes a possible side-effect of running the method. Like with checked exceptions, effect systems often suffer from *effect propagation*; a method that calls another method with some effect must declare that effect itself, meaning that effects have a tendency to spread throughout the program.

[3] Separate uniqueness is a special case of external uniqueness, where the objects in an aggregate cannot name any of the owners outside of this aggregate.

In concurrent systems, effects can be used to reason about which data is read and written when executing some expression. As long as two expressions have disjoint effects (any data touched by both expressions is only ever read), running them in parallel is safe from a data-race perspective. In the presence of aliasing, it is not enough to know which variables will be touched by an effect, since an effect on one variable will be visible through any alias of that variable. Instead, *regions* are used to denote the "maximum reach" of an effect. Regions may be explicitly declared by the programmer (*e.g.*, [20, 115]), or be based on some other construct in the language, like stack frames [77] or owners (effects on two non-nested owners will be disjoint) [48].

Deterministic Parallel Java (DPJ) is an extension of Java which adds support for regions and effects [20]. In DPJ, a class may declare regions and associate each field with one of these regions[4]. Additionally, each method is annotated with the effects (`reads` or `writes`) that the method has on each region. By requiring that any two expressions run in parallel have disjoint effects, meaning that they will not race to update the same fields, the language guarantees that programs are *deterministic*, *i.e.,* have the same result regardless of how threads are scheduled. DPJ also supports partitioning of arrays into disjoint parts, making parallel operations on a single array free from data-races as well.

The active-object language Joelle (*cf.*, Section 2.2.2) uses regions and effect annotations in a similar way to DPJ, but with the purpose of allowing safe parallelism within active objects [115]. In addition to allowing programmers to write parallel code in methods of active objects, if two messages in the queue of an active object have disjoint effects, these can automatically be run in parallel. Joelle also supports user-specified abstract effects which can be used to express the allowed interactions of more high-level operations. Abstract effects can either be assigned to owners, or be declared as disjoint (in which case safety is not checked by the compiler), allowing methods with disjoint effects to be run in parallel.

[4]Regions in DPJ can also be nested, making them similar to ownership contexts, but without the encapsulation guarantees (and restrictions).

4.1.4 Capabilities and Permissions

Central to the work in this thesis is the concept of a *capability*, sometimes also known as a *permission*. The terminology is not consistent in the literature, and both terms are overloaded. In this thesis, the terms are used to mean an abstract token associated with an alias (usually tracked in its type) which describes both which operations are permitted on the underlying object (*e.g.*, which methods may be called) and which operations are permitted on the reference itself (*e.g.*, if it may be copied). Capabilities can thus be used to control how the underlying object is accessed, as well as for tracking aliasing information, such as uniqueness or ownership [26].

The rest of this section overviews and compares work related to this thesis, which uses permissions or capabilities to avoid data-races in concurrent programs. Their relation to Kappa is discussed closer in Chapter 5.

Fractional permissions

Fractional permissions capture the notion that it is safe to read a piece of data as long as no other thread is writing it at the same time [25]. Data is created with a full permission, which allows updating the data. This permission may not be duplicated, but may be *split* into several permissions which hold a fraction of the original permission, and which only allows reading the data. The smaller permissions may be further split, but importantly, the individual fractions always "add up" to the original permission. This means that the full permission can be restored by collecting all the fractions, facilitating the pattern of having a single mutable reference, splitting it up into several read-only aliases which may be accessed concurrently, and then restoring the original mutable reference when all aliases have been collected.

An extension to Habanero Java called Habanero Java with Permissions (HJp) uses fractional permissions to ensure that all objects are either referenced by a single mutable reference, or by one or more read-only references, which may be shared between threads [142]. The single mutable reference can be fully transferred between threads, or temporarily split and distributed across threads. Additionally, in order to simplify programming with thread-local data, HJp allows private write permissions which may be duplicated but not shared between threads.

```
isolated IntBox increment(isolated IntBox b) {
  // implicitly convert b to writable
  b.value++;
  // convert b *back* to isolated
  return b;
}
```

Figure 4.3. An example of how isolation can be recovered, taken from [74]

SafeJava

SafeJava is a type system for of Java-like languages which uses a combination of capabilities, ownership types and effect clauses to ensure the absence of data-races and deadlocks [22]. Types can be annotated as **unique**, meaning values must be treated linearly, or **immutable**, meaning values will never change after initialisation. Objects owned by **world** may be shared across threads and are therefore implicitly protected by locks. Thread-local objects are expressed by using the special owner **thisThread**. Importantly, classes can be written parametrically, deferring specifying their "protection mechanism" (*i.e.,* uniqueness, immutabililty, encapsulation *etc.*) until the class is used. This allows the same implementation to be used for different concurrency scenarios, for example reusing the same `List` class as a thread-local list on one place and a linearly referenced list in another.

Isolated Types

Gordon's *isolated types* are used to guarantee safe concurrent programming in an object-oriented setting (the prototype implementation is in C#) [74]. The type system provides four permission annotations:

writable – a "normal" read/write reference.

readable – a reference through which no **writable** references can be obtained, and no mutation can be performed (other **writable** aliases may still exist).

immutable – denoting a reference to an object that will never be mutated and through which only other **immutable** references can be obtained.

isolated – the only external reference to an aggregate object, which may contain arbitrary internal aliasing (*cf.,* Section 4.1.2). Outgoing references from such objects may only refer to **immutable** data, making transfer of **isolated** references between threads safe from data-races.

An **isolated** reference can be converted to an **immutable** reference, effectively "freezing" the whole object aggregate. The type system similarly allows **isolated** references to be converted to **writable** ones to allow updating them, but importantly, the type system also allows recovering an **isolated** reference from a **writable** one: in a context (*e.g.*, a method call) where all ingoing values (arguments) are **isolated** or **immutable** and the outgoing value (return value) is a single **writable** reference, this reference *must* be externally unique (it was either one of the ingoing **isolated** values, or a value that was created in the current context, in which case it is the only remaining reference to that object).

Figure 4.3 shows an example of a method which takes an **isolated** argument and converts it to a **writable** one in order to be able to update it. At the end of the method, converting `b` from **writable** to **isolated** is safe because no external references to the object (even if there were any) can remain; any aliases are either local variables, in which case they go out of scope, or they are internal to the **isolated** argument of the method. Combining isolation recovery with the ability to convert **isolated** references to **immutable** ones facilitates building immutable structures with arbitrary internal aliasing.

Methods are also annotated with a permission, denoting the permission of that method's **this** variable. Methods may only be called when the annotation on the receiver matches (or can be converted to) the annotation on the method. This motivates the existence of the **readable** annotation; a method annotated as **readable** can also be called on a **writable**, **immutable** or **isolated** receiver, as all these annotations can be converted to **readable**. If no annotation is given for a reference or method, the permission defaults to **writable**.

Pony

Pony (*cf.*, Section 2.2.2) is an actor language with a capability-based type system which prevents data-races, even in the presence of sharing between actors [119, 124]. A reference can have one of six *reference capabilities*, four of which are similar to Gordon's permission annotations above [74].

iso – an externally unique reference (*cf.*, Section 4.1.2), similar to Gordon's **isolated** annotation, allowing outgoing references to immutable data only. An **iso** reference may be transferred between actors.

val – a reference to deeply immutable data, which may be shared between actors. Similar to Gordon's **immutable** annotation.

ref – a "normal" read/write reference that allows aliases, but cannot be shared between actors. Similar to Gordon's **writable** annotation.

box – a reference that does not allow mutation, although other **ref** aliases may exist in the same actor. Similar to Gordon's **readable** annotation

trn – the only writable reference to an object that is "transitioning" from being mutable to immutable. A **trn** reference may have **box** ("read-only") aliases in the same actor, and may be converted into a **val** reference, after which it is permanently immutable and may be shared between actors.

tag – an "opaque" reference that may not be dereferenced, but which may be used as the receiver of a message send (if the underlying type is an actor). Any reference may have an unbounded number of **tag** aliases, even **iso** ones.

Like in Gordon's system, there are mechanisms for converting references between the different capabilities, and for recovering isolation. Also similarly, methods are annotated with capabilities and can only be called on a receiver with a matching (or convertable) capability. If nothing else is specified, references of class type default to **ref** and primitives to **val**, while references to actors are always **tag** (preventing other actors from accessing private state). Method capabilities default to **box**, meaning mutating methods must be identified with an annotation.

Scala Capabilities and LaCasa

In Haller and Odersky's capabilities for Scala [82], references and capabilities are decoupled; a reference can only be accessed if its capability is available. A variable marked as `@unique` may be aliased (locally), but its *capability* is treated linearly; passing one of the references together with the capability to another context renders all other aliases inaccessible. This removes the need for destructive reads of local variables (fields marked `@unique` must be swapped for another value when read). A method (or parameter) may be marked as `@transient` to denote that calling it on a `@unique` receiver (or with a `@unique` argument) does not consume the capability of that reference (akin to borrowing, *cf.*, Section 4.1.1).

