Abstractions to Control the Future

FRANCISCO RAMÓN FERNÁNDEZ REYES
Multicore and manycore computers are the norm nowadays, and users have expectations that their programs can do multiple things concurrently. To support that, developers use concurrency abstractions such as threads, promises, futures, and/or channels to exchange information. All these abstractions introduce trade-offs between the concurrency model and the language guarantees, and developers accept these trade-offs for the benefits of concurrent programming.

Many concurrent languages are multi-paradigm, e.g., mix the functional and object-oriented paradigms. This is beneficial to developers because they can choose the most suitable approach when solving a problem. From the point of view of concurrency, purely functional programming languages are data-race free since they only support immutable data. Object-oriented languages do not get a free lunch, and neither do multi-paradigm languages that have imperative features.

The main problem is uncontrolled concurrent access to shared mutable state, which may inadvertently introduce data-races. A data-race happens when two concurrent memory operations target the same location, at least one of them is a write, and there is no synchronisation operation involved. Data-races make programs to exhibit (unwanted) non-deterministic behaviour.

The contribution of this thesis is two-fold. First, this thesis introduces new concurrent abstractions in a purely functional, statically typed programming language (Paper I – Paper III); these abstractions allow developers to write concurrent control- and delegation-based patterns. Second, this thesis introduces a capability-based dynamic programming model, named Dala, that extends the applicability of the concurrent abstractions to an imperative setting while maintaining data-race freedom (Paper IV). Developers can also use the Dala model to migrate unsafe programs, i.e., programs that may suffer data-races, to data-race free programs.

**Keywords:** concurrent, programming, type system, future, actors, active objects

Francisco Ramón Fernández Reyes, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Francisco Ramón Fernández Reyes 2021

ISSN 1651-6214
urn:nbn:se:uu:diva-425128 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-425128)
To Janina and Kai
This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I. Fernandez-Reyes K., Clarke D., McCain D.S.  
**ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations.** 18th International Conference on Coordination Models and Languages (COORDINATION’16) [92]  
A parallel abstraction that can be seen as a collection of asynchronous values or a handle to a parallel abstraction. Combinators control the abstraction and developers can express complex parallel pipelines and speculative parallelism.

II. Fernandez-Reyes K., Clarke D., Castegren E., Vo HP.  
**Forward to a Promising Future.** 20th International Conference on Coordination Models and Languages (COORDINATION’18) [89]  
The paper presents a high-level concurrent language that uses futures, and explores the combinator *forward*, that permits promise-like delegation patterns on future-based languages, reducing synchronisation. Then, it shows a compilation strategy from the high-level future-based language to a low-level promised-based language. The translation is semantics preserving and serves to drive the runtime implementation in the Encore programming language.

III. Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E. B., Wrigstad, T.  
**Godot: All the Benefits of Implicit and Explicit Futures.** 33rd European Conference on Object-Oriented Programming (ECOOP 2019) [90]  
The paper discusses two approaches to concurrent programming depending on a future dichotomy: explicit and implicit typing, and control- and data-flow futures. From this dichotomy, it identifies the problems of implicit data-flow futures and explicit control-flow futures and proposes a new design that solves these problems, formalised as Godot. This design is formalised for two calculi: first an encoding of control-flow futures in terms of data-flow futures, and second an encoding of data-flow futures in terms of control-flow futures.

IV. Fernandez-Reyes, K., Noble, J., Gariano, I.O., Greenwood-Thessman, E., Homer, M., Wrigstad, T.  
**Dala: A Simple Capability-Based Dynamic Language Design For Data-Race Freedom.**  
This paper discusses the design of the *Dala* programming model, a simple dynamic, concurrent, object-oriented language that maintains data-race
freedom in the presence of shared mutable state and supports efficient inter-thread communication. Dala is a capability-based language that relies on safe and unsafe capabilities. There are three safe capabilities and these capabilities grant permission to their possessor to perform certain actions, e.g., read, write, or alias an object. Unsafe objects grant all permissions to their possessors. Safe and unsafe objects may interact and Dala guarantees data-race freedom on safe objects.

Reprints were made with permission from the publishers.

The Author’s Contributions


II. Main author. Formalisation and manuscript written (primarily) in collaboration with second author. Implementation written in collaboration with all authors.

III. Main author. Manuscript written together with all authors. Formalisation written in collaboration with second author. Sole contributor of proofs and implementation.

IV. Main author. Manuscript written with all authors. Formalisation written with third and last author. Sole contributor of proofs.

Related Publications

Other relevant publications by the author that are not included in the dissertation are listed below:

- Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L., Wrigstad, T., Yang, A.M.

  This paper discusses the ongoing features of the Encore language, motivation for a new concurrent language, and future directions.

  A Survey of Active Object Languages. ACM Comput. Surv. 2017 [74]
This paper surveys actor and active object languages and compares them across a carefully selected set of dimensions.

- Castegren, E., Clarke, D., Fernandez-Reyes, K., Wrigstad, T., Yang, A.M. **Attached and Detached Closures in Actors.** International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2018 [46]

  This paper discusses the problem of choosing which actors can run closures, without introducing race conditions, and shows the approach taken by the Encore language.

- Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E. B., Wrigstad, T. **Godot: All the Benefits of Implicit and Explicit Futures (Artifact).** DARTS 2019 [91]

  This artefact shows a minimalistic Scala library that encodes data-flow futures in terms of control-flow futures, and explains some of the current limitations and implementation deviations from the paper.


  This paper introduces a gradual capability-based language that guarantees data-race freedom. This paper is a work-in-progress report.


  This paper shows the runtime characteristics of 3 actor-based languages, based on the Savina benchmarks, and shows how many benchmarks have been categorised differently but show the same runtime characteristics. The paper proposes a new benchmark that can simulate most of the Savina benchmark programs.


  This experience report shows how the Encore team used Haskell to develop the Encore compiler.
Note:
To avoid confusion on how to cite my name, I removed my second name (Ramón), tildes, and used a hyphen between the last names (Fernandez-Reyes). There was another researcher with a name pretty similar to mine, Francisco Ramón Fernández, so I decided to change Francisco to the diminutive of Kiko. Thus, I have signed all papers as Kiko Fernandez-Reyes, instead of Francisco Ramón Fernández Reyes.
Sammanfattning på svenska

På 1960-talet publicerades de första artiklarna om *concurrent algorithms*, alltså algoritmer som involverar många diskreta processer som överlappar i tid, dock inte med nödvändighet exakt samtidigt\(^1\). Concurrent-programmering och -algoritmer uppstod av nödvändighet för att kunna konstruera operativsystem med förmåga till multitasking, och hantera många användare uppkopplade till ett system samtidigt.

Design av mjukvara som innefattar sådant beteende är svårt. Om flera processer gör åtkomst till samma minne samtidigt kan resultatet bli icke-deterministiskt och därför variera mellan körningar beroende på t.ex. schemaläggningen av processerna. Detta gör att sådana system kan lida av olika typer av "kapplöpningsproblem": när två processer utan synkronisering gör åtkomst samma plats i minnet samtidigt, och minst en av processerna skriver (eng. data-races) eller att schemaläggningsordningen av två processer påverkar programmets beteende (eng. race condition).

Mot 1960-talets slut stod det tydligt att världen stod inför en mjukvarukris om det inte var möjligt att tygla kapplöpningsproblemen med hjälp av kontrollmekanismer och programspråkliga abstraktioner.

Idag är system som involverar parallella och samtidiga processer norm. Programspråk och programbibliotek möjliggör olika avväganden mellan t.ex. prestanda och komplexitet, och olika språk eller programmeringsmiljöer erbjuder olika programmeringsmodeller som lyfter fram eller görmer dessa aspekter. Vissa programspråk garanterar frihet från vissa typer av kapplöpningsproblem (data-races) i korrekta program, bl.a. genom att kontrollera hur förändringsätt data kan delas samtidigt mellan processer (programspråk som Encore, Erlang

---

\(^1\)På svenska används ibland ordet "samtidighet" som översättning för "concurrency" men det leder ofta till språkliga konstigkeiten; "många samtidiga diskreta processer". Vi använder därför den ibland engelska termen även på svenska.
och Pony); andra språk lämpar istället över ansvaret på programmeraren som i utbyte får större kontroll och möjlighet att optimera ett system på låg nivå och skriva “osäker kod” (programspråk som Java, Scala och bibliotek i dessa språk som t.ex. Akka). Anmärkningsvärt är att ren funktionell programmering är fri från vissa kapplöpningsproblem (data-races) genom sin konstruktion. Språk i den imperativa familjen, typiskt många objektorienterade språk, ger inga sådana garantier då det är vanligt att objekt delas mellan olika delar av ett program, och att dessa delar kommunicerar med varandra genom att förändra objekten. De 5 mest populära programspråken idag (enligt TIOBE-listan) – C, Java, Python, C++ och C# – är alla imperativa och ger inga sådana garantier.

Denna avhandlings huvudsakliga bidrag är ett antal högnivå-abstraktioner för att uttrycka samtliga beräkningar, inklusive spekulativa beräkningar, samt design av en programmeringsmodell som är garanterat fri från kapplöpningsproblem (data-races). I de fall där programmerare vill använda konstruktioner som bryter garantin är det möjligt att kontrollera förekomsten av kapplöpningsproblem på objektivnå.

Abstraktionernas bygger på “uppgifter” (eng. tasks, dvs. välavgränsade se-kvenser av instruktioner som kan exekveras av trådar), utlovade värden (promises), framtidiga värden (futures), och – i en imperativ miljö – “förmågor” (capabilities) för att utesluta kapplöpningsproblem (data-races). Informellt kan uppgifter betraktas som trådar, utlovade värden som värden som kan skrivas en enda gång men läsas fritt, framtidiga värden kan ses som utlovade värden som infras av uppgifter, och förmågor är biljetter som ger rätt att utföra operationer på objekt, t.ex. rätt att aliasera, uppdatera, etc.

Arbetet tar avstamp i en ren funktionell miljö där vi utvecklar en abstraktion för parallella beräkningar, kallad ParT. ParT bygger på framtidiga värden och möjliggör konstruktion och koordinering av komplexa mönster., t.ex. parallella pipelines av spekulativa beräkningar i ett nätverk av uppgifter. En uppgifts slutresultat propageras genom systemet med hjälp av framtidiga värden, vilket gör det möjligt att specificera mönster där en uppgift bygger vidare på resultatet från flera andra, eller väljer något av flera spekulativa resultat. För att möjliggöra mönster där uppgifter delegerar arbete mellan sig utvecklar vi en ny programmedräkskonstruktion kallad forward. Med hjälp av denna konstruktion kan en uppgift delegera beräkningen av ett framtidigt värde på en annan uppgift och på så sätt ta sig själv från den kritiska vågen för världets beräkning. Som ett led i detta arbete utvecklas ny teori för att hantera framtidiga värden från ett typperspektiv som skiljer mellan sykronisering i kontrollflödet och dataflödet i ett program utan att för den skull pådyvla en typmodell på programmet, vilket var fallet innan vårt arbete.

Framtidiga värden som bygger på kontrollflöden kan användas för att skapa delegeringsstrukturer med flera led där varje led kan informeras om framsteg i hur ett resultat propageras genom systemet. Framtidiga värden som bygger på dataflöden åtnjuter en enklare och delvis renare programmeringsmodell, till priset av mindre kontroll.

Med avseende på implementation går avhandlingens huvudbidrag att applicera på actorspråk.
Acknowledgements

I would like to thank people that supported me during my PhD studies.

Dave, you were a great supervisor and always had my back behind all the math symbols we wrote. You have taught me how to have fun with mathematical non-sense and I cannot wait to send this thesis to the printer to try (again) to understand category theory. You went to industry a bit too early, and I miss our conversations about monads, pro-functors, and how to add pre- and post-promise chaining functions. I hope we can keep in touch.

Tobias, you are great supervisor and person. I will remember all the good moments we had, all the pizza and hackathon events, and I will never forget how you supported me in the bad moments of the journey.

Janina, you have been my main anchor in this journey and have always taken one for the team. I will always be grateful to you for giving me strength and support, and for always believing in me. I love you!

Kai, what can I say? You have made the PhD much more difficult, but I really enjoy going home and having family time. You are the best person in this world, always maintain your innocence. Love you!

Davide and Pierre, thank you so much for all the talks, lunch, and coffees that we have shared during my last year of PhD studies. Davide, I am going to miss our early coffee and talks, I am not sure what I will do anymore! The early coffee was setting a good vibe for work.

Stephan, Kim, Albert, Kike, and Elias, thank you so much for making me have fun during the PhD, and for not judging possible stupid questions throughout all these years.

Raphaela, you are the best TA one can get for the course Advanced Software Design. Thanks to you, I never had to worry about the students and they showed how good you are in the course evaluations :)
Einar, Sophia, and Juliana, you have always encourage us (PhD students) to get the best of ourselves, challenge our opinions in a constructive way, and get us to be the best we can be. I will really miss working with the three of you. Please, do not hesitate to contact me in the future if there is any possibility for collaboration :)

Isaac, Erin, Michael, and James, thank you so much for taking me under your research group, for supporting my work, and for refining ideas. Hopefully I can repay you back by having our work published in a good venue.

Ulrika, Anna-Lena, and Eva, thank you for making so easy all the paperwork, financial stuff that had to be sorted out, and all the work that you all did so that me and my family could go on the research visit to Wellington.

Loreto and everyone in the administration, thank you so much for always helping me out to navigate the Swedish Ladok or Uppdok system. You have always been super helpful.

Alberto and Ale, thank you for being really good brothers; that said, Ale, you need to visit your nephew more often, less partying! Alberto, we have always been together in this Computer Science journey, modulo the PhD studies. Thank you for listening to my frustations, and the research that I do.

Mum and dad, you thought that research was relaxed and a “real” job was stressful. I think you are wrong :) but thank you for supporting me all these years and for all the travelling you have done to Uppsala, to help us when we needed it.

