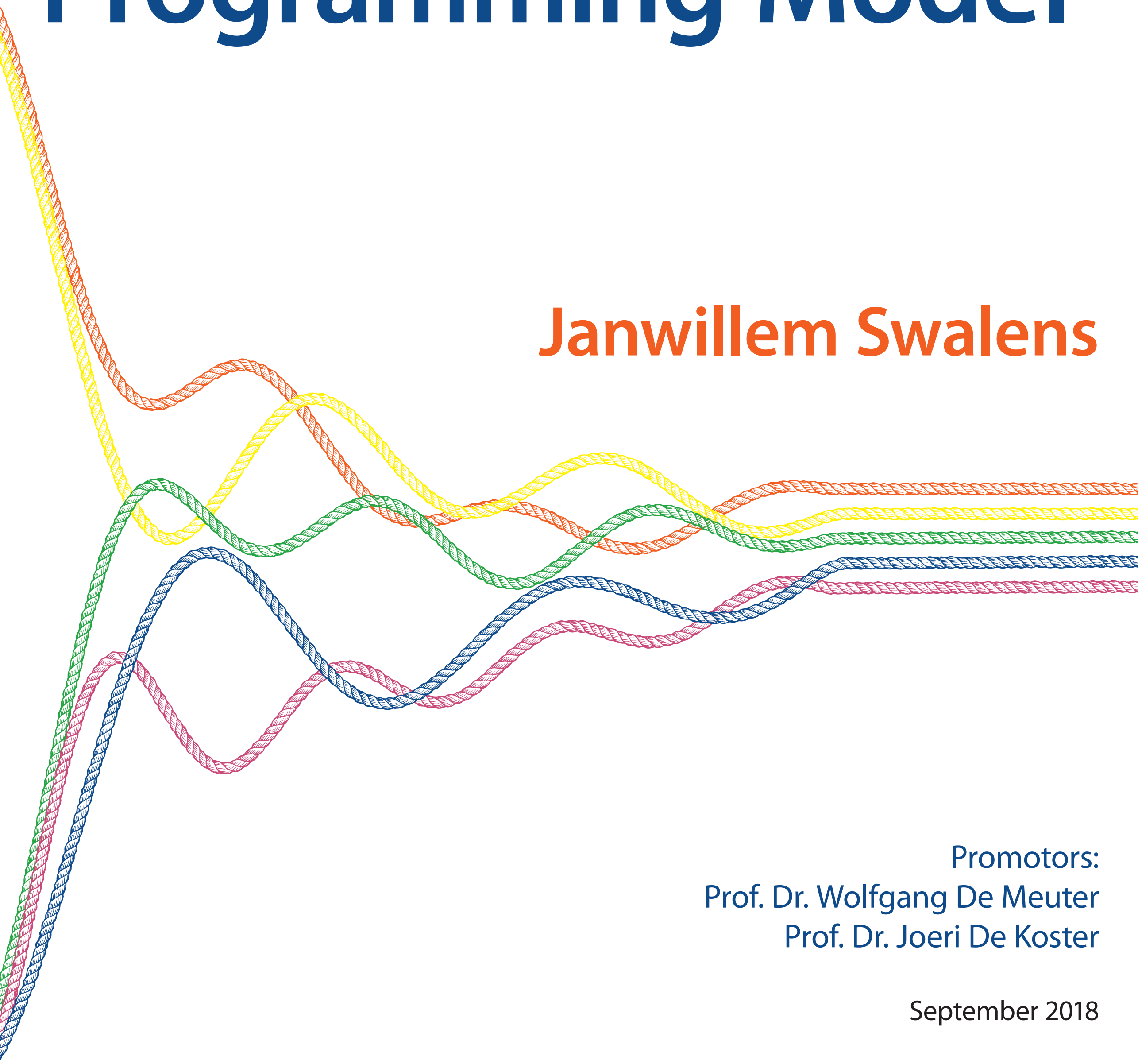
A Multi-Paradigm Concurrent Programming Model



Janwillem Swalens

Promotors:
Prof. Dr. Wolfgang De Meuter
Prof. Dr. Joeri De Koster

September 2018

VUB
VRIJE
UNIVERSITEIT
BRUSSEL

A Multi-Paradigm Concurrent Programming Model

Janwillem Swalens

A Multi-Paradigm Concurrent Programming Model

Janwillem Swalens

A dissertation submitted in fulfillment of the requirements for the award of the degree of Doctor of Science

September 2018

Jury:
Prof. Dr. Viviane Jonckers (chair)
Prof. Dr. Beat Signer (secretary)
Prof. Dr. Mira Mezini
Prof. Dr. Hridesh Rajan
Prof. Dr. Jan Lemeire
Prof. Dr. Wolfgang De Meuter (promotor)
Prof. Dr. Joeri De Koster (promotor)

Vrije Universiteit Brussel
Faculty of Science and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by Crazy Copy Center Productions
Pleinlaan 2, 1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN: 978-94-923-1296-9
NUR: 989

Abstract

Since the introduction of multicore processors, programmers can no longer rely on increasing clock frequencies to make their programs run faster “for free”. Instead, they have to explicitly use concurrency. However, concurrent programming is notoriously difficult. To this end, developers can use concurrency models: techniques that introduce parallelism in a controlled manner and provide guarantees to prevent common errors such as race conditions and deadlocks. In this dissertation, we look at three concurrency models from three categories: futures (which guarantee determinacy), transactions (which guarantee isolation and progress), and actors (which guarantee the isolated turn principle and deadlock freedom).

An empirical study has shown that existing programs and programming languages often combine multiple concurrency models. We study these combinations and show that they can annihilate the guarantees of their constituent models. Hence, the assumptions of developers are invalidated and the errors that were prevented by the separate concurrency models can resurface. For each combination, we examine which guarantees are broken when used in a naive, ad-hoc combination. Next, we study how the guarantees of both models can be maintained without limiting performance. We focus on two interesting cases in particular.

First, the combination of transactions and futures leads to *transactional futures*: futures created in a transaction with access to the encompassing transactional context. Using transactional futures, parallelism inside transactions can be exploited, benefitting from determinacy within the transaction and isolation between transactions.

Second, the combination of transactions and actors leads to *transactional actors*. These make it possible both to create transactions in actors, and vice versa, to send messages to actors in transactions. Our semantics maintains the isolation and progress guarantees of transactions, while guaranteeing low-level race freedom and deadlock freedom for the actors.

Finally, we combine all three models into one unified framework, called *Chocola* (COmposable COncurrency LAnguage), which we implemented as an extension of Clojure. We specify the operational semantics of Chocola and demonstrate its properties. Starting from three benchmarks from the commonly used STAMP benchmark suite, we demonstrate that by combining multiple concurrency models using Chocola, additional parallelism can be introduced in these programs, while requiring only a small effort from the developer.

To the best of our knowledge, this dissertation is the first to comprehensively study the combination of three radically different concurrency models – futures, transactions, and actors – and specify a semantics for their combinations that aims to introduce additional parallelism while maintaining their guarantees wherever possible. Using Chocola, developers can freely pick and mix the appropriate concurrency models for their use cases.

Samenvatting

Sinds de opkomst van multicore processors rond het jaar 2005 moeten programmeurs expliciet concurrency en parallellisme gebruiken om hun programma's sneller te maken. Programmeren met concurrency is echter erg moeilijk. Programmeurs gebruiken hiervoor "concurrencymodellen": technieken die parallellisme toevoegen aan een programma, maar op een gecontroleerde manier, zodat ze garanties bieden die vaak gemaakte fouten voorkomen (zoals race conditions of deadlocks). Dit proefschrift start van drie concurrencymodellen uit drie categorieën: futures (die determinisme garanderen), transacties (die isolatie en progress garanderen), en actors (die het isolated turn-principe en de afwezigheid van deadlocks garanderen).

Een empirische studie toonde aan dat bestaande programma's en programmeertalen vaak verschillende concurrencymodellen combineren. Wij tonen aan dat het combineren van modellen hun garanties kan teniet doen. Dit breekt de veronderstellingen van programmeurs, waardoor opnieuw de fouten opduiken die door de aparte concurrencymodellen werden voorkomen. Voor elke combinatie bestuderen we welke garanties gebroken worden in een naïeve, ad-hoc combinatie. Vervolgens onderzoeken we hoe de garanties van beide modellen toch kunnen behouden worden, zonder de performantie te beperken. We focussen op twee interessante gevallen.

Ten eerste leidt de combinatie van transacties en futures tot *transactional futures*. Dit zijn futures die aangemaakt worden in een transactie én toegang hebben tot die transactie. Met transactional futures kan het parallellisme binnenin transacties benut worden, met de garantie op determinisme binnen de transactie en isolatie tussen de transacties.

Ten tweede combineren we transacties en actors in *transactional actors*. Hiermee kunnen zowel transacties in actors gemaakt worden als, vice versa, berichten gestuurd worden naar actors in transacties. Onze semantiek behoudt de garanties op isolatie en progress van transacties en garandeert de afwezigheid van low-level races en deadlocks

van actors.

Ten slotte voegen we de drie modellen samen in één framework: *Chocola* (voor COmposable COncurrency LAnguage). We implementeerden Chocola als een uitbreiding van Clojure. We definiëren de operationele semantiek van Chocola en tonen aan welke garanties het biedt. Als evaluatie breiden we drie programma's uit de vaak gebruikte STAMP benchmark suite uit, door verschillende concurrencymodellen te combineren. Hiermee demonstreren we dat met Chocola parallellisme kan toegevoegd worden aan deze programma's, wat de performantie verhoogt, en dat dit slechts weinig moeite vereist van de programmeur.

Voor zover wij weten is dit proefschrift het eerste dat systematisch de combinaties van drie radicaal verschillende concurrencymodellen bestudeert – futures, transacties en actors – en een semantiek definieert voor hun combinaties die parallellisme toevoegt terwijl hun garanties waar mogelijk behouden blijven. Zo kunnen programmeurs met Chocola een rijk palet aan concurrencymodellen gebruiken en mengen in hun programma's.

Acknowledgements

First of all, I would like to thank my promotors, Wolf and Joeri. Their guidance and support has been essential throughout my PhD and their suggestions greatly contributed to improving this text. Furthermore, I would like to thank Tom and Stefan for their help in finding and defining an interesting thesis topic and for their contributions to numerous grant applications.

I also thank the members of the jury – Mira Mezini, Hridesh Rajan, Jan Lemeire, Beat Signer, and Viviane Jonckers – for their insightful remarks during the defense.

Moreover, I would like to thank all current and ex-SOFTies, both for their scientific contributions during PPP meetings and research presentations, as well as for the atmosphere they created during discussions at lunch, on Slack, and at Opinio. Every SOFTie contributes in their own unique way to make SOFT what it is.

Enkele collega's wil ik in het bijzonder bedanken. Mattias, just voor de muziekjes van de dag. Nathalie, voor het plannen van de Canada-reis. Simon, voor de Benvolios-broodjes. Laure, voor het klavertjevier dat nog steeds bloeit. Thierry, voor de droge opmerkingen. Jesse, voor de directe opmerkingen. Jens, mijn beste klant. Quentin, for proofreading this text. Daarnaast bedank ik ook de Zevergroep (Jean, Sjnico, Didi, Carrein, Piet, Banny, Meire en Leiderkamp) voor de gesprekken en trips, en ten slotte ook mama, papa en Maud om er altijd voor mij te zijn.

Veel leesplezier!

Contents

1 Introduction

During the previous decade, a turning point was reached in the development of processors: while from 1970 until 2000 processors got faster by increasing their clock frequency, in accordance to Moore's law [Moore 1998], this became impossible around 2005 due to exponential increases in power usage and heat generation [Geer 2005]. As a result, chip makers introduced **multicore processors**: processors that contain multiple cores that execute instructions in parallel. The number of cores on multicore processors has increased since their introduction and this trend is predicted to continue [Bright 2017a,b].

This "multicore revolution" also affects software developers: while previously they could rely on increasing clock frequencies to make their programs run faster "for free", they now have to exploit the parallel hardware explicitly [Sutter 2005]. To this end, programs use **parallelism** and **concurrency**: multiple computational activities are active at the same time. Besides performance, a second reason to use concurrency is to separate logically independent computations [Sutter 2005]: even on single-core processors programmers may decide to use concurrency to structure programs, for fault tolerance, or for security.

Moreover, powerful computers have become more accessible than ever since the introduction of Infrastructure-as-a-Service in the cloud in 2005. The decreasing price of cloud computing infrastructure has made big clusters of many-core machines available even to medium-sized businesses. This hardware is used to run intensive computations, e.g. in the field of machine learning or 'Big Data', and services available to large numbers of users, such as typical cloud applications. In those cases, parallelism and concurrency are indispensable: computations must be distributed over the available hardware to handle large amounts of data and user-facing services must scale as the number of users increases.

Unfortunately, concurrent programming is notoriously difficult [Farchi et al. 2003, Hovemeyer and Pugh 2004b,a, Lu et al. 2008]. In low-level programming languages, developers exploit concurrency by manually creating threads and managing shared resources using locks. However, threads can interact in unexpected and undesirable ways. There are three common problems: race conditions, which cause an incorrect result or a crash due to the order in which read and write instructions from multiple threads are interleaved; deadlocks, which cause the program to hang when multiple threads are waiting on each other because they acquire the same locks; and livelocks, which cause the program to reexecute a part of the code over and over, e.g. when a lock cannot be acquired.[1]

To prevent these problems, researchers have developed concurrency models. A **concurrency model** provides constructs to introduce parallelism and to manage concurrent access to shared resources. However, at the same time it imposes restrictions, in order to provide **guarantees** to the programmer that prevent common errors. For example, Communicating Sequential Processes (CSP) is a concurrency model in which parallelism can be introduced by creating processes [Hoare 1978]. It restricts the ability to share data by requiring the use of messages, and therefore, the programmer can reason about the program at the level of messages, as they are the only form of communication between processes. Another example is Nested Data Parallelism (NDP), a concurrency model to process lists and matrices in parallel [Chakravarty and Keller 2001]. It restricts operations that are executed in parallel to have no side effects, and can thus guarantee determinism.

Concurrency models can be implemented as a library or embedded into the programming language. Concurrency models that are embedded into the programming language can enforce stronger restrictions, enabling them to provide certain guarantees that a library cannot provide. For instance, when CSP is embedded into the programming language, the language can statically ensure that data can only be exchanged between processes using messages. This helps to avoid race conditions, as no two processes will ever write to the same memory location.

In this dissertation, we will study multiple concurrency models and their guarantees when they are embedded in a programming language.

Categories of concurrency models

Over several decades, researchers and programming language designers have been developing a plethora of concurrency models. Each concurrency model provides different programming language constructs, imposes different restrictions, and guarantees

[1] More formal definitions of these three bugs have been given by Tanenbaum and Bos [2014]. Deadlocks occur when four conditions hold, first defined by Coffman et al. [1971] and summarized by Tanenbaum and Bos [2014, page 440].

		Non-deterministic	
Category	**Deterministic**	**Shared Memory**	**Message Passing**
Example models	• Futures • Fork/Join • NDP	• Locks • Transactions	• CSP • Actors • Active objects
General use case	Parallelism for performance	Multiple threads that modify a central data structure	Separable components that occasionally exchange data
Example use case	Matrix multiplication	Shared queue of work	Web service

Table 1.1: An overview of the three categories of concurrency models and their use cases.

different properties. As a result, each concurrency model is suited to specific use cases.

Van Roy and Haridi [2004] systematize this variety of concurrency models by partitioning them into three categories. Concurrency models are first grouped into deterministic and non-deterministic models; the non-deterministic models are further subdivided into those that have shared memory and those that rely on message passing. (We will describe these categories in more detail in Section 2.1 of Chapter 2.)

The models in each category share general use cases:

- **Deterministic models** guarantee that a parallel program returns the same result no matter in which order threads are interleaved. They are typically used to exploit performance: their guarantee of determinism frees the developer from worrying about correctness, to focus on performance instead. Some example use cases for these models are parallel operations on lists or matrices, such as searching in a list or multiplying matrices.
- **Shared-memory models** are used in programs that contain shared data structures, to ensure safe access to the shared data. One specific example is a shared queue of work, from which several threads process elements.
- **Message-passing models** are typically used to implement separable components that occasionally exchange data. For instance, a web service can process incoming requests in parallel using a message-passing model, as each request is independent but may communicate with common resources such as a database.

Table 1.1 lists the three categories, some models in each category, and their use cases.

Model	**Futures**	**Transactions**	**Actors**
Category	Deterministic	Shared memory	Message passing
Guarantees	• Determinacy	• Isolation • Progress	• Isolated turn principle • Deadlock freedom

Table 1.2: The three concurrency models we study in this dissertation and their guarantees.

Futures, transactions, and actors

In this dissertation, we will focus on three radically different concurrency models, one from each category: futures, transactions, and actors. Each provides certain guarantees to the developer (summarized in Table 1.2; expanded upon in Chapter 2):

- **Futures** are placeholder values that represent the result of a concurrent computation [Flanagan and Felleisen 1995]. They are a deterministic model, thus guaranteeing *determinacy*.
- **Transactions** are sections of the code in which memory, represented as transactional variables, can be safely accessed and modified, to be shared with other transactions [Shavit and Touitou 1997]. Each transaction is executed atomically: it appears to execute in a single step, thereby guaranteeing that transactions run in *isolation*. Furthermore, when multiple transactions are running simultaneously, they are guaranteed to make *progress*, so that one transaction cannot 'hold back' another forever.
- **Actors** are entities with private memory and an inbox. They run concurrently and communicate by exchanging messages [Hewitt et al. 1973, Agha 1985]. Actors process each message in a 'turn'. Once a turn starts, it is isolated from changes outside the current actor: this is the *isolated turn principle*. Furthermore, actors guarantee the *absence of deadlocks*, as they do not provide any blocking operation.

1.1 Problem Statement

In practice, **it is useful to combine different concurrency models**. We base this assertion on three observations (further elaborated in Chapter 3, Section 3.1):

1. Existing applications already combine concurrency models. An empirical study of 15 Scala projects that use the actor model has shown that 80% of them combine it with another concurrency model [Tasharofi et al. 2013].

2. Many programming languages (and libraries developed for them) support different concurrency models. For instance, Clojure has built-in support for six models: atomic variables, Software Transactional Memory, futures, promises, asynchronous agents and Communicating Sequential Processes. Likewise, Java provides threads, locks, atomic variables, futures, and Fork/Join. These languages allow their concurrency models to be mixed freely by the developer, but unfortunately doing so correctly is far from trivial.
3. As each concurrency model targets specific use cases, complex applications can consist of different parts that each suit different concurrency models.

This dissertation studies the combinations of concurrency models. In Chapter 3, Section 3.2, we start with a case study of Clojure. We show that **when multiple concurrency models are combined, their guarantees are often invalidated**. For instance, sending a message inside a transaction can cause the message to be sent multiple times. Hence, the assumptions of developers are broken and the errors that were prevented by the separate concurrency models can resurface. This is not desirable.

In summary, this dissertation aims to tackle the following problem:

> Programs often combine multiple concurrency models, but this can invalidate the guarantees of the individual models, breaking the assumptions of the developer. How can these guarantees be maintained even when models are combined?

1.2 Research Goal and Approach

The goal of this dissertation is to **provide a linguistic framework that unifies futures, transactions, and actors**. We have two requirements:

1. When used separately, the semantics of each model should remain unchanged, so that existing programs work as before.
2. When models are combined, we seek a semantics for their combinations that maintains each model's guarantees – except when it is impossible to do so. In those cases, we replace the original guarantee with a less restrictive one.

This dissertation takes a programming language-based approach, allowing us to focus on the essential elements of each model. First, we define the semantics and guarantees of each concurrency model separately, working on top of a standard functional language. Then, we build our unified framework in several steps:

- We study 'naive', ad-hoc combinations of the three concurrency models. This consists of examining each pairwise combination, and studying which guarantees are broken in the naive combination.

- Next, we examine whether it is possible to specify a semantics of each pairwise combination that maintains the guarantees. If this is not possible, we define a less restrictive guarantee.
- Finally, we unify these semantics into one framework that combines the three concurrency models. We formalize its operational semantics, to precisely define its operations in any context and to establish its guarantees. Furthermore, we implement it and evaluate its performance using benchmark applications.

1.3 | Contributions

This dissertation makes the following contributions:

- To the best of our knowledge, this dissertation is the first to **comprehensively and systematically study the combination of three concurrency models**: futures, transactions, and actors. We study these combinations and, for each combination, we examine which guarantees are broken in a naive, ad-hoc combination.
- This dissertation introduces **transactional futures**: futures created in a transaction with access to the encompassing transactional context. Using transactional futures, parallelism in transactions can be exploited, benefitting from determinacy within the transaction and isolation between transactions.
- The dissertation also introduces **transactional actors**. These make it possible both to create transactions in actors, and vice versa to send messages to actors in transactions. Our semantics maintains the isolation and progress guarantees of transactions, while guaranteeing low-level race freedom and deadlock freedom for the actors.
- Finally, we combine futures, transactions, and actors into **one unified framework: Chocola** (COmposable COncurrency LAnguage). Chocola maintains the semantics of each model when used separately. When the models are combined, Chocola defines a semantics that maintains the guarantees of each model wherever possible. Hence, developers can mix multiple models in one program, in each part of the program using whichever model fits best for the problem at hand. We make three contributions:
  - A **specification of the operational semantics** of Chocola, **PureChocola**, which we use to demonstrate its guarantees.
  - An **implementation** of Chocola on top of Clojure, a programming language that already supports multiple concurrency models.