A `@unique` reference is an external reference to a separately unique aggregate (*cf.,* Section 4.1.2), and its capability guards access to that entire region. This means there may be references which are not aliases, but which share the same capability. A method parameter `x` may be marked as `@peer(y)` to denote that `x` is guarded by the same capability as `y`, allowing *e.g.,* assigning `x` into a field of `y`. Consuming a capability renders all references with that capability inaccessible. The type system also allows merging two regions, for example when a `@unique` reference is passed into, and stored in, another separately unique aggregate.

A similar system, but based on Scala implicits rather than annotations, is found in LaCasa [80]. LaCasa focuses on communication between actors, and requires references passed between actors to be wrapped in a special `Box` object. In order to interact with a `Box` (and the encapsulated object within), an actor needs to supply a special permission value associated with the box. Certain operations, such as sending a `Box` to another actor, consumes the permission, making the `Box` inaccessible to the sender (effectively transferring ownership). By using Scala's path dependent types, the type system can guarantee that a `Box`'s permission cannot be forged. Scala implicits are used to avoid having to explicitly chain the permission value through the program. By enforcing a strict object-capability discipline [60] (*cf.,* the E language [106], Section 2.2.2) within a `Box`, the type system ensures that the internal reference cannot leak (an external reference into a `Box` could cause data-races).

Mezzo

Mezzo [17] is another programming language where references are decoupled from the abstract tokens which guard their use. In Mezzo, using a reference requires access to a corresponding permission, and unlike for example the Scala capabilities in the previous section, these permissions may be defined by the programmer. Passing a reference to a function may consume its permission, but also return new permissions. This allows programmers to use the type system to enforce protocols for user-defined data. For example, a `Cell` data type may be created with a use-once linear permission which only allows calling the function that sets the value of the `Cell`. This function may in turn return a duplicable permission to the same `Cell` that only allows reading it.

When concurrency is involved, the type system prevents data-races by ensuring that only duplicable permissions (which do not allow mutation) are shared across threads. Additionally, locks can be used to safely share access to mutable state. The type system allows expressing the *protocol* of a lock, enforcing *e.g.*, that a lock may only be released if it was first held. This also aids the programmer in correctly implementing other data structures which use locks. However, the actual *implementation* of a lock can not be written in the language as it requires sharing mutable state (and thus duplication of linear permissions).

Comparison

All the systems discussed in this section have one or more things in common, apart from using capabilities or permissions. All systems use linearity in one form or another to ensure that mutable state is not shared, while still allowing data to migrate without copy between threads (or actors). Another reoccurring theme is relying on immutability for safe sharing (Scala capabilities being an exception). Mezzo, Pony and Gordon's isolated types, all allow "freezing" a linear object (aggregate) by making it immutable. Unlike with fractional permissions, mutability cannot be recovered once an object has been frozen. Pony and Gordon's isolated types additionally allows fixing a mutable object to a thread or actor by dropping linearity together with the ability to share the object.

In SafeJava, code can be reused across different concurrency scenarios, but after creating an object, its protection mechanism is fixed. Pony and Gordon's isolated types instead rely on being able to convert between capabilities to change the interface and aliasing restrictions of an object for different use-sites. Mezzo and SafeJava allows shared access to mutable state protected by locks and ensure that locks are not bypassed. Mezzo is the only one of these systems whose types are also used to enforce that the protocol of *e.g.,* locks is followed.

4.2 Verification Techniques

The techniques presented in Section 4.1 help the programmer to write safe and correct code. Type systems are modular and light weight—types can tell a programmer something useful about a single line of code in isolation—but this also forces reasoning to be conservative. For example, unless some borrowing annotation is used, a type system must conservatively assume that a linear reference passed to a function is consumed, regardless of how the function is implemented. Additionally, new languages and state-of-the-art type systems do not help with all the legacy code that has already been written in other languages. This section briefly overviews some more powerful (but also more involved) techniques for program verification. They are generally used to reason about more high-level correctness properties than "just" alias control.

4.2.1 Program Logics

A program logic is a formal system for reasoning about the behaviour of programs. The most widely used program logic is probably *Hoare logic* [87], which facilitates reasoning about pre- and post conditions by using *Hoare triples*. The Hoare triple $\{P\}e\{Q\}$ should be read as "if P holds in the initial state, and the program expression e is executed, Q will hold in the resulting state" (P and Q can for example describe the values of the variables in scope).

Hoare logic works well for reasoning about programs which manipulate numerical values, but less well for programs with pointers and potential aliasing. This is addressed by *separation logic* [122], which extends *Hoare logic* with an explicit notion of a heap and a store (or stack) and logical connectives for reasoning about the contents of the heap. Importantly, the "separating conjunction" $P * Q$ states that the current heap can be partitioned into two non-overlapping heaps h_1 and h_2 such that P holds for the heap h_1 and Q holds for the heap h_2. Conversely, the "separating implication" $P \mathrel{-\!\!*} Q$ (also known as the "magic wand") states that if two disjoint heaps h_1 and h_2 are merged into a heap h, and P holds for h_1, then Q holds for h. *Concurrent separation logic* additionally extends separation logic with support for reasoning about programs running in parallel.

To handle the complexity of using concurrent separation logic, there are implementations for theorem provers (*cf.*, Section 3.3.1), such as Iris [96] and FCSL (Fine-grained Concurrent Separation Logic) [128]—both implemented in Coq. These systems have been used to prove correctness of fine-grained concurrency algorithms (*cf.*, Section 2.1.2) [59, 96, 128], to reason about weak memory models [94], and to model the programming language Rust [93].

Rely/guarantee reasoning [92] is another technique often used in verification of concurrent programs. In short, with a *rely* R and a *guarantee* G, a thread can rely on the other threads to behave as specified by R, and must guarantee that it behaves according to G. Together, R and G can be used to reason about the possible interference between threads. Systems using rely/guarantee reasoning have been used to guarantee safe interference over shared memory [104], to reason about functional correctness of fine-grained concurrency algorithms [74], and for automatically proving that non-blocking algorithms actually do not block [75]. Rely/guarantee techniques have also been combined with separation logic into RGSep, which has been used to prove linearisability of several fine-grained concurrency algorithms [139].

4.2.2 Model Checking

Another approach for reasoning about concurrent programs is to use *model checking* [53], which, given a model of a program, exhaustively explores all possible states of this model to see if the program can reach an erroneous state. An erroneous state can for example be a deadlock between two threads, a failing assertion for a single thread, or a situation where *e.g.*, linearisability is violated for a lock-free data structure (*e.g.*, [83]). Naïvely exploring all possible interleavings of the threads in a concurrent program would not be feasible for non-trivial programs. Instead, the search space is pruned by using techniques such as dynamic partial order reduction to avoid exploring redundant interleavings (interleavings which are known to be equivalent to an already explored interleaving) [3, 66]. Two examples of tools using model checking to find concurrency errors are Concuerror for Erlang [47], and Nidhugg for C with PThreads [2].

After this overview of related work, we now move on to the main contribution of this thesis: Kappa, a capability based type system for concurrency control.

5. Kappa

This chapter presents Kappa, a capability-based type system for race-free concurrent programming. The details of the type system can be found in Papers I–III. Section 5.1 covers the contents of Papers I and II, while Section 5.2 focuses on Paper III. All examples in this chapter are written with the *syntax* of Encore, the active object language in which Kappa has been implemented, but not all examples are valid Encore code as some features of Kappa have not yet been implemented in Encore. The current state of the implementation in Encore is discussed in Chapter 6.

5.1 Reference Capabilities for Concurrency Control

In Kappa, the reference concept is unified with the concept of a capability. A reference capability is a handle to some resource, which can be an object, a part of an object, or an entire object aggregate[1]. Importantly, a reference capability guarantees that while the underlying resource is being accessed (read or written), no other thread will write to it (concurrent reads are allowed). In other words, accesses through a reference capability is always free from data-races.

A capability is specified by a type, which defines the operations available on the underlying object, and a *mode*, which defines the protection mechanism of the underlying object. The original formulation of Kappa uses six different modes[2]:

linear – a linear reference, potentially externally unique (*cf.,* Section 4.1.1). It may be passed between threads, and is trivially safe from data-races since it is the only (external) reference to its resource. Linearity can be maintained through destructive reads, or by some more advanced static tracking, *e.g.,* alias burying [23].

[1]Our terminology is different from *e.g.,* Pony, where the reference capability is the annotation on the type (*cf.,* Section 4.1.4).
[2]This presentation uses the names of the modes from the implementation in Encore. The names are therefore slightly different from the ones used in Paper I.

local – a thread-local reference, which may be freely aliased, but never passed between threads. It is trivially safe from data-races since it may only be accessed by a single thread. In order to ensure that **local** capabilities do not implicitly escape their thread, only **local** objects are allowed to have fields containing **local** capabilities.

locked – a reference to a resource implicitly guarded by a lock, and which may be freely shared across threads. Calling a method through the capability will acquire the lock for the duration of the call (*cf.,* Java-style **synchronized**), which guarantees mutual exclusion.

subord – a reference to a *subordinate* object, which is strongly encapsulated inside another object. Kappa ensures the owners-as-dominators property for subordinate objects (*cf.,* Section 4.1.2), meaning that the subordinate object is implicitly protected by the same mechanism as its owner.

read – a reference that does not allow mutating operations on the underlying object, and which may be freely shared across threads. The read-only property is shallow, meaning that a method call may result in (safe) mutation of some other object. The **read** mode does not exclude mutable aliases of the same resource (but does ensure that the object will not be mutated while it is being accessed through the **read** capability).

unsafe – a "normal" object reference, without any protection. This mode can either be used as an "escape hatch" for expressing aliasing patterns that are not captured by the other modes (this is the approach taken in Encore), or as an indication that some other synchronisation is needed, *e.g.,* acquiring a lock (this is the approach taken in Paper I).