To everyone else that I may have forgotten, Thank you!
Contents

1 Introduction ................................................................. 17
  1.1 Contributions ...................................................... 18
  1.2 Outline ............................................................... 21

2 Concurrency and Communication Abstractions .................... 22
  2.1 Concurrency ......................................................... 22
  2.2 Synchronisation and Communication Patterns ............. 24
    2.2.1 Futures and Promises ........................................ 24
    2.2.2 Channels .......................................................... 26
  2.3 Concurrency Problems .......................................... 28
    2.3.1 Data-Races ...................................................... 29
    2.3.2 Deadlocks ......................................................... 29
    2.3.3 Performance and Synchronisation Granularity ....... 31
  2.4 Concurrency and Synchronisation in Context ............... 32

3 Object-Oriented and Functional Programming .................... 34
  3.1 Object-Oriented Programming ................................ 34
    3.1.1 Concurrency Perspectives In Object-Oriented Languages ........................................ 35
  3.2 Functional Programming ....................................... 36
    3.2.1 Concurrency Perspectives In Functional Languages ............................................. 37
    3.2.2 Task-based Simply Typed Lambda Calculus ...................................................... 38

4 Related Work ................................................................ 42
  4.1 Actor-Based Concurrency Models ................................ 42
  4.2 Concurrent Asynchronous Abstractions ....................... 44
  4.3 Speculative Computations ...................................... 46
  4.4 Futures & Promises ................................................. 48
    4.4.1 History ............................................................. 48
    4.4.2 A Future Categorisation ...................................... 50
  4.5 Capability-Based Languages .................................... 52
    4.5.1 Introduction To Capability-Based Languages ....... 52
    4.5.2 Ideas Adopted In Capability-Based Languages ....... 53
1. Introduction

In the early 60s, researchers published first results on concurrent algorithms [82, 76, 192, 102, 221, 191]. Concurrent algorithms are those that allow multiple computations or processes to overlap in time, though not necessarily executing at the same instant [111]. The theory behind concurrent algorithms and concurrent programming was born out of the necessity to create operating systems that could perform multiple tasks (multiprocessing) and allow users to connect to a single computer, concurrently [20, 21, 209, 69].

Designing concurrent software is hard. If multiple processes access and modify the same memory cell concurrently, then the execution of a program may return different results on each run. This is dependent on the scheduling of processes and on the read and write operations on the memory cell. Thus, concurrent software is subject to race conditions [120] and data-races, defined as follows: a race condition happens when the ordering of the events affects the behaviour of the program, and a data-race happens when two concurrent memory operations target the same location, at least one of them is a write, and there is no synchronisation operation involved (definition adapted from [88]).

In the late 60s, researchers realised that without any kind of structure or abstraction that could prevent concurrency issues (data-races and race-conditions among were among these issues), designing a concurrent system was a monumental effort and they started to speak about a software crisis [174].

Nowadays, concurrent and parallel programming is the norm [35, 14, 148, 63, 110, 223, 74]. Programming language and library designers offer different trade-offs between the concurrency models and the language guarantees. Some programming languages offer data-race freedom guarantees by restricting sharing of mutable state (e.g., [35, 63, 14]) while others leave more control to the developer at the expense of unsafe (e.g., not data-race free) guarantees (e.g., [148, 223, 181]). For example, programming languages under the (pure) functional paradigm are data-race free by definition [158]. On the other end, (imperative) object-oriented languages are usually not data-race free. Today, the 5 most used programming languages are imperative at their core\(^1\) [212], and 4 out of 5 of these languages mix the object-oriented and the functional paradigm. Thus, concurrency abstractions cannot guarantee data-race freedom.

\(^1\)C, Java, Python, C++, and C#, Tiobe Index June 2020
1.1 Contributions

The main contribution of this thesis consists of high-level purely functional abstractions to express concurrent and speculative computations, and the design of a concurrent programming model that extends the applicability of the concurrent abstractions to an imperative setting while maintaining data-race freedom, when desired. In cases where developers do not want to maintain data-race freedom, the programming model allows data-races on a per-object granularity.

The concurrent abstraction relies on tasks, promises, and futures, while the object-oriented language uses capabilities to maintain data-race freedom. Informally, we can think of tasks as virtual threads, promises as value placeholders that can be written once and read multiple times, of futures as promises that are implicitly fulfilled by the task’s returned value, and of capabilities as tokens granting special permissions to its (object) possessor, e.g., ability to read, write, or alias. (We describe tasks, promises, and futures in Section 2.1, and capabilities in Section 4.5.)

Our work starts in a task-based concurrent purely functional setting, where we develop a parallel abstraction, named ParT, that uses futures at its core and allows to easily create complex coordination patterns, such as the creation of concurrent and parallel pipelines of speculative tasks. In this task-based concurrent setting, the task’s returned value implicitly fulfils a future. The implicit future fulfilment semantics prevents developers from writing common delegation patterns when using futures, e.g., the delegation of a future’s fulfilment to another task. To delegate future fulfilment, this thesis investigates a construct named forward and introduces a compilation strategy from a high-level future-based programming language to a promise-based low-level programming language, that can encode certain delegation patterns. Then, this thesis uses the forward delegation core idea, and new combinators and types to express control-flow futures (the mainstream futures similar to those found in Java, Scala, or Python) and data-flow futures. Whereas control-flow futures can nest futures and control individual access to each future layer (Section 2.2.1), data-flow futures abstract nesting and synchronisation operations traverse the possible nested futures and return a non-future value. For example, a control-flow future $f$ with type $\text{Fut}[\text{Fut}[\text{Int}]]$ can synchronise on each future layer, while a typed data-flow future $f$ cannot statically exhibit future nesting, i.e., the previous future $f$ would be typed as $\text{Fut}[\text{Int}]$ and represents a future that may have nested future layers at runtime and synchronisation operations cannot show intermediate synchronisation steps.

The abstractions and delegation patterns introduced in this thesis (Papers I–III) are data-race free in a purely functional setting, but data-race freedom is lost in the presence of mutation. To address this problem, this thesis proposes

---

2 These computations are speculative in the sense that the user may or may not be interested in all the results [123].
a capability-based programming model, *Dala*, that extends the applicability of Papers *I–III* to an imperative setting, retaining data-race freedom. The programming model also permits transitioning from an unsafe program, *i.e.*, a program that is subject to data-races, to a program that maintains data-race freedom.

From the implementation point of view, most of the ideas of this thesis can be applied to an actor or active object programming language. We leave as future work to statically type the *Dala* model and to re-write the functional abstractions in the *Dala* model.

Below we give a summary of our work.

**PAPER I**

ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations

We develop a concurrent abstraction, *ParT*, that allows developers to express pipelines of concurrent speculative computations in a task-based language. Spawning a task returns immediately a (control-flow) future, and futures are placeholders for values that may not have been yet computed. Values and futures can be lifted to the *ParT* abstraction, and *ParT*s are monoids, *i.e.*, a bunch of *ParT* abstractions can be grouped under a new *ParT*. Developers can write complex concurrent (speculative) coordination patterns using *ParT*’s high-level non-blocking combinators. Speculative termination of tasks stops tasks that are not needed.

**PAPER II**

Forward to a Promising Future

Control-flow futures from *Paper I* can be lifted to the *ParT* abstraction. But we noticed how the semantics of control-flow futures are often too rigid, *i.e.*, tasks implicitly fulfil a future upon termination which prevents users from writing common delegation patterns. For example, a client that communicates with a (proxy) server immediately gets back a future. If the proxy spawns a worker to handle the request and returns the worker’s future, the client is exposed to the internal structure of the (proxy) server, *e.g.*, a future to a future. If the proxy spawns a worker to handle the request and blocks on the worker’s future, the proxy cannot attend new requests and may become the bottleneck of the server. A fulfilment delegation pattern can transfer “ownership” of the fulfilment of the future.

---

3There is only one blocking combinator, needed to convert the *ParT* abstraction into an array.
We introduce the forward combinator, which allows some flexibility degree on delegation of the future’s fulfilment. Following the example from above, the forward combinator allows a direct response from the worker to the client, without involving the proxy, in a transparent way. This delegation pattern removes the intermediate synchronisation from the proxy and delegates the fulfilment of the client’s future to the worker.

**PAPER III**
Godot: All the Benefits of Implicit and Explicit Futures

Control-flow futures cannot abstract over nested futures without peeling the futures layers using synchronisation combinators, among other issues. The use of the forward construct (*Paper II*) allows developers to encode data-flow futures using control-flow futures. But this requires manually inserting the forward construct to encode data-flow futures, and data-flow futures do not have explicit types. Thus, they are not recognisable to the developer in the type signature.

To allow the co-existence of control- and data-flow futures, our work uses the core ideas of *Paper II* and adds new data-flow combinators and a data-flow type. This paper introduces typed data-flow futures and its combinators which implicitly delegate the fulfilment of the future. This is the first work, that we know of, that allows interoperability of control- and data-flow futures.

**PAPER IV**
Dala: A Simple Capability-Based Dynamic Language Design For Data-Race Freedom

The abstractions and delegation patterns introduced in this thesis are data-race free in a purely functional setting. But data-race freedom is lost in the presence of mutation. To solve this problem, this thesis proposes a capability-based programming model, named *Dala*. Dala distinguishes between safe and unsafe capabilities. Safe capabilities grant special permissions to its (object’s) possessor, *e.g.*, ability to read, write, or alias an object. Unsafe capabilities grant all permissions. Dala allows interaction between safe and unsafe objects and guarantees data-race freedom for safe objects. This guarantee is enforced via runtime checks.
1.2 Outline

The following outline shows the organisation of this thesis:

Chapter 2. Background on concurrent programming (Section 2.1), synchronisation and communication patterns (Section 2.2), concurrency problems (Section 2.3), and connection to our work (Section 2.4).

Chapter 3. Covers basic notions of the object-oriented (Section 3.1) and functional paradigms (Section 3.2), and introduces a task-based lambda calculus, used in Papers I – III.

Chapter 4. Overviews actor-based concurrency models (Section 4.1), concurrent asynchronous abstractions (Section 4.2), speculative computations (Section 4.3), and futures (Section 4.4). It also presents a new future categorisation based on four dimensions, introduces capability-based languages and features commonly used in them (Section 4.5), an overview of concurrent programming languages (Section 4.6), and finishes with a discussion section (Section 4.7).

Chapter 5 Concludes.
2. Concurrency and Communication Abstractions

This chapter reviews common concurrency abstractions, some of the problems introduced by these abstractions, and synchronisation patterns needed to understand this thesis. Section 2.1 explains common concurrency abstractions (threads and tasks); Section 2.2 reviews synchronisation and communication patterns in concurrent programs; Section 2.3 explains problems in concurrent programs; Section 2.4 connects the concurrency abstractions, synchronisation, and trade-offs between the concurrency abstractions and language guarantees, captured on a per-paper level.

2.1 Concurrency

Threads are the minimal computational unit scheduled by an operating system (OS); multiple threads have access to the same virtual memory and threads from different processes access disjoint virtual memory [201, 208]. Multi-threaded programs are concurrent by definition. By concurrent we mean that the OS gives each thread some amount of time to run and, when the thread runs out of its given time slice, the OS stops the running thread and runs another thread. Thus, two (or more) threads run concurrently but not at the same time. A multi-threaded program runs in parallel when multiple threads run at the same time.

Programming languages provide libraries for operating with threads. The most common operations are the creation and joining operations, typically named \texttt{fork} and \texttt{join} [70, 67, 179]. These operations create and run new threads and wait for a thread to finish. For example, Fig. 2.1 shows a parent thread spawning a child thread (Line 10) to perform a calculation (Line 2). Then, the parent thread continues doing other work (Line 14), and blocks until the child thread finishes (Line 16).

The creation and destruction of too many threads affects the performance of a running program, as each thread needs to allocate OS resources, free them, and the OS needs to context switch between threads [124, 151].

To mitigate this problem, researchers added new abstractions for cheap creation and distribution of work, namely tasks and work stealing [95, 193]. Under this programming model, developers specify possible points of concurrency and parallelism using a construct named \texttt{spawn}. Each thread maintains

---

1The parent and child thread have access to the same mutable state, the \texttt{counter} variable. Section 2.3 overviews common problems when sharing mutable state.
// Function executed by child thread
def calculation(counter):
    counter += 1
    ...

// parent thread
counter = 0

// Run function in child thread
child = Fork(calculation, counter)

// Perform other work
counter += 1
...
counter.join()

Figure 2.1. Pseudo-code where a parent thread forks a child thread to perform a calculation, performs some work and waits for the child thread to finish. Parent and child shared access to the variable named counter.

A doubly-ended queue of tasks to run. A thread tries to steal a task from other thread’s queue when there is no more local work to do [193, 95, 149, 151]. The await construct waits for the completion of a spawned job.

An update of the previous parent-child example is in Fig. 2.2. In this example, the only syntactic change was the use of the function spawn instead of Fork and await instead of join (Fig. 2.2, Lines 10 and 16, respectively). However, from the runtime perspective, spawning a new task expresses the desire that the task’s computation may run concurrently (or in parallel), but the runtime could also sequentialise it.

// Function executed by child task
def calculation(counter):
    counter += 1
    ...

// parent thread
counter = 0

// Run function in child task
child = spawn(calculation(counter))

// Perform other work
counter += 1
...

child.await()

Figure 2.2. Pseudo-code where a parent task spawns a child to perform a calculation, performs some work and waits for the child task to finish. Parent and child shared access to the variable named counter.
2.2 Synchronisation and Communication Patterns

Threads and tasks are similar, and their main difference is that while a thread is the minimal computational unit scheduled by an operating system, a task is a computational unit scheduled on a thread. Threads and tasks have similar synchronisation constructs, namely join and await, and rely on shared memory to indirectly return a value from a child thread/task. This indirect way to synchronise, or get a value as a result of a multithreaded computation, is error prone [183]. Low-level synchronisation and communication patterns such as locks, monitors, and semaphores, among others [81, 116, 36, 83, 37], provide facilities for synchronisation and getting a value as a result of a spawned computation, but the logic becomes difficult to understand [183]. This thesis focuses on higher level abstractions for synchronisation and communication of threads and tasks.

In this section we review two high-level communication and synchronisation concepts: futures and promises, and channels. Future and promises decouple the return of an asynchronous computation from how the value is computed (Section 2.2.1). We define an asynchronous computation as a computation that does not block the current thread (task) and takes place at some other point in time. Channels are an abstraction to explicitly control the sending and receiving of values between threads (tasks), such that synchronisation can happen explicitly and without waiting for a task to finish (Section 2.2.2).

2.2.1 Futures and Promises

Futures were originally introduced in an untyped setting [17, 133, 224], and Liskov et al. later moved them to a typed setting and renamed them as promises in Argus [156]. The main idea is that futures and promises decouple the return of a value from how the value is computed, and are placeholders for asynchronous computations.2 (From now on, in this chapter we will refer to a task to mean either a thread or a task, abstracting over implementation details.)

In the recent literature, concurrent programming languages maintained the names of these abstractions but changed its semantics slightly [165, 71, 35, 176, 187, 7]. Our work uses the following semantics when we refer to futures and promises:3

**Definition 1 (Future)** A future is a read-only placeholder for the result of an asynchronous computation, where the callee implicitly fulfils the future upon returning of a value.