- An **evaluation** using three benchmark applications from the commonly used STAMP benchmark suite. By extending a program that uses one concurrency model with another, we demonstrate that additional parallelism can be exploited, leading to better performance, while requiring only a small effort from the developer.

The implementation of Chocola, an executable implementation of the semantics of PureChocola, and the benchmark applications used to evaluate Chocola are available at http://soft.vub.ac.be/~jswalens/chocola/.

1.4 Outline

This dissertation is organized as follows:

Chapter 2: Concurrency Models: Futures, Transactions, and Actors describes how concurrency models can be classified into three categories – deterministic, shared-memory, and message-passing models – and then picks one model from each category to study in detail: futures, transactions, and actors. We describe each model's language constructs, use cases, guarantees, and operational semantics.

Chapter 3: Combining Concurrency Models looks into the naive combinations of concurrency models and the problems this causes. First, we explain why developers combine concurrency models in practice. Next, we use Clojure as a case study of a programming language in which such combinations lead to unexpected results. Finally, we expand on the goal of this dissertation and outline our approach to tackling these problems.

Chapter 4: Transactional Futures: Parallelism in Transactions examines the combination of futures and transactions. We motivate the use of futures in a transaction using an example and show the problems that occur when this is done using a naive combination of both. Hence, we introduce transactional futures: futures created in a transaction with a well-defined semantics and useful guarantees.

Chapter 5: Transactional Actors: Communication Between Transactions examines the combination of actors and transactions. We explain the reasons for combining both models and show which problems occur in a naive combination. Then, we introduce transactional actors as a way to introduce communication using actors in transactions with a well-defined semantics that maintains the expectations of developers.

Chapter 6: Chocola: a Language That Unifies Futures, Transactions, and Actors unifies the three concurrency models we examined into one framework, called Chocola. We first consider the third combination not discussed before: futures and actors. Then, we describe all concepts in Chocola and summarize its guarantees.

Chapter 7: PureChocola: an Operational Semantics presents PureChocola, a formalization of the operational semantics of Chocola, and shows how its properties can be inferred from the semantics.

Chapter 8: An Implementation of Chocola describes how Chocola modifies existing implementations of the three separate concurrency models to support transactional futures and transactional actors.

Chapter 9: Evaluation evaluates the benefits of Chocola. In a quantitative evaluation, we extend three benchmark applications that use transactions with futures and actors and demonstrate that the additional parallelism can improve performance. Using a qualitative evaluation, we determine that this requires only a small effort from the developer.

Chapter 10: Conclusion presents our conclusions and some ideas for future research.

1.5 Publications

Supporting publications

Parts of this dissertation appear in the following publications:

Transactional Tasks: Parallelism in Software Transactions
Janwillem Swalens, Joeri De Koster, Wolfgang De Meuter
Published in the *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016)*
Received a *Distinguished Paper Award* at ECOOP 2016.
This paper combines transactions and futures to create transactional futures. It describes one contribution of this dissertation and forms most of Chapter 4.

Transactional Actors: Communication in Transactions
Janwillem Swalens, Joeri De Koster, Wolfgang De Meuter
Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems (SEPS 2017)
This paper combines transactions and actors, introducing transactional actors. It describes another contribution of this dissertation and forms most of Chapter 5.

Chocola: Integrating Futures, Actors, and Transactions
Janwillem Swalens, Joeri De Koster, Wolfgang De Meuter
Proceedings of the 8th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE 2018)
This paper describes Chocola as an integration of futures, actors, and transactions. It compiles the contributions of this dissertation, describing Chocola's semantics and evaluation, as in Chapters 6, 7 and 9.

Towards Composable Concurrency Abstractions
Janwillem Swalens, Stefan Marr, Joeri De Koster, Tom Van Cutsem
Proceedings of the Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES 2014)
This paper systematically studies all pairwise combinations of the six concurrency models supported by Clojure and reports in which cases their guarantees are maintained or broken. It is discussed in Section 3.2 and describes the problem that we aim to solve in this dissertation.

Other publications

I contributed to the following other publications during the course of my research:

Cloud PARTE: Elastic Complex Event Processing based on Mobile Actors
Janwillem Swalens, Thierry Renaux, Lode Hoste, Stefan Marr, Wolfgang De Meuter
Proceedings of the 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE 2013)
This paper introduces Cloud PARTE: a distributed version of the Rete algorithm that can be used to implement complex event detection systems. It is implemented using mobile actors: these are actors that can move between machines.

Just-in-time inheritance: a dynamic and implicit multiple inheritance mechanism
Mattias De Wael, Janwillem Swalens, Wolfgang De Meuter
Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)
This paper introduces Just-in-Time Inheritance, a form of multiple inheritance in which one parent is “favored” over the others. It is the first implicit and dynamic multiple inheritance mechanism.

2

Concurrency Models: Futures, Transactions, and Actors

This chapter describes the three concurrency models at the basis of this dissertation. In Section 2.1, we describe a taxonomy that partitions concurrency models into three categories: deterministic, shared-memory, and message-passing models. In Section 2.2, we select one model from each category to study in the rest of this dissertation: futures, transactions, and actors. In sections 2.3, 2.4, and 2.5, we describe each in detail: we define the relevant terminology, list their language constructs, demonstrate their use cases, and list their properties. We also provide a formalization of an operational semantics of each model, which we will use as building blocks in subsequent chapters.

Appendix A defines some common notation used in the operational semantics throughout this dissertation.

2.1 Categories of Concurrency Models

A concurrency model provides programming language constructs to introduce parallelism and to manage concurrent access to shared resources. However, at the same time it imposes restrictions, in order to provide **guarantees** to the programmer that prevent common errors and make the code easier to understand, maintain, and debug. Over several decades, researchers and practitioners have developed a wide variety of concurrency models. They each provide different constructs and different guarantees, and are thus each aimed at certain use cases.

In this dissertation, we use the taxonomy of Van Roy and Haridi [2004], who systematize the breadth of existing concurrency models into three categories: deterministic, shared-memory, and message-passing models. The last two are non-deterministic, as illustrated in Figure 2.1. In this section, we describe the three categories, list some representative concurrency models for each, and describe some of the use cases and properties common to the models of each category.

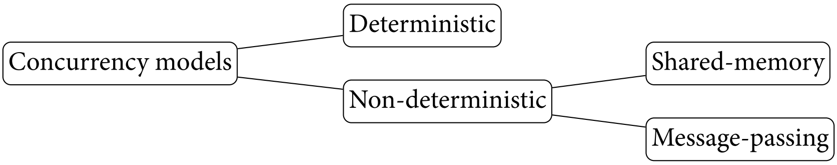


Figure 2.1: Categories of concurrency models [Van Roy and Haridi 2004]

Deterministic models

Deterministic models guarantee that, given the same input, a program will always produce the same output. When using such a model, a programmer can verify a program's output for given inputs and be sure that it will always work for those inputs. The order in which the threads will be interleaved during execution does not affect the end result, so there cannot be any race conditions or deadlocks that only appear in some executions. Hence, these models are suitable when parallelizing for performance: if a program produces the correct result for a given input, it will always do so, and the programmer can focus on tweaking the program to get the best performance.

Examples of deterministic models are futures [Baker and Hewitt 1977, Halstead 1985], promises [Liskov and Shrira 1988], Fork/Join [Blumofe et al. 1995, Lea 2000, Cavé et al. 2011], Nested Data Parallelism [Chakravarty and Keller 2001], and dataflow [Van Roy and Haridi 2004]. Some typical use cases are the parallelization of search, matrix addition or multiplication, or operations on large lists. In these examples, determinism is required: the output of these algorithms should only depend on the input, no matter the number of threads or how they are interleaved.

Typically, these models do not allow operations with side effects to occur in parallel. By requiring that all operations are purely functional, the order in which they are interleaved does not matter and thus determinism is achieved. We give a few examples of deterministic models and how they achieve determinism:

- Futures are *semantically transparent*: a functional program with futures evaluates to the same result as that program with the futures elided [Flanagan and Felleisen 1995]. This is true no matter the order in which threads are scheduled, and therefore the result is deterministic. Futures can thus safely be added wherever the programmer suspects their benefits will outweigh their costs, without affecting the correctness of the program.

- Cilk is an implementation of Fork/Join. It defines the *serial elision* of a program as that program with all Cilk-specific keywords removed. The serial elision of a Cilk program has the same semantics as the parallel version [Randall 1998]. Again, this is true no matter the order of interleaving, and therefore guarantees determinism.
- Data Parallel Haskell is an implementation of Nested Data Parallelism for Haskell that provides several operations on lists, such as mapping, filtering, concatenation, and transposition [Jones et al. 2008]. It guarantees determinism by using the type system to require that the applied functions are purely functional, i.e. that they have no side effects.

Shared-memory models

Shared-memory models coordinate access to shared memory from multiple threads. These models are suited to programs where multiple tasks read and modify one or more centralized data structures. An essential focus of these models is preserving the consistency of the shared data structure: one thread should not be able to observe the intermediate actions of another. For instance, when $100 is transferred between two bank accounts, it should be impossible to observe the intermediate state where the money was taken out of one account but not yet put into the other.

A common technique is to protect access to shared memory using locks or semaphores [Dijkstra 1965]. By holding one or several locks during the execution of a section of the code that accesses shared memory, called a *critical section*, the developer can ensure that one thread has exclusive access to that data. Thus, the critical sections are serialized: multiple sections that access the same data cannot execute in parallel, but will be executed serially.

Transactions are another technique to protect access to shared memory [Herlihy and Moss 1993, Shavit and Touitou 1997]. In this model, the programmer encapsulates code that accesses shared memory in a *transaction*. While the transaction is running, modifications made to shared memory are stored locally. Only at the end of the transaction will it attempt to commit these changes, persisting them so they can be observed by other threads. When two transactions modify the same data, a conflict is detected during (or even before) the commit, and the transaction will roll back and retry. This ensures all changes in a transaction are made visible in a single, atomic step, and that a transaction always has a consistent view of the shared memory.

Transactional systems avoid some of the common pitfalls of locks. In lock-based systems, the programmer must manually safeguard critical sections by acquiring locks. Failing to do so can lead to race conditions; acquiring locks multiple times or in the wrong order can cause deadlocks. Transactional systems typically avoid these problems by automating conflict detection.

Both critical sections protected with locks as well as transactions guarantee *atomicity*: the changes to shared memory they contain become visible in a single, indivisible step. They ensure that no other thread can observe the intermediate states, therefore, they should contain code that moves the shared memory from one consistent state to another, while the inconsistent intermediate states within a critical section or transaction are hidden from other threads.

Note that atomicity is a less strict guarantee than determinacy. Atomicity hides the intermediate states of one critical section or transaction from another, but the order in which these sections are interleaved is still undetermined. A program can therefore still have several distinct outputs for the same input.

Message-passing models

Message-passing models consider the different computational activities of a program as active entities, each of which has its own, isolated memory. These components exchange data by sending each other messages. Message-passing models are thus well-suited for programs that consist of easily separable components that only exchange data sparingly.

Message-passing models are popular in distributed settings, in which they naturally map onto the hardware. However, they are also used on shared-memory systems such as multicore or multiprocessor machines (which we focus on here), when they naturally map onto the problem. In the former case, the developer decides to use a concurrency model that fits the hardware and maps the problem onto it, often leading to better performance but sometimes requiring an 'unnatural' design. In the latter case, the developer decides to use a concurrency model that fits the use case and relies on the system to map this onto the hardware, leading to a natural design but possibly suboptimal performance.

A distinction is made between models that use synchronous versus asynchronous message passing. When messages are passed **synchronously**, the sender and receiver run at the same time, and they wait until both are ready to pass the message, at which point they are said to *rendez-vous*. If a sender tries to send when no receiver is present, it will block until there is a corresponding receiver. On the other hand, when messages are sent **asynchronously**, the receiver has a buffer of messages, and messages can be sent even when the receiver is busy or inactive by storing them in the buffer. In this model, sending a message is immediate: the sender puts the message in the buffer and can immediately continue.

We give some examples of message-passing models:

- Communicating Sequential Processes (CSP) [Hoare 1978] is a synchronous message-passing model. Concurrent processes can be spawned and communicate by sending messages to each other.
- Concurrent ML [Reppy 1991, Reppy et al. 2009] is a synchronous message-passing model in which messages are sent over channels. Here, the sending and receiving of messages over a channel are first-class values called events, which can be composed using a set of operators.
- The actor model [Hewitt et al. 1973, Agha 1985] is an asynchronous message-passing model. Actors are entities with a private memory and an inbox. They communicate by placing messages in each other's inbox. Each actor can send to any other actor (of which it has the address), but can only read its own inbox.
- In Concurrent Object-Oriented Programming, an object can execute in a separate process; it is then called an active object [Karaorman and Bruno 1993]. Active objects have private memory, and communicate asynchronously using remote method invocation.
- MPI (Message Passing Interface) [Message Passing Interface Forum 1994, 2015] is a message-passing model widely used in industry, amongst others to implement scientific software. It has been implemented in libraries for languages such as Java, C, C++, Fortran, MATLAB, Python and others. MPI was originally designed for distributed memory architectures, but can also be used in shared-memory systems and hybrids (e.g. a distributed system of multicore machines). It provides constructs to handle different patterns common in message-passing programs, such as broadcasting, scattering, gathering, and reducing data. MPI allows both blocking (synchronous) and nonblocking (asynchronous) communication.

Why not always choose a deterministic model?

After studying these three categories, one might wonder why deterministic models are not always preferable, as they guarantee that the parallel program is correct regardless of the order in which threads are interleaved. Unfortunately, sometimes non-determinism is unavoidable. For instance, imagine a program that sends several messages to different servers, and would like to continue as soon as one is answered. Here, non-determinism is necessary: which message will be answered first depends on external factors. Van Roy and Haridi [2004], as well as Lee [2006] and Bocchino et al. [2009a], recommend using a deterministic model wherever possible, and to only introduce non-determinism in exactly those places where it is absolutely necessary. They recommend using the *least* expressive model – that is the *most* restrictive – whenever possible, as this is the model that provides the most guarantees to the programmer. They reason that the most restrictive model makes the code easier to understand and to check for correctness.

2.2 From Three Categories to Three Concurrency Models

Building on this taxonomy, we study one concurrency model from each category in this dissertation:

- As a deterministic model, we study *futures*. They are introduced in Section 2.3. Futures are one of the simplest deterministic concurrency models, as they only introduce two new constructs: forking a task and joining it later.
- As a shared-memory model, we look at *Software Transactional Memory* (STM), which is described in Section 2.4. While locks are traditionally used to protect shared memory, transactions avoid many of the common pitfalls of locks: they prevent the developer from forgetting to acquire a lock, they avoid deadlocks when locks are taken in the wrong order, and transactions can be nested without causing deadlocks [Harris et al. 2005].
- As a message-passing model, we study the *actor model*. Actors are introduced in Section 2.5. They are a popular concurrency model supported in languages such as Erlang, Scala, Java, and C++.[1] The actor model also has a long history in research [De Koster et al. 2016b].

In the following three sections, we describe and formalize these three models in detail. There exist many different variants and formalizations of each model in literature. We pick and study a simple variant of each, to reduce the model to its essentials. We also define a simple formal semantics of each model, which is loosely based on existing formalizations but defined in a uniform manner for the three models. This will allow us to focus on the combinations of these models in the following chapters.

All our implementations are modifications of Clojure, a Lisp-like language built on top of the Java Virtual Machine. Clojure already supports futures and transactions, and is therefore the perfect testbed for the problems tackled in this dissertation. We briefly describe Clojure in Appendix B.

Our formalizations are built around a *base language*: a standard, functional calculus. The syntax of this language is inspired by Clojure: it uses S-expressions and supports conditionals (`if`), local variables (`let`), and blocks (`do`). Also like Clojure, it is a functional language that does not contain assignments; once our three models are combined, any mutable memory must therefore be represented as transactional memory. We do not further formalize the base language in this text.

The formalisms introduced in the next sections build upon each other. This is visualized in Figure 2.2. L_b is the base language. In Section 2.3, we extend it to support futures, creating L_f. In Section 2.4, we define the language with transactions L_t. STM

[1] Erlang has built-in support, other languages use libraries (listed in Appendix C).

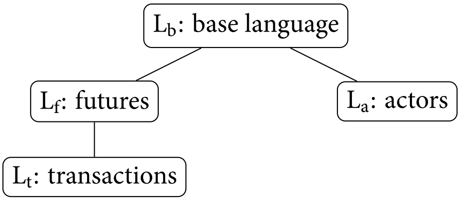


Figure 2.2: The relations between the languages presented in the following sections.

merely defines how transactions are created but not how the tasks are created in which transactions run: L_t must therefore rely on L_f. In Section 2.5, we again build upon the base language to create a language with actors L_a.

In all models we study, concurrency is explicit: the programmer must explicitly use certain language constructs to introduce parallelism or manage access to shared resources. We do not consider concurrency models which introduce concurrency implicitly, by for example automatically parallelizing expensive computations. This matches our language-based approach: we aim to study which (explicit) constructs of one model can be embedded inside the constructs of another model.

2.3 Futures

We start by describing futures: placeholder values that represent the result of a parallel computation (Section 2.3.1). Futures are semantically transparent and guarantee determinacy (Section 2.3.2). We show some examples (Section 2.3.3) and provide some details about the implementation (Section 2.3.4). Finally, we specify a formalization (Section 2.3.5).

2.3.1 Concepts

A parallel **task** is a fragment of the program that can be executed in parallel with the rest of the program. A run-time system schedules these tasks over the available processing units (detailed in Section 2.3.4).

A parallel task can be created using the expression `fork` *e*. This begins the evaluation of the expression *e* in a new task, and immediately returns a future. A **future** is a placeholder variable that represents the result of a concurrent computation [Baker and Hewitt 1977, Halstead 1985]. Initially, the future is **unresolved**. Once the parallel evaluation of *e* yields a value *v*, the future is said to be **resolved** to *v*. This result can be retrieved by other tasks by calling `join` *f*. If the future is resolved, `join` returns its value immediately; if the future is still unresolved, this call will block until it is resolved and then return its value.

```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fib (- n 1))
          b (fib (- n 2))]
      (+ a b))))
```

(a) Sequential recursive implementation of Fibonacci.

```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fork (fib (- n 1)))
          b (fork (fib (- n 2)))]
      (+ (join a) (join b)))))
```

(b) Recursive implementation of Fibonacci, parallelized using futures.

```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fork (fib (- n 1)))
          b (fib (- n 2))]
      (+ (join a) b))))
```

(c) In this implementation, only one recursive call is executed in a new task.

Listing 2.3: A sequential and two parallel implementations of Fibonacci. The sequential implementation is equivalent to the parallel ones with `fork` and `join` elided, hence, it is their serial elision.

Clojure implements futures as described here, except for some syntactical differences. Scala and Haskell also offer similar constructs.[2] We aim to approximate these languages closely, to demonstrate how the problems we will describe later also apply to them. A list of (minor) syntactical and semantical differences between our formal model and the implementations of Clojure and Haskell is given in Appendix D.

2.3.2 Guarantee: Determinacy

The **serial elision** of a program with futures is the program with `fork` *e* replaced by *e*, and `join` *f* replaced by *f*. In a functional programming language, where there are no side effects, a program with futures is equivalent to its serial elision. This property is called the **semantic transparency** of futures [Flanagan and Felleisen 1995]: futures are 'transparent' to the semantics of the program; when they are elided the semantics of the program remains unchanged. As a result, futures are a deterministic concurrency model: no matter in which order the tasks are scheduled, the program has the same result, which is equivalent to the result of its serial elision. This is called **determinacy** (or also *observable determinism*[3]): any execution of the program with the same input must lead to the same output.

[2] `fork` *e* is `(future e)` in Clojure, `Future { e }` in Scala, and `forkIO e` in Haskell.

[3] Some literature, such as Denning and Dennis [2010], distinguishes between *determinism*, which requires that the tasks are interleaved in the same way for each execution, and *determinacy*, which only requires the same output for a given input. We are concerned with determinacy, not determinism.

2.3.3 Examples and Use Cases

An example program using futures is shown in Listing 2.3b: an implementation of the Fibonacci function in which the recursive calls occur in futures. Listing 2.3a shows the program's serial elision.[4] For this program, determinacy is desired, as the result of this function should not depend on the order in which tasks are scheduled, and therefore futures are a good fit.