The selection of modes is similar to the annotations offered by other capability systems (*cf.,* Section 4.1.4). Like SafeJava [22], Kappa relies on a combination of annotations and ownership to achieve data-race freedom, but unlike SafeJava, Kappa does not need explicit ownership types. The **linear**, **local** and **read** modes are similar to the **isolated**, **writable** and **readable** annotations from Gordon's isolated types [74] (or **iso**, **ref** and **box** of Pony [124]), although **read** capabilities are safe to share between threads, which reduces the need for the deep immutability offered by Gordon's **immutable** (or Pony's **val**)[3]. There is no conversion between modes, but since Kappa does not require annotations on

[3]Deep immutability can be expressed in Kappa by constructing `read` capabilities which (transitively) can only reach other `read` capabilities.

methods, a capability can always be used to the full extent allowed by its type. Patterns such as turning a **linear**, mutable object into an immutable one is possible by using composite capabilities (*cf.,* Section 5.1.2). There is no equivalent of Pony's **trn** capability (a unique writable reference which may have read-only aliases) in Kappa.

The ownership guarantees given by the **subord** mode can be though of as a combination of the special owners **rep** and **owner** in ownership types (*cf.,* Section 4.1.2). This means that there is no way for an object o to distinguish between privately owned objects and sibling objects that have the same owner as o. This is less expressive than full ownership types, but also requires less complicated machinery. Having subordinate objects with a **local** or **unsafe** dominator expresses flexible alias protection, while a **linear** dominator additionally expresses external uniqueness. Having a **locked** dominator lets a single lock protect a whole object aggregate. Since a **read** capability may be accessed concurrently, it may only grant access to other capabilities which are also safe to access concurrently (*i.e.,* **locked** capabilities and other **read** capabilities). This disqualifies **read** capabilities from accessing encapsulated state in the form of **subord** capabilities.

In order to make programming with **linear** capabilities simpler, method parameters can be annotated as **borrowed** to allow **linear** capabilities to be passed as arguments without destructive reads (*cf.,* Section 4.1.1). Additionally, if `x` is a linear capability which grants access to a linear field `x.f`, Kappa allows non-destructively borrowing `x.f` as long as `x` is buried for the duration of the borrowing (this generalises to longer chains of linear fields).

Figure 5.1 shows a hierarchy of different kinds of capabilities. The three top-level categories are the *exclusive* capabilities, **linear** and **local**, the subordinate capabilities and the *shared* capabilities, which may all be shared between threads. The shared capabilities can be categorised as *safe* and **unsafe**, where the latter does not provide any concurrency control of its own. The safe capabilities can further be grouped according to the kind of concurrency control they provide.

The *optimistic* capabilities allow concurrent updates, but roll back in case of conflicts (the **lockfree** mode is detailed in Section 5.2). The *pessimistic* capabilities are based on mutual exclusion, like **locked**. The **active** mode, for expressing active objects, is explained in Chapter 6. Finally, the *oblivious* capabilities do not need concurrency control because they only provide non-racing (*i.e.,*

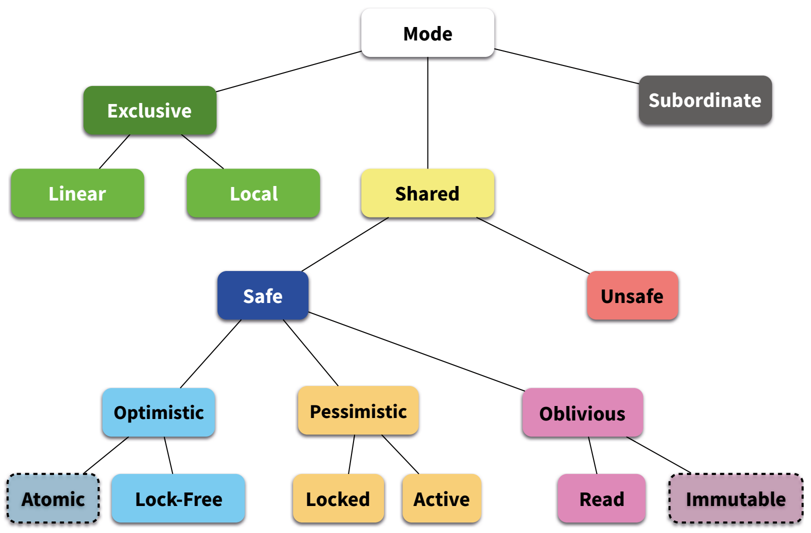

Figure 5.1. A hierarchical view of the modes of reference capabilities.

read-only) operations. The figure also includes two modes that have not yet been implemented: an **immutable** mode, which is a "deep" version of the **read** mode, and an **atomic** mode, which wraps each method call in a transaction (*cf.*, Section 2.1.3).

As the hierarchy suggests, allowing a **safe** mode which abstracts over the underlying modes introduces a kind of *polymorphic concurrency control*: a method which takes a **safe** capability knows that this capability can be safely used and shared, but does not need to care *how* this safety is achieved. Section 6.3 explains how parametric polymorphism is implemented Encore, including this kind of "mode-bounded polymorphism".

5.1.1 Concurrency Agnostic Code Reuse with Traits

The mode of a capability expresses why that capability is safe to access. The interface of a capability is defined by its type, which is introduced via traits (*cf.*, Section 3.1). When a trait type is used (*e.g.*, included by a class), it must be provided with a mode in order to form a capability. Like in SafeJava [22], separating the business logic from the mechanism of concurrency control allows traits to be reused across different concurrency scenarios. A trait can always

```
trait Inc
  require var cnt : int
  def inc() : unit
    this.cnt += 1
  end
end

trait Get
  require val cnt : int
  def get() : int
    this.cnt
  end
end

class LinearCounter : linear Inc + read Get
  var cnt : int
end

class LockedCounter : locked Inc + read Get
  var cnt : int
end

class LocalCounter : local Inc + read Get
  var cnt : int
end
```

Figure 5.2. A simple example of how traits can be reused for different concurrency scenarios. **var** and **val** denote mutable and immutable (Java **final**) fields respectively.

be given any mode, except **read** which requires that all required fields are immutable and have types which are safe to access concurrently.

An important property of Kappa is that the implementor of a trait can assume exclusive access to all the required fields of the trait. Regardless of which mode the trait type is given when used, mutual exclusion will be guaranteed (except for the **unsafe** mode). This is made possible by typechecking the methods of a trait with **this** as a **subord** capability, meaning methods may not leak **this** to the outside. If **this** was not encapsulated, a **linear** reference could return itself from a method, and a **local** reference pass itself to another thread. Optionally, the programmer can provide a trait declaration with a *manifest mode*, meaning methods will be typechecked with **this** having that mode. These traits cannot have their mode overridden at use-site, and may only be included together with other traits of the same mode[4]. Providing a manifest mode thus makes the implementation less restricted, but reduces the amount of possible code reuse.

Figure 5.2 shows a simple example of how traits can be reused to implement classes for different concurrency scenarios. The trait `Inc` provides a method `inc`, which increments the required field `cnt`. The trait `Get` provides a method `get` which simply reads the field. These traits are included by three different classes, which all give different modes to the `Inc` trait, respectively producing a counter that must be handled linearly, a counter that is protected by a lock, and a counter that is thread local. Using the **read** mode for the trait `Get` is allowed as `Get` only requires an immutable (**val**) fields whose type (**int**) is safe to access concurrently. Conversely, it would not be allowed to use **read** for `Inc` as it mutates its required field. In the case of `LockedCounter`, the fact that `Get` is a **read** capability—and thus will not mutate the underlying object—facilitates safely using a readers-writer lock (*cf.*, Section 2.1.1) to allow concurrent calls to `get` but keep calls to `inc` mutually exclusive. In general, all compositions of **locked** and **read** capabilities can be implemented to use readers-writer locks.

The design decisions for Kappa have been aided by studies of aliasing patterns in extant Java programs [31] using the trace-based analysis tool Spencer [32]. A comprehensive study of over 1 million lines of Java code shows that the amount of aliasing found in Java programs is quite small, and that aliasing patterns for a declaration are mostly stable across its use-sites. We hope to further validate our designs using Spencer in future work, but it should also be noted that it is not clearly established how well Java-results translate to *e.g.*, Encore.

5.1.2 Composite Capabilities

In addition to introducing capabilities through traits, capabilities can be formed by composing existing capabilities. When a class includes more than one trait, this class name becomes synonymous with the corresponding composite capability. A composite capability offers the union of the interfaces of its constituents, and must adhere to the aliasing restrictions of all its modes. For example, the `LinearCounter` class in Figure 5.2 is synonymous with the composite capability **linear** `Inc + ` **read** `Get`, which must be treated linearly due to its sub-capability **linear** `Inc`.