---

2 We will refer to a concurrent computation to mean that two or more computations have interleaving semantics, possibly running in the same thread, and these computations do not execute at the same time, and we refer to an asynchronous computation as a computation that does not execute immediately (synchronously) and runs at some non-specified point in time.

3 A more technical definition is given at the end of Section 4.4.
Table 2.1. Operations typed in systems with futures and promises, respectively, including operation style (blocking or non-blocking) where “–” means not applicable.

<table>
<thead>
<tr>
<th>Future Type</th>
<th>Promise Type</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>async  () → Fut[t]</td>
<td>−</td>
<td>Non-blocking</td>
</tr>
<tr>
<td>get(f) Fut[t] → t</td>
<td>Prom[t] → t</td>
<td>Blocking</td>
</tr>
<tr>
<td>Prom – t → Prom[t]</td>
<td>−</td>
<td>Non-blocking</td>
</tr>
<tr>
<td>fulfill – Prom[t] → t → Unit</td>
<td>−</td>
<td>Non-blocking</td>
</tr>
<tr>
<td>$f \rightsquigarrow (\lambda x. e)$ Fut[t] → (t → t')</td>
<td>Prom[t] → (t → t') Prom[t']</td>
<td>Non-blocking</td>
</tr>
</tbody>
</table>

Definition 2 (Promise) A promise is a data structure that can be fulfilled (written) once and read multiple times.

From the definitions, it is implicit that a future is tied to the asynchronous computation (task) that fulfils it and that a future is fulfilled only once; promises are lower-level abstractions not tied to any asynchronous computation. When used in asynchronous computations (e.g., a promise shared with another task), promises decouple values from asynchronous computations, but promises must be explicitly managed. Because promises are not implicitly linked to a task or implicitly fulfilled upon task termination, promises can simulate futures, but futures cannot simulate all the behaviours of a promise. For this reason, we consider futures as higher-level constructs than promises. Paper II uses this reasoning to define a future-based higher-level language, and a compilation strategy to a promise-based lower-level language that allows future-based programs to encode promise-like delegation patterns.

Futures and promises have a small core set of combinators to operate on them, listed in Table 2.1 together with their type signature and asynchronous operational description. (There are other derived combinators, but we will not cover them here.) The asynchronous operational description merely asserts whether the combinator is blocking or not. A blocking combinator can be considered a synchronisation point, as it guarantees the presence of a value in a future (promise) before it can continue. Next, we provide a description of these combinators:

- **async** Spawns an asynchronous computation, returning immediately a future.
- **get** A synchronisation operation that blocks on the future or promise until it is fulfilled (has a value).
- **Prom** Creates a promise. A promise can be fulfilled once and raises an error if it is fulfilled multiple times.
- **fulfill** Fulfils a promise with a given value.
- **$f \rightsquigarrow (\lambda x. e)$** Future- and promise-chaining operation: returns immediately a new future (promise), and attaches the computation $\lambda x. e$ as a callback to $f$. For example, $f \rightsquigarrow (\lambda x. e)$ returns a future $g$ and attaches the com-
Figure 2.3. Pseudo-code that spawns a task to perform a calculation, performs some work and waits for the future (left listing) or promise (right listing) to finish. The current task and its spawnee have access to the mutable variable named `counter`.

### Channels

Channels are abstractions that allow direct communication between two tasks [117], and can be unidirectional or bidirectional. In an unidirectional channel one end of the channel allows tasks to send (but not receive) messages, and the other end of the channel allows tasks to receive (but not send) messages [115].

Channels can be synchronous or asynchronous. A channel is synchronous when a send operation blocks the sending task until another task receives a value.

---

4 We assume that the `spawn` computation in the promise listing has type: () → ()
Figure 2.4. Graphical representation of a synchronous channel where two tasks exchange a value. (1) Task #1 sends a value to a channel; (2) Task #1 blocks until another task receives the value; (3) Task #2 receives the value from the channel; (4) No task is blocked.

For this reason, channels are considered synchronisation points between two tasks, where two tasks wait for each other to exchange a value, and continue afterwards. A channel is asynchronous when it incorporates a buffer of size $S$, where messages accumulate in FIFO order until the buffer is full. In the most common case, when the buffer is full the sender blocks (in other designs the buffer may drop messages once it is full). Buffered channels of infinite size (also known as unbuffered channels) accept all messages and never block the sender [50]. (More channel designs and alternatives in [205]).

In Paper IV, we adopt channels from the CSP programming model, i.e., channels are bidirectional and synchronous [117], with the restriction that we forbid passing a channel to another channel. This restriction allows us to focus on key concepts, and we do not think that this limitation invalidates the programming model developed in Paper IV. The use of channels was a pragmatic choice that allows developers to pass values back and forth between tasks (c.f., futures, Section 2.2.1). For simplicity Paper IV unifies task and channel creation; Table 2.2 summarises the channel combinators:

- $\text{spawn}(x) \ldots$ Spawns a new computation that executes $\ldots$ and immediately returns a channel; the variable $x$ represents the channel that the spawned task uses to communicate with the caller.
- $x \leftarrow \text{ch}$ Sending operation that places the value $x$ inside the channel $\text{ch}$, blocking if the channel is full.
- $\leftarrow \text{ch}$ Receiving operation that extracts a value from the channel $\text{ch}$, blocking if the channel is empty.
Table 2.2. *Channel’s combinator types and blocking semantics*

<table>
<thead>
<tr>
<th>Signature</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spawn(x){...}</code></td>
<td>unit → chan Non-blocking</td>
</tr>
<tr>
<td><code>ch ← value</code></td>
<td>chan → t → chan Blocking</td>
</tr>
<tr>
<td><code>← ch</code></td>
<td>chan → t Blocking</td>
</tr>
</tbody>
</table>

1 // Function executed by task
2 `def job(counter, ch):`
3 `counter += 1`
4 ...
5 `ch ← counter`
6 `counter = 0`
7 `ch = spawn(x) { job(counter, x) }`
8
9 `counter += 1`
10 ...
11 `← ch`
12

*Figure 2.5.* Pseudo-code that spawns a task to perform a calculation; the tasks synchronise using channels.

Fig. 2.5 adapts the example from Section 2.1 of a parent and a child task that share a counter, but use channels in its stead. The spawning constructor (`spawn(x){ ... }`) returns a channel, where variable `x` represents the channel name used by the spawnee task to communicate with its caller (Line 7). Channels have two operations: one for sending a value to a channel (`ch ← counter`, Line 5) and one for receiving a value (`← ch`, Line 12). In the example (Fig. 2.5), the child task finishes by placing a value in the channel (Line 5), and the parent task receives the value (Line 12).

2.3 Concurrency Problems

Designing a concurrent (parallel) program is not an easy task, specially because it is easy to introduce data-races and these may have harmful consequences [138, 227]. We use the following definition of a data-race, adapted from [88]:

**Definition 3 (Data-Race)** A data-race happens when two concurrent memory operations target the same location, at least one of them is a write, and there is no synchronisation operation involved.

The use of synchronisation operations may remove data-races, but they may also introduce deadlocks and over-synchronisation may impact the performance of the program [66, 105, 200].

The next sections show examples of synchronisation operations that may suffer from data-races and deadlocks, or performance regressions due to over-synchronisation. This section finishes stating which abstractions from our work may suffer data-races and deadlocks.
2.3.1 Data-Races

Fig. 2.6 shows an example (borrowed from Fig. 2.3) of a concurrent program that uses a pass-by-reference evaluation strategy, futures, and suffers from a data-race. The program starts by setting the variable `counter` to 0 (Line 7). The parent task spawns a child, delegating the computation of some `job` operation (Line 8), and gets back immediately a future – parent and child (may) run concurrently. Assume the following scheduling, where the parent gets to run first: the parent increments the counter (Line 9); then the child is scheduled. The child updates the counter (Line 3), and this introduces a data-race. This is a data-race because there are two accesses involved on the same memory location, at least one of them is a write, and the operations are not synchronised (Definition 3).

The main implication of a data-race is non-deterministic behaviour, and nothing can be said about the result contained in the `future` variable; the result of the future variable may not even coincide with the result of `counter` [19]. Obviously, this is not the intent of the programmer. The intent of the programmer was to update the counter once, in the child task, and once in the parent task.

The `spawn` computation of Fig. 2.6 captures the variable `counter`, and this variable should not be used again in the parent process until the child returns it. If one forbids any access to captured variables (and memory locations reachable from the reachable object graph of the captured variable) until the variable is retrieved from a future, then the programmer has a guarantee that there are no data-races. This can be written as follows:

```python
counter = get(async(job(counter))); counter += 1.
```

2.3.2 Deadlocks

A deadlock happens when a group of tasks wait for a condition to change before they can continue [36, 126]. A circular dependency among tasks causes a deadlock since each task waits for another task to remove its blocking condition. The consequences of deadlocks range from blocked operating systems to standstill cranes in automated container terminals or blocked trains that compete for the same track when there is not enough trackage.
buffer [132, 150, 157]. The coming subsections show how tasks may suffer from deadlocks, and deadlocks produced between the interaction of tasks and futures, promises, and channels.

**Deadlocks in Tasks**

Task-based programs deadlock when there is a circular dependency between tasks waiting for each other to remove their blocking condition.

Dijkstra introduced a simple example of a concurrent program that may deadlock, the dinning philosophers problem [117] (definition follows). There is a round table where $N$ philosophers sit next to each other to eat a bowl of spaghetti. There is a single fork between each pair of philosophers. Each philosopher alternates between thinking and eating, but a philosopher must pick both forks before eating. A philosopher that finishes eating must release the forks so that other philosophers may pick them up. The amount of spaghetti is infinite.

```python
class Philosopher:
    id: Int
    left: Fork
    right: Fork

    def think():
        ...
    def eat():
        ...
    def pick(forks : List[Fork]):
        while !this.left.isFree():
            this.think()
        forks[this.id].notFree()
        this.left = forks[this.id]
        while !this.right.isFree():
            this.think()
        rFork = this.id+1 % X
        forks[rFork].notFree()
        this.right = forks[rFork]
        this.eat()
        this.left.release()
        this.right.release()
        this.run(forks)

class Main:
    def main():
        forks = ...
        philosophers = List
        for x in 0..5:
            phi = new Philosopher()
            spawn(phi.pick(forks))
            philosophers.add(phi)

    def main():
        forks = ...
        philosophers = List
        for x in 0..5:
            phi = new Philosopher()
            spawn(phi.pick(forks))
            philosophers.add(phi)
```

*Figure 2.7. Philosophers problem using tasks, possibly deadlocking.*

Fig. 2.7 shows a possible implementation, where the class Philosopher contains the attributes id, used to know which fork to pick up, and then two forks. Each philosopher can think (Line 6), eat (Line 8), or pick up forks (Line 11). When a philosopher tries to pick up a fork, the philosopher checks whether the left fork is available (Line 12) and only try to pick up the right fork when it has the left fork (Line 18).

It is easy to see how the code in Fig. 2.7 does not prevent deadlocks. All philosophers could concurrently decide to pick their left fork, waiting now for other philosophers to release their right fork. But all of them are waiting on the same condition, that someone releases the right fork before they can continue, so there is no progress.
Deadlocks in Futures and Promises

Futures and promises may suffer from deadlocks when a future (promise) is not fulfilled. Fig. 2.8 (left), shows a deadlock produced when a shared object \((x.f, \text{Line 5})\) that contains a future is concurrently updated (in the parent task) with the own spawnee generated future \((x.f = \text{fut}, \text{Line 9})\). Depending on the scheduling, the child may block on the initial future in \(x.f\) (no deadlock) or on the spawnee future, creating a deadlock. Fig. 2.8 (right) shows a deadlock produced by an unfulfilled promise, which does not even need of any concurrency construct. The example creates a promise (Line 6), calculates some decimals of \(\pi\) and waits forever for the promise to be fulfilled (Line 9).

```python
1  def pi_decimals(dec, prom):
2      -- It does not fulfil the promise
3      ...
4  -- Main task.
5  prom = Promise()
6  pi_decimals(50, prom)
7  -- Blocks parent task.
8  get(prom)
```

Figure 2.8. Deadlock using futures (left); deadlock using promises (right).

Deadlocks in Channels

Channels are subject to deadlocks when there is a mismatch between the number of send and receive operations. Fig. 2.9 shows an (adapted) example of a deadlock that side steps the deadlock detector of the Go language [175]. In this example, all scheduling permutations reach to the same deadlock state. After spawning an initial task, the child waits to receive a value from the channel (Line 13). The parent task continues by spawning a new child task (Line 15), the child blocks on the sending operation, as there is no receiver for that channel (Line 5). The parent task places a value on the channel (Line 18), unblocking the channel on Line 13. Assume that the unblocked child continues and is blocked again when sending a value to the channel (Line 9). The parent task takes over, spawns a new task which immediately blocks (Line 21), and performs a receive operation (Line 27) that unblocks the child task on Line 9. After that, the parent task tries to retrieve a new value from the channel and blocks forever (Line 28). There are two deadlocks, one in Line 21 and one in Line 28 due to a mismatch between sends and receives.

2.3.3 Performance and Synchronisation Granularity

Introducing the right amount of synchronisation granularity is an active research area with performance implications from operating systems to databases, among other domains [22, 66, 84].
Synchronisation operations may prevent the introduction of data-races, but over-synchronisation may introduce deadlocks or performance regressions [66, 84, 129]. Performance regressions can happen due to synchronisation overhead and contention [84, 129]. For example, the use of locking synchronisation introduces new atomic operations to acquire and release locks, and lock contention may serialise (reduce) the amount of parallelism of an application [84, 129]; a coarse locking policy improves the performance in some situations [84], and compiler researchers found synchronisation heuristics to remove redundant synchronisation [24, 27, 52, 84, 226], but these heuristics use simple syntactic rules [136].

Other approaches may use software and/or hardware transactional memory for lock-based synchronisation [204, 78, 190, 41, 144, 77]. This approach executes a (lock-protected) critical section in a software or hardware atomic transaction; transactions succeed when there are no conflicts, and fail when multiple transactions conflict at runtime. Upon a conflict, one of the transactions aborts and depending on the conflict strategy, either the aborted transaction tries speculatively to commit (again) using transactional memory, or opts to acquire the lock. One of the main benefits of using software and/or hardware transactional memory is to elide locks [190, 77, 85]. These techniques may also perform runtime analysis and statistics collection to dynamically decide whether critical sections should run in a transaction or using other synchronisation means [214].