In general, futures are commonly used to the parallelize homogenous operations over lists, such as searching and sorting [Halstead 1985]. The operation is deterministic and can be applied to each element in parallel. For this purpose, Clojure provides a parallel map function: `(pmap f xs)` will apply `f` to each element of `xs` in parallel. If `f` has no side effects, `(pmap f xs)` is equivalent to `(map f xs)`. Futures are also used to increase the responsiveness of an application by executing long-running operations concurrently. For example, in graphical applications expensive computations or HTTP requests often return a future so as not to block the user interface.

Thanks to the semantic transparency of futures, developers can add futures to a program, simply by wrapping an expression in `fork`, in any place where they believe the parallel execution of the expression outweighs the cost of the creation of a task. Listing 2.3c shows another implementation of the Fibonacci function, in which only one of the recursive calls is parallelized. As only one task needs to be created, in many implementations this will result in better performance. Some 'smart' compilers even automatically insert or remove futures after analyzing the program, using heuristics to determine where they probably provide a benefit [Flanagan and Felleisen 1995, Harris and Singh 2007, Zhang et al. 2007, Surendran and Sarkar 2016].

2.3.4 Implementation Notes

Usually, a distinction is made between user threads and kernel threads [Herlihy and Shavit 2011, Section 16.3]. When a program spawns a parallel task using `fork`, this task will run in a *user thread* (sometimes also referred to as a *green thread*). The programming language provides a virtual machine, which creates a number of *kernel threads* and maps the user threads onto these kernel threads. Finally, the operating system's kernel schedules the kernel threads onto the hardware processors. This is illustrated in Figure 2.4.

This separation is made to hide the cost of kernel threads. Creating a kernel thread involves a kernel call and the reservation of a section of memory for the new thread, while creating a user thread is (often) no more costly than a function call, and (often) only allocates a small amount of memory. For instance, in Erlang, a newly spawned

[4] All our code examples use Clojure's syntax. If this example looks unfamiliar, a brief description of Clojure is given in Appendix B.

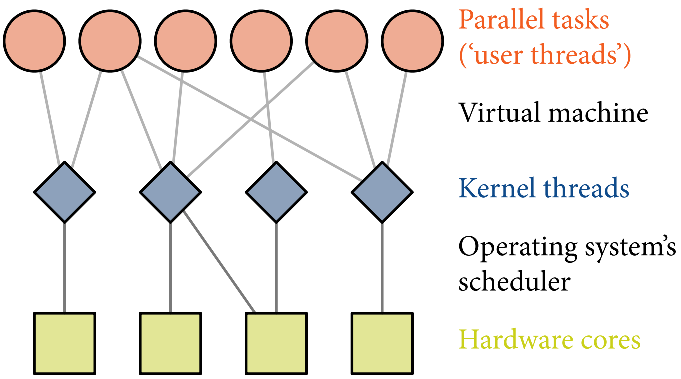


Figure 2.4: Kernel vs. user threads. The programmer creates parallel tasks, sometimes also referred to as user threads. The virtual machine schedules these onto kernel threads. The operating system's kernel in turn schedules these onto the hardware.

process only uses 309 words of memory (about 2.5 kB on a 64-bit architecture), while the Java HotSpot VM has a minimum stack size of 64 kB for a newly spawned kernel thread.[5] Hence, creating millions of kernel threads exceeds current machines' memory limits, while mapping millions of user threads upon a few kernel threads prevents this. Furthermore, context switches between user threads are more efficient as they do not require any interaction with the kernel [Tanenbaum and Bos 2014].

In this dissertation, parallel tasks are implemented as user threads. We will not further focus on how these are mapped onto kernel threads and hardware processors; these are 'details' of the implementation of the virtual machine and operating system.

2.3.5 $\mathrm{L_f}$: Formalization of Futures

Figure 2.5 defines $\mathrm{L_f}$: the base language extended to support parallel tasks and futures. This formalism is based on the work of Flanagan and Felleisen [1995] and Welc et al. [2005]. We will describe the operational semantics piece by piece.

The syntax consists of our base language – supporting conditionals (`if`), local variables (`let`), and blocks (`do`) – augmented with references to futures (f) and the expressions `fork` e and `join` e.

A program's state, p, is a tuple containing only one element:[6] a collection of tasks. A task is a tuple containing two elements: the future that represents its value and the expression it is currently evaluating. The future f associated with each task is unique,

[5] This varies depending on the specific virtual machine and version, operating system, and processor architecture. Sources: http://erlang.org/doc/efficiency_guide/processes.html and http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#threads_oom

[6] In the next sections, we will extend this tuple with more elements.

Syntax

$$
\begin{array}{llll}
c \in \text{Constant} & ::= & \texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{""} \mid \texttt{"a"} \mid \ldots & \\
x \in \text{Variable} & & & \\
f \in \text{Future} & & & \\
v \in \text{Value} & ::= & c & \\
 & \mid & x & \\
 & \mid & \texttt{fn}\ [\overline{x}]\ e & \text{Anonymous function} \\
 & \mid & f & \text{Future} \\
e \in \text{Expression} & ::= & v & \\
 & \mid & (e\ \overline{e}) & \text{Function application} \\
 & \mid & \texttt{if}\ e\ e\ e & \\
 & \mid & \texttt{let}\ [x\ e]\ e & \\
 & \mid & \texttt{do}\ \overline{e;}\ e & \\
 & \mid & \texttt{fork}\ e & \text{Fork a future} \\
 & \mid & \texttt{join}\ e & \text{Join a future}
\end{array}
$$

State

$$
\begin{array}{rlll}
\text{Program state} & \mathsf{p} & ::= & \langle \mathrm{T} \rangle \\
\text{Tasks} & \mathrm{T} \subset \mathsf{Task} & & \\
\text{Task} & \mathsf{task} \in \mathsf{Task} & ::= & \langle f,\ e \rangle
\end{array}
$$

Evaluation contexts

$$
\begin{array}{ll}
\mathcal{P} ::= & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E} \rangle \rangle \\
\mathcal{E} ::= & \square \mid (\overline{v}\ \mathcal{E}\ \overline{e}) \mid \texttt{if}\ \mathcal{E}\ e\ e \mid \texttt{let}\ [x\ \mathcal{E}]\ e \mid \texttt{do}\ \overline{v;}\ \mathcal{E}\ \overline{;e} \mid \texttt{join}\ \mathcal{E}
\end{array}
$$

Reduction rules

$$
\begin{array}{rlll}
\text{congruence}|_{\mathrm{f}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[e] \rangle \rangle & \rightarrow_{\mathrm{f}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[e'] \rangle \rangle & \text{if } e \rightarrow_{\mathrm{b}} e' \\
\text{fork}|_{\mathrm{f}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\texttt{fork}\ e] \rangle \rangle & \rightarrow_{\mathrm{f}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[f_\star] \rangle \mathbin{\dot\cup} \langle f_\star, e \rangle \rangle & \text{with } f_\star \text{ fresh} \\
\text{join}|_{\mathrm{f}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\texttt{join}\ f_\star] \rangle \mathbin{\dot\cup} \langle f_\star, v \rangle \rangle & \rightarrow_{\mathrm{f}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[v] \rangle \mathbin{\dot\cup} \langle f_\star, v \rangle \rangle &
\end{array}
$$

Figure 2.5: Operational semantics of $\mathrm{L_f}$, a language with futures.

and thus can be considered an identifier for the task. To kickstart evaluation, a program e is converted into the initial state $\langle \{ \langle f_0, e \rangle \} \rangle$, i.e. it contains one "root" task that executes e.

We use evaluation contexts to define the evaluation order within expressions. The program evaluation context $\mathcal{P}$ can choose an arbitrary task, and use the term evaluation context $\mathcal{E}$ to find the active site in the term. The fact that any task can be chosen in which a reduction is possible, models that a parallel execution of the program can interleave the execution of different tasks in any order. $\mathcal{E}$ is an expression with a "hole $\square$". We write $\mathcal{E}[e]$ for the expression obtained by replacing the hole $\square$ with e in $\mathcal{E}$. Note that there is no form for "`fork` $\mathcal{E}$": the expression in a `fork` is not evaluated in place, it will be executed in a new task.

We define the operational semantics using transitions $\mathsf{p} \to_{\mathrm{f}} \mathsf{p}'$. The subscript f denotes all reduction rules that apply to futures.

The rule congruence$|_{\mathrm{f}}$ defines that the base language can be used in each task. Transitions in the base language are written $e \to_{\mathrm{b}} e'$. They define a standard λ calculus, supporting the constructs defined in the syntax, but they are not detailed here.

The rule fork$|_{\mathrm{f}}$ specifies that the expression `fork` e creates a new task in which e will be evaluated, and reduces to a freshly created future $f_\star$. After the expression e has been fully reduced to a value v, `join` $f_\star$ will also reduce to v. A task can be joined multiple times; each join reduces to the same value. A join can only be resolved by the rule join$|_{\mathrm{f}}$ if the corresponding task has been fully reduced to a value; this detail encodes the blocking nature of our futures.

The semantic transparency of futures can be proven from these reduction rules [Flanagan and Felleisen 1995]. When a task is created, its expression is put in a new task and a placeholder $f_\star$ is returned. In the new task, the expression is reduced using the rules of the base language. As the base language is purely functional, there is no non-determinism: there is always only one rule that applies. Hence an expression always evaluates to the same value, regardless of the task it is executed in. When the task is joined, the future $f_\star$ is used to look up the value of the expression after reduction. This is equivalent to evaluating the future's expression in place.

2.4 | Transactions

In this section, we describe Software Transactional Memory, a concurrency model that coordinates access to shared memory using transactions. We start by describing its constructs and defining related terminology (Section 2.4.1), illustrated using an example (Section 2.4.2). We describe its two core guarantees: isolation and progress (Section 2.4.3). One implementation technique, multiversion concurrency control, is discussed in detail (Section 2.4.4). Finally, we specify a formal operational semantics of STM (Section 2.4.5).

2.4.1 Constructs and Terminology

Software Transactional Memory (STM) is a concurrency model that allows multiple parallel tasks to access shared memory locations [Herlihy and Moss 1993, Shavit and Touitou 1997]. It introduces transactions in the programming language, based on the concept of database transactions. A difference is made between Hardware Transactional Memory, which requires special support from the hardware, and Software Transactional Memory, which is implemented fully in software.

To use STM, each memory location that is shared by multiple parallel tasks is encapsulated in a **transactional variable**. Accesses to shared memory can only occur in a **transaction**: a block of code in which transactional variables can be read and modified. In a transaction, the developer has a consistent view of the shared memory: reading a transactional variable multiple times in the same transaction always yields the same result, even if another task modified it in the meantime. Furthermore, all changes made to shared memory in a transaction are made visible to other tasks atomically: it is not possible for other tasks to observe intermediate states.

In contrast to mechanisms based on locking, which are said to be 'pessimistic', STM is optimistic [Herlihy and Shavit 2011]. In *pessimistic* mechanisms, a task has to wait before entering a critical section to ensure it is the only task accessing the shared variables. In **optimistic** mechanisms, the code of the critical section is immediately executed, without taking locks. When leaving the critical section, the task needs to verify whether it was the only one accessing the variable(s), and if this is not the case, its changes are rolled back and the critical section is retried.

A transaction thus executes one or several **attempts**, which end in either a successful **commit** or an **abort**. Aborts are caused by **conflicts** between different transactions attempting to read and/or write to the same transactional variable(s). (Which types of conflicts can lead to aborts depends on the algorithm used to implement STM.) An aborted transaction is **retried**, which means that its changes are discarded or rolled back, and its contents are reexecuted in a new attempt.

The advantage of an optimistic approach is that it eliminates the cost of locking, and therefore often performs better when the chance of conflicts is low (hence the name 'optimistic'). A disadvantage of optimistic mechanisms is that transactions may be executed multiple times, and therefore should not contain any irrevocable operations, such as input/output or other operations with side effects.

A transactional variable is created using `ref` *v*, containing the initial value *v*. A transaction is a block `atomic` *e* that encapsulates an expression, which can contain reads (`deref` *r*, abbreviated to `@r`) and writes (`ref-set` *r v*) on the shared memory locations.

Software Transactional Memory is implemented in Clojure and in Haskell's GHC compiler [Harris et al. 2005]. Except for a few syntactic differences, the model described here is very similar to what is offered by Clojure and Haskell.[7] A list of differences between our model and those of Clojure and Haskell is given in Appendix D.

2.4.2 Examples and Use Cases

Transactions are used to allow safe access to shared memory in programs with multiple parallel tasks. These applications typically contain complex data structures of which pieces are encapsulated in transactional variables. For example, Atomic Quake [Zyulkyarov et al. 2009] implements the server of a multiplayer game, using transactions to concurrently process incoming requests from different players which each update a subset of the shared objects in the game. Another example is a parallel Sudoku solver [Perfumo et al. 2008], where each cell of the board is encapsulated in a transactional variable and multiple concurrent transactions solve the board. More examples include storing parts of rich text documents, HTML pages, and CAD models in transactional variables to operate on them in parallel [Guerraoui et al. 2007]; and networks, mazes, graphs, and lists which are manipulated by multiple tasks [Minh et al. 2008]. STM is particularly suited if it is difficult to predict statically which objects will be accessed in a critical section, for instance during graph traversals: (pessimistic) locks may require the developer to be overly conservative, locking any shared object that *might* be accessed in the critical section, while (optimistic) STM allows the developer to access objects freely and only synchronizes on those *actually* accessed [Harris et al. 2010].

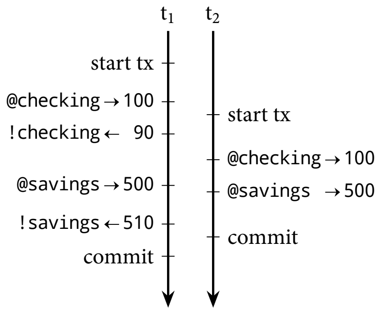
Listing 2.6 demonstrates the use of STM in a code snippet that manages bank accounts. There are two bank accounts: a checking and a savings account. They are created on lines 1 and 2. By wrapping their value in a transactional variable (`ref`), they can be accessed safely from multiple tasks using STM. Next, two tasks are forked that will execute in parallel (lines 3 and 7). The first starts a transaction that transfers €10 from the checking to the savings account. Hence, both variables are read (using `deref`) and written to (using `ref-set`). In the second task, a transaction is started to calculate the sum of the balances of both accounts.

STM guarantees serializability, meaning that the result of this program is equivalent to either serially running the first transaction followed by the second, or the second followed by the first. In both cases, the program prints `"You own €600"`. The second transaction cannot observe the intermediate state of the first transaction (its state between lines 5 and 6), so the program cannot print `"You own €590"`. Because it

[7] In Haskell transactional variables are called "TVars" and are created, read, and modified with `newTVar`, `readTVar`, and `writeTVar` respectively. A transaction is encapsulated in the construct `atomically`. In Clojure transactional variables are referred to as "refs" and manipulated using the same constructs that we use. Clojure encapsulates transactions in a `dosync` block.

```
(def checking (ref 100))
(def savings  (ref 500))
(fork
  (atomic  ; Transaction 1: transfer €10
    (ref-set checking (- (deref checking) 10))
    (ref-set savings  (+ (deref savings)  10))))
(fork
  (atomic  ; Transaction 2: sum accounts
    (println "You own €"
      (+ (deref checking) (deref savings)))))
```

(a) Code



(b) Timeline of a parallel execution, with read (@) and write (!) operations on transactional memory.

Listing 2.6: A program that transfers money between bank accounts, implemented using STM.

disallows these interleavings, STM prevents race conditions.

2.4.3 Guarantees: Isolation and Progress

We now discuss the two crucial guarantees of STM – isolation and progress – in more detail.

Isolation

Transactional systems provide a form of isolation between the transactions: one transaction can never see the changes of another until the latter has committed. Different isolation 'levels' have been defined, reflecting a range of trade-offs between satisfying the expectations of developers, integrating with existing programming languages, and improving the performance for transactional programs with specific characteristics. We discuss three isolation levels: serializability, opacity, and snapshot isolation.

The traditional guarantee of transactional systems (including databases) is **serializability**: transactions appear to execute serially, i.e. the steps of one transaction never appear to be interleaved with the steps of another [Herlihy and Moss 1993]. This only appears so to the developer, who cannot observe intermediate states of one transaction in another; in practice, the transactions actually are executed in parallel. Formally, serializability requires that the result of a transactional program, which may execute transactions concurrently, must always be equal to the result of *a* serial execution of the program, i.e. one in which no transactions execute concurrently.

The definition of serializability, originating in the world of database transactions, has been refined specifically for STM. Serializability only considers successful executions of a transaction. **Opacity** is a property that extends serializability to require that executions of a transaction that eventually abort also have a consistent view of the shared memory [Guerraoui and Kapałka 2008]. This ensures that no exceptions or other irrevocable side effects occur due to an inconsistent view in a transaction that later aborts. (This requirement was not necessary for database systems, where such actions cannot occur.) Opacity is stricter than serializability.

Some transactional systems provide a more relaxed form of isolation. **Snapshot isolation** requires that (1) a transaction sees a consistent view of the memory (this is its snapshot), and (2) a transaction can only commit if none of its updates conflict with any concurrent updates made since the snapshot [Berenson et al. 1995]. In contrast to opacity, snapshot isolation allows a transaction to commit even if it has seen an outdated version of some transactional variables, as long as these were only read. As snapshot isolation ignores these conflicts, it avoids retries in certain situations, increasing performance. However, it can lead to unserializable results, as we will demonstrate later in this chapter (Section 2.4.4).

Existing languages and libraries implement different isolation levels:

- Clojure's STM provides snapshot isolation. In Clojure, variables are immutable by default and the built-in collection types are implemented using persistent data structures [Driscoll et al. 1989], which cannot be destructively updated. Hence, creating a snapshot is cheap: it can simply keep references to the immutable values that the transactional variables had before the transaction started. When stricter isolation is desired, it is up to the developer to indicate which transactional variables must be checked for inconsistencies at commit time (using the explicit construct `(ensure r)`).
- ScalaSTM's reference implementation guarantees opacity [Bronson et al. 2010]. It assumes the developer does not access non-transactional memory in the transaction, but does not verify or enforce this. An important concern in the development of ScalaSTM was to implement it fully as a library.
- Haskell's STM implements an isolation level that is stricter than snapshot isolation, but weaker than opacity [Bieniusa and Thiemann 2011a]. This is due to its `orElse` construct: which branch is executed depends on the scheduling of the tasks. Haskell's STM without the use of `orElse` is opaque. Furthermore, Haskell harnesses its type system to statically ensure that transactional memory is only accessed in transactions and that no other side effects can occur in transactions.

Progress

In addition to isolation, transactional systems also guarantee progress. While traditional locking systems are prone to issues such as deadlocks, livelocks, and starvation, transactional systems aim to free the programmer from worrying about these issues. Similar to isolation levels, different STMs provide one of a range of different 'progress guarantees', each with its specific advantages and disadvantages. STMs can be divided into two categories in regard to progress: nonblocking and blocking (lock-based) algorithms [Harris et al. 2010, Herlihy and Shavit 2011].

The earliest STM implementations used nonblocking algorithms. Nonblocking algorithms require that, when one transaction is pre-empted, it should not prevent other transactions from being able to make progress. This requirement rules out the use of locks, because if the transaction holding the lock is pre-empted, no other transaction can acquire it. There are three common progress guarantees in this category: wait freedom, lock freedom, and obstruction freedom. Wait freedom is the strictest, requiring that a transaction should make progress on its own work as long as it is executing. Lock freedom is weaker, only requiring that when a transaction is executing, some transaction should make progress. Obstruction freedom requires that a transaction can make progress with its own work if other transactions do not run at the same time.

A disadvantage of nonblocking algorithms is that they are often slow (especially wait-free and lock-free algorithms). Ennals [2006] argues that nonblocking progress guarantees are not necessary for STM, as STM does not run on distributed systems, in contrast to the database systems for which these algorithms were originally designed. Abandoning nonblocking guarantees can result in much faster implementations.

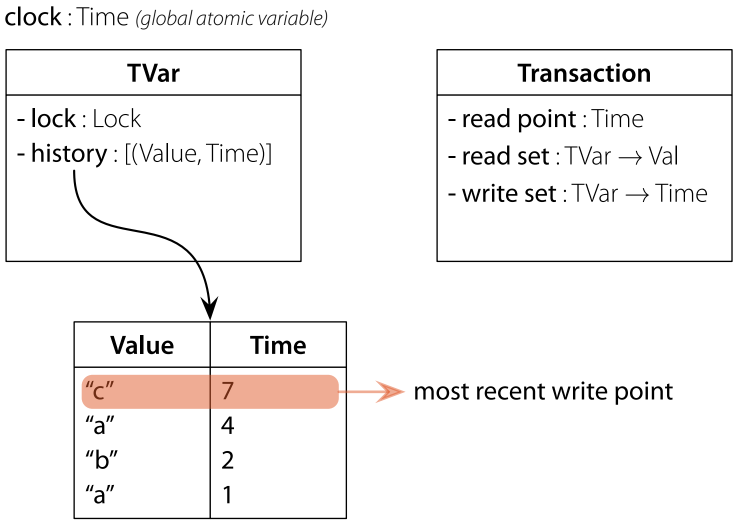
Hence, many blocking (i.e. lock-based) STM algorithms were developed. This category of STM algorithms guarantees **deadlock freedom**: when two transactions conflict, progress is guaranteed by a contention manager, a mechanism that decides which transaction(s) to delay so that another can always make progress. Clojure, Haskell, and ScalaSTM all use blocking STM algorithms. This is also the strategy followed by our implementation, presented in the next section.