[4] If for example a class could include traits where some methods see **this** as **local** and others see **this** as **linear**, linearity could be violated.

```
linear trait Left
  require var left : Tree
  def getLeft() : Tree
    consume this.left
  end
  def setLeft(var left : Tree) : unit
    this.left = consume left
  end
end

linear trait Right
  // Analogous to Left
end

linear trait Elem
  require val elem : T
  def apply(f : T -> unit) : unit
    f(this.elem)
  end
end

class Tree : Elem * Left * Right
  var left : Tree
  var right : Tree
  var elem : T
end

fun foreach(var t : Tree, f : T -> unit) : Tree
  unless t == null then
    var e : Elem, l : Left, r : Right = consume t // Split t
    finish
      // Operate on aliases concurrently
      async {e.apply(f)}
      async {l.setLeft(foreach(l.getLeft(), f))}
      async {r.setRight(foreach(r.getRight(), f))}
    end
    t = consume e * consume l * consume r // Restore t
  end
  return t
end
```

Figure 5.3. An example of traits composed into a conjunctive capability. T is some elided type.

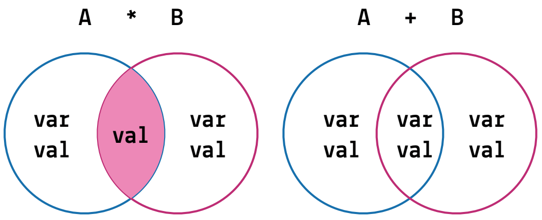

Figure 5.4. Restrictions for overlapping fields in composite capabilities.

When aliasing a composite capability, the new capability must generally retain the same modes as the original capability. Forgetting *e.g.*, that a capability is thread local is not allowed, as this could lead to a thread local object leaking. It is however safe to drop all **linear** sub-capabilities of a capability (*cf.*, strong updates, Section 4.1.1). For example, using the code from Figure 5.2 it is possible to drop the capability **linear** `Inc` from `LinearCounter` and get a sharable **read** capability (**consume** denotes a destructive read):

```
var cnt1 = new LinearCounter()
val cnt2 = cnt1 // Not allowed! cnt is linear
val g1 : read Get = consume cnt // Drop linear Cnt capability
val g2 = g1 // Allowed! g1 is a read capability
```

This pattern is similar to converting from **isolated** to **immutable** in Gordon's isolation types (*cf.*, Section 4.1.4). Paper I additionally defines machinery for *temporarily* dropping a **linear** capability, allowing a mutable capability to be split into several **read** capabilities which may be used concurrently, before restoring the original **linear** capability, akin to fractional permissons [25] (*cf.*, Section 4.1.4). This is not supported in SafeJava, Pony, nor Gordon's isolated types.

Capabilities can be either be composed as disjunctions (using "+") or conjunctions (using "*"). Disjunctive capabilities `A + B` can be used as an `A` *or* a `B`, but not at the same time[5]. There are no restrictions on what data `A` and `B` may access concurrently. Conversely, conjunctive capabilities `A * B` can be used as an `A` *and* a `B`, possibly at the same time. This capability is only allowed if the fields shared by `A` and `B` are immutable and have types that are safe to access concurrently. Figure 5.4 illustrates the restrictions on field sharing between conjunctions and disjunctions.

[5]This doesn't really hold for **read** capabilities, as two composed **read** capabilities can safely be used at the same time, but it works as a mnemonic.

Concurrent access to conjunctive capabilities is handled by splitting the composite capability into its constituents. Figure 5.3 shows the definition of a `Tree` class, conjunctively composed by three **linear** capabilities `Elem`, `Left` and `Right`. This allows a `Tree` capability to be split into three aliases of type `Elem`, `Left` and `Right` respectively, which can be operated on concurrently, and later be merged to restore the original capability, as shown in the function `foreach`. On Line 30, the variable `t` is split into three aliases, presenting three different "views" of the same object, and on Line 37, the aliases are merged again. If one of the capabilities would not be an alias of the others, the operation would fail, and throw an exception. The dynamic check could be omitted using escape analysis or a blocked splitting construct. This splitting and merging of capabilities extends the ideas of fractional permissions [25]—which support the pattern single writer/multiple readers—to also allow multiple writers.

Paper I contains the same example as Figure 5.3, but uses borrowing to avoid having to reconstruct the `Tree` after each call to `foreach`. A more clever local analysis could infer that concurrent usages of a single `Tree` capability are non-interfering, removing the need for an explicit split[6]:

```
fun foreach(t : borrowed Tree, f : T -> unit) : unit
  unless t == null then
    finish
      // Infer that non−overlapping capabilities are used
      async {t.apply(f)}
      async {foreach(t.getLeft(), f)}
      async {foreach(t.getRight(), f)}
    end
  end
end
```

Paper I also shows how a *nested capability* **linear** `Tree[A*B]` can be split into two aliases of type **read** `Visit[A]` and **read** `Visit[B]`, where `Visit` is a trait that implements iterating over the tree. This allows applying two different (non-overlapping) functions concurrently to the elements of a data structure.

Conjunctive capabilities allow similar concurrency patterns as regions and effects (*cf.*, Section 4.1.3). While effects are more fine-grained, Kappa avoids the introduction of explicit effect annotations on methods which must be tracked

[6] implementing such an analysis is future work

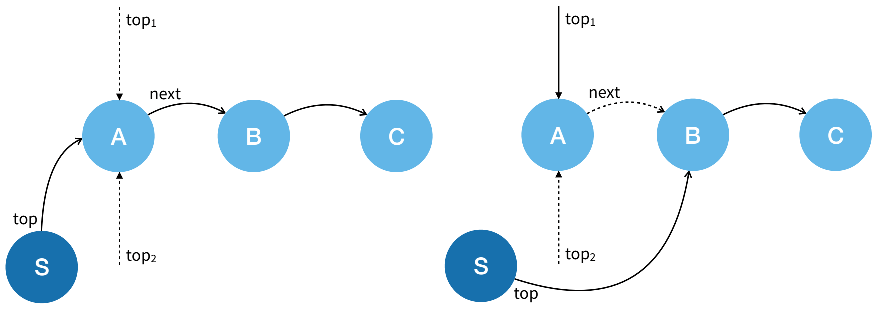

Figure 5.5. A Treiber stack before and after a successful pop. Dashed lines are references without ownership.

and propagated. In Kappa, the required fields of a trait can be seen as defining its maximum “effect footprint”, where the modifier on the field (**var** or **val**) decides if the effect is a read or a write. The `Tree` class from Figure 5.3 could be reinterpreted with regions and effects as a class with three regions `Left`, `Right` and `Elem`, with one of the fields in each. The effect annotations on the methods from the trait `Left` would be “**writes** `Left`” because of the **var** annotation on the `Left` field (and similarly for `Right`). The effect annotations on the `apply` method would be “**reads** `Elem`”, since `elem` is a **val** field. The rules for when two methods have non-conflicting effects coincides with the rules for when a disjunctive capability is well-formed. This is discussed closer in Paper II.

5.2 Reference Capabilities for Fine-Grained Concurrency

The modes presented in Section 5.1 guarantee mutual exclusion whenever a thread is accessing mutable state. This is a powerful property, but does not allow implementing the kind of fine-grained concurrency discussed in Section 2.1.2, where threads follow some protocol to safely compete for access to some shared memory. Such algorithms are important to avoid scalability bottlenecks caused by high contention on data structures that require synchronisation. This section discusses an extension of Kappa, called LOLCAT[7], which allows the implementation of data structures based on fine-grained concurrency, while still guaranteeing the absence of (uncontrolled) data-races.

[7]Paper III presents LOLCAT as a stand-alone type system which does not explicitly mention reference capabilities.

In contrast to systems like behavioural types (*cf.,* Section 4.1.1) or more heavy-weight verification techniques (*cf.,* Section 4.2), LOLCAT cannot specify or verify high-level protocols for data structures. Instead, the guarantee given by the type system is simpler: access to owned data (mutable **var** fields) is always exclusive, and any occurrences of shared mutable state must be explicitly allowed. The type system captures and enforces existing patterns from the literature on fine-grained concurrency, such as speculating on values and using atomic operations to acquire ownership of these values.

The ideas in LOLCAT stem from two observations. Firstly, many fine-grained concurrent data structures, *e.g.,* the lock-free Treiber stack in Figure 2.6, follows a pattern of *linear ownership*: each element in a stack is "owned" by the data structure itself, until a thread manages to assert ownership of it by popping it from the stack. In a correct implementation, there can never be more than one owner of an element. Note that using destructive reads (*cf.,* Section 4.1.1) to maintain linearity will not work in a concurrent setting—even temporarily setting a shared field to **null** will prevent the progress of other threads, destroying concurrency. Secondly, having more than one reference to an object is safe from a data-race perspective as long as at most one of these references is ever used to access the mutable resources of that object. For the rest of this presentation, this reference is called the *owning reference*.

As a concrete example, Figure 5.5 shows the Treiber stack from Figure 2.6 before and after a successful `pop`. The fully drawn lines are owning references; before the `pop`, all owning references are references in the data structure. In particular, the `top` reference from the `Stack` object S owns the node A. Two threads are accessing the stack concurrently, shown as the two non-owning references (dashed lines) to A, top_1 and top_2. After the `pop` succeeds, top_1 is upgraded to an owning reference. The `top` reference from S has been moved forward from A to B, meaning `top` now owns B. In order to maintain linear ownership, the `next` reference from A must be downgraded to a non-owning pointer (if A.`next` could also be used to access the element of B, there could be races).