2.4 Concurrency and Synchronisation in Context

This section shows a summary of our work with respect to concurrency concepts (futures, promises, task, and channels) and concurrency problems (deadlocks and data-races), showing the reader the expected background for each paper.
Table 2.3. Necessary background to understand our work w.r.t. concurrency abstractions (futures, promises, tasks, and channels) and concurrency problems (deadlock and data-races), captured on a per-paper level.

<table>
<thead>
<tr>
<th></th>
<th>Future</th>
<th>Promise</th>
<th>Task</th>
<th>Channels</th>
<th>Deadlock</th>
<th>Data-Race</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper I</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Paper II</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Paper III</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Paper IV</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2.3 shows the necessary background to understand our work, captured on a per-paper level. But this table does not specify whether abstractions are data-race or deadlock free (e.g.,), merely that the concept is a prerequisite to understand the paper. We elaborate on this as follows:

1. Our work starts with a concurrent functional abstraction, ParT (Paper I), in a task-based language that uses control-flow futures (brief explanation in Section 1.1; detailed explanation in Section 4.4) that maintains data-race and deadlock freedom guarantees. Data-race freedom holds because there is no mutable state, and deadlock freedom holds because there are no cyclic dependencies between tasks. (These guarantees extend to Paper II and Paper III for the reasons above.)

2. The ParT abstraction relies on control-flow futures and we noticed that certain delegation patterns could not be expressed in the ParT abstraction. Based on this realisation, we introduce an existing construct named forward [58] into a concurrent future-based calculus (simpler than the ParT calculus), and show how the use of this new construct allows developers to write certain promise-based delegation patterns in a future-based language.\(^5\) The calculus is based on core fragments of the ParT calculus and thus, remains data-race and deadlock free.

3. The forward construct does not allow to differentiate between control- and data-flow futures, i.e., futures that may synchronise in all nested futures from those that abstract over the nesting, and requires manual intervention to create delegation patterns. Paper III uses a minimal concurrent calculus (in line with previous work) and shows how we can use combinators to manage control- and data-flow futures.

4. Our last paper (Paper IV) shows how we can extend the applicability of our previous work (Papers I–III) to an imperative setting, while retaining data-race freedom. We define a programming model that is similar to the core calculus from previous work, but uses channels instead of futures and a capability-based system. The programming model is data-race free, when desired, but not deadlock free.

\(^5\)As stated in Section 2.2.1, futures and promises are similar but futures enforce a single writer to the future, while promises cannot statically maintain this guarantee.
3. Object-Oriented and Functional Programming

The calculus from papers I – III use a functional language, and Paper IV uses an object-oriented language. This chapter reviews relevant features of functional and object-oriented languages. Section 3.1 summarises the difference between objects and classes, and explains why data-race freedom is difficult in a concurrent object-oriented language (Section 3.1.1). Section 3.2 introduces functional programming concepts, explains why functional programs are data-race free (Section 3.2.1), and gives a brief summary to a concurrent simply typed lambda calculus (Section 3.2.2).

3.1 Object-Oriented Programming

In object-oriented programming, objects represent abstract or concrete concepts and they interact with each other via method calls. Method calls may change the object’s internal state [12]. For the purpose of this thesis, we refer to objects as anything that contains identity, state, and methods (borrowed definition from [28]). But this definition does not state anything about how developers instantiate objects.

Language designers are responsible for the object instantiation strategy, which affects other aspects of the language, e.g., inheritance. Common instantiation strategies are class- or object-based instantiation. A class-based strategy instantiates objects from classes, while in an object-based strategy each object definition is also its object instantiation.

Fig. 3.1 shows an example of an object-based program that updates an image on the screen, using a common technique that avoids flickering of the image during a refreshing cycle, known as page flipping with double buffering [2]. Page flipping consists on simply swapping a pointer that points to video memory, to point to some other video memory address. The example defines a constant function \texttt{scene} that creates an object; this object will write in the off-screen buffer and do the page flipping of the video pointer. The object has two fields, \texttt{current} and \texttt{next}, where the former refers to the video pointer that the graphics card constantly reads, and the latter refers to an off-screen frame buffer. Upon creation of a scene (Line 14, initialisation details have been omitted), the method \texttt{draw} is called to draw a new image on the screen (Line 16). The method clears the off-screen frame buffer, draws some
image, and calls on the method swap (Lines 3 and 8), which simply swaps the video pointer current to display the content pointed by next (Lines 9 to 12).

Our work on the Dala programming model (Paper IV) uses an object-based instantiation strategy, and omits inheritance. This decision is purely pragmatic, to remove “noise” from the core calculus. Based on [131], who implemented multiple inheritance models on object-based languages, we believe that the omission of inheritance should not affect much our programming model.

3.1.1 Concurrency Perspectives In Object-Oriented Languages

Concurrent programming in object-oriented languages is hard. Most common object-oriented languages are not data-race free. Examples of these languages are Java, C#, JavaScript, and Scala among others, where either developers protect objects or accept “unpredictable and surprising behaviour” [98].

There are a number of dynamic and static approaches to forbid data-races and it is an active research area [202, 53, 56, 137, 135, 134, 196, 23, 94]. Dynamic detection tools such as Eraser [202] are based on a lockset algorithm. Roughly speaking, the lockset algorithm ensures that shared variables are always accessed by threads that hold a lock, and infers at runtime which locks should protect which variables [202]. Other approaches improve upon Eraser by mixing static and dynamic information [53, 23, 196]. To avoid the runtime overhead of dynamic detection tools and guarantee data-race freedom, large programs may only use static analysis techniques. As suggested by Rinald [197], “augmented type systems are the most promising approach for activity management programs”\(^1\), which is the approach taken by capability-based languages such as Encore [35, 48], Pony [63, 64], or Gordon’s C# extension [99] (see Section 4.5).

\(^1\)Activity management programs refer to programs that use threads to manage concurrent computations.
3.2 Functional Programming

Functional programming and their ideas have been steadily spreading and influencing many modern languages, perhaps due to the benefits of working with immutable values in concurrent and parallel settings.

Many functional programming languages are build upon a simple yet powerful functional typed language, the simply typed lambda calculus [57, 130]. Many researchers consider the simply typed lambda calculus as a bare bones programming language and there has been plenty of research into adding more advanced features to it, ranging from adding parametric polymorphism [96, 195] to encoding objects [122], among others.

When it comes to concurrency and parallelism, we highlight two key features from functional programming languages [118]: immutability and referential transparency. Immutability allows developers to work with parallel algorithms and abstractions without worrying about mutable side effects. Referential transparency refers to the ability to change variables and expressions by their values, without altering the program semantics. This has many implications, spanning from memoization of functions, where functions cache their results to application of equational reasoning [119].

```
1  typecheck env (Program cls) = Program <$> mapM (typecheck env) cls
2
3  typecheck env cdef@ClassDef{cname, fields, methods} = do
4     let env’ = addVariable env "this" (ClassType cname)
5     fields’ <- mapM (typecheck env’) fields
6     methods’ <- mapM (typecheck env’) methods
7     return cdef{fields = fields’, methods = methods’}
```

Figure 3.2. Implementation of a type checker in Haskell [47]

For example, consider the implementation of a type checker for an object-oriented language using a pure functional language (Fig. 3.2, example borrowed from [47]). Type checking classes in parallel cannot introduce data-races, even in the presence of a shared environment – the environment is immutable. In this example, the type checker receives an abstract syntax tree node, named Program, that contains a list of classes that need to be type checked. The operation mapM (Lines 1, 5 and 6) runs in a serial fashion and applies the typecheck function to each class, field, and method. Compiler writers can type check classes (and fields and methods) in parallel using parallel combinators such as parMapM, from the Par monad library [159].

In the next section, we argue about why these concepts are crucial for data-race freedom in concurrent functional languages (Section 3.2.1) and introduce the common parts of a concurrent functional core calculus used in Paper I – Paper III (Section 3.2.2).
// Function executed by child task
def job(counter):
    counter += 1
...
return counter

counter = 0
future = async(job(counter))
counter += 1
...
get(future)

Figure 3.3. Pseudo-code that spawns a task to perform a calculation, performs some work and waits for the future.

3.2.1 Concurrency Perspectives In Functional Languages

Immutability and referential transparency are key features for a data-race free concurrency model. Immutability forbids mutable side effects and referential transparency help developers apply equational reasoning [118, 119]. Thus, two expressions can run concurrently (or in parallel) without introducing data-races.

As an example, Fig. 3.3 shows a parent task that delegates work to a child task, where the end result is non-deterministic in an object-oriented language, due to data-races. (This same example has been studied in Fig. 2.6.) We show two possible concurrency schedules that produce different results on the future variable (Line 11). In both schedules, the counter has an initial value of 0, and a parent task shares the counter variable with a child task. Under one scheduling, the child task may dereference the counter and read value 0 (Line 3); next, the scheduler runs the parent task, which dereferences the counter variable and reads value 0 (Line 9). The parent task writes 1 to the counter variable (counter = 0 + 1); the scheduler context switches to the child task, which overwrites the value of the counter variable to be counter = 0 + 1. The parent task reads the value from the future, i.e., the value from the counter variable, 1. Under another scheduling the child task executes without any context switch interruption, placing the value 1 in the counter variable; when the parent task executes (Line 9), it increments the value counter, producing the value 2. The returned result in the future is the value of the counter variable, 2.

The same code is completely deterministic in a functional programming language. The main reason is that values are immutable. Hence, concurrent and parallel (pure) computations do not have any effect on each other. In the

---

2Paper I introduces an exception to referential transparency at the task level, with the introduction of a combinator called prune. This combinator allows multiple tasks to race to fulfil a future, and referential transparency is broken because given the same input, the output may be different depending on the scheduling of tasks. This is not a data-race, but a race condition. If the prune combinator is not used, our model maintains referential transparency.
example of Fig. 3.3, regardless of the scheduling, the helper task always reads the value of \( x \) as 0 (Line 3) and the parent task always reads the value of the counter variable as 0 (Line 9). The returned future result is always 1.

Next, we show the core semantics of the task-based simply typed lambda calculus from Papers \( I – III \). These papers perform minor modifications to this core calculus.

### 3.2.2 Task-based Simply Typed Lambda Calculus

In this section we give a summary of the task-based simply typed lambda calculus used in Papers \( I – III \); the main differences between the papers are at the term and expression level constructs and their properties, leaving the core task-based calculus almost unchanged.

The syntax of the task-based simply typed lambda calculus contains expressions and values (Fig. 3.4). Expressions are values \( (v) \), application \( (e \ e) \), a combinator for spawning a task \( (async \ e) \), which returns a future \( f \), the \( get \ e \) combinator to extract a value from a future, and future chaining \( (e \rightsquigarrow e) \), Section 2.2.1). Values are constant values \( (c) \), variables \( (x) \), futures \( (f) \), and abstractions \( (\lambda x.e) \).

\[
e ::= v | e \ e | async \ e | e \rightsquigarrow e | get \ e
\]

\[
v ::= c | x | f | \lambda x.e
\]

**Figure 3.4.** Task-based simply typed lambda calculus.

Configurations represent running programs. A global configuration represents the global state of the run time system, e.g., \((fut_f) (task_f e)\), and a partial configuration \( \text{config} \) shows a view of the run time; configurations are a multi-set of tasks, futures, and chained configurations. The empty configuration is \( \varepsilon \); an unfulfilled future configuration is \((fut_f)\) and a fulfilled future is \((fut_f v)\); a task configuration represents a running expression inside the task \((task_f e)\) that fulfils \( f \) when \( e \) evaluates to a value \( v \); a chain configuration \((chain_f g e)\) represents a configuration waiting on the fulfilment of future \( g \); when future \( g \) contains a value \( v \), the chain configuration may run \((e v)\) and write its resulting value in future \( f \).

\[
\text{config} ::= \varepsilon | (fut_f) | (fut_f v) | (task_f e) | (chain_f f e) | \text{config} \ \text{config}
\]

The operational semantics are based on small-step, reduction-context rules for evaluation of expressions within tasks, and non-deterministic reduction rules for evaluation across configurations. Evaluation context \( E \) contains a hole \( \bullet \) that denotes the location of the next reduction step, in the standard fashion [222].
The reduction step relation \( \text{config} \to \text{config}' \) takes a single reduction step from configuration \( \text{config} \) to \( \text{config}' \). Configurations are commutative monoids under concatenation, with \( \varepsilon \) as its unit; configurations are equivalent modulo associativity and commutativity (Fig. 3.5) and these equivalences can be applied at any time during the reduction step.

\[
\text{config} \to \text{config}' \\
\text{config} \equiv \text{config}' \to \text{config}'' \equiv \text{config}'' \\
\text{config} \equiv \varepsilon \equiv \text{config}' \equiv \text{config}''
\]

\( \text{config} \equiv \text{config}' \equiv \text{config}'' \)

\( (\text{config} \text{config}') \equiv \text{config}'' (\text{config} \text{config}') \)

Figure 3.5. Configuration equivalence rules modulo associativity and commutativity.