2.4.4 Implementation: Multiversion Concurrency Control

There exist many different algorithms to implement STM. Clojure uses Multiversion Concurrency Control (MVCC) [Herlihy and Shavit 2011], a technique originating from databases [Bernstein and Goodman 1981]. In this section, we describe the implementation of STM using MVCC as it appears in Clojure. This implementation is similar to the one described by Herlihy and Shavit [2011] but adds a few optimizations. We choose to describe MVCC as we will build on this implementation in the rest of this dissertation.

Basic algorithm

The essence of Multiversion Concurrency Control is that it keeps multiple versions of the transactional variables. Each transactional variable contains a history, consisting



Value	Time
"c"	7
"a"	4
"b"	2
"a"	1

Figure 2.7: The most important components of an STM implementation.

of its previous values and the time at which that value was set. Each transactional variable also contains a lock. Additionally, MVCC uses a global clock: an atomic counter shared by all transactions. These components are illustrated in Figure 2.7.

When a transaction starts, it records the current clock value: this is its **read point**. To read a transactional variable, the system looks up the variable's most recent value before or at the read point. Thus, the transaction has a consistent view of the shared memory, which is the state that existed at the moment the transaction started. The set of all variables read in a transaction is called its **read set**, and may be cached to avoid the cost of looking up through the history. When a transactional variable is written to, its new value is not written to the variable yet. Instead, it is stored in the transaction's **write set**, a map containing all variables modified by the transaction along with their new value.[8] Hence, changes remain invisible to other threads while the transaction is still running.

When the transaction ends, it attempts to **commit** by taking the following steps:

1. All variables in the write set are locked.
2. To detect conflicts, for each variable in the write set, it is checked whether its most recent write point is not larger (i.e. more recent) than the transaction's read point. If any check fails, a conflict was detected and the transaction is aborted.
3. The global clock is atomically incremented (e.g. using a compare-and-set instruction). Its old value is the transaction's *write point*.

[8] Even though it is typically referred to as a write *set*, it is actually a *map*.

4. The new values and the write point are written to all variables in the write set.
5. Finally, all locks are released.

If a transaction aborts, it simply discards its read and write set and its read point, and it restarts.

Properties: snapshot isolation and deadlock-freedom

MVCC guarantees snapshot isolation. Each transaction's snapshot is captured by the read point of the transaction: all read operations return the value that existed at that point. Concurrent updates are detected at commit time and lead to an abort.

MVCC is a blocking algorithm as it takes locks. Deadlocks are prevented in step 1 of the commit protocol above, in one of two ways. One way is to define an order on all transactional variables, and acquire locks in this order. Another way is to take locks in no particular order, but to abort the transaction if a lock cannot be acquired in time.

Optimizations

Clojure modifies the algorithm in three ways.

First, the history of transactional variables is bounded: only a fixed number of old versions are kept for each transactional variable. (10 by default, but this can be changed per variable.) As a result, when reading a variable, it is possible no version can be found before the read point of the transaction. In that case, the transaction aborts and retries. Thus, this optimization limits memory usage but might increase the number of attempts per transaction.

Second, **barging** is an optimization that detects conflicts between transactions early. This means that a transaction that is doomed to abort will do so as soon as possible, reducing time spent on transactions that will abort anyway. It works as follows. When a transaction writes to a transactional variable, it already acquires the corresponding lock. If another transaction later attempts to write to the same variable, it cannot acquire the lock, and thus a conflict between the two transactions is detected. The oldest transaction is allowed to continue, while the newer one is aborted.[9] As a result, steps 1 and 2 in the commit protocol are no longer necessary: (1) the locks for all variables in the write set were already acquired during the transaction, and (2) checking for concurrent updates is no longer necessary as the locks ensure exclusive write access to the variables. This optimization maintains deadlock freedom as the oldest transaction is allowed to continue.

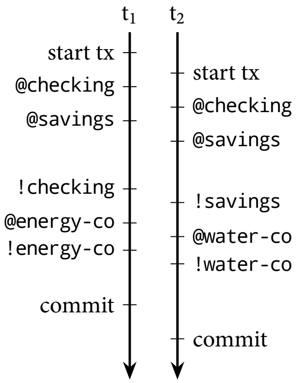
[9] The age of a transaction is based on the read point of its first attempt. Whenever a transaction acquires a read point, the global clock is incremented atomically, ensuring no two transactions have the same age.

```
(def checking  (ref 100))
(def savings   (ref 100))
(def energy-co (ref 0))
(def water-co  (ref 0))

(defn transfer-without-debt [from to amount]
  (atomic
    (if (> (+ @checking @savings) amount)
      (do (ref-set from (- @from amount))
          (ref-set to (+ @to amount)))
      (println "Insufficient funds"))))

(fork (transfer-without-debt
        checking energy-co 150)))
(fork (transfer-without-debt
        savings  water-co  150)))
```

(a) Code



(b) Timeline of a parallel execution

Listing 2.8: Example program that leads to a write skew anomaly, and read (@) and write (!) operations on transactional memory.

Third, Clojure counts how often a transaction restarts, and throws an exception if does so more than 10 000 times. This prevents livelocks.

Advantages and disadvantages

MVCC is well suited for read-heavy scenarios. First, it does not require locks for reading, making read operations much cheaper than write operations. This also means that when multiple transactions read the same variable, they can still execute in parallel, possibly reading different versions. Second, by keeping older versions, MVCC can avoid read–write conflicts. When a transaction reads a variable that has been modified since the transaction started, it does not need to abort, instead it will find an older version of the variable. In fact, if transactional variables have an unlimited history, a read operation will never lead to an abort, eliminating read–write conflicts.

A disadvantage of MVCC is that it requires additional memory to maintain multiple versions of each transactional variable. Limiting the histories attempts to find a balance between memory usage and the number of attempts per transaction.

Write skew anomalies

As MVCC implements snapshot isolation, it is vulnerable to write skew anomalies. This may sometimes lead to unexpected results. We illustrate this problem using an example.

Listing 2.8 shows a program that implements transfers between bank accounts. There are four bank accounts (lines 1–4): the user's checking and savings account, and two accounts belonging to the user's energy and water supply companies. On lines 13–16, the user pays two bills in parallel: one paying €150 from the checking account to the energy company, the other paying €150 from the savings account to the water company. The function `transfer-without-debt` transfers money from one of the user's account to a destination account. On line 8, it checks whether the user has sufficient funds: the bank allows accounts to have a negative balance, as long as a user's total funds remain positive.

Listing 2.8 also shows the timeline of a parallel execution of this program using MVCC. Task 2 starts its transaction immediately after task 1. Both transactions perform the check on line 8 at the same time, and confirm that the sum of both accounts (€100 + €100) is larger than the amount to transfer (€150). Next, they perform the transfer. Transaction 1 writes to the checking account and the energy company. Transaction 2 writes to the savings account and the water company. Transaction 1 reaches its commit phase first: it locks `checking` and `energy-co`, and verifies whether there are no conflicts on these variables. As there are none, their values can be updated. Next, transaction 2 commits. It locks `savings` and `water-co`, confirms there are no conflicts on these variables, and updates their values. Hence, both transactions have successfully committed, and the user's checking and savings account now both contain -€50, which was not allowed.

This inconsistency is a write skew anomaly. A **write skew anomaly** occurs when two concurrent transactions read from overlapping data sets but write to disjoint data sets, while there is a constraint over the data they read [Berenson et al. 1995]. Snapshot isolation allows the transactions to commit, as neither sees the updates of the other. In a serializable system, write skew anomalies are not possible: either t_1 or t_2 would have to occur first, and its result would be visible to the other transaction.

The possibility of write skew anomalies is a disadvantage of systems with snapshot isolation. Clojure requires the developer to use the construct `(ensure r)` to indicate that a transactional variable `r` is only read in the transaction, but has a constraint. To fix the example, the variables `checking` and `savings` must be ensured. By providing snapshot isolation instead of serializability, Clojure makes a trade-off between better performance but an arguably more intricate semantics for the developer.

2.4.5 L_t: Formalization of Transactions

This section formally defines L_t, a language with transactions. As STM does not provide any constructs that create parallel tasks, L_t will extend L_f, the language with futures from Section 2.3.5. The syntactical elements from L_f are reused and extended, and we define a new reduction relation $\rightarrow_t$. Afterwards, we consider how the guarantees of STM can be inferred from the operational semantics.

The formal semantics describes an algorithm that is simpler than MVCC, but provides the same guarantees. It guarantees snapshot isolation in a trivial way, by taking a complete snapshot of the transactional memory whenever a transaction is started. It therefore accurately represents an STM that provides snapshot isolation, while ignoring implementation details such as the versioning of transactional variables and taking locks.

Syntax

$r \in$ TVar			
$v \in$ Value	$::=$	$\ldots$	
	$\mid$	r	Transactional variable
$e \in$ Expression	$::=$	$\ldots$	
	$\mid$	`atomic` e	Transaction
	$\mid$	`atomic`$\star$ e	Running in transaction (intermediate state)
	$\mid$	`ref` e	Create a TVar
	$\mid$	`deref` e	Read a TVar
	$\mid$	`ref-set` e e	Write to a TVar

We inherit all the syntactical elements from L_f and add references r to transactional variables. The set of values is extended to include these.[10] The set of expressions is extended: `ref`, `deref`, and `ref-set` operate on transactional variables, and `atomic` encapsulates a transaction. Finally, we also add a construct `atomic`$\star$ e: this syntax cannot be used by the programmer in the source program, but will be used in the reduction rules to represent a transaction that is being executed.

State

Program state	p	$::=$	$\langle T, \tau, \sigma \rangle$
Task	$\mathsf{task} \in \mathsf{Task}$	$::=$	$\langle f, e, \mathsf{n}^{?} \rangle$
Transactions	$\tau : \mathsf{TransactionNumber} \rightharpoonup \mathsf{Transaction}$		
Heap, snapshot, local store	$\sigma, \overleftarrow{\sigma}, \delta : \mathrm{TVar} \rightharpoonup \mathrm{Value}$		
Transaction	$\mathsf{tx} \in \mathsf{Transaction}$	$::=$	$\langle \circ, \overleftarrow{\sigma}, \overleftarrow{e}, \delta \rangle$
Transaction id	$\mathsf{n} \in \mathsf{TransactionNumber} = \mathbb{N}^{+}$		
Transaction state	$\circ$	$::=$	$\triangleright \mid ✓ \mid ✗$

The state of a program in execution was previously modeled as a set of tasks T. This is now extended to contain the transactions and the transactional memory. The transactions are stored in τ, which maps identifiers to a representation of the transaction.

[10] The notation $v ::= \ldots \mid$ indicates we extend the definition of v from L_f.

The heap σ represents the transactional memory, mapping transactional variables to their value. The heap can be accessed by all tasks. Both τ and σ are initially empty.

Tasks are extended to contain an additional, optional element: n, the identifier of the transaction that is active in that task. In the practice this is implemented as a thread-local variable. If no transaction is running, this will be $\bullet$.[11] There is at most one transaction active per task. (We will consider nested transactions later.) Transactions are bound to exactly one task: they cannot span over multiple tasks.

A transaction contains the following elements:

- $\circ$: the current *state* of the transaction, one of $\triangleright$ (running), ✓ (committed), or ✗ (aborted).
- $\overleftarrow{\sigma}$: a *snapshot* of the heap created when the transaction started. This is a copy of the heap as it existed when the transaction started, used to get a consistent view of the transactional memory in the transaction.
- $\overleftarrow{e}$: the *original expression* in the transaction. This is used when the transaction is aborted, to restore and retry the original expression.
- δ: the *local store* of the transaction, mapping all transactional variables that the transaction modified to their new values. This is the transaction's write set.

Evaluation contexts

$$\begin{array}{rl} \mathcal{P} ::= & \langle \mathrm{T} \uplus \langle f, \mathcal{E}, \mathsf{n}^? \rangle, \tau, \sigma \rangle \\ \mathcal{E} ::= & \cdots \mid \texttt{atomic}\star\, \mathcal{E} \mid \texttt{ref}\ \mathcal{E} \mid \texttt{deref}\ \mathcal{E} \mid \texttt{ref-set}\ \mathcal{E}\ e \mid \texttt{ref-set}\ r\ \mathcal{E} \end{array}$$

`ref`, `deref`, and `ref-set` reduce their arguments before proceeding. `atomic` does not appear in the list of evaluation contexts, as its argument is not immediately reduced: a transaction is started first (rule $\mathrm{atomic}|_{\mathrm{t}}$ below). The expression encapsulated by `atomic`$\star$ will be evaluated, until it is reduced to a single value, after which the transaction can commit (see below).

Reduction rules

Finally, we can define the reduction relation $\rightarrow_{\mathrm{t}}$ for L_{t}.

$$\begin{array}{lll} \mathrm{congruence}|_{\mathrm{t}} & \langle \mathrm{T} \uplus \langle f, \mathcal{E}[e], \mathsf{n}^? \rangle, \tau, \sigma \rangle & \rightarrow_{\mathrm{t}} \langle \mathrm{T}' \uplus \langle f, \mathcal{E}[e'], \mathsf{n}^? \rangle, \tau, \sigma \rangle \\ & \text{if } \langle \mathrm{T} \uplus \langle f, \mathcal{E}[e] \rangle \rangle & \rightarrow_{\mathrm{f}} \langle \mathrm{T}' \uplus \langle f, \mathcal{E}[e'] \rangle \rangle \end{array}$$

This first rule specifies that all rules from the language with futures can be used in the language with transactions. This applies whether a transaction is active or not. This includes all rules from the base language, as well as the rules to fork and join new

[11] The question mark indicates an optional element, i.e. $\mathsf{n}^? ::= \mathsf{n} \mid \bullet$. (Defined in Appendix A.)

tasks (which may read or modify T). We will extensively discuss what happens when a task is forked while a transaction is active in Chapter 4.

$$
\begin{array}{rll}
\text{atomic}|_{\mathrm{t}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\mathtt{atomic}\ e], \bullet\rangle, \tau, \sigma\rangle & \\
 & \rightarrow_{\mathrm{t}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\mathtt{atomic}{\star}\ e], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \sigma, e, \varnothing\rangle], \sigma\rangle & \text{with n fresh} \\
\text{atomic}_{\mathrm{tx}}|_{\mathrm{t}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\mathtt{atomic}\ e], \mathsf{n}\rangle, \tau, \sigma\rangle & \\
 & \rightarrow_{\mathrm{t}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[e], \mathsf{n}\rangle, \tau, \sigma\rangle &
\end{array}
$$

When `atomic` e is encountered, a new transaction is started. A new identifier is created and stored in the current task, which will point to a new transaction in the map τ. This transaction is running, its snapshot is a copy of the current heap, it copies the original expression, and its local store starts empty.

`atomic` e is replaced by `atomic`$\star$ e. In the following transitions e will be reduced, as it is the active site of `atomic`$\star$'s evaluation context. Eventually, it will be reduced to a single value, after which one of the rules $\text{commit}_{\checkmark}|_{\mathrm{t}}$ or $\text{commit}_{\text{✗}}|_{\mathrm{t}}$ below can be applied.

The rule $\text{atomic}_{\mathrm{tx}}|_{\mathrm{t}}$ specifies that, if an `atomic` block is encountered when a transaction is already running ($\mathsf{n} \neq \bullet$), the inner transaction will simply become part of the outer transaction. This design decision corresponds to the "closed nesting" of transactions, which we discuss in Section 3.3.3 on page 64. Note that the child and parent transaction are running in the same task; the child transaction is merely a portion of the parent encapsulated in another `atomic` block. The writes of the child transaction are written to the local store of the parent and if the child aborts, the parent aborts too.

$$
\begin{array}{rll}
\text{ref}|_{\mathrm{t}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\mathtt{ref}\ v], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma\rangle & \\
 & \rightarrow_{\mathrm{t}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[r], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta[r \mapsto v]\rangle], \sigma\rangle & \text{with } r \text{ fresh} \\
\text{deref}|_{\mathrm{t}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\mathtt{deref}\ r], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma\rangle & \\
 & \rightarrow_{\mathrm{t}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[(\overleftarrow{\sigma} :: \delta)(r)], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma\rangle & \\
\text{ref-set}|_{\mathrm{t}} & \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[\mathtt{ref\text{-}set}\ r\ v], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma\rangle & \\
 & \rightarrow_{\mathrm{t}} \langle \mathrm{T} \mathbin{\dot\cup} \langle f, \mathcal{E}[v], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta[r \mapsto v]\rangle], \sigma\rangle &
\end{array}
$$

`ref`, `deref`, and `ref-set` operate on transactional variables. These operations can only be used in a transaction ($\mathsf{n} \neq \bullet$); there are no rules that apply when they are used outside a transaction, in practice this will raise an error.

`ref` creates a new transactional variable, and updates the local store to contain its initial value.

`deref` looks up the value of a transactional variable. This happens first in the local store, which contains the value if the variable was written to in this transaction, and next in the snapshot.[12] Since the snapshot is a copy of the heap made when the transaction started, this ensures a consistent view of the transactional memory as it existed at the read point of the transaction.

[12] The operator :: concatenates two maps and is right-preferential: $(\overleftarrow{\sigma} :: \delta)(r)$ first looks up r in δ, then in $\overleftarrow{\sigma}$ if it is not present in δ. (Its formal definition can be found in Appendix A.)

`ref-set` updates a variable's value in the local store.

$$
\begin{array}{rl}
\text{commit}_{\checkmark}|_{\mathrm{t}} & \langle T \mathbin{\dot{\cup}} \langle f, \mathcal{E}[\texttt{atomic}\star\ v], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma\rangle \\
& \rightarrow_{\mathrm{t}} \langle T \mathbin{\dot{\cup}} \langle f, \mathcal{E}[v], \bullet\rangle, \tau[\mathsf{n} \mapsto \langle \checkmark, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma :: \delta\rangle \\
& \text{if } \forall r \in \mathrm{dom}(\delta) : \sigma(r) = \overleftarrow{\sigma}(r) \\
\text{commit}_{\text{✗}}|_{\mathrm{t}} & \langle T \mathbin{\dot{\cup}} \langle f, \mathcal{E}[\texttt{atomic}\star\ v], \mathsf{n}\rangle, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma\rangle \\
& \rightarrow_{\mathrm{t}} \langle T \mathbin{\dot{\cup}} \langle f, \mathcal{E}[\texttt{atomic}\ \overleftarrow{e}], \bullet\rangle, \tau[\mathsf{n} \mapsto \langle \text{✗}, \overleftarrow{\sigma}, \overleftarrow{e}, \delta\rangle], \sigma\rangle \\
& \text{if } \exists r \in \mathrm{dom}(\delta) : \sigma(r) \neq \overleftarrow{\sigma}(r)
\end{array}
$$

A transaction can commit successfully if none of the transactional variables in its write set have been modified by another transaction since the start of this transaction. In other words, we need to validate whether, for all variables in the local store, the latest value in the current heap is still equal to the value in the snapshot that was taken of the heap when the transaction started.

If this validation succeeds, the transaction can commit: its changes are written to the transactional memory by appending its local store to the heap. This occurs in a single step, hence these changes cannot be interleaved with transitions from other tasks, ensuring that they atomically become visible to the other tasks. Finally, the transaction is marked as committed, and the transaction's identifier is removed from the task, indicating no transaction is active anymore.

If the validation fails, the transaction aborts, rolls back, and retries. The transaction is marked as failed, and the task is restored to the state it had before the transaction started: the original expression is restored and the transaction identifier is removed. In the next transition, the rule $\text{atomic}|_{\mathrm{t}}$ will be triggered and restart the transaction. Note that different attempts of the same `atomic` block have different identifiers, so in fact n identifies the transaction attempt and not the transaction.

Property: snapshot isolation

This semantics guarantees snapshot isolation, as it satisfies its two requirements (defined in Section 2.4.3):

1. A transaction sees a consistent view of the memory: every read operation in a transaction returns values from the snapshot, which is taken atomically when the transaction starts.
2. A transaction can only commit if none of its updates conflict with any concurrent updates made since the creation of the snapshot: this is the condition on the reduction rule for commits.

In this semantics, and throughout the rest of this dissertation, we choose to provide snapshot isolation instead of opacity. This is because our implementations are built upon Clojure, which provides snapshot isolation, and thus the semantics reflects the implementation.

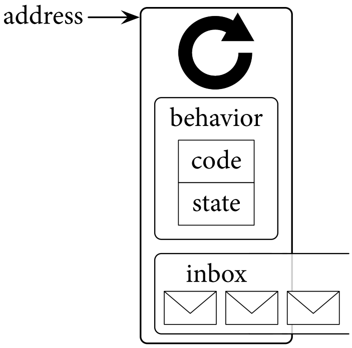


Figure 2.9: An actor and its parts.