LOLCAT uses reference capabilities to track this kind of ownership information and maintain linear ownership. Figure 5.6 shows a partial implementation of a Treiber stack in the extended version of Kappa (the full code is available in Paper III). The **lockfree** mode (*cf.,* Figure 5.1) denotes a capability that is safe to use because its implementation follows the protocol enforced by LOLCAT. Since the

```
 1  linear class Node
 2    var elem : Elem
 3    val next : Node
 4  end
 5
 6  lockfree trait Pop
 7    require spec top : Node // Field may be accessed concurrently
 8
 9    def pop() : Elem
10      while true do
11        val t = speculate this.top // t : Node | elem
12        if(t == null) then
13          return null
14        end
15        if (CAT(this.top, t, t.next)) then
16          // t : Node ~ next
17          return consume t.elem
18        end
19      end
20    end
21  end
```

Figure 5.6. A partial implementation of a Treiber stack in Kappa. `Elem` is some elided type. The full code is available in Paper III.

`top` field may be updated concurrently, any values read from it are considered speculative. It is therefore marked with the keyword **spec**. A **spec** field may be speculatively read to yield a non-owning reference, as in Line 11[8].Updates to a **spec** field must be performed with an atomic `CAT` (*compare-and-transfer*) operation, as in Line 15.

A `CAT` has the same dynamic semantics as a `CAS` (*cf.*, Section 2.1.2), but leverages types to ensure that correct ownership is in place. `CAT(`**this**`.top, t, t.next)` should be read as "atomically transfer ownership of `t.next` to **this**`.top`, and from **this**`.top` to `t`". The intuition behind this is as follows:

- **this**`.top` stores an owning reference, and because of linear ownership, it is the *only* owning reference to its referent.
- The `CAT` only succeeds if **this**`.top` and `t` are aliases, meaning that `t` must be a non-owning reference.
- If the `CAT` succeeds, ownership of `t.next` is transferred to **this**`.top` (through assignment), meaning that `t.next` must also be an owning reference.

[8]We use a **speculate** keyword in the implementation to explicate speculative reads.

– If **this**.top, which is the *only* owning reference to its referent, is overwritten, there is no longer an owning reference to the Node being popped. It is therefore safe to transfer ownership to t.

The types in LOLCAT track ownership on a per-field basis. A type Node can be used to access any field. The speculative read on Line 11 yields a reference of type Node | elem, denoting that the mutable **var** field elem may not be accessed (the immutable **val** field may still be accessed though). If the CAT on Line 15 succeeds, the type of t changes to Node ~ next, denoting that the field next no longer contains an owning pointer[9] (*cf.,* Figure 5.5). Because the restriction on elem is lifted for the duration of the **if**-statement, the field may safely be destructively read on Line 17. Restricting individual fields is similar to dropping capabilities from a composite capability (*cf.,* Section 5.1.2).

Tracking the ownership of individual fields gives a meaningful type to the compare-and-swap operation (in the form of CAT), enforcing the pattern of linear ownership. The types in LOLCAT help programmers to modularly reason about ownership in the context of fine-grained concurrency. Paper III also defines constructs for tracking stability (non-mutation) of mutable fields, as well as the global absence of owning references (the type Node | elem seen above still allows owning aliases). There are also additional examples of lock-free data structures based on CAS implemented in LOLCAT, including spin-locks that can be used to protect a linear value. These locks could be used to implement Kappa's **locked** capability as a library, without using the **unsafe** escape hatch, as warranted *e.g.,* by Rust (*cf.,* Section 4.1.1).

After a brief overview of the type systems presented in Papers I–III, we now continue by discussing the implementation of Kappa in a real programming language.

[9] This prevents the Node from *e.g.,* being re-inserted in another stack, introducing sharing between the two stacks.

6. Implementing Kappa in a Language with Active Objects

This chapter discusses the implementation of Kappa in the object-oriented language Encore, which supports concurrency and parallelism through active objects (*cf.*, Section 2.2) [29]. Just as the object calculi discussed in Section 3.3 model a subset of Java, the formal description of Kappa in Paper I omits many features and programming conveniences that belong to a full-fledged langugae. This chapter explains how Kappa was adapted to a language with active objects (Section 6.1), and goes through the addition of method overriding (Section 6.2) and polymorphism (Section 6.3). Section 6.4 discusses the work presented in Paper IV, which allows switching from synchronisation based on isolation to synchronisation based on delegation.

The Encore compiler, runtime and tooling is open-source and is available on GitHub [62].

6.1 Active Objects for Concurrency Control

Encore distinguishes between active and passive objects, where passive objects are normal objects which may be interacted with synchronously, and active objects are objects with their own logical thread of control which may only be interacted with asynchronously via message sends. Messages end up in the mailbox of the active object, where they are picked up and handled one by one. This way, the mailbox serialises all interactions with an active object, analogously to how locks achieve synchronisation by blocking interfering threads. Encore uses active objects as its main source of concurrency; there is no explicit spawning of threads.

The modes discussed in Chapter 5 all describe passive data. Kappa handles active objects by introducing an **active** mode. An **active** capability is a reference to an active object which may only be interacted with through message sends.

Active objects serialise all external interactions in their message queue, much like locks serialise interactions by making threads wait for their turn. Since interaction with an active object is always safe, **active** capabilities can be freely shared between active objects. A **local** capability is encapsulated inside an active object, and will never escape from the active object that created it. Therefore active objects are allowed to have fields holding **local** capabilities. A **linear** capability can be transferred between actors, together with any encapsulated **subord** capabilities. As usual, **read** capabilities can be freely shared.

Encore's concurrent garbage collection protocol (which it shares with that of Pony, *cf.,* Section 2.2.2) [54], is based on the absence of shared mutable state. This complicates the inclusion of locks, and explains why the **locked** capability has not been implemented in Encore yet. In related work, Yang and Wrigstad recently introduced Isolde [145], a pluggable garbage collection protocol which uses the information given by the LOLCAT type system (*cf.,* Section 5.2) to allow threads to share garbage collection responsibility of lock-free data structures. This allows the Encore garbage collector to handle shared mutable state in special cases, including library implementations of simple locks. The omission of the **locked** capability has not been a problem so far though, as the synchronisation offered by **active** objects has been enough.

To facilitate the common case where all the included traits of a class have the same mode, giving a class a manifest mode will implicitly assign that mode to all the included traits. For example, here is "Hello World" in Encore, with the business logic broken out into a trait:

```
trait Hello
  def hello() : unit
    println("Hello, World!")
  end
end

active class Main : Hello
  def main() : unit
    this.hello()
  end
end
```

6.2 Preserving Safety under Method Overriding

In the original formulation of Kappa, classes can only define fields; all methods come from included traits. This makes programming somewhat tedious, as even small classes need to be factored into one or more traits. At the same time, Kappa requires all capabilities to have a mode which explains why its methods are safe to use. If classes could define methods of their own without "protecting" the code with a mode, Kappa's safety guarantees would be lost.

In Encore, the only method that all classes may define is the constructor method `init`. To be allowed to define other methods, the class must either provide a manifest mode which explains why that code is safe, or assign each method to one of its included traits (which by necessity has a mode), thereby *extending* that trait. For example, the following code defines a class which includes the `Hello` trait from above and extends it with a method for additionally giving compliments:

```
class Greeter : active Hello(compliment())
  def compliment() : unit
    println("You are looking swell today!")
  end
end
```

The inclusion of the trait `Hello(compliment())` creates an *anonymous trait*, which is equivalent to `Hello` extended with a requirement of `compliment`:

```
trait Hello'
  require def compliment() : unit
  ... // the rest of Hello
end
```

The implementation of `compliment` in `Greeter` is therefore seen as overriding the abstract method in the anonymous trait. In order for an overridden method to be valid, it must typecheck as if it was defined in the requiring trait. In the example above, the `compliment` method is therefore typechecked twice: once with **this** as the class type `Greeter`, and once with **this** as the trait type `Hello`. If the added method uses fields or calls other methods, these must be added to the anonymous trait as well.

To see why extending traits is necessary, consider the code in Figure 6.1, which does *not* typecheck. If this code was allowed, an active object could create a

```
read trait Box
  require val f : int
  def get() : int
    return this.f
  end
end

class BadBox : Box
  var f : int
  def init(f : int) : unit
    this.f = f
  end
  def reset() : unit
    this.f = 0 // Potentially racing operation!
  end
end
```

Figure 6.1. A non-compiling program showing why all code must be "protected" by a mode.

`BadBox`, create an alias of type **read** `Box` which can be shared with other actors, and then call `reset` to cause a data-race. If on the other hand `Box` is extended with `reset()`, the compiler gives the error message "Overridden method '`reset`' writes field '`f`', which is marked as immutable in requiring trait '`Box`' ".