The types of the task-based simply typed lambda calculus are

\[ \tau ::= K \mid \tau \to \tau \mid \text{Fut} \tau \]

where \( K \) are the basic types, \( \tau \to \tau \) is abstraction, and \( \text{Fut} \tau \) represents a future type containing \( \tau \).
The typing judgement for expressions are written \( \Gamma \vdash \rho \ e : \tau \) which asserts that under the environment \( \Gamma \), expression \( e \) has type \( \tau \), and the return type of the task is \( \rho \). An environment \( \Gamma \) contains the types of free variables and futures. Below we show the expression typing rules, and emphasise on rule T-ASYNC, which spawns a computation returning a future type where the spawned task running \( e \) sets the return type of the task to be the return type of the expression.

\[
\begin{align*}
\text{\textbf{(T-CONST)}} & \quad \frac{}{\Gamma \vdash \rho \ c : \tau} \\
\text{\textbf{(T-FUT)}} & \quad \frac{f : \text{Fut} \ \tau \in \Gamma}{\Gamma \vdash \rho \ f : \text{Fut} \ \tau} \\
\text{\textbf{(T-VAR)}} & \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash \rho \ x : \tau} \\
\text{\textbf{(T-APP)}} & \quad \frac{\Gamma \vdash \rho \ e : \tau' \to \tau \quad \Gamma \vdash \rho \ e' : \tau'}{\Gamma \vdash \rho \ e \ e' : \tau} \\
\text{\textbf{(T-ABS)}} & \quad \frac{\Gamma \vdash \rho \ e : \tau' \quad \Gamma \vdash \rho \ \lambda x . e : \tau \to \tau'}{\Gamma \vdash \rho \ \lambda x . e : \tau \to \tau'} \\
\text{\textbf{(T-ASYNC)}} & \quad \frac{\Gamma \vdash \rho \ e : \tau}{\Gamma \vdash \rho \ \text{async} \ e : \text{Fut} \ \tau} \\
\text{\textbf{(T-CHAIN)}} & \quad \frac{\Gamma \vdash \rho \ e : \text{Fut} \ \tau' \quad \Gamma \vdash \rho \ e' : \tau' \to \tau'}{\Gamma \vdash \rho \ e \to e' : \text{Fut} \ \tau}
\end{align*}
\]

Well-formed configurations (Fig. 3.6), denoted \( \Gamma \vdash \text{config ok} \), express that a configuration \( \text{config} \) is well-formed under environment \( \Gamma \), that gives the types of futures. Futures are well-formed if the future exists in the environment \( \Gamma \); task and chains are well-formed if the futures are well-formed and the expressions are well-typed; a bunch of configurations are well-formed if the individual configurations are well-formed.

\[
\begin{align*}
\text{\textbf{(WF-UFUT)}} & \quad \frac{f \in \text{dom}(\Gamma)}{\Gamma \vdash (\text{fut}_f \ ) \ ok} \\
\text{\textbf{(WF-FFUT)}} & \quad \frac{f : \text{Fut} \ \tau \in \text{dom}(\Gamma)}{\Gamma \vdash (\text{fut}_f \ v) \ ok} \\
\text{\textbf{(WF-CHAIN)}} & \quad \frac{f : \text{Fut} \ \tau_2 \in \text{dom}(\Gamma) \quad g : \text{Fut} \ \tau_1 \in \text{dom}(\Gamma)}{\Gamma \vdash \tau_2 \ e : \tau_1 \to \tau_2} \\
\text{\textbf{(WF-TASK)}} & \quad \frac{f : \text{Fut} \ \tau \in \text{dom}(\Gamma)}{\Gamma \vdash (\text{task}_g \ e) \ ok} \\
\text{\textbf{(WF-CONF)}} & \quad \frac{\Gamma \vdash \text{config ok} \quad \Gamma \vdash \text{config}' \ ok}{\Gamma \vdash \text{config} \ \text{config}' \ ok}
\end{align*}
\]

\textbf{Figure 3.6.} Well-formed configurations.

This section finishes with a running example of the dynamic semantics. The example starts in a well-formed configuration that spawns a task, and blocks until the task has produced a value (Eq. (3.1)). The only possible reduction step is R-ASYNC, which spawns a new task. This creates a new future \( g \) and a new task that runs the expression contained in the body of the \text{async} expression (Eq. (3.2)). The next reduction can only be R-FULFIL since the other task,
(task_f get g), is blocked waiting on a value on future g; this reduction step (R-FULFIL) fulfils the future g with the value 42 (Eq. (3.3)). Future g is fulfilled, and the get combinator can now extract the value from the future and place it in the task (Eq. (3.4)). The final reduction, simply finishes the task and fulfils its future (Eq. (3.5)).

\[
\begin{align*}
(fut_f) (task_f get (async 42))) & \rightarrow^{R-ASYNC} \\
(fut_f) (task_f get g) (fut_g) (task_g 42) & \rightarrow^{R-FULFIL} \\
(fut_f) (task_f get g) (fut_g 42) & \rightarrow^{R-GET} \\
(fut_f) (task_f 42) (fut_g 42) & \rightarrow^{R-FULFIL} \\
(fut_f 42) (fut_g 42) & 
\end{align*}
\]
4. Related Work

This chapter overviews common actor-based concurrency models, concurrent abstractions and futures, termination strategies for speculative computations, and capability-based languages. The chapter finishes giving an overview of existing concurrent programming languages, connecting to the different concurrency models, termination strategies, and features used in capability-based languages.

Section 4.1 overviews existing task- and actor-based concurrency models; Section 4.2 shows concurrent collections and abstractions. Section 4.3 shows speculative termination strategies and adds new languages to the initial categorisation from Kolesnichenko’s et al [143]. Section 4.4 follows the history of futures and promises, how these concepts have been used indistinguishably, and presents a new categorisation of futures based on four-dimensions: implicit/explicit futures, control/data-flow synchronisation, synchronous/asynchronous futures, and its typing. Section 4.5 introduces object and reference capability languages and ideas that have seen adoption in capability-based languages. Section 4.6 summarises concurrent languages, connecting concepts from concurrent abstractions, futures, and capability-based languages whenever relevant (for the languages under study). Section 4.7 comments on how the Dala programming model extends the applicability of our previous work (Papers I–III) to an imperative setting, using as example the ParT abstraction in an object-oriented setting.

4.1 Actor-Based Concurrency Models

This section covers related work in the area of actor-based systems. Concretely, the actor model, active objects, communicating event loops, and concurrent object groups.

Actors [114, 104, 8] and active objects [224] are means to concurrency and parallelism. In both systems, an actor / active object represents an entity with its own thread-of-control and a message queue. Actors and active objects can send asynchronous messages, change their behaviour upon processing of messages, and spawn new actors [74]. (For the purposes of this thesis, we will refer to active objects as actors, except in a context where the difference matters.)

Active object languages distinguish between active objects and passive objects, and both of them are first-class citizens. Active objects have their own
Table 4.1. *Comparison of concurrency programming models.*

<table>
<thead>
<tr>
<th></th>
<th>Actor</th>
<th>Active Object</th>
<th>Event Loop</th>
<th>COG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>Actors</td>
<td>Active Object</td>
<td>Vats</td>
<td>COG</td>
</tr>
<tr>
<td>First-class</td>
<td>Actors</td>
<td>Active &amp; Passive Objects</td>
<td>Objects</td>
<td>Objects</td>
</tr>
<tr>
<td>Other objects</td>
<td>–</td>
<td>Passive Objects</td>
<td>Objects</td>
<td>Objects</td>
</tr>
<tr>
<td>Async. Calls</td>
<td>Actors</td>
<td>Active Objects</td>
<td>Objects</td>
<td>Object</td>
</tr>
<tr>
<td>Message Queue</td>
<td>Actor</td>
<td>Active Objects</td>
<td>Vats</td>
<td>Method calls</td>
</tr>
<tr>
<td>Communication</td>
<td>Message</td>
<td>Method calls</td>
<td>Method calls</td>
<td>Method calls</td>
</tr>
</tbody>
</table>

Thread-of-control; passive objects are common objects, *i.e.*, objects without a thread-of-control such as a Python or Java object. Communication with active objects involves asynchronous method calls, while passive objects can only receive synchronous method calls. Without any concurrency control, passive objects are subject to data-races when shared among active objects.

Communicating event loop languages such as E [165] or AmbientTalk [71] use “actor containers” (known as *vats*) to achieve concurrency. An event loop is conceptually an actor with its own thread-of-control and message queue; the event loop actor continuously picks up messages and executes them. Messages contain the target of the method call, arguments passed, and the method to execute. Objects are owned by a single event loop actor, and multiple event loops run concurrently; there is no concept of first-class event loop, *c.f.*, active object model [74]. Objects accept synchronous and asynchronous method calls. Synchronous method calls execute directly, while asynchronous method calls place messages in the message queue of the object’s event loop owner.

Languages based on concurrent object groups [127] (*cogs* for short) present many similarities to the event loop concurrency model. A cog represents a thread-of-control, and owns a set of objects; multiple cogs may run concurrently, or in parallel. Each object has a message queue and can perform synchronous or asynchronous method calls to other objects, where an asynchronous method call places a message in the target object. In this concurrency model, a cog picks up an object, dequeues a message from its message queue, and executes the computation (of the message) to completion.

Table 4.1 summarises the main differences between actors, active objects, communicating event loops, and concurrent object groups. One of the main differences between the actor and the event loop model is that in an actor-based system, actors are first class citizens while vats and cogs (in the event loop and concurrent object group models, respectively) are not; in the event loop and cog model all objects can be called synchronous and asynchronously, while in the active object model only active objects are the target of asynchronous calls. Actors, active objects, and the event loop concurrency model place their message queues in their concurrent abstraction, *i.e.*, actor, active object and event loop (vat), respectively. Concurrent object groups differ from these models in...
the placement of the message queue, and attaches a message queue to each single object, instead of on its cog.

4.2 Concurrent Asynchronous Abstractions

This thesis uses high-level abstractions to create and coordinate complex asynchronous and concurrent computations, such as parallel pipelines of speculative computations; we overview them in this section.

**Concurrent Abstractions.**

Programming languages offer high-level abstractions to facilitate writing concurrent applications. One simple approach towards this goal is to offer standard collection implementations that may execute concurrently (or in parallel). For example, the languages Scala and C# added parallel methods to their collections [188, 164], and Clojure and Haskell added parallel functions to operate on their collections [87, 128, 158]. As an example, `val list = (1 to 10000).toList` creates a list in Scala; developers can manipulate the list concurrently (and in parallel) when invoking the `par` attribute: `list.par.map(_ + 42).

**Distributed Abstractions.**

Distributed programming models in the vein of Map-Reduce [75, 51, 220, 125, 11, 225] are suitable in the context of concurrent abstractions. These models build execution plans with three simple combinators: map, shuffle, and reduce, which may operate in parallel. The model works as follows (omitting implementation details): values from a collection (or other medium) are passed to a mapping operation. The mapping operation outputs key-value pairs based on a user-defined function; the shuffling operation takes the output of the mapping operation and groups key-value pairs that have the same key, passing this data to the reduce operation; the reduce operation receives a group of key-values and executes user-defined computations on them. The execution plans can be deferred and optimised until the values are needed [51, 11].

**Functional Approaches.**

A common approach in functional languages is to use monads and concurrent combinators to control them [159, 210, 39, 147]. In Haskell, the `Par` monad uses the `fork` combinator to spawn a concurrent (parallel) computation. The forked computation can share values with other concurrent computations when it captures an I-structure [178]. An I-structure is a write-once structure with semantics analogous to promises (Section 2.2.1), with combinators `new`, `get`, and `put` to create, read, and write an I-structure, respectively.

Other languages such as F# offer computation expressions [210, 185] to write non-standard semantics using common syntax to developers, *e.g.*,
let getLength url = async {
    let! html = fetchAsync url
    return html.Length
}

getLength url = do
    fut ← new
    fork $ do
        htmlIVar ← fetchAsync url
        html ← get htmlIVar
        put fut (length html)
    get fut

Figure 4.1. Asynchronous monadic computation using F# computation expressions (left) and Haskell’s Par monad (right). The types of the functions are fetchAsync :: Async t, fork :: Par () → Par (), get :: IVar t → Par t, new :: Par (IVar t), put :: NFData a ⇒ IVar a → a → Par ()

monadic computations. Developers can assign different “interpretations” to the syntax in a computation expression, by implementing an interface. Fig. 4.1 shows an example (borrowed from [185]) that retrieves the HTML of a website and returns its length, written in F# using a computation expression, and in Haskell using the Par monad. The computation expression starts an async block (Fig. 4.1, left), uses the function fetchAsync :: String → Async<’T> where the let! binding is reinterpreted as a monadic bind, i.e., let! :: Async<’T> → (’T → Async<’U>) → Async<’U>. The return expression simply lifts the value to the expected monad, i.e., return :: ’T → Async<’T>. In Haskell (Fig. 4.1, right), we observe how the fork captures the future (I-structure) fut; the put combinator fulfils the future, and the get operation blocks until the future is fulfilled.

Orchestration Languages

Orchestration languages [169, 189, 170] coordinate (external) concurrent and distributed computations towards certain goal. In the Orc language [139], clients send and receive data from sites. A site represents (possibly external) computations or services, e.g., matrix multiplication or communication with a web server, and a site may return at most one value, but they could also not respond. Expressions are sites and parallel combinators. Whereas a site may return at most one value, an expression may return multiple values. We highlight four combinators from the Orc language [140]:

1. F | G represents parallel execution of expressions F and G, and publishes values from expressions F and G.
2. F >>x> G the sequence combinator executes F; each published value from F gets bound to x and starts execution of G. When F does not publish a value, G cannot execute.
3. G <x< F the prune combinator executes F and G in parallel and G executes until it encounters the variable x. Upon finding x, G stops execution until F publishes a value. When F publishes a value, the value is bound to x, the expression F terminates, and the expression G continues its execution.

45
4. F; G the otherwise combinators executes F and, if F does not publish any result and halts, then G starts execution.

Other combinators are signal and stop, that publish a value and stop execution (without publishing a value), respectively.

Using these combinators one can easily orchestrate concurrent (speculative) computations, such as parallel splits (running multiple computations in parallel, e.g., F \( \| \) G) or multi-merges (each published result from an expression executes a function, e.g., \((F \| G) > \times > H\)), among others [68].

The Orc language has a “monadic feeling” (suggested in [139]) where one could consider the signal combinator as the monadic unit combinator, that lifts a value to an Orc monad, and \( > \times > \) as its bind combinator. The parallel composition combinator \( \| \) makes the Orc monad a monoid [147, 39] (a Monad Plus in Haskell terms [215]).

The ideas of Orc have been ported to an object-oriented language, OrcO [184]. OrcO inherits Orc’s core ideas, and adds an object-oriented programming style with notions of objects, classes, inheritance, and mixins. Objects are immutable records and fields are (ongoing) Orc expressions, bound on the first published value. The creation of an object returns immediately and the field computations can be seen as transparent futures (Section 4.4), that block upon accessing a field that is not bound to a (published) value; concurrency stems from the Orc combinators and OrcO does not guarantee data-race freedom.

4.3 Speculative Computations

Many concurrent and asynchronous abstractions spawn speculative tasks to solve a problem. These computations are speculative in the sense that the user may or may not be interested in all the results [123], e.g., constraint solving or search strategies. Kolesnichenko’s et al classified termination strategies of speculative tasks as client-, supplier-, and client-/supplier-based cancellation strategies [143]. The name of the categorisation highlights who is in control of the cancellation process: in client-based cancellations there is a client task requesting cancellation to supplier tasks, in supplier-based cancellation a supplier task informs its client that the supplier will cancel its execution, and in client-/supplier-based cancellation the client and supplier tasks cooperate during the cancellation process.

Client-Based Cancellation.