To provide opacity instead of snapshot isolation, it suffices to adapt the reduction rules in two ways [Bieniusa and Thiemann 2011b]. First, transactions must keep track of their read set: the transaction is extended with a read set R, and each `deref` r operation adds r to R. Second, the validation at commit time checks whether references in both the read and write set have not been modified, i.e. $\forall r \in R \cup \mathrm{dom}(\delta) : \sigma(r) = \overleftarrow{\sigma}(r)$.

2.5 Actors

In this section, we discuss the actor model, a message-passing concurrency model. After describing the model (Section 2.5.1), we use an example to introduce its constructs (Section 2.5.2). The actor model provides two guarantees: the isolated turn principle and deadlock freedom (Section 2.5.3). We again formalize its operational semantics (Section 2.5.4).

2.5.1 Concepts

The actor model is a message-passing model that was originally introduced by Hewitt et al. [1973] and later revised by Agha [1985]. Actors are entities that run concurrently and can receive messages. In response to a message, an actor can send messages to other actors, create new actors, and change its own state. In the actor model, messages are sent asynchronously.

The actor model has a long history [De Koster et al. 2016b], and is widely used in many programming languages (a.o. Erlang, Scala, SALSA, E, AmbientTalk) and frameworks (a.o. Akka, Kilim, Pulsar, Quasar, Orleans).[13] De Koster et al. [2016b]

[13] An extensive list of actor languages, libraries, and frameworks can be found at https://en.wikipedia.org/w/index.php?title=Actor_model&oldid=819989553#Programming_with_Actors.

```
1 (def chat-room
2   (behavior [history]
3     [user msg]
4     (let [line (format "[%s] %s: %s" (now) user msg)]
5       (become chat-room (cons line history)))))
6
7 (def general (spawn chat-room ["Welcome to this chat room!"]))
8 (send general "Janwillem" "Hello!")
```

Listing 2.10: A chat application implemented using actors.

divide actor models into four categories: classic actors, active objects, processes, and communicating event loops. In this dissertation, we use a classic actor model, based on the model of Agha et al. [1997]. We will mention how our results apply to other actor models where applicable.

An actor consists of three elements: an address, an inbox, and a behavior (visualized in Figure 2.9). Each actor has a unique and immutable **address**, used to send it messages. Its **inbox** is a queue of messages. In our model, a message is simply a tuple of values. Finally, a **behavior** specifies how an actor responds to a message. It is parameterized by two types of parameters: the internal state of the actor and the values of the received message.

2.5.2 Example and Use Cases

The example in Listing 2.10 implements a chat application, in which each chat room is represented by an actor. Users, represented by actors as well, can send messages to the chat room actors to communicate. `chat-room` is a behavior. In its body (lines 4–5), two types of parameters can be used. The first type represent the internal state of the actor – here `history`, a list of all messages ever sent to the chat room. The second set of parameters correspond to the values contained in the received message – here `user` and `msg`.

On line 7, a new actor is created using `spawn`, with `chat-room` as initial behavior and a list containing a welcome message as the initial internal state. `spawn` returns the address of the new actor, and the new actor runs concurrently with the spawning actor.

On line 8, the construct `send` sends a message to this actor: it puts a message containing the values `"Janwillem"` and `"Hello!"` in the inbox of the actor with address `general`. When the receiving actor processes the message, it will execute the code in the behavior (lines 4–5), with `history` bound to the list of messages given when the actor was spawned, and `user` and `msg` bound to the message's values.

An actor can change its behavior and internal state using `become`. On line 5 in the example, `become` updates the `general` actor, keeping its behavior the same but updating its internal state to add the received message to its history.[14]

In general, the actor model is suitable in applications that consist of independent components that each work on their own data and communicate occasionally. This includes chat and communication applications, web services, HTTP servers, databases, event-driven systems (e.g. processing financial transactions), and simulations (e.g. of traffic or physical systems) [Tasharofi et al. 2013].

Actors have several qualities that make them suitable for these applications [Armstrong 2007]. They are scalable: as actors are cheap to create, many small actors can be created and distributed over the available cores, no matter the number of cores. Actors are also used for their fault tolerance: when one actor fails, the other actors do not crash. The failed actor can simply be restarted.

Actors are popular in distributed settings, where they naturally map onto the hardware. However, they are also used on shared-memory systems such as multicore processors, when they naturally map onto the problem (e.g. consider the use of actors by Phoenix, a framework for web applications on top of the Erlang VM[15]). In the context of this dissertation, we only consider programs that run on a single machine.

2.5.3 Guarantees: Isolated Turn Principle and Deadlock Freedom

An actor alternates between two states: it is either **idle** and ready to process a message, or it is **busy** processing a message. A **turn** is the processing of a single message by an actor, that is, the process of an actor taking a message from its inbox and processing that message to completion [De Koster et al. 2016b].

The actor model enforces three constraints:

Isolation An actor's state cannot be observed by other actors except through messages; there is no shared state.

Consecutive message processing An actor processes the messages in its inbox one by one. The processing of one message cannot be interleaved with the processing of another by the same actor. There is no parallelism inside an actor.

Continuous message processing The actor model does not contain nor allow any blocking operations. This guarantees that once a turn starts, it always runs to completion.

A consequence of these restrictions is that the actor model provides two useful guarantees. First, it guarantees that programs are free from races within turns: this is called

[14] In contrast to the actor model defined by Agha et al. [1997], in our system become does not immediately and concurrently process the next message with the new behavior, but instead switches the behavior of the current actor at the end of its current turn, sequentially. (As explained in the formal semantics later in this section.)

[15] https://phoenixframework.org/

the **isolated turn principle**. Thanks to actors' isolation and consecutive message processing, an actor has a consistent view of the program during a turn. Hence, developers do not need to care how individual instructions within a turn are interleaved with those from other actors. Instead, they can reason about their program at the level of turns. High-level races can still occur due to an unexpected interleaving of messages, in which case turns are interleaved in an unexpected way. Still, the isolated turn principle makes the program easier to understand, reason about, and debug, as it hugely reduces the number of interleavings developers have to consider.

Second, the actor model guarantees **deadlock freedom**: as there are no blocking operations, an actor can never deadlock. However, it is still possible for an actor to not make progress, e.g. when an actor is waiting for a message that never arrives.

2.5.4 $\mathrm{L_a}$: Formalization of Actors

This section defines the actor language $\mathrm{L_a}$ with a corresponding reduction relation $\rightarrow_{\mathrm{a}}$. This language again extends the base language. Afterwards, we consider how the isolated turn principle can be inferred from the operational semantics.

Syntax

$a \in$ Address			
$b \in$ BehaviorDef	::=	behavior $[\overline{x}_{\mathrm{beh}}]$ $[\overline{x}_{\mathrm{msg}}]$ e	Behavior definition
$v \in$ Value	::=	$\ldots$	
	|	a	Address
	|	b	Behavior
$e \in$ Expression	::=	$\ldots$	
	|	spawn $e\ \overline{e}$	Spawn an actor
	|	become $e\ \overline{e}$	Become a behavior
	|	send $e\ \overline{e}$	Send a message
	|	self	Address of current actor

We introduce two new syntactical elements: addresses and definitions of behaviors. An address is a unique reference to an actor. A behavior definition is similar to a function definition, but takes two lists of parameters. Addresses and behavior definitions can both be used as values. Further, we add four expressions: `spawn`, `become`, and `send` as discussed earlier, as well as the keyword `self`, which will always be the address of the current actor.

State

$$\begin{array}{rlll}
\text{Program state} & \mathsf{p} & ::= & \langle \mathrm{A}, \mu \rangle \\
\text{Actors} & \mathrm{A} \subset \mathsf{Actor} & & \\
\text{Inboxes} & \mu : \mathsf{Address} \rightharpoonup \overline{\mathsf{Message}} & & \\
\text{Actor} & \mathsf{act} \in \mathsf{Actor} & ::= & \langle a, e^{?}, \mathsf{beh} \rangle \\
\text{Behavior} & \mathsf{beh} \in \mathsf{Behavior} & ::= & \langle b, \overline{v} \rangle \\
\text{Message} & \mathsf{msg} \in \mathsf{Message} & ::= & \langle a_{\text{from}}, a_{\text{to}}, \overline{v} \rangle
\end{array}$$

The state of a program consists of a collection A of actors and a map μ of inboxes. μ maps each actor (using its address) to its inbox: a queue that is processed in the order in which messages are added to it.

An actor consists of three elements:

- Its unique address.
- The expression it is currently reducing; or • between turns, when the actor is idle.
- Its current behavior. A behavior actually consists of two parts: the behavior definition that specifies the code to execute and the values for its first list of parameters $\overline{x}_{\text{beh}}$. We call these values the **internal state** of the actor.

A message contains references to its sender and receiver and the list of values passed in the message.

Evaluation contexts

$$\begin{array}{rl}
\mathcal{P} ::= & \langle \mathrm{A} \uplus \langle a, \mathcal{E}, \mathsf{beh} \rangle, \mu \rangle \\
\mathcal{E} ::= & \cdots \mid \mathsf{spawn}\ \mathcal{E}\ \overline{e} \mid \mathsf{spawn}\ b\ \overline{v}\ \mathcal{E}\ \overline{e} \mid \mathsf{become}\ \mathcal{E}\ \overline{e} \mid \mathsf{become}\ b\ \overline{v}\ \mathcal{E}\ \overline{e} \\
 & \mid \mathsf{send}\ \mathcal{E}\ \overline{e} \mid \mathsf{send}\ a\ \overline{v}\ \mathcal{E}\ \overline{e}
\end{array}$$

The program evaluation context $\mathcal{P}$ allows an arbitrary actor to be chosen whose expression will be reduced next. The three expressions spawn, become, and send evaluate their arguments before their reduction.

Reduction rules

$$\begin{array}{rlll}
\text{congruence}|_{\mathrm{a}} & \langle \mathrm{A} \uplus \langle a, \mathcal{E}[e], \mathsf{beh} \rangle, \mu \rangle & \rightarrow_{\mathrm{a}} \langle \mathrm{A} \uplus \langle a, \mathcal{E}[e'], \mathsf{beh} \rangle, \mu \rangle & \text{if } e \rightarrow_{\mathrm{b}} e' \\
\text{self}|_{\mathrm{a}} & \langle \mathrm{A} \uplus \langle a, \mathcal{E}[\mathtt{self}], \mathsf{beh} \rangle, \mu \rangle & \rightarrow_{\mathrm{a}} \langle \mathrm{A} \uplus \langle a, \mathcal{E}[a], \mathsf{beh} \rangle, \mu \rangle &
\end{array}$$

congruence$|_{\mathrm{a}}$ specifies that the base language can be used in any actor. Additionally, we introduce the keyword `self`, which always resolves to the address of the actor it is evaluated in.

$$\begin{array}{rl}
\text{spawn}|_{\mathrm{a}} & \langle \mathrm{A} \uplus \langle a, \mathcal{E}[\mathsf{spawn}\ b_{\star}\ \overline{v}], \mathsf{beh} \rangle, \mu \rangle \\
 & \rightarrow_{\mathrm{a}} \langle A \uplus \langle a, \mathcal{E}[a_{\star}], \mathsf{beh} \rangle \uplus \langle a_{\star}, \bullet, \langle b_{\star}, \overline{v} \rangle \rangle, \mu[a_{\star} \mapsto []] \rangle \quad \text{with } a_{\star} \text{ fresh}
\end{array}$$

When a new actor is spawned, it is added to the collection of active actors. Its initial expression is set to $\bullet$, indicating that the actor is idle. Its behavior and internal state are initialized as specified in the call. It obtains a new address, which is returned as result of the call to `spawn`. Additionally, an empty inbox is created for the actor.

$$\begin{array}{rll}
\text{receive}|_{\text{a}} & \langle \mathrm{A} \uplus \langle a, \bullet, \mathsf{beh} \rangle, \mu[a \mapsto \langle a_{\text{from}}, a, \overline{v}_{\text{msg}} \rangle \cdot \overline{\mathsf{msg}}] \rangle & \\
 & \rightarrow_{\text{a}} \langle \mathrm{A} \uplus \langle a, e_\star, \mathsf{beh} \rangle, \mu[a \mapsto \overline{\mathsf{msg}}] \rangle & \\
 & \text{with } e_\star = \text{apply-behavior}(\mathsf{beh}, \overline{v}_{\text{msg}}) & \text{(as defined in footnote 16)} \\
\text{turn-end}|_{\text{a}} & \langle \mathrm{A} \uplus \langle a, v, \mathsf{beh} \rangle, \mu \rangle & \\
 & \rightarrow_{\text{a}} \langle \mathrm{A} \uplus \langle a, \bullet, \mathsf{beh} \rangle, \mu \rangle &
\end{array}$$

When an actor is idle and there is a message in its inbox, it can start a turn. The expression encapsulated in its current behavior (e) will be evaluated, with the first list of parameters $\overline{x}_{\text{beh}}$ bound to the values stored in the actor and the second list of parameters $\overline{x}_{\text{msg}}$ bound to the values passed in the message (this results in $e_\star$).[16] The message is removed from the inbox.

After the rule receive$|_{\text{a}}$ was triggered, the expression in the actor will be reduced. Eventually, this will result in a single value. At that point, the rule turn-end$|_{\text{a}}$ is triggered. This rule resets the actor to its idle state.[17] If there are more messages in the actor's inbox, another turn can start.

$$\begin{array}{rl}
\text{become}|_{\text{a}} & \langle \mathrm{A} \uplus \langle a, \mathcal{E}[\mathtt{become}\ b_\star\ \overline{v}], \mathsf{beh} \rangle, \mu \rangle \\
 & \rightarrow_{\text{a}} \langle \mathrm{A} \uplus \langle a, \mathcal{E}[\mathtt{nil}], \langle b_\star, \overline{v} \rangle \rangle, \mu \rangle \\
\text{send}|_{\text{a}} & \langle \mathrm{A} \uplus \langle a, \mathcal{E}[\mathtt{send}\ a_{\text{to}}\ \overline{v}], \mathsf{beh} \rangle, \mu[a_{\text{to}} \mapsto \overline{\mathsf{msg}}] \rangle \\
 & \rightarrow_{\text{a}} \langle \mathrm{A} \uplus \langle a, \mathcal{E}[\mathtt{nil}], \mathsf{beh} \rangle, \mu[a_{\text{to}} \mapsto \overline{\mathsf{msg}} \cdot \langle a, a_{\text{to}}, \overline{v} \rangle] \rangle
\end{array}$$

Calls to `become` update the behavior and internal state of the current actor. `send` adds a new message to the end of the inbox of the receiver.

Properties: isolated turn principle and deadlock freedom

As said, the isolated turn principle and deadlock freedom are a consequence of three restrictions imposed by the actor model, which can be inferred from the semantics:

Isolation When an actor is evaluating a turn, the only parameters it receives are its internal state ($\overline{x}_{\text{beh}}$) and the values passed in the message ($\overline{x}_{\text{msg}}$), in the rule receive$|_{\text{a}}$. These are substituted in the turn's body using the function apply-behavior, and thus never change throughout the reduction of the turn. The internal state is set when the actor is spawned, and can only be modified afterwards by the actor itself, using `become`. Hence, messages are the only way to transfer data between actors.

[16] apply-behavior$(\langle \mathtt{behavior}\ [\overline{x}_{\text{beh}}]\ [\overline{x}_{\text{msg}}]\ e, \overline{v}_{\text{beh}} \rangle, \overline{v}_{\text{msg}}) = \mathtt{let}\ \overline{[x_{\text{beh}}\ v_{\text{beh}}]}\ (\mathtt{let}\ \overline{[x_{\text{msg}}\ v_{\text{msg}}]}\ e)$

[17] The rule turn-end$|_{\text{a}}$ is not strictly necessary: we could say any actor that has reduced its expression to a single value is in an idle state, instead of bothering to replace this value with $\bullet$. However, extracting this step into a separate rule will prove handy in subsequent chapters.

Consecutive message processing An actor processes the messages in its inbox one by one. Turns of an actor are never interleaved: one turn needs to run to completion before the next can start (see rules receive$|_a$ and turn-end$|_a$).

Continuous message processing The semantics does not specify any blocking operations, hence, once a turn starts it always runs to completion without blocking.

2.6 Summary

Using a concurrency model, developers can exploit parallelism while enjoying certain guarantees or properties with respect to the program's semantics, which make it easier to understand, maintain, and debug. There are many concurrency models, and they can be partitioned into three categories: deterministic, shared-memory, and message-passing models. In this chapter, we described and formalized three concurrency models, one of each category: futures, transactions, and actors.

Table 2.11 summarizes the language constructs that are at the heart of these three concurrency models along with their guarantees. In the next chapter, we will look at combinations of these models. We make three observations that will prove relevant then:

- Each model provides one construct that contains an expression: `fork` for futures, `atomic` for transactions, and `behavior` for actors. We call these **concurrent constructs**: they contain an expression that will run concurrently with the rest of the program, in a new task, actor, or transaction. In the next chapter, when concurrency models are combined, it will be important to look at what can be embedded in these concurrent constructs.
- The operations on transactional variables (`ref`, `deref`, `ref-set`) are only allowed in a transaction. When used out of a transaction they will raise an error.
- The construct `behavior`, defining an actor's behavior, is a value; it cannot be further reduced. In contrast to all other constructs, it has no side effect.

Futures	**Transactions**	**Actors**
Deterministic	*Shared memory*	*Message passing*
`fork` e	`atomic` e	`behavior` $[\overline{x}]$ $[\overline{x}]$ e
`join` f	`ref` v	`spawn` $b\ \overline{v}$
	`deref` r	`send` $a\ \overline{v}$
	`ref-set` $r\ v$	`become` $b\ \overline{v}$
Determinacy	Isolation	Isolated turn principle
	Progress	Deadlock freedom

Table 2.11: Summary of the constructs and guarantees of each model.

3 Combining Concurrency Models

In this chapter, we look at combinations of concurrency models and the problems these cause. In Section 3.1, we explain the need to combine concurrency models and we confirm that this actually occurs in practice. Next, in Section 3.2, we use Clojure as a case study of a language in which combining concurrency models can lead to unexpected results. Finally, Section 3.3 defines the goal of this dissertation and describes how we will tackle these problems in the following chapters.

3.1 | Motivations for Combining Concurrency Models

We motivate why it is desirable to combine concurrency models, based on three observations.

■ Observation 1: Existing applications combine concurrency models.

An empirical study [Tasharofi et al. 2013] has shown that in a collection of 15 large, mature, and actively maintained Scala projects that use the actor model, 80% combine it with another concurrency model (illustrated in Figure 3.1).

8 of the 15 applications (53%) combine actors and futures. Here, a future is used to represent the 'return value' of an asynchronous message sent to an actor. This pattern is a common combination of actors and futures, supported by Scala [Nash and Waldron 2016, Chapter 4] but dating as far back as ABCL [Yonezawa et al. 1986].

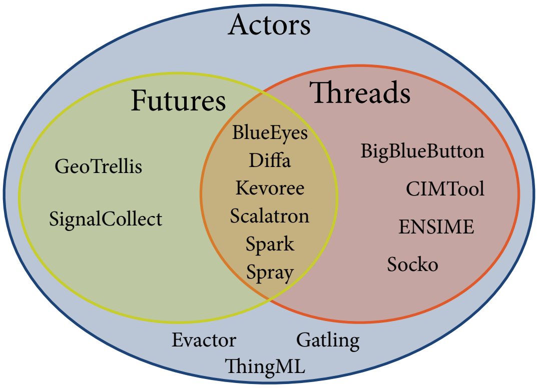


Figure 3.1: Out of 15 Scala projects that use the actor model, 80% combine it with futures and/or threads.

10 of the 15 applications (67%) combine actors with Scala's `Runnable`. A `Runnable` is an object containing a single `run` method, which is executed on a separate thread. The authors of the study contacted the developers of these programs, who gave several reasons for combining actors and `Runnable`. In five cases, developers found actors to have too much overhead – either when dealing with I/O (four cases) or when using low-end devices (one case) – and therefore they combined actors with a lower-level model: threads. In one case, developers found asynchronous message passing unsuitable to handle their use case, in which close coordination between concurrent tasks was required, and therefore they combined it with a shared-memory model: locks. Finally, in four cases the developers were inexperienced with actors, had legacy code that used `Runnable`, or preferred the 'traditional' method, and therefore ended up with a combination of concurrency models.

Further, we note that 6 out of the 15 applications (40%) use actors, futures, and threads, thus combining three concurrency models. This again confirms that developers choose to use different concurrency models throughout their application.

These results are corroborated by a survey on the use of concurrency among Microsoft employees [Godefroid and Nagappan 2008]. In this survey, around 45% of respondees indicated that they combine shared-memory and message-passing concurrency in their product.

Observation 2: Programming languages support multiple concurrency models and allow them to be combined.