Requiring included traits to be extended with the fields and methods used by any overridden methods also prevents creating malformed composite capabilities. In Figure 5.3, the class `Tree` could be split into its `Left` and `Right` subcapabilities, which could then safely be operated on concurrently since they do not share fields. The following bad code, which tries to cause a data-race by updating the `left` field when calling `getRight`, does not compile:

```
class BadTree : Left * Right
  var left : Tree
  var right : Tree
  def getRight() : Tree
    this.left = null // Potentially racing operation!
    consume this.right
  end
end
```

The error given by the compiler is "Overridden method '`getRight`' requires access to field '`left`' which is not in requiring trait '`Right`'. Consider extending the trait on inclusion: `Right(left)`" (this error is caught when typechecking the overridden method in the original trait). Following the suggested fix instead gives the error message "Conjunctive traits '`Left`' and '`Right`' cannot share mutable field '**var** `left : Tree`' ", preventing the definition of a conjunctive capability with data-races.

6.3 Polymorphism

An important source of code reuse for programming languages is polymorphism. In Kappa however, the presence of capabilities which have different aliasing restrictions complicates the matter. Since a fully polymorphic value does not "remember" its mode, care must be taken so that polymorphic code cannot violate the aliasing restrictions of incoming capabilities. In particular, **linear** capabilities must no be duplicated, **local** capabilities must not be passed to another actor, and **subord** capabilities must not escape its enclosing object aggregate.

Encore solves this problem by introducing *mode bounded polymorphism*. When introducing a type parameter, the programmer can optionally specify a mode whose aliasing restrictions values of that type must respect. For example, the following function promises to treat its argument as a **local** capability:

```
fun duplicate[local t](x : t) : (t, t)
  (x, x)
end
```

Calling the function with a **linear** capability gives a compiler error (as the **local** mode allows aliasing). Calling the function with a **read** capability is however allowed, as the **read** mode is strictly more permissive than the **local** mode. Without a specified mode, type parameters default to the **local** mode, meaning it can be instantiated with any type that can be freely aliased within the same actor (*i.e.,* any type except **linear** and **subord**), including primitives. In addition to the concrete modes, Encore allows the abstract mode **sharable**, which can be instantiated with any mode that allows sharing across active objects (*i.e.,* **read**, **active** and **locked**). This is the same as the **safe** mode in the capability hierarchy in Figure 5.1.

Generic classes and traits follow the same rules as polymorphic functions and methods. A parameterised capability `Foo[T]` must follow the aliasing restrictions of `T`. For example, if `T` is a **local** capability, `Foo[T]` gives access to local state, and must therefore be treated as a **local** capability as well[1]. With similar reasoning, closures which capture capabilities of certain modes must follow the aliasing restrictions of these modes.

The need for destructive reads when reading **linear** capabilities from fields inherently complicates the story for data structures that are generic over both linear and non-linear data, as *e.g.,* iterating over such a data structure either destroys it, or only allows mapping functions which explicitly use borrowing. Since a destructive read changes the value being read, instantiating a **linear** type parameter with a primitive type is also problematic as it is not clear *e.g.,* what the value of a destructively read **int** should be. Encore currently solves this problem by only allowing destructive reads of **linear** polymorphic values when they are wrapped in a `Maybe` (destructively reading a value of `Maybe` type sets it to `Nothing`).

Gordon's isolation types [74] allow polymorphism over type qualifiers, but do not allow using **isolated** types to instantiate type parameters (for the reasons mentioned above). Pony [119] also allows capability generic types, and also provides a kind of abstract capabilities in the form of *capability constraints*, for example #**read**, which abstracts over all types that can be read, and #**send** which abstracts over all types that can be sent between actors. Whenever a type parameter allows instantiation with **iso** types (*e.g.,* an unconstrained parameter, or one with the `#send` constraint), values of that type must be handled linearly.

6.4 Switching between Isolation and Delegation

As discussed in Section 2.2.2 many actor languages avoid shared mutable state by relying on isolation, achieved either with language support or through programmer diligence. With the help of the **local** capabilities of Kappa, programmers can achieve encapsulation of an active object's private state and trust that this state remains isolated from other active objects and can be accessed without fear of data-races. This section overviews an extension to Kappa that allows

[1]An exception is **active** capabilities, which when parameterised over **local** data can still be safely shared since the **local** data is going to be local to the active object itself.

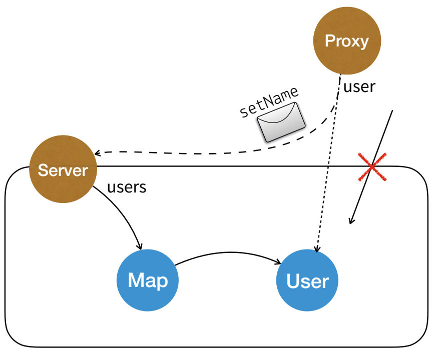

Figure 6.2. A graphical depiction of the program in Figure 6.3. The dashed lines denote a bestowed capability to a `User` object, whose messages will be relayed to the owning `Server`.

switching from protection based on isolation to protection based on delegation, similar to the far references of E [106], AmbientTalk [140] and CoBoxes [126] (*cf.*, Section 2.2.2).

This extension is based on the observation that the existence of external references to the private state of an active object is benign, as long as these references are never dereferenced by another active object[2]. Active objects are allowed to create such sharable references by *bestowing* **local** capabilities with activity (using a **bestow** operation). A bestowed capability behaves like an **active** capability and can be aliased as such, but implicitly forwards all messages sent to it to the active object that created it. This way, all interaction through the bestowed capability is synchronised in the mailbox of the owning actor, effectively serialising all operations. Figure 6.2 shows a graphical depiction of the difference between isolation and delegation.

Figure 6.3 shows a simplified program using bestowed capabilities. An active `Server` object stores a `Map` of **local** `User` objects. When a new connection comes in, the `Server` looks up the corresponding `User` and returns a new active object that acts as the proxy for the client to interact with. The `ClientProxy` holds a **bestowed** capability to the `User` object of the logged in user (note that the `User` is actually a **local** object in the `Server`). When the `ClientProxy` receives a request to change *e.g.*, the name of the logged in user, it sends a message to its **bestowed** `User` capability. Because this capability is bestowed, the message

[2]The same reasoning exists in Pony [124], where any reference can have an unbounded number of `tag` aliases, which do not allow dereferencing (*cf.*, Section 4.1.4)

```
local class User
  var name : String

  def setName(newName : String) : unit
    this.name = newName
  end
  ...
end

active class Server
  val users : Map[String, User]

  def newLogin(userId : String) : ClientProxy
    val user = this.users.lookup(userId)
    return new ClientProxy(bestow user)
  end
  ...
end

active class ClientProxy
  val user : bestowed User

  def init(user : bestowed User) : unit
    this.user = user
  end

  def updateName(newName : String) : unit
    this.user ! setName(newName)
  end
  ...
end
```

Figure 6.3. An example of a program using bestowed capabilities.

will be placed in the mailbox of the `Server`, where it will be handled eventually. In Figure 6.2, a `ClientProxy` has just sent a `setName` message through its **bestowed** `User` capability.

The same program could have been written by extending the `Server` class with methods for performing all possible operations on users, for example:

```
def setUserName(userId : String, newName : String) : unit
  val user = this.users.lookup(userId)
  user.setName(newName)
end
```

With bestowed capabilities, this coupling can be avoided. The same kind of safe relaxation of encapsulation is also possible for objects protected by locks; it is safe for a **locked** capability to leak an internal object (*i.e.*, a **subord** capability) if accessing the leaked object requires taking the lock of its owner.

The details of bestowed capabilities is discussed in Paper IV, including variants where the owner of a bestowed object can change during runtime. The paper also presents constructs for grouping messages to guarantee that two (or more) messages are performed back to back by the receiving active object, without any other messages interleaved. This allows composing new atomic operations from the existing methods of an active object, analogous to locking an object over several method calls.

We have now seen the motivation for this work, surveyed the necessary background, and examined the relation between our contributions and related work. Following that, we looked briefly at Kappa, and touched on the implementation of Kappa in the Encore programming language. Finally, we are ready to give some concluding remarks.

7. Concluding Remarks

This thesis contributes with novel programming language technology for controlling concurrency in the presence of shared memory. Kappa's vision is to take the fear out of programming in the multi-core age, and allow even novices to write concurrent programs with confidence. At the core of Kappa is the concept of viewing references as capabilities which restrict the operations allowed on the underlying object as well as the reference itself. The key to controlling concurrency is controlling the creation and propagation of reference capabilities, to ensure that no two concurrent processes, whether they are threads, actors, or active objects, can ever hold aliasing references which provide unsynchronised access to mutable state.

Concurrency control can be achieved in many different ways: by restricting which operations are available through a reference, for example by only allowing read operations or operations wrapped in locks; by restricting how a reference may be shared, for example by confining it to a single thread or encapsulating it inside an object aggregate; or by making sure that there is never more than one reference that gives access to the underlying object, for example by banning aliasing altogether or by tracking which reference currently owns the object. As this thesis has shown, all of these things can be expressed naturally using the capability abstraction.

Reference capabilities, as described in this thesis, provide a versatile toolkit for developing programming language technologies for the multi-core era. With a small number of primitives, they are capable of expressing a wide variety of intentions for how references may be—or are intended to be—used. Allowing the placement of annotations to vary between declaration-site and use-site introduces a "sliding scale" which balances reusability and freedom of implementation. The closer to declaration-site an annotation is, the lower the syntactic overhead and implementation restriction, but also the less flexible the program is. Allowing a single program to combine both approaches is both powerful and sensible. Future work involves extending the range of the sliding scale further by allowing programmers to defer annotation until object *creation*.