A simple client-based cancellation stops the remaining speculative tasks upon finding a result, preemptively. This solution may leave objects in inconsistent state or create deadlocks [98]. Other client-based cancellation strategies use constructs such as abort. In Cilk-5, the abort statement marks all child speculative computations as non-runnable, so that the runtime ignores them when
they are scheduled [95]. OpenMP uses implicit checks within the executing program to stop speculative computations, and these implicit checks happen more often than just before starting a new task [6]. Java and Scala follow a similar approach to OpenMP, but they use interrupts. An interrupt requests a task to stop its execution [98], but the execution may not stop immediately, but on safe points. Interrupts may become client-/supplier-based, more on this in client-/supplier-based categorisation.1

Kumar’s work [146] on termination of speculative computations shows how modern managed runtimes can leverage existing runtime checks to implicitly stop speculative computations at well-defined points. The authors introduce the Featherlight programming model which adds two new constructs to the async-finish model: finish_abort and abort. The finish_abort construct allows cancellation of speculative tasks created inside of them, via abort, without the introduction of manual checks. Upon execution of an abort operation, all (workers) tasks will stop on yield points. (Yield points are locations in the program where is safe to run the garbage collector, e.g., method prologue [146], compiler-dependent.) The task that requested the abort operation walks the workers’ stacks and identifies workers that execute tasks that belong to the same finish_abort scope. These workers are marked to throw a special exception instead of continue with their work, and the exception makes them jump to a safe point where they can continue execution of other work. The task that requested the abort operation continues right after the finish_abort, as expected.

Supplier-Based Cancellation.
Supplier-based cancellation strategies make supplier threads to throw exceptions and indicate client threads that the request was not handled as expected [143, 172]. For example, Erlang processes use exit signals to indicate to other linked processes the reason of their termination [14, 172].

Client-/Supplier-Based Cancellation.
In a client-/supplier-based cancellation, client and supplier cooperate during the task cancellation process. Interrupts (mentioned earlier) become client-/supplier-based when the API exposes methods to interrupt and test for interruption [103], which has as end goal to increase the responsiveness of the cancellation strategy.

C#’s Task Parallel Library [151] uses cancellation tokens to cooperatively stop speculative computations [1]. These cancellation tokens are passed between tasks and developers either poll the token, to test its state and manually terminate the task, or register callbacks to exit gracefully.

1 Since Orc runs on the JVM, we believe that Orc’s prune combinator may use also a client- or client-/supplier-based cancellation strategy, but we could not find these implementation details in the literature.
Imam’s et al work [123] on the Eureka model combines explicit and implicit cancellation points in an async-finish computation. In this model, the finish construct can register a “token” (eureka) that represents a placeholder for the value being computed; speculative tasks (and their children) belong to their immediate enclosing finish and can resolve its “token”. Developers can create new “tokens” (within an existing async-finish) and link them to a new finish construct to further control the scope and synchronisation of these child tasks. Implicit cancellation points check whether tasks should stop before they start execution; explicit cancellation points are entered by developers to manually check whether the “token” has been fulfilled and the speculative task should be stopped. The thread executing a finish construct blocks until all speculative (and transitively spawned) tasks stop.

4.4 Futures & Promises

The original definition of futures and promises has changed over time, i.e., researchers and practitioners have used them interchangeably. In practice, futures are considered placeholders for values that are computed asynchronously, where the fulfilment of the future happens implicitly [90, 109, 35, 127, 43]; a promise is a pair: the first item of the pair can only be read (as a future), while the second item of the pair grants its owner write access to the promise, but a promise can only be fulfilled once. Fulfilling a promise multiple times throws an error.

This section continues with a brief overview on the history of futures and promises (Section 4.4.1), and finishes with a new categorisation of futures (and promises) that takes into account four dimensions when categorising futures (Section 4.4.2). Our hope is that this categorisation succinctly captures the semantics of the various forms of futures.

4.4.1 History

Originally, futures were introduced by Baker et al [17] as a reduction strategy (call-by-future) that evaluates arguments of function calls in parallel, returning a future. A future was therefore a placeholder for an asynchronous computation. Operations that need the result of the future to continue execution block until the future has a value.

MultiLisp gave control to developers to perform call-by-future explicitly [133], using the construct Future \{ e \} (async in some languages) which spawns an asynchronous computation to execute the expression e. These futures are first-class citizens and block implicitly when their values are needed.

The actor paradigm quickly adopted futures [224], with a first twist. ABCL/1 featured asynchronous message passing placing the eventual result in
a future object, which was explicitly named. A future object had queue-like behaviour and multiple asynchronous calls could place their results under the same future. Synchronisation was blocking. An interesting feature was that asynchronous calls could get the handle to fulfil the future, pass it around or delegate the fulfilment of the future to another future computation.

Argus’ promises from Liskov et al [156] were heavily influenced by MultiLisp’s futures and (probably) the first work to typed them. Asynchronous message sends returned promises, and these were second-class citizens, i.e., promises could neither be passed as arguments nor returned. Promises introduced explicit blocking synchronisation constructs and error propagation, and were implicitly fulfilled by the returned argument.

Caromel created an active object language with notions of implicit (transparent) futures [42], named await objects. Futures were present at runtime but the typing did not reflect this fact. When the value of the runtime future was needed, the executing thread synchronised on the future, blocking until the future was fulfilled. This synchronisation is known as wait-by-necessity [42]. These futures were similar to MultiLisp’s, but in a typed language; more recent work on wait-by-necessity and active object languages culminated in the ASP calculus [44] and its implementation, named ProActive [43].

ABCL/f was (probably) the first active object language to use explicit typed futures [211]. Instead of using futures in a queue-like fashion as ABCL/1, ABCL/f had first-class futures and allowed developers to create empty futures that could be fulfilled by anyone.

The language E [165] is untyped and uses promises in a distributed event loop language. Promises are placeholders for eventual values, and promises have asynchronous synchronisation methods. Asynchronous message sends on unfulfilled promises accumulate the messages, which are sent to their recipients when the promises are fulfilled.

Industrial programming languages such as Java or Scala adopted (typed) futures with synchronous and asynchronous control-flow synchronisation constructs, where futures are first class values [187, 3]. Promises in Scala are similar to futures (CompletableFutures in Java), but they must be explicitly fulfilled [187].

When we talk about asynchronous synchronisation constructs we generally think of the future chaining operation, i.e., $\text{fut} \rightarrow (\forall x \rightarrow \ldots)$. Recent work identified the future chaining operation to have either attached or detached closures [46]. An attached closure is executed in the current actor, while a detached closure can be executed by any available actor. In the Encore programming language, closures are attached when they capture internal state from an actor, and detached otherwise.

Henrio noticed that futures can be categorised as implicit or explicit. An implicit future is transparent to the developer and only visible to the runtime; an explicit future is typed and visible to the developer [112]. In most implementations, implicit futures have data-flow synchronisation i.e., transparent
futures (may) wait for the completion of multiple nested futures to return a value, and values flow from one future to the next in a transparent way. Unlike implicit (data-flow) futures, explicit futures have control over the synchronisation steps of nested futures, for which they are said to have control-flow synchronisation [112]. Based on these findings, Henrio proposed DeF (Data-flow Explicit Future). DeF is based on a typed active object language where futures are explicit and have data-flow synchronisation. Nested futures are squashed by the type system as singleton futures and the future synchronisation operations return the inner-most value of the future.

Our work on Paper III proposes a way to incorporate explicit control- and data-flow futures under the same programming language, which is currently not supported in DeF [112].

4.4.2 A Future Categorisation

After this quick history on futures, we observe that futures are typed or untyped, implicit or explicit, with control- or data-flow synchronisation, and with synchronous and asynchronous synchronisation (e.g., contrast the get operation with future chaining). For this reason, we introduce a new future categorisation based on these observations and diverge from the notion of implicit and explicit futures introduced by Henrio [112]. Under this new categorisation, we define implicit futures as those transparent to the developer, i.e., futures where there is no explicit (and visible to the developer) future synchronisation, such as futures from the ASP calculus or the ProActive language; we define explicit futures as those futures with visible synchronisation constructs, e.g., JavaScript promises or Java futures.

Table 4.2 classifies futures in various languages according to the explicit/implicit synchronisation “visibility”, their types, their control-/data-flow synchronisation operations, and on whether these synchronisation operations are synchronous (e.g., get operation) or asynchronous (e.g., future chaining).

Example 1. Languages in the implicit and typed categories have futures that are not visible to the developer (futures exists but not at the surface level), and the fact that they are typed implies that the type is some \( \tau \neq \text{Fut} \, \tau' \). The Panini language [16] uses implicit control-flow futures with wait-by-necessity semantics [42].

---

2 DeF can introduce data-flow and control-future futures, but we foresee a substantial amount of work to integrate them.
3 Our categorisation does not make any distinction between the fulfilment of futures and promises; for the categorisation we will refer to futures as futures and promises indistinguishably, except in a context where the difference matters.
4 The formal semantics seem to allow returning nested futures; the dereferencing operation works on locations and implicit futures, but this operation does not traverse nested futures.
Table 4.2. Categorisation of futures based on implicit / explicit dichotomy, synchronous/asynchronous synchronisation, and control-/data-flow synchronisation

<table>
<thead>
<tr>
<th></th>
<th>Control-Flow</th>
<th>Data-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
<tr>
<td>Implicit Typed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Panini</td>
<td>ABS</td>
</tr>
<tr>
<td>Implicit Untyped</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Future (Baker)</td>
<td>MultiLisp</td>
</tr>
<tr>
<td>Explicit Typed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Java</td>
<td>Scala</td>
</tr>
<tr>
<td></td>
<td>Encore</td>
<td>ABS</td>
</tr>
<tr>
<td></td>
<td>Paper I – III</td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>Java</td>
<td>Scala</td>
</tr>
<tr>
<td></td>
<td>Encore</td>
<td>ABS</td>
</tr>
<tr>
<td></td>
<td>ABCL/1</td>
<td>ABCL/f</td>
</tr>
<tr>
<td></td>
<td>Argus</td>
<td>Paper I – III</td>
</tr>
<tr>
<td>Explicit Untyped</td>
<td>Asynchronous</td>
<td>Python</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ruby</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Python</td>
<td>Ruby</td>
</tr>
</tbody>
</table>

Example 2. Some languages may belong to more than one category. It is common under the explicit and typed category to have synchronous and asynchronous future synchronisation operations, e.g., Java, Encore, ABS, and Scala. For example, the get operation on a future in the Encore language is explicit because the combinator exists at the surface level, is typed because it happens in a typed language, and has synchronous semantics.

Example 3. E, AmbientTalk, and JavaScript are untyped and use explicit futures (promises). E and AmbientTalk do not have blocking synchronisation operations — they are deadlock free. Their promises are overloaded as far references, and have data-flow synchronisation. JavaScript's promises have synchronous and asynchronous synchronisation with data-flow semantics.

Surprisingly, ABS appears in the categories explicit and typed, and implicit and typed. ABS is an actor-based object-oriented language with a concurrent object groups (concurrency) model and has explicit and typed futures at the surface level, i.e., the future type and their synchronisation constructs are accessible to developers. The implicit and typed futures appear as a side effect of its data-race freedom guarantees, i.e., synchronous method calls on objects...
that belong to a different concurrent object group are interpreted at runtime as asynchronous method calls that return futures not visible to the developer – implicit futures – followed by a blocking synchronisation operation.

Regarding this thesis work, Paper I – Paper III use explicit and typed futures with control-flow synchronisation and synchronous and asynchronous synchronisation operations, similar to Java or Scala. Paper II adds a new construct, forward, which allows data-flow synchronisation (to some extent) by delegating the fulfilment of the future to other asynchronous task. Paper III adds explicit and typed futures with control- and data-flow synchronisation, where the types reflect this fact.

4.5 Capability-Based Languages

This section introduces the notion of object and reference capabilities (Section 4.5.1), and shows an overview of ideas that have been adopted in reference capability languages (Section 4.5.2).

4.5.1 Introduction To Capability-Based Languages

A capability is a token that grants permissions to its owner to access an object. First capability designs consider the capability an abstraction on its own, where the initial bits of the capability abstraction encoded the granted permissions on the object, and the remaining bits were the pointer to the object. To access an object, the capability must grant the appropriate permissions; thus, “capabilities are the basis for object protection” [153].

Object-capabilities need not be separate tokens of their own. In languages that adopt the object-capability model, e.g., E or Pony [165, 63, 4], objects can only interact via message sends with other objects in their reference graph (which becomes their access graph). Object $o_1$ can exchange information with object $o_2$ only if $o_1$ has a reference to $o_2$, i.e., there are no global variables nor static objects, and fields are private and can only be accessed via message sends. The main benefit of object-capability languages is that they satisfy the principle of least authority [165]. Objects can only access objects sent to them, i.e., objects can only perform operations on the objects that they have been explicitly granted permission to.

Reference capability languages express what (owner and other) objects are allowed to do. These systems use extra annotations (qualifiers) to restrict / control objects and their usage. For example, the Encore language qualifies classes and traits with reference capabilities for concurrency control [45]. Fig. 4.2 shows an example written in Encore, where the declaration of a Tuple class is qualified with the $\text{read}$ capability; the type checker enforces that $\text{read}$ qualified classes must be immutable.
read class Tuple[t, u]  
  val first: t  
  val second: u  
  def fst() : t => this.first  
  def snd() : u => this.second  
end

Figure 4.2. Immutable tuple.

This section introduced object-capabilities and reference capabilities. The next section (Section 4.5.2) shows ideas that have been adopted in reference capability languages, and Section 4.6 overviews concurrent languages and shows how these ideas have been integrated in capability languages.

4.5.2 Ideas Adopted In Capability-Based Languages

Linear logic [97] has seen traction in the programming language community. In linear logic, propositions cannot be used multiple times, i.e., propositions are consumed. This idea is interesting to programming language designers as a way to (e.g.,) restrict aliasing [48, 47], track object state [9], and safely reclaim memory [217]. But the application of linear logic in programming languages may impose too strong semantics. For this reason, researchers have found ways to relax these semantics, and they introduced concepts like borrowing. Borrowing allows variables to relax aliasing restrictions for some delimited scope [30, 108, 35, 141].

Ownership types [62] enforce object encapsulation. In ownership types, the top object is usually known as world, and objects build nested ownership layers with the objects they own. The type system guarantees that two objects identified with different owners cannot be aliases [62, 45]. There are many variations of ownership types [59] and we highlight two well-known invariants that affect the design of ownership systems: owners-as-dominators and owners-as-modifiers. Before we summarise these two invariants, we define an external object $o_1$ (w.r.t. to another object $o_2$) as an object $o_1$ whose reachable object graph does not contain $o_2$. In systems with owners-as-dominators an external object $o_1$ can only communicate with object $o_2$ via $o_2$’s owner – objects with the same owner are not subject to this restriction. The owners-as-modifiers invariant relaxes the communication invariant and allows an external object $o_1$ to read an unowned object $o_2$, but modifications can only happen via the object’s $o_2$ owner [59]. Applications of ownership types range from safe memory allocation and immutability enforcement to deadlock and data-race freedom [141, 182, 29], to name a few; there is also vast research on the verification of the invariants mentioned above [173, 59, 60, 152, 80].