Many programming languages and frameworks already support more than one model. A selection is shown in Table 3.2. An annotated version of this table can be found in

	Clojure	**Scala**	**Java**	**Haskell**	**C++**
Deterministic models					
Futures	✓	✓	✓	•	✓
Promises	✓	✓	✓	•	✓
Fork/Join	✓*	✓*	✓		•
Parallel collections	✓*	✓	✓	•	•
Dataflow	•	•	•	•	
Shared-memory models					
Threads	✓*	✓*	✓	✓	✓
Locks	✓*	✓*	✓	✓	✓
Atomic variables	✓	✓*	✓	✓	✓
Transactional memory	✓	•	•	✓	•
Message-passing models					
Actors	•	•	•	•	•
Channels	✓	✓	•	✓	•
Agents	✓				

Table 3.2: Concurrency models supported by selected programming languages and their libraries. ✓ indicates that support for the concurrency model is built into the language or its standard library; • indicates that an external library exists. Because Clojure and Scala are built on top of the Java Virtual Machine, they provide access to Java's concurrency models, this is indicated with ✓*. An annotated version of this table can be found in Appendix C.

Appendix C.

Clojure is the best example of a programming language with support for many concurrency models. It has constructs for no less than six concurrency models: futures, promises, atomic variables, transactional memory, channels, and agents. Moreover, as it is built on top of the JVM, it provides access to four more models: Fork/Join, parallel collections, threads, and locks. Scala similarly supports eight different concurrency models: four through its own constructs and four built on top of Java. The designers of these languages evidently consider it necessary to support a smorgasbord of concurrency models. Other programming languages also support multiple models: Java supports futures, promises, Fork/Join, parallel collections, threads, locks, and atomic variables; Haskell supports threads, locks, atomic variables, transactions, and channels; and C++ gained support for different concurrency models in C++11.

Moreover, we see that when languages support fewer models, for instance Haskell or C++, libraries have been developed to support many others. In that case, programmers decide they need to develop libraries to extend the language they use with support for additional models.

In all of these examples, the languages impose no restrictions on combinations of concurrency models and developers can freely mix multiple models in a single program. However, as we will see in the next section, these naive combinations can break the guarantees of the separate models, leading to bugs.

■ Observation 3: Complex applications consist of different parts that suit different concurrency models.

Even if many developers combine multiple models and programming languages support this, one might still wonder whether this is a good idea. We argue it is.

Originally, concurrency models were devised to each address a specific concurrency issue that occurs in a specific scenario. However, a typical application consists of many different parts, which may each benefit from concurrency. As each concurrency model is aimed at specific types of problems, different parts may need different concurrency models, and thus it is desirable to combine concurrency models.

For instance, the developers of an Integrated Development Environment (IDE) could choose to use Fork/Join to implement its search functionality, that is, a deterministic model to increase performance while obtaining the same result. They may implement its plug-in system using actors, as each plug-in runs simultaneously and independently, only needing occasional coordination. Finally, the code model, a shared and central data structure that must remain consistent, may be exposed using STM. Similarly, in a web browser each tab (an independent task) might be exposed as an actor, the Document Object Model (a shared data structure) stored in transactional memory, and its parser (a deterministic job that benefits from parallelism for performance) parallelized using futures.

■ Summary

Based on these three observations, we conclude that combining concurrency models is desirable. First, developers today actually combine multiple concurrency models. Second, many modern programming languages and libraries support many concurrency models and allow these to be combined (naively), demonstrating that the designers and users of these languages and libraries consider this useful. Third, combining concurrency models is reasonable, as large and complex applications often consist of different parts that are best expressed using different concurrency models.

3.2 | Motivating Case Study: Clojure

In Chapters 1 and 2 (Section 2.1), we explained that each concurrency model offers a number of guarantees that make it easier for developers to reason about their program and prevent bugs. In the previous section, we showed that several programming languages already support many different concurrency models and that these are often combined by developers. Now, we demonstrate that when concurrency models are naively combined, their individual guarantees sometimes no longer hold, defying the expectations of the developer.

We demonstrate this using Clojure as a case study. As shown in Table 3.2, Clojure is a language that supports many different concurrency models and developers are therefore especially likely to combine models in it. We limit our case study to the six concurrency models 'natively' supported by Clojure. While Clojure also provides access to four more models it inherits from Java, we do not consider these in our study, because Clojure either does not provide built-in constructs to access them (for Fork/Join and parallel collections) or its constructs are merely thin wrappers (for threads and locks).

We start this section by briefly describing these six concurrency models and their constructs (Section 3.2.1). Next, we define three specific bugs that can arise when they are combined: race conditions, deadlocks, and livelocks (Section 3.2.2). Finally, we examine which combinations of Clojure's concurrency models can give rise to these bugs and for which combinations they are prevented (Section 3.2.3).

These results were first described by Swalens et al. [2014].

3.2.1 Clojure's Concurrency Models and Their Constructs

We briefly describe the six concurrency models built into Clojure. Table 3.3 lists their constructs. Note that the implementation of these concurrency models in Clojure sometimes differs from their original specification or how they were described in Chapter 2. Here, we specifically describe Clojure's implementation.

■ Atomic variables

Atomic variables, or atoms, are variables that support concurrent access to shared memory, using a number of low-level atomic operations such as compare-and-swap. Compare-and-swap compares the value of an atomic variable with a given value and, only if they are the same, replaces it with a new value. Compare-and-swap is an atomic operation implemented by the hardware, so it can be used to safely modify a shared variable. Operations affecting multiple atomic variables are not coordinated, consequently when modifying two atomic variables race conditions can occur. Hence, atoms are typically used to share independent pieces of data that do not require coordinated updates.

In Clojure, atomic variables are created using `(atom v)`, where `v` is the atom's initial value. The value of the atom `a` can be read using `(deref a)`. `(reset! a v)` writes the value `v` to the atom. An atom is usually modified using `swap!`, e.g. `(swap! counter inc)`. `swap!` is a higher-order function that evaluates the given function (here `inc` to increment the counter[1]) with the current value of the atom, and attempts to write the return value to the atom. This write is protected using an atomic compare-and-swap instruction: in case another thread wrote to the atom in the mean time, the current thread will retry the `swap!` operation, by reevaluating the function `inc` with the newest value. As such, the code in the function passed to `swap!` can be executed multiple times and any side effects it contains may occur multiple times.

Agents

Clojure's agents implement a message-passing concurrency model. An agent consists of a user thread, a memory location owned by the agent (its internal state), and an inbox. `(agent v)` creates an agent with initial value `v`. `(send counter inc)` asynchronously puts a message `inc` in the inbox of agent `counter`. When the agent processes the message, on its own thread, it calls the function in the message (here `inc`) with the current state of its memory location, and the return value of this call is stored as the new state.

The state of an agent can be read using `(deref a)`, which immediately returns the current state. If the agent is processing a message at the time, the state is still unmodified; when the message is processed, the state is replaced in a single atomic step. Further, Clojure provides `(await a)` to block until all messages sent to the agent so far have been processed.

Agents are similar to actors, as both use asynchronous message passing. The difference is that an agent does not contain any behavior, only state. Instead, the message is the behavior, taking the current state as an argument and updating it.

Transactions (Software Transactional Memory – STM)

As explained in Section 2.4 (page 22), transactions allow concurrent tasks to safely access shared memory. Clojure's Software Transactional Memory is identical to the formal description we gave in Section 2.4 except that transactions are encapsulated in `dosync` instead of `atomic`. As explained before, transactions may retry multiple times.

Futures

Futures in Clojure have the same semantics as the model described in Section 2.3 (page 17). Only their syntax differs: in Clojure, a future is created using `(future e)` and read using `(deref f)`, which blocks until its result is available.

[1] `inc` is a built-in function that increments its argument, i.e. `(defn inc [i] (+ i 1))`.

	Atoms	Agents	STM	Futures	Promises	Channels
Create	`atom`	`agent`	`ref`	`future`	`promise`	`chan`
Read	`deref`	`deref`	`deref`	`deref` ⊗	`deref` ⊗	`<!` ⊗
Set	`reset!`		`ref-set`		`deliver`	`>!` ⊗
Update	`swap!` ⟲	`send`	`alter`			
Block			`dosync` ⟲			`go`
Other		`await` ⊗				`take!` `put!`

Table 3.3: Constructs supported by Clojure's concurrency models. ⊗ indicates a (potentially) blocking operation, ⟲ an operation that might be retried automatically.

Promises

A promise is a placeholder for a value that will be filled in later. A promise is similar to a future, but it is created independently from its task: its value is 'delivered' later using an explicit construct. In Clojure, a promise is created using `(promise)`, and delivered using `(deliver p v)`. `(deref p)` reads the promise, blocking until it has been delivered.

Channels (variant of Communicating Sequential Processes – CSP)

Clojure's core.async library implements a variant of the CSP model: a message-passing concurrency model in which concurrent tasks exchange messages over channels. A channel can have multiple readers and writers. Communication is synchronous: the sender and receiver wait until both are ready to pass the message (*rendez-vous*).

A new 'process' is started using a go block, and channels are created using `(chan)`. Inside a go block, the message `v` can be sent over channel `c` using `(>! c v)`, and a message can be received using `(<! c)`. These operations block until their complementary operation is executed on another thread. There are also nonblocking, asynchronous versions of these operations: `take!` and `put!`.

Summary

Table 3.3 lists the constructs described above. There are some striking similarities between all models:

- Each model encapsulates values in a 'container': an atom, agent, ref, future, promise, or channel.
- The value in these containers can be read using `deref`, for each model except channels, which requires `<!`. Moreover, Clojure provides the special syntax `@x` that expands to `(deref x)`, and works on all models except channels.

- All models provide constructs to write to the container, except futures, which are delivered implicitly. We distinguish constructs that directly set a value, e.g. `(reset! a 5)`, and those that update the current value using a function, e.g. `(swap! a inc)`. Unlike `deref`, these constructs have a different name for each model.
- Finally, transactions, channels, and futures provide a construct that contains a block of code, wrapping a transaction (`dosync`), concurrent process (`go`), or future (`future`). (`future` has a double function in both creating a future and wrapping the code it executes.)

In the table we also highlight constructs with potentially dangerous properties:

- **Blocking operations** (indicated with ⊗) wait until a result is made available by another thread. We will pay particular attention to these operations, as they potentially cause deadlocks when combined with another concurrency model.
- **Retrying operations** (indicated with ⟲) can reexecute code automatically. There are two such operations: `swap!` on atoms and `dosync` for transactions. We will also pay attention to these operations, as they may cause unexpected behavior or incorrect results when combined with certain operations of another concurrency model.

3.2.2 Typical Bugs in Concurrent Programs

We look at the three bugs that typically arise in concurrent programming: race conditions, which may cause incorrect results, and deadlocks and livelocks, which may prevent a program from terminating.

Race conditions

In concurrent programs, an important source of bugs are race conditions [Lu et al. 2008]. A race condition occurs when an incorrect result is reached due to an unexpected ordering of the instructions of multiple threads. Many concurrency models are designed to prevent (certain) race conditions. They can do this by managing shared resources; for instance, STM only allows shared memory to be accessed inside transactions, while the actor model only allows threads to share data through explicit message passing.

When two models are combined in a naive way, new races could be introduced unexpectedly. For example, some implementations of STM have been proven linearizable [Shavit and Touitou 1997] or opaque [Guerraoui and Kapałka 2008]. However, this assumes that all shared memory is managed by the STM system. This is not true if a thread communicates with other threads, e.g. when a transaction communicates over a channel, opacity can be broken. This can cause unexpected interleavings that eventually lead to race conditions. In the next section, we study which pairwise combinations of concurrency models can introduce race conditions.

Deadlocks

Another important source of bugs in concurrent programs are deadlocks [Lu et al. 2008]. A deadlock occurs when two or more threads are waiting for each other to take action, and thus neither ever does. Deadlocks are introduced by operations that block. Some concurrency models guarantee an absence of deadlocks, e.g. STM and actors (see Chapter 2). Others try to confine the problem by limiting blocking to a small set of operations. For example, futures only provide one blocking operation: reading a future waits until it is resolved. As long as all futures eventually resolve, no deadlocks will occur.

When concurrency models are combined without the necessary precautions, unexpected deadlocks might arise. For instance, a transaction that uses the blocking operations of another model, such as reading a channel, may suddenly become prone to deadlocks. Or, a future that contains blocking operations from another model may never be resolved, potentially leading to a deadlock. In the next section, we study which pairwise combinations of concurrency models can introduce deadlocks by looking at their blocking operations.

Livelocks

Livelocks appear when code is reexecuted under a certain condition, and another thread causes this condition to be always true. Some concurrency models, such as some STMs, guarantee the absence of livelocks [Harris et al. 2010, Herlihy and Shavit 2011].

Again, when concurrency models are naively combined, unexpected livelocks may arise. The `swap!` construct on an atom restarts until its write operation no longer conflicts, but in combination with another concurrency model a conflict may always occur. We will study which pairwise combinations of concurrency models can introduce livelocks by looking at their retrying operations.

3.2.3 Integration Bugs in Clojure

In this section, for each pairwise combination of models, we study whether race conditions, deadlocks, or livelocks can arise from the interactions between the two models.

Note that a concurrency model can already be prone to these bugs when used separately. For instance, it is possible to construct a program using promises that leads to a deadlock, or a program in which two atomic variables are accessed in a way that causes a race condition. We consider this behavior expected by the programmer: the

(a) Race conditions

→ used in ↓	Atoms	Agents	STM	Futures	Promises	Channels	
Atom's swap!	✗	✗	✗	✗	✗	✗	← ①
Agent's action	✓	✓	✓	✓	✓	✓	
STM's dosync	✗	✓	✓	✗	✗	✗	← ②
Future	✓	✓	✓	✓	✓	✓	
CSP's go	✓	✓	✓	✓	✓	✓	

(b) Deadlocks

→ used in ↓	Atoms	Agents	STM	Futures	Promises	Channels
Atom's swap!	✓	✓	✓	✓	✓	✗
Agent's action	✓	✓	✓	✓	✗ ④	✗
STM's dosync	✓	✓	✓	✓	✓	✗
Future	✓	✗	✓	✗ ⑤	✓	✗
CSP's go	✓	✗	✓	✓	✓	✗
		↑ ⑥				↑ ③

(c) Livelocks

→ used in ↓	Atoms	Agents	STM	Futures	Promises	Channels
Atom's swap!	✗ ⑧	✓	✓	✓	✓	✓
Agent's action	✓	✓	✓	✓	✓	✓
STM's dosync	✓	✓	✓ ⑦	✓	✓	✓
Future	✓	✓	✓	✓	✓	✓
CSP's go	✓	✓	✓	✓	✓	✓

Table 3.4: These tables show when race conditions, deadlocks, or livelocks can arise by combining two models in Clojure. The model in the column is *used in* the model in the row. Combinations that potentially give rise to bugs are indicated using ✗. For the combinations highlighted in green, Clojure takes into account the combination and prevents a potential bug, for those highlighted in orange Clojure throws an exception to make the developer aware of the problem.

concurrency model works as specified. We look specifically for new bugs that arise due to a combination of two concurrency models, as we consider this unexpected behavior: a guarantee offered by the separate models no longer holds.

We examined the pairwise combinations of Clojure's concurrency models, by lexically embedding each of the concurrency models in each of the models – including itself – and examining whether the bugs arise. The complete set of results is shown in Tables 3.4a, 3.4b, and 3.4c, corresponding to the three types of bugs – race conditions, deadlocks, and livelocks. For instance, the second column of Table 3.4a indicates whether a race condition can occur (✓) or not (✗) when a `send` to an agent is embedded in:

- an atom's `swap!` block (as in the code example shown in Listing 3.5a on page 57),
- another agent action,
- an STM transaction (as in Listing 3.5b on page 57),
- a future, and
- a `go` block.

Below we discuss the cases where a bug can occur. These results are indicated with circled numbers (① to ⑧) in the tables. A detailed enumeration of all cases, including those where no bugs occur, is given in the appendix of Swalens et al. [2014].

When a combination of two models can lead to one of these bugs, we say that they are not composable. However, the absence of all three bugs does not guarantee that they *are* composable. Concurrency models may provide other guarantees besides the absence of race conditions, deadlocks, and livelocks. For example, it is possible to construct a program that combines transactions with atoms, that does not lead to a race condition, deadlock, or livelock, but still breaks the serializability expected of STM.

Race conditions (Table 3.4a)

We first focus on the possibility of race conditions, which are caused by an incorrect interleaving between two models.

When using any concurrency model in the function given to an atom's `swap!` operation, race conditions are possible, because the function may be reexecuted ①. For example, when this function sends a message to an agent, it could be sent twice (Listing 3.5a). Moreover, because operations on *multiple* atoms are not coordinated, even combining two atoms is unsafe.

For transactions ②, actions inside a `dosync` block are reexecuted if the transaction is retried, and therefore the general rule is that "I/O and other activities with side effects should be avoided in transactions"[2]. However, there are two safe combinations.

[2] As suggested in Clojure's documentation at https://clojure.org/reference/refs.

First, when a message is sent to an agent inside a `dosync` block (Listing 3.5b), Clojure does not send this message immediately. Instead, it delays the send until the transaction is successfully committed. Second, when one `dosync` block is used in another, the inner transaction is merged into the outer one (this is called closed nesting [Moss 1981, Moss and Hosking 2006]), and as a result transactions are combined safely. In both of these cases, Clojure anticipates the combination of these concurrency models and ensures their semantics remain the same.

Based on these results we conclude that *if the outer model provides a retrying operation, and the inner model can perform irrevocable actions, unexpected interleavings can happen and therefore safety is not guaranteed.* We call these **spurious retries**. Clojure prevents this bug in two specific cases.

Deadlocks (Table 3.4b)

Next, we look at deadlocks, which are introduced by blocking operations.

Communication over channels is blocking by default, and therefore deadlocks are possible when this is done in another model ③. This is particularly problematic when embedded in `swap!` (atoms) or `dosync` (transactions): synchronous communication is irrevocable and should not be reexecuted.

Reading a future or a promise is an operation that blocks until its value becomes available. This can cause a deadlock when a promise is embedded in an agent ④. A code example is shown in Listing 3.6a: when one thread sends an action to an agent that reads a promise, and another sends an action that delivers this promise, the messages can be scheduled in an order that leads to a deadlock. Reading futures inside another future can also cause a deadlock when mutually dependent futures are allowed, as is the case in Clojure ⑤ (Listing 3.6b).

Finally, agents support one blocking operation, `await`, which waits until the agent has finished processing all its messages. In an atom's `swap!` block, this does not pose a problem: the `await` operation will always proceed eventually. However, in a `go` block or a future ⑥, this can cause a deadlock, when those models' blocking operations (`<!`, `>!`, or `deref`) are combined with `await`. In agent actions and transactions this situation is prevented: Clojure raises an exception to indicate this behavior is unsafe.

We conclude that *if the inner model provides blocking operations, a deadlock is possible.* We refer to this as **unexpected blocking**. In two cases, Clojure makes the developer aware of the potential bug by throwing an exception, but does not attempt to solve it.

Listing 3.5: Sending a message to an agent (in the context of a mail client), in a retrying operation (swap! and dosync):

```
(def unread-mails (atom 0))
(def notifications (agent ’()))
(swap! unread-mails
  (fn [n]
    (send notifications (fn [msgs] (cons ″New mail!″ msgs)))
    (inc n)))
```

(a) send to an agent in an atom’s swap!: send may happen more than once.

```
(def mail (ref {:subject ″Hi″ :archived false}))
(def notifications (agent ’()))
(dosync
  (ref-set mail (assoc @mail :archived true))
  (send notifications
    (fn [msgs] (cons (str ″Archived mail ″ (:subject @mail)) msgs))))
```

(b) send to an agent in a transaction (dosync): send is delayed until the transaction commits.

Listing 3.6: Deadlocks when reading a promise or a future:

```
(def result (promise))
(def an-agent (agent 0))
;Thread 1:
(send an-agent (fn [_] (deref result)))
;Thread 2:
(send an-agent (fn [_] (deliver result 1)))
```

(a) Using a promise in an agent can lead to a deadlock, depending on the order in which messages are sent.

```
(declare f2)
(def f1 (future … (deref f2) …))
(def f2 (future … (deref f1) …))
```

(b) Mutually dependent futures lead to deadlocks.

Listing 3.7: Livelocks when modifying one atom in another:

```
(def counter (atom 0))
(swap! counter
  (fn [n]
    (swap! counter inc)
    (inc n)))
```

(a) Modifying an atom inside the *same* atom always leads to a livelock: the outer swap! retries forever because it conflicts with the inner swap!.

```
(def counter1 (atom 0))
(def counter2 (atom 0))
;Thread 1
(swap! counter1
  (fn [n] (swap! counter2 inc)
          (inc n))))
;Thread 2
(swap! counter2
  (fn [n] (swap! counter1 inc)
          (inc n))))
```

(b) When two threads modify two different atoms, the swap! blocks may be retried, until eventually they happen to not overlap and they succeed. No livelock occurs.