Kappa-style reference capabilities provide a means of constructing and deconstructing objects in ways tied to object-oriented, trait-based reuse. The key cost for the programmer to allow a piece of code to be used across different concurrency scenarios is giving up the leaking of **this** to outside of the current aggregate. Where such leakage is warranted, reference capabilities will allow it (where sound), and constrain the versatility of that piece of code accordingly. This last note is important, as it highlights how reference capabilities differentiate code that involves sharing from code that does not. The annotation clarifies how a programmer must reason about correctness about that particular code, and statically prevents code to be used in ways which might lead to data-races.

An important direction of future work is defining the semantics of new modes, as well as further exploring the combination of capabilities with different modes. For example, the **active** mode opens up for some interesting combinations, where parts of an active object could be interacted with synchronously, or where acquiring a lock could provide prioritised access to an active object. Composing non-overlapping **active** capabilities naturally expresses an active object where several messages can be safely processed in parallel. Further directions for future work is discussed in the individual papers.

On the hardware-side, we entered the age of ubiquitous parallelism in the early 2000s, but so far, software has been lagging behind. With the help of reference capabilities, we hope to have opened the door enough for object-oriented programming to take the plunge.

The rest of the thesis consists of Papers I–V, which have all been briefly summarised in previous chapters. The appendix after the references contains some notes for the reader of these papers.

References

[1] Martín Abadi and Luca Cardelli. An Imperative Object Calculus. *TAPSOFT'95: Theory and Practice of Software Development*, pages 469–485, 1995.

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless Model Checking for TSO and PSO. *Acta Informatica*, 54(8):789–818, Dec 2017.

[3] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM*, 64(4):25:1–25:49, August 2017.

[4] Gul Agha. Actors: a Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence. *MIT Press*, 11(12):12, 1986.

[5] Akka library. `https://akka.io`. Accessed November 2017.

[6] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In European Conference on Object-Oriented Programming, *ECOOP*, volume 4, pages 1–25. Springer, 2004.

[7] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-Oriented Programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object Oriented Programming Systems Languages and Applications*, pages 1015–1022. ACM, 2009.

[8] Paulo Sérgio Almeida. Balloon types: Controlling Sharing of State in Data Types. In *Proceedings ECOOP'97*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag, June 1997.

[9] Gene M Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[10] Nada Amin and Ross Tate. Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 838–848. ACM, 2016.

[11] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Mikroelektronik och Informationsteknik, 2003.

[12] Joe Armstrong. A History of Erlang. In *HOPL III*, pages 6–1–6–26. ACM, 2007.

[13] Atomic Operations Library for C11. `http://en.cppreference.com/w/c/atomic`. Accessed November 2017.

[14] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A Formally Verified Proof of the Prime Number Theorem. *ACM Transactions on Computational Logic (TOCL)*, 9(1):2, 2007.

[15] Henry Baker and Carl Hewitt. The Incremental Garbage Collection of Processes. In *ACM Sigplan Notices*, volume 12, pages 55–59. ACM, 1977.

[16] Henry Baker and Carl Hewitt. Laws for Communicating Parallel Processes. 1977.

[17] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Transactions on Programming Languages and Systems*, 38(4):14:1–14:94, August 2016.

[18] Hendrik Pieter Barendregt et al. *The Lambda Calculus*, volume 2. North-Holland Amsterdam, 1984.

[19] Gavin M Bierman, MJ Parkinson, and AM Pitts. Mj: An Imperative Core Calculus for Java and Java with Effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

[20] Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. *ACM Sigplan Notices*, 44(10):97–116, 2009.

[21] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A Survey of Active Object Languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017.

[22] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2003.

[23] John Boyland. Alias burying: Unique Variables Without Destructive Reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[24] John Boyland. The Interdependence of Effects and Uniqueness. In *Workshop on Formal Techs. for Java Programs*, 2001.

[25] John Boyland. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pages 55–72, 2003.

[26] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pages 2–27, 2001.

[27] Gilad Bracha and William Cook. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages

303–311, New York, NY, USA, 1990. ACM.

[28] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. In *ACM SIGPLAN Notices*, volume 48, pages 133–144. ACM, 2013.

[29] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 1–56. Springer International Publishing, 2015.

[30] Stephan Brandauer, Dave Clarke, and Tobias Wrigstad. Disjointness Domains for Fine-Grained Aliasing. In *ACM SIGPLAN Notices*, volume 50, pages 898–916. ACM, 2015.

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

[32] Stephan Brandauer and Tobias Wrigstad. Spencer: Interactive Heap Analysis for the Masses. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 113–123, 2017.

[33] C# reference. `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference`. Accessed November 2017.

[34] Luís Caires and João C. Seco. The Type Discipline of Behavioral Separation. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 275–286, New York, NY, USA, 2013. ACM.

[35] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple Ownership. *SIGPLAN Not.*, 42(10):441–460, October 2007.

[36] Elias Castegren, Johan Östlund, and Tobias Wrigstad. Refined Ownership. In *Formal Methods for Multicore Programming*, pages 179–210. Springer, 2015.

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

[38] Elias Castegren and Tobias Wrigstad. Kappa: Insights, Current Status and Future Work. In *International Workshop on Aliasing, Capabilities and Ownership in object-oriented programming: IWACO at ECOOP*, 2016.

[39] Elias Castegren and Tobias Wrigstad. Reference Capabilities for Concurrency Control. In European Conference on Object-Oriented Programming, *ECOOP*,

2016.

[40] Elias Castegren and Tobias Wrigstad. Reference Capabilities for Trait Based Reuse and Concurrency Control. Technical Report 2016-007, 2016. Uppsala University.

[41] Elias Castegren and Tobias Wrigstad. Types for CAS: Relaxed Linearity with Ownership Transfer (Extended Abstract). In *NWPT*, 2016.

[42] Elias Castegren and Tobias Wrigstad. Relaxed Linear References for Lock-free Data Structures. In European Conference on Object-Oriented Programming, *ECOOP*, 2017.

[43] Elias Castegren and Tobias Wrigstad. Types for CAS: Relaxed Linearity with Ownership Transfer. *In submission*, 2017.

[44] Castegren, Elias and Wrigstad, Tobias. Actors without Borders: Amnesty for Imprisoned State. *PLACES*, 2017.

[45] Castegren, Elias and Wrigstad, Tobias. Bestow and Atomic: Concurrent Programming using Isolation, Delegation and Grouping. *In submission*, 2017.

[46] Castegren, Elias and Wrigstad, Tobias. OOlong: an Extensible Concurrent Object Calculus. 2017.

[47] M. Christakis, A. Gotovos, and K. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163, March 2013.

[48] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *ACM SIGPLAN Notices*, volume 37, pages 292–310. ACM, 2002.

[49] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[50] Dave Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In European Conference on Object-Oriented Programming, *ECOOP*, pages 176–200, 2003.

[51] David Clarke, Tobias Wrigstad, and James Noble. *Aliasing in Object-oriented Programming: Types, Analysis and Verification*, volume 7850. Springer, 2013.

[52] David G. Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, pages 48–64, 1998.

[53] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[54] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. Orca: GC and Type System Co-Design for Actor

Languages. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):72, 2017.

[55] The Coq Proof Assistant. `https://coq.inria.fr`. Accessed November 2017.

[56] Silvia Crafa and Luca Padovani. The Chemical Approach to Typestate-Oriented Programming. In *ACM SIGPLAN Notices*, volume 50, pages 917–934. ACM, 2015.

[57] Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A Complete Guide to the Future. In *ESOP*, volume 4421, pages 316–330. Springer, 2007.

[58] Benjamin Delaware, William R Cook, and Don Batory. Fitting the Pieces Together: a Machine-Checked Model of Safe Composition. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 243–252. ACM, 2009.

[59] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent Data Structures Linked in Time. *arXiv preprint arXiv:1604.08080*, 2016.

[60] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, March 1966.

[61] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic Object Re-Classification. In European Conference on Object-Oriented Programming, *ECOOP*, pages 130–149. Springer, 2001.

[62] Open source GitHub repository for Encore. `https://github.com/parapluu/encore`. Accessed November 2017.

[63] The Elixir Programming Language. `https://elixir-lang.org`. Accessed November 2017.

[64] Joel A Farrell, Stephen E Record, and Brian K Wade. Preemptive and Non-Preemptive Scheduling and Execution of Program Threads in a Multitasking Operating System, September 21 1993. US Patent 5,247,675.

[65] Introducing the New Firefox: Firefox Quantum. `https://blog.mozilla.org/blog/2017/11/14/introducing-firefox-quantum`, 2017.

[66] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[67] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 171–183. ACM, 1998.

[68] Simon Fowler, Sam Lindley, and Philip Wadler. Mixing Metaphors: Actors as Channels and Channels as Actors. *arXiv preprint arXiv:1611.06276*, 2016.

[69] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[70] Anwar Ghuloum. Face the Inevitable, Embrace Parallelism. *Communications of the ACM*, 52(9):36–38, 2009.