Many object-oriented languages allow developers to statically forbid object mutation. We distinguish between object immutability and reference im-
### Table 4.3. Summary of features of capability-based static languages. (A = Actor, Th = Thread, Tsk = Task, AO = Active Objects, FJ = Fork-Join.)

<table>
<thead>
<tr>
<th>Complexity \ Lang.</th>
<th>Complexity</th>
<th>Encore</th>
<th>HJp</th>
<th>Pony</th>
<th>RefImm</th>
<th>Rust</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>AO</td>
<td>Tsk</td>
<td>A</td>
<td>FJ</td>
<td>Th</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Capabilities</td>
<td>7+</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Capability Subtyping</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Promotion\Recovery\Borrowing</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Compositional Capabilities</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Deep copying</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

mutability (commonly referred as readonly references) [228, 213, 186]. Object immutability forbids mutation of an object, and (commonly) of its reachable object graph; reference immutability forbids mutation of an object via its referent, but other (mutable) aliases may mutate the object. One of the main implications of the various forms of immutability in a concurrent language is w.r.t. data-race freedom. Immutable objects are always data-race free because no mutation is ever allowed; reference immutability cannot immediately guarantee data-race freedom on its own as there may be aliases that may cause mutation.

Fractional permissions [31, 32] act as a bridge between mutable and readonly references. A “full” fraction allows mutation on the object; developers use combinators to get partial fractions, and partial fractions only allow readonly operations. Carefully crafted type systems may (implicitly or explicitly) use fractional permissions to maintain data-race freedom in the presence of reference immutability (e.g., [99, 160]).

This section identified features such as linear variables, ownership types, readonly references, immutable objects, and fractional permissions. In the next section we overview concurrent programming languages and link capabilities to the ideas that we identified here.

### 4.6 Concurrent Programming Languages Summary

In this section we give an overview of concurrent programming languages and refer to the concepts introduced in previous sections of the chapter, namely the concurrent model, future “style”, concurrent abstractions, and capabilities used. First we overview reference capability languages (summary in Table 4.3), then we continue with concurrent languages (summary in Table 4.4).

#### Encore

Encore is an active object language with a reference capability-based type system that guarantees data-race freedom [48, 35]. The Encore language
Table 4.4. Summary of features of concurrent programming languages. (AT = AmbientTalk, Erl = Erlang, CapSh = Capabilities for Sharing, CnsJava = ConstraintJava, DrSES = Distributed Electronic Rights For ECMAScript, AO = Active Objects, COG = Concurrent Object Group, Obj = Object-Capability, Own = Ownership Types, WW = Web Workers.)

<table>
<thead>
<tr>
<th>Complexity \ Lang.</th>
<th>ABS/ASP</th>
<th>AT</th>
<th>CapSh</th>
<th>CnsJava</th>
<th>DrSES</th>
<th>E</th>
<th>Erl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>COG/AO</td>
<td>Vat</td>
<td>✗</td>
<td>✗</td>
<td>WW</td>
<td>Vat</td>
<td>Actor</td>
</tr>
<tr>
<td>Capabilities</td>
<td>✗</td>
<td>✗</td>
<td>7</td>
<td>Own</td>
<td>Obj</td>
<td>Obj</td>
<td>✗</td>
</tr>
<tr>
<td>Compositional Cap.</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deep copying</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Far References</td>
<td>✗</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Futures\Promises</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

has active objects, passive objects, tasks, and asynchronous messages return control-flow futures. Futures are explicit, typed, and with synchronous and asynchronous constructs. Asynchronous operations, e.g., future chaining, use attached and detached semantics (Section 4.4), and this is transparently controlled by the runtime, to forbid data-races.

Futures can be lifted to a concurrent collection, ParT, which allows developers to coordinate asynchronous workflows, e.g., asynchronous pipelines of speculative computations. The ParT abstraction is data-race free in a concurrent and pure functional language. Under Encore’s type system, the ParT abstraction maintains data-race freedom when dealing with immutable data and active objects. Encore also incorporates a delegation construct, forward, that allows an actor to delegate the implicit fulfilment of its future, to another actor [89]. (The ParT abstraction and forward construct are further explain in their respective papers, Paper I and Paper II.)

Encore’s type system has 7 (seven) capabilities. Capabilities are manifest at class or trait declaration, and classes are made of trait compositions. The reference capabilities of Encore are: linear (linear), locked (object implicitly protected by a lock), read (immutable object), subord (owners-as-dominators), active (active object), thread (actor-local), and unsafe (objects must explicitly be protected by locks).

For example, we can describe a Pair class as follows (borrowed from [48]):

```plaintext
class Pair = (linear Fst ⊗ linear Snd) ⊕ linear Swap {...}
```

Pair is packed as the conjunction of two traits (Fst and Snd) and the disjunction of the Swap trait. This means that objects of the Pair class can operate concurrently on the fields of the Fst and Snd traits, and contains mutable state in fields of the Swap trait. If the pair is unpacked to perform concurrent operations, then the Swap trait (disjunction trait) is lost forever; details for recovery mechanisms are in [48].

5Details regarding trait composition can be found in [48].
Encore offers capability polymorphism for code reusability [35, 48]. The polymorphic capabilities are categorised as \textit{exclusive}, \textit{safe}, \textit{optimistic} and \textit{pessimistic} (more abstract capabilities and its details in [35]). For example, an exclusive capability denotes resources available to a single thread and accept linear (linear) and thread-local capabilities (thread).

\section*{Habanero Java With Permissions}

Habanero Java with Permissions (HJp) [219] is an extension to Habanero Java (HJ), an object-oriented language with task-based concurrency [49]. The extension adds permissions to HJ and guarantees data-race freedom. HJp has subtyping via inheritance and interfaces, and parametric and F-bounded polymorphism [40], all inherited from Java [103].

HJp uses fractional permissions [32] to maintain data-race freedom. The permissions are: \textbf{read} (readonly), \textbf{write} (mutable reference), \textbf{shared read} (readonly and sharable), and \textbf{exclusive} (either read write or shared read). Certain operations split permission fractions, and this is key to sharing data while maintaining data-race freedom. For example, regardless of whether there are permission fractions, an object can have permissions \textbf{read} and \textbf{write} to allow thread-local reads and writes, and disallow sharing, or have \textbf{shared} and \textbf{read} permissions to allow sharing and reading, and disallow modification. Exclusive permissions are permissions that retain all the fractions; exclusive permissions can be \textbf{read} and \textbf{write}, or \textbf{read} and \textbf{shared} permissions, but must have all fractions. There are promotion rules to make the permission system flexible and allow turning an exclusive \textbf{write} permission to an exclusive \textbf{shared read}, and \textit{vice versa}.

HJp has a gradual permission system [206, 207], accepting partial permission annotations. The compiler will insert dynamic checks to maintain data-race freedom. Thus, all programs are data-race free modulo errors produced from dynamic checks.

\section*{Pony}

Pony is an object-oriented actor-based language that uses the object-capability model and a reference capability-based type system that guarantees data-race freedom [64, 63, 65].

Actors communicate via asynchronous message passing; synchronisation operations are asynchronous, which makes the language deadlock free by design. Actors may share linear references, immutable objects, and other actors, which makes the language data-race free.

Pony’s type system has 6 (six) reference capabilities: iso (linear), val (immutable object), ref (thread-local), box (thread-local and readonly), trn (readonly and single owner has write privilege), and tag (identity comparison
and message sends). The capabilities put restrictions on usage and the type system forbids violation of such restrictions.

Pony has three kinds of capability subtyping: plain subtyping, aliased subtyping, and ephemeral subtyping. To maximise reusability, Pony incorporates parametric capabilities and uses viewpoint adaptation to ensure that the parametric capabilities do not break the data-race freedom guarantees [154, 79]. (We refer the reader to [64, 154] for further details.)

The Pony runtime was optimised taking into consideration the type system’s static guarantees. One of such optimisations is the ability to do asynchronous message sends without deep copying of its actual values.

C# Reference Immutability

Gordon et al extended C# with permissions to guarantee data-race freedom [99]. Their concurrency model uses a fork-join model (see Section 2.1) and assume a non-deterministic scheduler.

The language include 4 (four) reference permissions: isolated (linear), readable (readonly), writable (mutable object), and immutable (immutable object). The permissions apply transitively to its objects. That is, a field assignment of a readable object returns always a readable reference, even when the referenced field has writable permissions. This guarantees that new references do not violate the intended semantics of a readable owner, and are enforced through viewpoint adaption [79, 99].

The type system supports permission subtyping where isolated is the bottom permission and readable is the top permission, and the subtyping relation is reflexive and transitive. Recovery plays a central part in the language flexibility, allowing to recover an isolated reference from a writable reference and an immutable reference from a readable reference (under some circumstances [99]).

The type system guarantees data-race freedom, even in the presence of writable references. This is because the type system can promote writable references to readable ones when aliases are shared in a fully strict fork-join style [26], and the type system recovers the writable reference when the threads join. Sharing isolated references is also safe because each thread works on disjoint parts of a reachable object graph.

Rust

Rust is an imperative (object-oriented) systems programming language that uses ownership types [62] to guarantee data-race freedom and memory safety [161, 141]. Functional programming concepts influenced Rust, such as type classes [218] (named traits in Rust), pattern matching, immutable values, algebraic data types [38] (named Enums), and higher-order functions [57]
among other features [141]. Rust offers a thread-based concurrency model with channel-based synchronisation [117] but other concurrency patterns are also allowed [141].

Explicit annotations control Rust’s objects lifetimes, its mutability, and its uniqueness; these annotations are (to some degree) compositional. The annotations in Rust can be seen as reference capabilities, and they have 5 (five) main annotations: no annotation (readonly and linear), \texttt{const} (immutable object), \texttt{mut} (mutable object), \& (borrow reference), ‘a (named lifetime to guide borrow checker), and \texttt{static} (singleton object, \textit{i.e.}, alive for the duration of the program).\footnote{We do not show special cases here, for further information we refer to [141].}

As an example of how the type system uses these different ideas, we show an example. The following assignment, \texttt{let x = Box::new(5)}, represents a readonly linear variable. Assignment implicitly consumes the variable and the Rust borrow checker forbids usage of implicitly consumed variables, \textit{e.g.}, \texttt{let y = x; println!(x)} throws an error because the variable \texttt{x} was consumed. The \& annotation allows borrowing a variable for some duration of time, \textit{e.g.}, \texttt{let y = \&x; println!(x)} allows \texttt{y} to access the content of \texttt{x} in a delimited scoped, but \texttt{x} is the owner of the content (further details in [141]).

The type system prevents data-races and forbids sharing mutable state that could lead to data-races. For example, a closure that captures state is safe to share as long as sharing the closure removes ownership from the current thread.

Scala

Scala is a multi-paradigm programming language that mixes the object-oriented and functional paradigm. Scala has support for concurrent and parallel programming via the actor standard library and Akka [110, 223]. In terms of object-oriented features, Scala offers support for traits [86], mixins [171, 34, 155], inheritance, F-bounded polymorphism [40], and a powerful type system that includes type level programming and path-dependent types [194].

This thesis reviews two implementations of capability-based systems on top of Scala [108, 107]. Both of them try to achieve the same goal: use of linear references to achieve data-race freedom, but one uses Scala annotations [180] and the other uses Scala implicits [72, 73].

Haller’s \textit{et al} work on \textit{Capabilities for Uniqueness and Borrowing} [108] uses Scala annotations to enforce unique (linear) references and data-race freedom of the Scala actor model. Unique variables are guarded by a capability, and the capability serves two purposes: identification of (disjoint) heap regions and access permission. Unique variables are ownership closed (\textit{c.f.}, [60]), \textit{i.e.}, forbids internal objects from referring to external objects. Thus,
the capability of a unique variable guards access to its object aggregate. The creation of a unique variable uses the @unique annotation on object instantiation, and stack-bound local aliases are allowed; consumption of capabilities happen implicitly on method calls when the formal arguments are annotated as @unique, which invalidates the use of aliases and variables that were guarded by the consumed capability. Other annotations exist for borrowing and merging permissions, i.e., specifying that two unique objects belong to the same (logical) region of the heap. The type system rejects references to implicitly consumed variables (similar to Pony, Section 4.5.1). The motivation of this work was to forbid data-races in the implementation of the Scala actor library. The formalism does not include any concurrency construct but the implementation uses actors to show that the type system rejects programs that may cause data-races.

LaCasa [107] requires actors to communicate using a special Box object. Objects within a box can only be updated via its constructors or via method calls, and cannot acquire global references. Because of this, boxes enforce the object-capability model. To access and operate on a box one needs a special permission value that is “linked” to the box; sending a box to another actor consumes the permission of the box, which prevents concurrent access to the box from the sender and receiver. The fields of a Box object are externally unique [60], which is guaranteed from the enforcement of the object-capability model within a box. Scala’s powerful type system, with its use of implicits and path dependent types [10, 72], can infer most of the permission usages, so that developers do not thread permissions explicitly through the program. The type system guarantees data-race freedom in a process-based calculus, among other properties.

ABS

The Abstract Behavioural Specification Language (ABS) is a modelling language suitable for automated analysis of complex concurrent behaviour, realised via concurrent object groups (cogs) [127]. ABS has several backends, ranging from Erlang to Scala, is data-race free and has tooling support to guarantee deadlock freedom [74].

A cog selects an object and the object gets to execute a message from its message queue in FIFO order. Cogs have cooperative scheduling, i.e., once the cog schedules an object, the cog cannot preemptively stop the execution of the object. Extra annotations on object construction specify whether new objects run in the current cog or under a new cog.

All objects are subject of synchronous and asynchronous method calls, and asynchronous method calls place messages in the target objects’ message queue and return a future. Future constructs can stop the method execution and release the object from the cog (cooperative scheduling), such that the cog
can pick another object to run [74, 127]. ABS has explicit and typed control-flow futures with synchronous and asynchronous synchronisation constructs. Surprisingly, ABS also has implicit control-flow futures. These futures are not visible to the developer and happen as a side-effect of data-race freedom. Concretely, when an object tries to synchronously perform a method call on an object owned by another cog, the runtime performs an asynchronous message send that returns a future, and this future is immediately synchronised with a blocking `get` operation [127]. Unlike futures in task- and active object-based programming languages, the blocking `get` synchronisation not only blocks the object, but (it) blocks the whole cog.