Livelocks (Table 3.4c)

Finally, we look at livelocks, which appear when code is reexecuted. A transaction is retried when it conflicts with another one, causing a livelock if the conflict would consistently occur. However, Clojure's STM prevents such a livelock ⑦: if a transaction retries too often (fixed at 10,000 attempts), an exception will be thrown [Emerick et al. 2012]. When `swap!` is called on an atom inside a `swap!` on the *same* atom, a livelock will occur ⑧ (Listing 3.7). This is not prevented by Clojure.

In general, *a livelock can appear when a model that provides retrying operations is combined with a model that causes this reexecution to continually happen*. We call these **perpetual retries**. In one case, this can cause a bug; in another, Clojure will detect this and throw an exception.

Conclusion

From this case study of Clojure, we distinguish three typical problems that arise when concurrency models are combined naively: spurious retries, unexpected blocking, and perpetual retries. These problems manifest themselves as race conditions, deadlocks, and livelocks, and – in the studied cases – occur when the concurrency models provide constructs that retry or block.

Some of these potential bugs are anticipated by Clojure. They are handled in two ways. In some cases, Clojure prevents the bug by maintaining the semantics of the concurrency models as if they were used separately. In other cases, Clojure does not prevent the bug, but instead notifies the developer of the problem through an exception (hence, it only surfaces at run time, and may not do so every time).

All anticipated bugs occur when agents and STM are combined with each other or with themselves. Both agents and STM were present in Clojure since version 1.0 and their integration was well considered.[3] Futures, promises, and channels were introduced later. Their integration with existing models and each other seems more haphazard, leading to unhandled problems.

In general, we see that when several concurrency models are adopted by a language in a naive way (either as language features or as a library), their combinations can introduce several new bugs that break the guarantees of the individual models. It therefore becomes necessary to study how they interact when their operations are nested, to ensure the guarantees of the individual models are preserved. When more than two models are introduced, this must be done for each combination. This is not dissimilar to the problem of 'feature interaction' [Calder et al. 2003], where several

[3] As evidenced by talks by Hickey [2012] and the documentation at https://clojure.org/about/concurrent_programming.

features of a system that each function correctly separately might behave incorrectly when they are combined.

3.3 Combining Futures, Transactions, and Actors

In this section, we define the goals of this dissertation (Section 3.3.1) and describe the problems we must tackle to achieve these (Section 3.3.2). We also look at nesting each model in itself, which we refer to as the 'trivial' combinations (Section 3.3.3).

3.3.1 Goal and Requirements

In this dissertation, we will develop a unified model of futures, transactions, and actors – three concurrency models, each from a different category. The goal of this dissertation is to find a suitable semantics for the unified model, even when concurrency models are combined. We define two requirements:

1. First, **the semantics of the separate models should remain unchanged**. Existing programs should work unchanged in our unified framework so that developers can harness their existing expertise of the separate concurrency models. Therefore, when a program uses only the constructs of a single model, its semantics should be unchanged.
2. Second, to meet the assumptions of developers, **the guarantees of the individual models should be maintained even when they are combined**. We will do this wherever possible, however, in some cases it is impossible to combine the guarantees of all models because they inherently conflict. For instance, when a non-deterministic model is used in a deterministic one, it is impossible to maintain determinism. In this case, we will need to relinquish one of the original guarantees and we will define a modified, less restrictive guarantee that is provided by our combination.

3.3.2 Approach

In this section, we take a first, brief look at how combining our three models affects their guarantees. In the following chapters, we consider each combination in more detail.

Using the operational semantics of the models as defined in the previous chapter, we examine the 9 (3×3) ways in which the three models can be nested. The resulting combinations are tabulated in Table 3.8. Each model has one concurrent construct that can contain nested expressions: `fork` for futures, `atomic` for transactions, and `behavior` for actors. The three rows correspond to these three concurrent constructs.

	Future	Transaction	Actor

Future	`(fork`
	`  (fork …)`
	`  (join …))`
	Nested futures
		`(fork`
		`  (atomic …))`
		Parallel transactions
			`(fork`
			`  (spawn …)`
			`  (send …)`
			`  (become …))`
			Communication in future

Transaction	`(atomic`
	`  (fork …)`
	`  (join …))`
	Parallelism in transaction
		`(atomic`
		`  (atomic …)`
		`  (ref …)`
		`  (deref …)`
		`  (ref-set …))`
		Nested transactions
			`(atomic`
			`  (spawn …)`
			`  (send …)`
			`  (become …))`
			Communication in transaction

Actor	`(behavior [] []`
	`  (fork …)`
	`  (join …))`
	Parallelism in actor
		`(behavior [] []`
		`  (atomic …))`
		Shared memory in actor
			`(behavior [] []`
			`  (spawn …)`
			`  (send …)`
			`  (become …))`
			Actors

Table 3.8: Pairwise combinations of futures, transactions, and actors.

Each cell in the table injects the constructs of the model of the column in the concurrent construct of the row.

We consider the *dynamic extent* of each construct: if one model is used in another at execution time, we say they are *(dynamically) nested*. Note that this does *not* require their constructs to be nested *lexically*. For instance, if a library function that uses futures is called in a transaction, the construct `fork` will not appear in the `atomic` block in the code (lexically), but at execution time a future will be created while a transaction is running (dynamically), so we say that they are nested and we will study this combination. In the rest of this text, whenever we say that two constructs are nested, we refer to this type of *dynamic* nesting.

A few constructs have been omitted from Table 3.8:

- STM explicitly only allows operations on transactional variables (`ref`, `deref`, and `ref-set`) in `atomic` blocks. Therefore, we do not nest these operations in futures or actors directly; they will always appear first in an `atomic` block, which in turn can be nested in another model (second column).
- The construct `behavior`, defining an actor's behavior, is a value; it cannot be further reduced. In contrast to all other constructs, it has no side effect, and therefore we do not need to examine how it can be nested in other models.

Note that there is a sort of 'anti-symmetry' in the table (indicated through the colors). The diagonal contains models nested in themselves. All other cells have an opposite across the diagonal, e.g. the top-right cell shows actors in futures while the bottom-left cell shows futures in actors.

For each pairwise combination, we check whether the guarantees of the two models are maintained in a naive combination. These results are shown in Table 3.9, and will be discussed throughout this dissertation. Each cell lists the guarantees of both models and indicates whether these are maintained in a naive combination. For instance, when a future is forked in a transaction (second row, first column), the determinacy of the futures is broken and the isolation of the transactions is no longer guaranteed.

We discuss the cells on the diagonal, in which each model is nested in itself, in the following section. These 'trivial' combinations all maintain the guarantees.

In the other cells, in which *different* models are combined, the guarantees are broken. These cases are discussed in the subsequent chapters: Chapter 4 discusses the combination of futures and transactions, Chapter 5 the combination of transactions and actors, and Section 6.1 of Chapter 6 the combination of futures and actors. Each of these discussions will first describe the problems that arise when the models are combined, and then offer a modified semantics that does maintain the guarantees of both models if possible.

→ in ↓	Future	Transaction	Actor
Future	Nested futures (Section 3.3.3) Det	Parallel transactions (Section 4.1) ~~Det~~ Iso Pro	Communication in future (Section 6.1) ~~Det~~ ~~ITP~~ DLF
Transaction	Parallelism in transaction (Sections 4.2–4.4) ~~Det~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

■ **Guarantees**

Futures:	Det	Determinacy		
Transactions:	Iso	Isolation	Pro	Progress
Actors:	ITP	Isolated Turn Principle	DLF	Deadlock Freedom

Table 3.9: Maintained (in green) and broken (in red) guarantees of combined models.

3.3.3 Trivial Combinations

In the next three sections, we briefly discuss the combinations on the diagonal of Table 3.9, in which each model is nested in itself. These combinations have been studied in existing literature and maintain the model's guarantees.

Nested futures

Forking one parallel task in another is common and expected in programs that use futures. The example of a Fibonacci function we used to introduce futures in Listing 2.3, repeated in Figure 3.10 below, actually demonstrates this: the Fibonacci function uses futures to execute its recursive calls, creating (up to) two new futures in each future. Nesting futures does not break the determinacy of the program: no matter where futures are introduced, the program remains equivalent to its serial elision.

Executions of programs with futures can be represented using a **spawn tree** [Blumofe et al. 1995, Lee and Palsberg 2010]. For instance, Figure 3.10 shows the Fibonacci program and its spawn tree when used to calculate fib(4). Each node corresponds to a task. The edges connect parent tasks to the tasks they forked. This graph will always form a tree: there is one root task that encapsulates the whole program and can spawn new tasks, and each task can in turn spawn more tasks.

```
(defn fib [n]
  (if (< n 2)
    n
    (let [a (fork (fib (- n 1)))
          b (fork (fib (- n 2)))]
      (+ (join a) (join b)))))
```

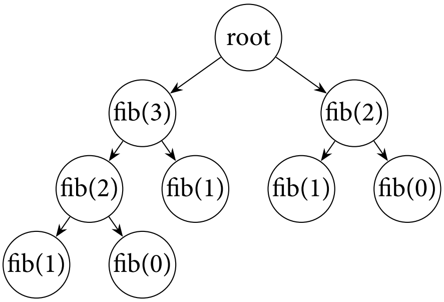


Figure 3.10: Spawn tree of the recursive Fibonacci implementation of a parallel Fibonacci program (Listing 2.3, repeated on the left), for $n = 4$, at the end of the program.

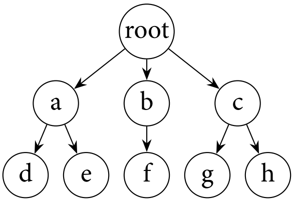
Furthermore, **nested futures can never lead to a deadlock**. This is the result of the tree structure of nested futures. It is illustrated in Listing 3.11. Each task is identified by a future, which is a reference that can be passed around and supports one operation: `join`. Each task can obtain the future of:

1. its direct children, as it created those;
2. any further descendants, if a child returns the future of a descendant (e.g. the root task can access `d` if `a` returns `d`);
3. its earlier siblings (e.g. `b` can access `a`); and

4. any descendants of its earlier siblings, if a sibling returns the future of one of its descendants (e.g. `f` can access `d` if `a` returns `d`).

Tasks cannot obtain their own future. Writing the spawn tree in post-order notation defines a strict total order on the tasks, in which each task can access the futures of the tasks that come before it. This is a consequence of the lexical scoping of the language. This means that there cannot be cyclical dependencies between futures, and therefore deadlocks are impossible.

```
(let [a (fork (let [d (fork …)
                    e (fork …)] …))
      b (fork (let [f (fork …)] …))
      c (fork (let [g (fork …)
                    h (fork …)] …))]
  …)
```



Listing 3.11: A program with nested futures and its spawn tree. Writing the spawn tree in post-order notation yields the sequence `d-e-a-f-b-g-h-c-root`. This shows that task `b` can join tasks `d`, `e`, `a`, and `f`. It may be less obvious that task `f` can also join task `d`: task `a` may return `d`, and `f` can access `a`.

Nested transactions

When a transaction is started while another transaction is already running, we say this is a **nested transaction**. It occurs in many cases: when a program with transactions calls into a library that also uses transactions, when a program consists of different components that each use transactions and call each other, or even simply when code is reused (e.g. when a set's replace operation is implemented as a transaction that contains a remove and add operation, each of which is also a transaction). The nesting of transactions is a well-studied problem.

Moss and Hosking [2006] distinguish two types of nesting: open and closed nesting. In both cases, a child transaction has access to the state of its parent. In **closed nesting**, when the child commits, its changes are *not* made globally visible yet, instead they become part of the parent. It is only when the parent commits that the changes become permanent. When the child aborts, the parent aborts too. In essence, in systems with closed nesting transactions are 'flattened' to a single level, and at run time there will only be a single transaction in execution.

On the other hand, in **open nesting**, when a child transaction commits, its changes immediately become visible to other transactions, even if its parent is still running. As a result, even when the parent aborts, the effects of the child transaction remain visible. Open nesting thus breaks isolation. To counter this, developers need to specify a *compensating action* for each child transaction, which is executed when it aborts.

Moreover, in some cases developers need to manually lock data structures to avoid breaking isolation. Open nesting is therefore aimed at expert developers [Ni et al. 2007], e.g. a possible scenario is that expert developers implement libraries of concurrent data structures that use nested transactions internally, and that these libraries can then be composed by the users of the library using (the top-level) transactions.

The advantage of open nesting is that it increases the concurrency, as transactions can be split into smaller parts, and therefore resources can be released earlier and the chance of conflicts decreases. Conversely, closed nesting leads to long transactions, making conflicts more likely and more costly.

In practice, closed nesting is the norm: Clojure, Haskell, and ScalaSTM all implement closed nesting. The complexity of open nesting does not seem to outweigh its performance benefits. The semantics we specified in Section 2.4.5 also implements closed nesting, in the rule $\text{atomic}_{\text{tx}}|_{\text{t}}$, which specifies that a nested transaction simply becomes part of its parent.

In this discussion, we assume that no parallel tasks are created in a transaction, as no `fork` can appear in the transaction, and therefore child transactions of the same parent cannot run in parallel. Like Haines et al. [1994], we distinguish between nested transactions, discussed here, and multithreaded transactions, discussed in Chapter 4 when futures and transactions are combined.

Nested actors

'Nesting' actors – creating one actor in another – is a standard part of the actor model. In fact, an actor program consists of only actors running concurrently, and therefore all actors except the initial one are nested actors. The guarantees of actors are maintained.

3.4 Conclusion and Roadmap

In this chapter, we first motivated why combining concurrency models is desirable: it happens in practice, it is allowed by programming languages, and it makes sense for large applications. Next, by studying Clojure, we showed that in existing languages naive combinations of concurrency models can lead to unexpected semantics and thus break the expectations of developers, as they no longer maintain the guarantees provided by their constituent parts.

In this dissertation, we will develop a unified model of futures, transactions, and actors – three concurrency models that belong to a different category. We explored the combinations of these three models, and found that, while nesting each model in itself works as expected, combining different models breaks their guarantees. In the rest of this text, we aim to find a semantics for these combinations that maintains the guarantees of the individual models wherever possible. When this is not possible, we instead define a less restrictive guarantee. When a program only uses constructs of one model, its semantics should remain unchanged.

The following chapters examine each combination of different models in detail:

- We first look at each pairwise combination separately: Chapter 4 examines the combination of transactions and futures; Chapter 5 focuses on transactions and actors; and in Chapter 6 we consider futures and actors. These chapters describe the semantics informally and using examples.
- In Chapter 6, we then join the three pairwise combinations into one framework, called Chocola (for "composable concurrency language").
- In Chapter 7, we formalize a precise semantics for the unified framework and show which guarantees it provides.
- In Chapter 8, we describe how we implement these combinations starting from three separate implementations of futures, transactions, and actors.
- In Chapter 9, we evaluate the performance of our unified framework by examining whether a combination of concurrency models allows parallelism to be more efficiently exploited. We do this by extending existing programs that currently use one model, combining it with another model.

4 Transactional Futures: Parallelism in Transactions

This chapter examines the combination of futures and transactions. These combinations are shown in Table 4.1. First, in Section 4.1, we examine the creation of transactions in a parallel task, which is standard in languages with transactions. The rest of the chapter looks at the opposite combination: creating futures in a transaction. In Section 4.2, we motivate the use of futures in a transaction using an example and show the problems that occur when this is done using a naive combination of both models. Section 4.3 introduces *transactional futures*: futures created in a transaction with a well-defined semantics. Afterwards, we discuss the guarantees and other properties of transactional futures in Section 4.4 and compare them with related work in Section 4.5.

We first introduced transactional futures in Swalens et al. [2016].

4.1 Transactions in Futures

We first focus on the use of transactions in parallel tasks. As shown in Table 4.1, this corresponds to the use of the construct `atomic` inside the dynamic extent of a `fork`.

As the transactional model does not provide any construct to create parallelism, this combination is a standard part of languages with transactions: any use of transactions requires the use of another model to create the tasks in which they run. This is the case in Clojure, where transactions run in futures, in Haskell, where they run in threads created using `forkIO`, and in Scala, where they run in `Threads`. The seman-

→in↓	Future	Transaction	Actor
Future	Nested futures (Section 3.3.3) Det	Parallel transactions (Section 4.1) ~~Det~~ Iso Pro	Communication in future (Section 6.1) ~~Det~~ ~~ITP~~ DLF
Transaction	Parallelism in transaction (Sections 4.2–4.4) ~~Det~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

▨ **Guarantees**

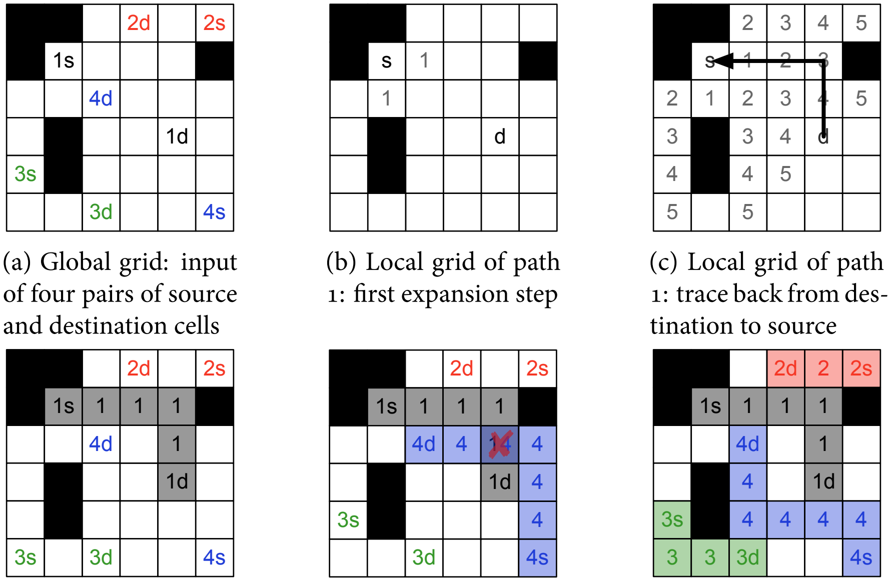
Futures:	Det	Determinacy		
Transactions:	Iso	Isolation	Pro	Progress

Table 4.1: Combinations of futures and transactions.

tics of this combination was already specified accordingly in Section 2.4.5 (page 31), where we define the language L_t with transactions as an extension of the language L_f with futures.

The question is whether this combination can preserve the guarantees of both of its constituents: the determinacy guarantee of futures and the isolation and progress guarantees of transactions. As shown in Section 2.4.5, this combination guarantees isolation and progress. However, the determinacy of futures is broken: a program with transactions is equivalent to *a* serialization of the transactions, but often there are many possible serializations. Determinacy allows only one possible output for a given input.

Breaking determinacy is unavoidable: as soon as a non-deterministic model such as transactional memory is introduced, there is no way to maintain determinacy of the whole program. As explained in Section 2.1 (specifically on page 15), it is generally recommended that developers use determinism wherever possible, and carefully introduce non-determinism only where it is inescapable [Van Roy and Haridi 2004, Lee 2006, Bocchino et al. 2009a]. We argue that the loss of determinacy thus does not pose a problem: when developers decide to use a non-deterministic model, they



(a) Global grid: input of four pairs of source and destination cells

(b) Local grid of path 1: first expansion step

(c) Local grid of path 1: trace back from destination to source

(d) Global grid updated with path 1

(e) Concurrent updates of paths 1 and 4 that conflict

(f) Final global grid with four non-overlapping paths

Figure 4.2: Different steps of the Labyrinth algorithm, illustrated using 4 paths on a two-dimensional 6 × 6 grid. The black squares are impenetrable 'walls'.

do so because they need its non-determinism, and therefore they are aware that determinacy is no longer guaranteed. Moreover, transactions still limit the number of possible outputs developers need to consider to the number of possible serializations of the transactions in the program. Further, the non-determinism is confined to the parts of the program that use transactions; code in which no transactions appear remains determinate.

4.2 Motivation for Futures Inside Transactions

This section discusses the use of parallelism in the dynamic extent of transactions, i.e. using `fork` inside `atomic`. Using an example, the Labyrinth application, we illustrate that it is desirable for certain applications to use parallelism inside their transactions. We discuss how this is realized in contemporary programming languages and demonstrate that those do not provide satisfying semantics.

Labyrinth application

As an example, we look at the Labyrinth application of the STAMP benchmark suite [Minh et al. 2008], a suite of applications that use transactional memory. The Labyrinth application implements a transactional version of Lee's algorithm [Lee 1961, Watson et al. 2007], an algorithm that is used in chip design to place electric components on a grid, connecting them without introducing overlapping wires.