[71] Jean-Yves Girard. Linear Logic. *Theoretical computer science*, 50(1):1–101, 1987.

[72] The Go Programming Language. `https://golang.org`. Accessed November 2017.

[73] Georges Gonthier. Formal Proof–The Four-Color Theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[74] Colin S Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, 2014.

[75] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving That Non-Blocking Algorithms Don't Block. In *ACM SIGPLAN Notices*, volume 44, pages 16–28. ACM, 2009.

[76] Jim Gray et al. The Transaction Concept: Virtues and Limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[77] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. *ACM Sigplan Notices*, 37(5):282–293, 2002.

[78] Olivier Gruber and Fabienne Boyer. Ownership-Based Isolation for Concurrent Actors on Multi-Core Machines. In *European Conference on Object-Oriented Programming*, pages 281–301. Springer, 2013.

[79] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with Webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200. ACM, 2017.

[80] Philipp Haller and Alex Loiko. LaCasa: Lightweight Affinity and Object Capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 272–291. ACM, 2016.

[81] Philipp Haller and Martin Odersky. Scala actors: Unifying Thread-Based and Event-Based Programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[82] Philipp Haller and Martin Odersky. Capabilities for Uniqueness and Borrowing. In European Conference on Object-Oriented Programming, *ECOOP*, pages 354–378. Springer, 2010.

[83] Frédéric Haziza, Lukáš Holík, Roland Meyer, and Sebastian Wolff. *Pointer Race Freedom*, pages 393–412. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[84] Maurice Herlihy and J Eliot B Moss. *Transactional Memory: Architectural Support for Lock-Free Data Structures*, volume 21. ACM, 1993.

[85] Maurice P Herlihy and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[86] Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 Formalisms for Artificial Intelligence a Universal Modular Actor Formalism for Artificial Intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[87] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[88] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *ACM SIGPLAN Notices*, volume 26, pages 271–285. ACM, 1991.

[89] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[90] Intel. Expanding Moore's Law. `https://www.cc.gatech.edu/computing/nano/documents/Intel%20-%20Expanding%20Moore's%20Law.pdf`. Accessed December 2017.

[91] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems. *Theoretical Computer Science*, 365(1):23 – 66, 2006. Formal Methods for Components and Objects.

[92] Cliff Jones. Development Methods for Computer Programs Including a Notion of Interference. 12 2017.

[93] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the 6th ACM SIGPLAN Workshop on Higher-Order Programming. Oxford, United Kingdom*, 2017.

[94] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[95] Naoki Kobayashi. Quasi-Linear Types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 29–42. ACM, 1999.

[96] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *ACM SIGPLAN-SIGACT Symposium om Principles of Programming Languges (POPL 2017)*. ACM, 2017.

[97] R Greg Lavender and Douglas C Schmidt. Active object–an Object Behavioral Pattern for Concurrent Programming. 1995.

[98] Xavier Leroy. *The CompCert C Verified Compiler: Documentation and User's Manual*. PhD thesis, Inria, 2016.

[99] Barbara Liskov. Keynote Address - Data Abstraction and Hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM.

[100] Barbara Liskov and Liuba Shrira. *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*, volume 23. ACM, 1988.

[101] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding Featherweight Java with Assignment and Immutability Using the Coq Proof Assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 11–19. ACM, 2012.

[102] Robert C Martin. Java and C++ a Critical Comparison. 1997.

[103] Maged M Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[104] Filipe Militão. *Rely-Guarantee Protocols for Safe Interference over Shared Memory*. PhD thesis, Carnegie Mellon University & Universidade de Lisboa, 2015.

[105] Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing Control with View-Based Typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, page 7. ACM, 2010.

[106] M. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, USA, May 2006.

[107] Gordon E Moore et al. Progress in Digital Integrated Electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[108] Mpi: A Message-Passing Interface Standard, Version 3.1. `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, June 2015. Accessed November 2017.

[109] Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, volume 263. Technical Report 263, Fernuniversität Hagen. http://www. informatik. fernuni-hagen. de/pi5/publications. html, 1999.

[110] Peter Müller and Arsenii Rudich. Ownership Transfer in Universe Types. In *ACM SIGPLAN Notices*, volume 42, pages 461–478. ACM, 2007.

[111] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Type and Effect Systems. In *Principles of Program Analysis*, pages 283–363. Springer, 1999.

[112] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*, volume 2283. Springer Science & Business Media, 2002.

[113] James Noble and Alex Potanin. On Owners-as-Accessors. In *International Workshop on Aliasing, Capabilities and Ownership in object-oriented programming: IWACO at ECOOP*, 2014.

[114] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In European Conference on Object-Oriented Programming, *ECOOP*, pages 158–185, 1998.

[115] Johan Östlund. *Language Constructs for Safe Parallel Programming on Multi-cores*. PhD thesis, Department of Information Technology, Uppsala University, Jan 2016.

[116] Johan Östlund and Tobias Wrigstad. Welterweight Java. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 97–116. Springer, 2010.

[117] Lawrence C Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. 2015.

[118] Benjamin C Pierce and David N Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming*, pages 187–215. Springer, 1995.

[119] The Pony Programming Langugae. `http://www.ponylang.org`. Accessed November 2017.

[120] Pthread Read-Write Locks. `http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_rwlock_init.html`. Accessed December 2017.

[121] Reentrantlock Java Class. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html`. Accessed December 2017.

[122] John C Reynolds. Separation logic: A logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[123] The Rust Programming Language. `https://www.rust-lang.org`. Accessed December 2017.

[124] S. Clebsch, S. Drossopoulou, S. Blessing and A. McNeil. Deny Capabilities for Safe, Fast Actors. *AGERE*, 2015.

[125] The Scala Programming Language. `http://www.scala-lang.org`. Accessed November 2017.

[126] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In European Conference on Object-Oriented Programming, *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.

[127] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In European Conference on Object-Oriented Programming, *ECOOP*, volume 2743 of *Lecture Notes in*

Computer Science, pages 248–274. Springer Berlin Heidelberg, 2003.

[128] Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-Style Specifications as Correctness Conditions for Non-Linearizable Concurrent Objects. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 92–110. ACM, 2016.

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

[130] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In European Conference on Object-Oriented Programming, *ECOOP*, volume 8, pages 104–128. Springer, 2008.

[131] R. Strňisa. *Formalising, Improving, and Reusing the Java Module System*. PhD thesis, St. John's College, United Kingdom, May 2010.

[132] *Multithreaded Programming Guide*. Sun Microsystems, September 2008.

[133] Herb Sutter. The Free Lunch is Over: a Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[134] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do Scala Developers Mix the Actor Model with Other Concurrency Models? In *European Conference on Object-Oriented Programming*, pages 302–326. Springer, 2013.

[135] Tiobe Index of Programming Languages. `https://www.tiobe.com/tiobe-index/`. Accessed December 2017.

[136] R Kent Treiber. *Systems Programming: Coping with Parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[137] Unique Pointer Library for C++. `http://en.cppreference.com/w/cpp/memory/unique_ptr`. Accessed December 2017.

[138] The UPSCALE Project. `https://upscale.project.cwi.nl`. Accessed November 2017.

[139] Viktor Vafeiadis. Modular Fine-Grained Concurrency Verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[140] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. Ambienttalk: Programming Responsive Mobile Peer-to-Peer Applications with Actors. *Computer Languages, Systems & Structures*, 40(3):112–136, 2014.

[141] Philip Wadler. Linear Types can Change the World. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

[142] Edwin Westbrook, Jisheng Zhao, Zoran Budimli, and Vivek Sarkar. Practical Permissions for Race-Free Parallelism. In European Conference on

Object-Oriented Programming, *ECOOP 2012*, volume 7313 of *LNCS*, pages 614–639. Springer, 2012.

[143] What's new in JDK 8. `http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`. Accessed November 2017.

[144] Tobias Wrigstad and Johan Östlund. Owners as Ombudsmen: Multiple Aggregate Entry Points for Ownership Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming: IWACO at ECOOP*, 2011.

[145] Albert Mingkun Yang and Tobias Wrigstad. Type-Assisted Automatic Garbage Collection for Lock-Free Data Structures. *SIGPLAN Not.*, 52(9):14–24, June 2017.

Appendix A.
Notes and Errata

- In Paper I, the well-formedness rule for classes (WF-CLASS) is missing a premise for checking that traits with manifest modes are not included together with traits of different modes. The missing premise should read:

$$\forall\ (k_1\mathtt{T}_1), (k_2\mathtt{T}_2) \in K\ .\ (k_1\mathbf{trait}\ \mathtt{T}\langle\_\rangle\{\_\_\} \in P) \Rightarrow k_1 = k_2$$

- In Paper I, appendices §**D** and §**E** define the target calculus κ_F, which is an extended version of OOlong, presented in Paper V. There is therefore some overlap between the two papers. The presentation in the latter paper is a better starting point than the appendices.
- Paper III presents LOLCAT as a stand-alone type system in a procedural setting with structs, rather than a class-based object-oriented setting as in Chapter 5. While the implementation follows the style seen in Chapter 5, the system is just as applicable in a procedural setting.
- Papers II and V have been reformated from double to single column to make reading the printed thesis easier. See the references for the original versions.

Distribution: publications.uu.se
urn:nbn:se:uu:diva-336021

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018