**ASP**

ASP is an asynchronous calculus developed by Caromel et al [44] that integrates active objects and passive objects. Communication between active objects happens via asynchronous method calls, and synchronous method calls on passive objects. An asynchronous method call returns immediately an implicit, typed, data-flow future with synchronous operators, i.e., wait-by-necessity synchronisation [42]; passive objects are shared by deep copy [74].

ProActive [43] is Java library implementation of the ASP object calculus. Unlike Pony or Encore, active objects are considered coarse-grained and the ProActive middleware does not expect to handle hundreds to thousands of active objects on the same machine.

Recently, ASP added multi-active objects [198, 113]. Multi-active objects allow parallel and concurrent execution of methods on the same active object, breaking the one-thread-of-control per active object reasoning. Methods are annotated with assertions (named concerns) and conflicting assertions do not run concurrently. This guarantees data-race freedom when methods are correctly annotated, but this is not a strong guarantee, as data-races may happen when developers do not identify correctly conflicting methods; nothing prevents deadlocks from happening.

**AmbientTalk**

AmbientTalk [71] is an untyped, distributed, object-oriented language based on communicating event loops with explicit data-flow futures and asynchronous synchronisation constructs. Futures are fulfilled with a far reference (as in E [165]) or by a deep copy of a value. Deep copying happens on isolate objects, i.e., objects that cannot capture free variables from the lexical scope.

AmbientTalk’s futures queue asynchronous messages until the value is resolved; once the future is resolved with a value, these messages are send to the (event loop) value’s owner.

7Other languages such as Joelle do this using effects and ownership types [61].
AmbientTalk was designed to run on devices connected to mobile ad-hoc networks, e.g., running on smart phones. In such distributed environment, programs use a publish-subscribe service to make themselves available to other peers. A program makes itself available to the network by publishing topics names (type tags), and subscribers can register themselves on topics (under the assumption that there is a categorisation scheme that uniquely identifies topics, i.e., type tags on multiple programs refer to the same topic).

AmbientTalk is data-race and deadlock free, as there are no blocking synchronisation constructs on futures.

Capabilities for Sharing

Boyland et al defined an object-oriented language with a core set of reference permission that allows them to express common capabilities features, such as readonly, immutability, and linear references [33]. (This work does not include concurrency features, but we thought it deserves a special mention in the list.)

The authors defined 7 (seven) compositional reference permissions, grouped as access permissions, exclusive permissions, and ownership permission. Access permissions include read (readonly), write (mutable), and identity permissions; exclusive permissions are analogous to access permissions but guarantee exclusiveness. For example, accessing a field requires a read access permission, field assignment requires a write permission, and identity comparison requires the identity permission. The combinators assert and limit remove permissions from all aliases to an object (global effect) and from the current reference (local effect), respectively.

A variable with an ownership permission is an owned variable; variables that lack ownership are borrowed variables. Assertions from borrowed variables only have an effect on borrowed variables; assertion of owned variables affect the reference permissions of all owned and borrowed references.

This core set of capabilities can express linearity (as in Encore and Pony), destructive reads, readonly, transitive readonly (as in Gordon's work), and immutability, among other features. The main idea of this work is to separate the capability semantics from the enforcement of the invariants, through the use of limit and assert.

ConstraintJava

ConstraintJava is a dynamic, object-oriented language incorporating dynamic ownership [100, 101]. It is a class-based language that builds on top of BeanShell [177], and supports subtyping via inheritance.

ConstraintJava incorporates dynamic ownership where objects can send messages only to visible objects, i.e., objects that have a common owner and do not cross ownership boundaries. That is, if we use the notation $a \rightarrow b$ to mean
that $a$ owns $b$, and we have the following ownership tree, $a \rightarrow b, a \rightarrow c, c \rightarrow d$, then $b$ and $c$ can exchange messages because they have the same owner, but $b$ cannot exchange messages with $d$, while $d$ can exchange messages with $b$ because they have the same root owner $a$. Thus, dynamic ownership is restricted to tree shape ownership and cannot cross ownership boundaries.

To forbid introducing dependencies between visible objects that break the ownership boundary, e.g., leaking mutable state from $d$ to $b$ when $b$ sends a message to $c$, ConstraintJava introduces three different message sends: internal (for creating object dependencies among owned objects) and external (for interacting with unowned objects). Within the external dependencies, ConstraintJava has pure messages, which do not access mutable state and can only return immutable objects (final fields in Java [103]), and oneway messages, which return $\text{void}$ but are allowed to update mutable state. Runtime checks maintain the dynamic ownership model.

(ConstraintJava does not have concurrency constructs; we believe the model could maintain data-race freedom, but it must be ensured by the developer.)

**Distributed Electronic Rights For ECMAScript**

The language *Dr. SES* (Distributed Resilient Secure ECMAScript) [166] adds object-capabilities to the JavaScript language and requires of a small subset of JavaScript and two key constructs: records (anonymous objects) and arrow functions [5], and the asynchronous message send operator ($!$).

*Dr. SES* builds on SES (Secure ECMAScript), an object-capability subset of ECMAScript 5 where objects can only interact with the references they hold, and global objects from JavaScript are transitively immutable (powerless [162]). The concurrency model uses communicating event loops; the web workers API creates isolated event loops that communicate via deep copying. The main goal of the language is to add smart contracts to JavaScript and the necessary glue for a distributed, secure, and persistent platform.

*Dr. SES* uses object-capabilities to safely execute third party code, where global objects are powerless. To control asynchronous communication, it incorporates promises with similar semantics to E’s promises [165]. The concurrency model use pass-by-copy on primitive and array values, and pass-by-reference otherwise.

**E**

E [165] is a distributed, object-oriented language with object-capabilities [165, 167, 168]. Objects may receive synchronous and asynchronous message sends. Synchronous message sends are the common call-return control-flow expression; asynchronous message sends return a promise. Promises are fulfilled implicitly by the callee on termination of the method call. But promises
can be explicitly created. Promises are explicit, typed, with data-flow semantics and with asynchronous synchronisation. Because there is no blocking synchronisation, the language is deadlock free.

E is a communicating event loop language. Each event loop has a single thread-of-control and owns a bunch of local objects. When a promise is fulfilled with an object owned by a different event loop process (vat), the promise contains a far reference (proxy) to the remote object. Only message sends are allowed on far references, and trying to do a synchronous call throws a runtime error. Unfulfilled promises accumulate asynchronous messages until they are fulfilled, and forward the messages when fulfilled.

Erlang

Erlang is an actor-based, functional programming language, born at the Ericsson Computer Science Lab with the aim of improving how to program telephony applications. Telephony applications are highly concurrent, parallel, and distributed, and must be fault-tolerant. [13]

Actors have their own heap and do not share state between them; communication happens via message passing, where data is immutable and copied on message send. It falls out that the language is data-race free by construction, from using immutable data [55], but not deadlock free [54].

Many Erlang applications rely on the Erlang/OTP libraries, which include the Erlang runtime system as well as an extensive set of libraries for facilitating writing fault-tolerant applications. The use of Erlang/OTP opens the door to data-races because Erlang/OTP has constructs that allow actors to share state between them [55], e.g., ETS [142, 199]. The absence of Erlang/OTP libraries makes Erlang a data-race free language.

4.7 Discussion

Papers I – III introduce purely functional concurrent (future-based) abstractions to allow developers to express complex concurrent coordination and delegation patterns. Paper IV extends the applicability of Papers I–III to the imperative domain. As an example, the functional ParT abstraction (Paper I) can share mutable state without introducing data-races. The example in Fig. 4.3 shows a language that uses the Dala programming model and the ParT abstraction to find a solution to the N-Queen problem [18]. The N-Queen problem consists of placing N queens in a NxN chess board such that no queen can attack any other queen (following the chess rules).

The program starts in the solveNQueen function, which receives an isolated (alias-free) array of isolated strategies. Each strategy uses a different algorithm to solve the N-Queen problem. Line 3 lifts the array with strategies to the ParT abstraction; Line 4 uses the bind combinator ( >>= ) to create
We have covered all the necessary background on concurrency and synchronisation, examined related work, and shown how our work on concurrent and future abstractions can be used in an imperative setting. We are ready to proceed to the concluding remarks.
5. Conclusion

This thesis proposes a concurrent functional abstraction to control futures (Paper I), future combinators to express control- and data-flow computations (Papers II–III), and a novel capability-based programming model that can be used to extend the applicability of the work done in previous papers (Papers I–III).

The contributions of this thesis are:

**Paper I** Design, formalisation, and implementation of a concurrent functional abstraction, ParT, with asynchronous combinators that allow developers to express complex speculative concurrent computations.

**Paper II** Introduces an existing construct into a concurrent future-based language, defines a translation strategy from the future-based language to a low-level concurrent promise-based language, and shows that the translation is semantics preserving.

**Paper III** First language (that the authors know of) that incorporates explicit, typed, control- and data-flow futures with synchronous and asynchronous synchronisation combinators.

**Paper IV** Design and formalisation of Dala, a concurrent, capability-based programming model that maintains data-race freedom in the presence of shared mutable state. Dala extends the applicability of the work done in previous papers.

Future work involves adding gradual typing to our capability-based language, such that developers can choose whether data-race freedom is a static or dynamic guarantee. We started some work towards this goal with the design of a gradual capability-based system. After this is in place, we would like to explore the addition of a gradual type system, and the interplay between the gradual capability system and gradual type system, extending Bañados et al work on gradual effects [203].
References


[26] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded

[27] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In Hailpern et al. [106], pages 35–46.


[51] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient


[61] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen.


[86] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and


[96] Jean-Yves Girard. Une extension de l’interprétation de gödel a l’analyse, et
son application a l’elimination des coupures dans l’analyse et la theorie des
types. In *Studies in Logic and the Foundations of Mathematics*, volume 63,

1987.

[98] Brian Goetz, Tim Peierls, Joshua J. Bloch, Joseph Bowbeer, David Holmes,

[99] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and
Joe Duffy. Uniqueness and reference immutability for safe parallelism. In
Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th
Annual ACM SIGPLAN Conference on Object-Oriented Programming,
Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012,

[100] Donald Gordon and James Noble. Dynamic ownership in a dynamic language.
Symposium on Dynamic Languages, DLS 2007, October 22, 2007, Montreal,


[102] J. A. Gosden. Explicit parallel processing description and control in programs
for multi- and uni-processor computers. In *Proceedings of the November 7-10,
1966, Fall Joint Computer Conference*, AFIPS’66 (Fall), pages 651–660, New
York, NY, USA, 1966. Association for Computing Machinery.

[103] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel

Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

history tells us about thread synchronization. In Elisabetta Di Nitto, Mark
Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint
Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo,

[106] Brent Hailpern, Linda M. Northrop, and A. Michael Berman, editors.
*Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented
Programming Systems, Languages & Applications (OOPSLA ’99), Denver,

[107] Philipp Haller and Alexander Loiko. Lacasa: lightweight affinity and object

[108] Philipp Haller and Martin Odersky. Capabilities for uniqueness and
borrowing. In Theo D’Hondt, editor, *ECOOP 2010 - Object-Oriented
Programming, 24th European Conference, Maribor, Slovenia, June 21-25,

[109] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland
Kuhn, and Vojin Jovanovic. Scala Documentation. Futures and Promises.
https://docs.scala-lang.org/overviews/core/futures.html. (Accessed November,


[162] Adrian Matthew Mettler. *Language and Framework Support for*


Appendix A.
Notes and Errata

Paper I

- Fig. 4, the expression-level evaluation context has a missing rule:

\[ E ::= \ldots \mid v \ll E \]

- Reduction rule TS-ASYNC has misspelled the return type of the task in the premise judgement, the rule should be:

\[
\frac{(T\text{-ASYNC})}{\Gamma \vdash e : \tau} \quad \Gamma \vdash \rho \text{async } e : \text{Fut } \tau
\]

Paper II

- The types in Section 2.2 should be \( \tau ::= K \mid \text{Fut } \tau \mid \tau \rightarrow \tau' \). The abstraction type was missing, even when we use it in rule T-APP.
- The types in Section 3.2 should be \( \tau ::= K \mid \text{Prom } \tau \mid \tau \rightarrow \tau' \). The abstraction type was missing, even when we use it in rule TI-APP.
- Type Preservation (Lemma 1) had a minor misspelling, and \( \Gamma' \supseteq \Gamma \) should have been \( \Gamma' \supseteq \Gamma \).
- For concreteness, the micro-benchmark is in Fig. 5.1.

Paper III

- Section Extending FutFlow with Data-Flow Futures can be simplified (Page 19). Rules T-TYPEABSTRACTION and T-TYPEAPPLICATION can be the same as the ones in the FlowFut calculus, and the \( \downarrow \) in rule T-MATCH (\( \Gamma \vdash \rho \ e_3 : \downarrow \text{Fut } \tau \)) is not really required; we assume that the types of the premises are normalised, same as in FlowFut calculus.
read class Job
val workload: int
def init(w: int): unit
  this.workload = w
end

active class Broker
val workers: [Worker]
var current: UINT
val wsize: UINT

def init(): unit
  this.workers = [new Worker, new Worker, new Worker, new Worker]
  this.wsize = |this.workers|
end

def runBaseline(job: Job): int
  this.current = (this.current + 1) % wsize
  val worker = this.workers(this.current)
  forward(worker!start(job))
end

def run(job: Job): int
  this.current = (this.current + 1) % wsize
  val worker = this.workers(this.current)
  for i ← [0..workload] do
    if (result % 4 == 0) then
      result += i
    else
      result = result + 4
    end
  end
  return result
end

active class Worker
def start(job: Job): int
  val result = 0
  for i ← [0..workload] do
    if (result % 4 == 0) then
      result += i
    else
      result = result + 4
    end
  end
  return result
end

active class Main
def main(argv: [String]): unit
  val repetitions = match argv(1).to_int() with
    case Nothing => 4000
    case Just(i) => i
  end
  val bkr = new Broker()
  for i ← [0..999] do
    -- Baseline synchronous code
    bkr!runBaseline(new Job(repetitions))
    -- Asynchronous send + get strategy
    bkr!run(new Job(repetitions))
    -- Await strategy
    bkr!runA(new Job(repetitions))
    -- Forward Strategy
    bkr!runF(new Job(repetitions))
  end
end

Figure 5.1. Micro-benchmark used in Paper II.
A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-425128