Listing 4.3 shows the main part of the Labyrinth application, translated to Clojure. Its main data structure is `grid`, a two-dimensional array of transactional variables. The aim of the application is to find non-overlapping paths between given pairs of source and destination cells. For example, Figure 4.2a depicts four source–destination pairs on an 6 × 6 grid, and Figure 4.2f shows a possible solution of connections.

```
1  (def grid (initialize-grid w h))       ; w × h array of cells, each cell is a ref
2  (def work-queue (ref (parse-input)))  ; list of [source destination] pairs
3
4  (loop [] ; 1
5    (let [work (pop work-queue)] ; atomically take element from work queue
6      (if (nil? work)
7        true ; done
8        (do
9          (atomic
10           (let [local-grid (copy grid)
11                 [src dst] work ; destructure pair using pattern matching
12                 reachable? (expand src dst local-grid)] ; ref-sets on local-grid
13             (if reachable?
14               (let [path (traceback local-grid dst)]
15                 (add-path grid path)))))                  ; ref-sets on grid
16         (recur)))))
```

Listing 4.3: Transactional version of Lee's algorithm in Clojure.

The transactional variable `work-queue` is initialized to the input list of source–destination pairs. As long as the work queue is not empty, a source and destination pair in the input will be processed in a new transaction (lines 9–15). This happens in four steps:

1. First, copy create a local copy `local-grid` of the shared `grid` (line 10).
2. Next, a breadth-first search expands from the source cell (line 12), recording the distance back to the source in each visited cell of the `local-grid` using `ref-set` (Figure 4.2b), until the destination cell is reached.
3. Afterwards, the `traceback` function finds a path from the destination back to the source (line 14; Figure 4.2c).
4. Finally, the function `add-path` updates the shared `grid` to indicate that the cells on the found path are now occupied (line 15; Figure 4.2d).

After the transaction has finished, this process is repeated until all work has been processed.[1]

To parallelize this algorithm, we have to make several "worker threads" that execute the loop simultaneously. Each thread repeatedly takes a source–destination pair from the work queue and attempts to find a connecting path in a transaction. If two threads result in overlapping paths, a conflict occurs when updating the global `grid` (on line 15), as the two threads attempt to write to the same transactional variable (that represents the cell where the paths collide; Figure 4.2e). As a result, one of the two transactions is rolled back and will look for an alternative path.

Performance of Labyrinth

Minh et al. [2008] measure various metrics of the applications in the STAMP benchmark suite, shown in Table 4.4. We compare Labyrinth with the other applications. First we observe that this application spends 100% of its execution time in transactions. Hence, the amount of parallelism in this program is maximally the number of transactions that are created, which is the number of input source–destination pairs. On a machine with more cores, the hardware will not be fully utilized. To exploit more fine-grained parallelism in this program, it is necessary to allow parallelism inside the transactions. Second, we infer that the transactions in this application take a long time to execute: an average transaction of the Labyrinth application contains several orders of magnitude more instructions than the other applications in the STAMP benchmark suite. This means that conflicts will be costly: retrying a transaction incurs a large penalty. Parallelizing the computation inside the transactions will reduce this cost.

Profiling reveals that, for typical inputs, more than 90% of the execution time of the program is spent in the expansion step (line 12 in Listing 4.3). This performs a breadth-first search. Listings 4.5a and 4.5b show a simplified version of the relevant code. The `expand` function starts with a queue containing the `src` cell (Listing 4.5a, line 2). In `expand-cell` (Listing 4.5a, line 8), the first cell in the queue is expanded, which updates the neighboring cells in `local-grid` for which a cheaper path has been found, using `ref-set` (Listing 4.5b, line 11), and returns these neighbors. These are then appended to the queue (Listing 4.5a, lines 7–8), and the loop is restarted. This continues until either the queue is empty or the destination has been reached.

[1] Clojure's loop construct `(loop [x 0] (recur 1))` defines and calls an anonymous function, in which `recur` executes a recursive call. It is equivalent to Scheme's named let: `(let l ([x 0]) (l 1))`.

Application	**Transaction length** (mean # of instructions per tx)	**Average time in transaction**
Labyrinth	219,571 🟥	100% 🟥
Bayes	60,584 🟥	83% 🟥
Yada	9,795 🟥	100% 🟥
Vacation-high	3,223 🟧	86% 🟥
Genome	1,717 🟧	97% 🟥
Intruder	330 🟩	33% 🟧
Kmeans-high	117 🟩	7% 🟩
SSCA2	50 🟩	17% 🟩

Table 4.4: Characterization of the STAMP applications, abridged from Minh et al. [2008]. These numbers were gathered on a simulated 16-core system. The transaction length and transactional execution time are color-coded high, medium, low.

Labyrinth with parallel search

In Listing 4.5c, additional parallelism is introduced by replacing the breadth-first search algorithm by a parallel version of this algorithm. It uses *layer synchronization*, a technique in which all nodes of one layer of the breadth-first search graph are expanded – in parallel – before the next layer is started [Zhang and Hansen 2006]. The queue now starts as a set containing only the `src` cell (line 2). In each iteration of the loop, `expand-layer` will expand all cells in the queue in parallel (lines 11–13), using Clojure's parallel map operation `pmap`. `pmap` divides the list of cells into partitions (not shown in the code) and processes each partition in a separate task. It gathers the results of all tasks in futures, here all neighbors of the current layer, joins their results, and returns these. The union of all returned neighbors is then used as the queue for the next iteration of the loop (line 10). As before, this continues until either the queue is empty or the destination has been reached.

However, this code does *not* work as expected in Clojure! Each partition created by `pmap` is processed in a new task, calling `expand-cell`, in which an `atomic` block appears. As transactions are thread-local in Clojure, it detects no transaction is running in the current task, and starts a *new* transaction. When the `atomic` block ends, this inner transaction is committed. However, the surrounding transaction may still roll back, while the inner transaction cannot be rolled back anymore. Spurious nested transactions are created which commit independently from their parent: this is the 'spurious retries' problem that was discussed in Section 3.2.3 (page 53), and can eventually lead to incorrect results.

```
(defn expand [src dst local-grid] ; Called in a transaction, in Listing 4.3
  (loop [queue (list src)]
    (if (empty? queue)
      false ; no path found
      (if (= (first queue) dst)
        true ; destination reached
        (recur (concat (rest queue)
                       (expand-cell (first queue) local-grid)))))))
```

(a) Sequential, breadth-first search of local grid.

```
(defn expand-cell [current local-grid]
  (atomic
    (let [neighbors (get-neighbors local-grid current) ; neighbors of current
          cheaper-neighbors ; neighbors of current for which we found a cheaper path
            (filter
              (fn [neighbor]
                (< (cost neighbor current) ; cost of path to neighbor, through current
                   (deref neighbor)))      ; cost of previous path to neighbor
              neighbors)]
      (doseq [neighbor cheaper-neighbors]           ; for each cheaper neighbor:
        (ref-set neighbor (cost neighbor current))) ; set new cost
      cheaper-neighbors)))
```

(b) Expand a cell. (Code modified for clarity.)

```
(defn expand [src dst local-grid]  ; Called in a transaction, in Listing 4.3
  (loop [queue (set [src])]
    (if (empty? queue)
      false ; no path found
      (if (contains? queue dst)
        true ; destination reached
        (recur (expand-layer queue local-grid))))))

(defn expand-layer [queue local-grid]
  (reduce union (set []) ; convert list of list of neighbors into a set (without duplicates)
    (pmap                ; parallel map, returning list of list of neighbors
      (fn [p] (expand-cell p local-grid))
      queue)))

(defn pmap [f xs] ; Simplified version of Clojure's pmap
  (let [futures (map (fn [x] (fork (f x))) xs)]
    (map join futures)))
```

(c) Parallel breadth-first search of local grid. The expansion occurs in layers, the cells in each layer are expanded in parallel in `expand-layer`. `pmap` is a parallel map built into Clojure.

Listing 4.5: Expansion step through Labyrinth grid. (a) shows the original, sequential search algorithm; (c) replaces this with a parallel version. The function `expand-cell` in (b) is used by both to expand a single cell.

Problem Analysis

In general, Clojure allows futures to be created in a transaction, but they are not part of that transaction's context. When an `atomic` block appears in a new task, a separate transaction is created with its own, possibly inconsistent, snapshot of the shared memory. This transaction will commit independently. Clojure does not consider the creation of futures as part of the transaction, hence it is not undone when the encapsulating transaction is rolled back. As such, **the isolation of the transactions is broken**. This is not the desired behavior of the presented example. The same problems occur in most library-based STM implementations, including ScalaSTM, Deuce STM for Java, and GCC's support for transactional memory for C and C++.[2]

Haskell, on the other hand, does not allow the code above to be written. The type system prohibits the creation of new futures in a transaction, as transactions are encapsulated in the STM monad while `forkIO` can only appear in the IO monad. As such, the isolation of transactions is guaranteed, but in effect, **the potential parallelism is limited**: every time transactions are introduced to isolate some computation from other tasks, the potential performance benefits of parallelism *inside* this computation are forfeited. This is throwing out the baby with the bathwater. This problem becomes apparent for programs containing long-running transactions. In the Labyrinth example, the maximal amount of parallelism is equal to the number of input source–destination pairs, even though additional parallelism could be exploited by the breadth-first search algorithm.

Furthermore, these design choices *hinder reusability* [Haines et al. 1994]. Inside a transaction, calling a library function or other part of the program that contains `fork` is unsafe: it is either not allowed or can lead to incorrect results. For instance, it is impossible to use a library that implements a parallel breadth-first search for the Labyrinth application.

Finally, we note that transactions are used to ensure isolation, e.g. to prevent overlapping paths in the Labyrinth example, but they also form the unit of parallelism, evidenced by the fact that the maximal amount of parallelism is equal to the number of transactions. Moore and Grossman [2008] and Haines et al. [1994] argue that isolation and parallelism are orthogonal issues, but the *notions of isolation and parallelism are conflated*. Parallelizing the search algorithm should be orthogonal to the isolation between transactions, but it is not.

The ideal solution is one where several tasks can be created in a transaction and can execute in parallel, i.e. allowing `fork` inside `atomic` (unlike Haskell's `forkIO`). Furthermore, these tasks should be able to access and modify the transactional variables, using the transactional context of the encapsulating transaction (unlike Clojure's or

[2] https://nbronson.github.io/scala-stm/, https://sites.google.com/site/deucestm/, and https://gcc.gnu.org/wiki/TransactionalMemory.

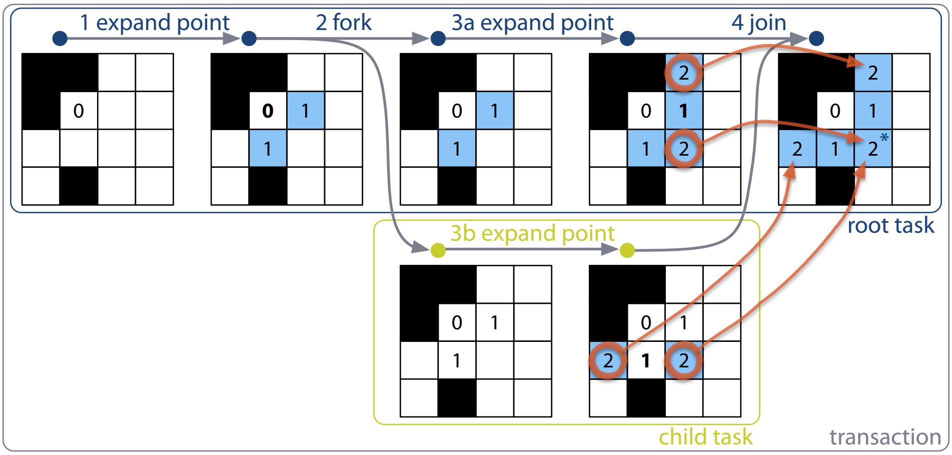


Figure 4.6: The timeline of the transactional tasks that are forked and joined when expanding the labyrinth grid. At each point in time, we show the grid as it exists in that task. The cells that are stored in the snapshot of the task are white and black, the modifications stored in the local store are blue. The orange arrows indicate how the local stores of two tasks are merged.

Scala's futures in transactions). With our approach, we want to preserve isolation between all transactions in the program.

4.3 Transactional Futures

In this section, we define transactional futures. A **transactional future** is the future associated with a so-called transactional task: a task that is forked while a transaction is running. We describe transactional futures and transactional tasks (Section 4.3.1) and the semantics of the `join` operation (Section 4.3.2) informally. (A formalization will be given in Chapter 7.)

4.3.1 Transactional Tasks

When a future is forked in a transaction, a new parallel task is spawned. We call this task a **transactional task**, to indicate that it is associated with the transaction that was active when it was spawned. As motivated in the previous section, a transactional task operates within the context of its encapsulating transaction, so that it can access and modify the state of the transaction.

Conceptually, each transactional task creates a copy of the transactional heap, and will access and modify that private copy. This ensures that two tasks can run concurrently without interfering with each other. To this end, a transactional task contains two data structures: a (read-only) **snapshot** containing the values of the transactional variables when the task was spawned, and a **local store** containing the values that the task has written to transactional variables.

Each transaction starts with one *root* task that evaluates the transaction's body. Its snapshot is a copy of the transactional heap; its local store starts empty. In Figure 4.6, a timeline of the `expand` operation of the Labyrinth application is shown. At the start of the transaction, the snapshot of the root task contains the source cell at distance 0 (in white). After step 1, the local store is modified with the expanded cells at distance 1 (in blue).

As before, tasks are spawned using the `fork` *e* construct, in the root task or any subsequently created task. Thus, `fork` can be used in the dynamic extent of a transaction. When a task is spawned, its snapshot should reflect the current state of the transactional variables, so it is the snapshot of its parent task modified with the current local store of the parent. The local store of a newly spawned task is empty. In Figure 4.6, step 2 spawns a task, the new task's snapshot (in white) consists of its parent's snapshot combined with the parent's local store.

While a task executes, it looks up transactional values in its snapshot, and modify them by storing their updated values in the local store. It only uses its own snapshot and local store, ensuring that each task runs in isolation. In steps 3a and 3b, the root task and its child both expand a cell and update their local stores (in blue).

When a task finishes its execution, its future is resolved to its final value. When a task is joined for the first time, its local store is merged into the task performing the join, and the value of its future is returned. This way, changes propagate from child tasks to their parent. In the figure, step 4 copies the modified cells of the child task (blue cells) into the root task. Subsequent joins of the same task will not repeat this, as their changes are already merged; they will only return the final value of the future.

At the end of the transaction, the modifications of all transactional tasks in the transaction should have been merged into the root task, and these are committed atomically. All changes from all tasks are committed in a single step, so the transaction remains an indivisible step to the outside, maintaining its isolation. If a conflict occurs at commit time, the whole transaction is aborted and retried. If a conflict occurs in one of the tasks while the transaction is still running, *all* tasks are aborted and the whole transaction is retried. In other words, the tasks within a transaction are coordinated so that they either all succeed or all fail: they form one atomic group.

4.3.2 Conflicts and Conflict Resolution Functions

When a task is joined into its parent, conflicts are possible: it may be the case that the child task has modified a transactional variable that the parent also modified since the creation of the child. In that case, a write–write conflict occurs. An example of such a conflict is marked on Figure 4.6 with an asterisk (*). Note that in this example both values happened to be the same, but this is not necessarily the case in general.

For these situations, a **conflict resolution function** can be specified per transactional variable.[3] The programmer can specify a `resolve` function when the transactional variable is created, using `(ref initial-value resolve)`. If a conflict occurs, the new value of the variable in question is the result of `resolve`$(v_{\text{original}}, v_{\text{parent}}, v_{\text{child}})$, where v_{parent} and v_{child} refer to its value in the parent and child respectively, and v_{original} refers to its value when the child was created (stored in the child's snapshot).

In the Labyrinth example, the new value of a conflicting cell should be the minimum of the joining tasks, so `resolve`$(o, p, c) = \min(p, c)$, as we want to find the cheapest path. Generally, conflict resolution functions are useful when each task performs a part of a calculation. For instance, when each task calculates a partial result of a sum, the resolve function is `resolve`$(o, p, c) = p+c-o$, as the total is the value in the parent plus what was added in the child since its creation. Similarly, if several tasks generate sets they are combined using `resolve`$(o, p, c) = p \cup c$, or if several tasks generate lists of results they can be combined with `resolve`$(o, p, c) = \text{concat}(o, p-o, c-o)$. If no conflict resolution function is specified, we default to picking the value in the child over the one in the parent, `resolve`$(o, p, c) = c$, as we reason that when a programmer explicitly joins a task he intends to merge all its changes. Conversely, the parent may be preferred by specifying `resolve`$(o, p, c) = p$.

Read–write "conflicts" are not considered to be actual conflicts in our model. If the parent reads a transactional variable while its child wrote to it, the parent still reads the 'old' value from its snapshot. The value will only be updated after an explicit `join` of the child. This avoids non-determinism, as the moment at which changes from one task become visible in another does not depend on how tasks are scheduled but on explicit `join` statements.

4.3.3 Summary

Using the concepts introduced in this section, the code in Listings 4.5c and 4.5b now behaves as expected. Each newly spawned task is part of the encapsulating transaction's context and has access to its state. Transactional tasks can observe the changes that occurred before they were created and they make their modifications in a private local store. When they are joined, their changes become visible in their parent task. The only required modification to the Labyrinth program is the definition of the conflict resolution function on the grid cells.

In Chapter 9, Section 9.2, we evaluate the performance of the Labyrinth application. We observe that transactional futures improve performance in two ways. First,

[3] This is inspired by a similar idea from Concurrent Revisions [Burckhardt et al. 2010], where conflicts on 'versioned variables' are resolved using such functions.

introducing finer-grained parallelism inside the transaction lowers the execution time of each transaction and thus of the whole program. Second, the lower execution time of transactions also means that the cost of conflicts is decreased, as each attempt takes less time.

4.4 | Properties of Transactional Futures

Transactional futures exhibit several useful properties. In this section, we first describe how transactional futures affect the guarantees of their constituent models (Section 4.4.1). This is discussed more formally in Section 7.3 of Chapter 7. Next, we discuss some additional useful properties arising from the combination of futures and transactions (Section 4.4.2).

4.4.1 Guarantees

As explained in Chapter 2, futures guarantee determinacy and transactions guarantee isolation and progress (deadlock freedom in our implementation). We discuss the effect of transactional futures on these guarantees.

■ Isolation of transactions

Transactional futures maintain isolation (whether this is serializability, opacity, or snapshot isolation). This can be seen as follows. We require that all tasks created in a transaction are joined before its end, otherwise an error is raised. All changes made to the transactional variables by all tasks in the transaction have therefore been applied to the local store of the root task before the transaction commits. Upon commit, they are applied to the transactional heap in a single atomic step, just as if no futures were created in the transaction. Consequently, transactions remain atomic and isolated: the time of the commit determines the order of the transaction in the serialization.

Transactional futures realize this guarantee by making `fork` and `join` a part of the transaction in which they run, instead of an independent side effect. This solves the 'spurious retries' problem that appeared in Clojure in Section 3.2.3 (page 53) for this combination.

It may appear strange to talk about serializability for a transaction that contains internal parallelism. However, serializability only requires that parallel executions of the program are equivalent to an execution in which the transactions run serially. It does not impose any restrictions on interleavings outside transactions nor on interleavings within a transaction. It is only the transactions as a whole that must be serializable.

■ Deadlock freedom

The guarantee of deadlock freedom of transactions is maintained as well. Transactional futures allow `join`, a blocking construct, to be used in transactions. This could potentially cause deadlocks when two tasks are waiting for each other to finish, as we saw when we discussed the 'unexpected blocking' problem in Clojure. However, this can never happen here: the tasks in a transaction form a spawn tree, just like they did when used outside transactions (as described Section 3.3.3 on page 63), and therefore can never contain a cycle.

■ Intratransaction determinacy

Transactions are a non-deterministic model because the order in which transactions are committed is not deterministic. Hence, breaking determinacy is unavoidable when futures and transactions are combined. However, we are still able to guarantee determinism *inside* transactions, which we refer to as **intratransaction determinacy**.

The fact that transactional futures do not introduce non-determinism inside transactions follows from two observations. First, it does not matter in which order the instructions of two tasks are interleaved, as they both work on their own copies of the data. Second, the join operation is deterministic as long as the conflict resolution function is. The changes made in one task only become visible in another one after an explicit and deterministic `join` statement has been executed. As a result, given the state of the transactional memory when a transaction started, it can only have one result. This property is "intratransaction determinacy".

More formally, intratransaction determinacy states that, given the initial state of the transactional memory, a transaction must always have the same result, assuming that all conflict resolution functions are determinate.[4] A transaction has two kinds of results: its final value, and its effects on transactional memory. We will discuss this guarantee more formally in Section 7.3.1 (page 144).

Intratransaction determinacy makes the behavior in a transaction easier to predict. Developers can trace back the value of a variable by looking where tasks were joined. Such a trace was shown in Figure 4.6, where two tasks modified the same data concurrently yet there was only one end result.

4.4.2 Additional properties

The interaction of transactions and futures gives rise to some additional properties, which we discuss here.

[4] Compare this with the definition of determinacy, which states that, given an input, a program must always have the same output.

Coordination of transactional tasks

While a transaction can contain many tasks, each task is fully contained in a single transaction. The changes of all tasks created in one transaction are committed at the same moment, when their encapsulating transaction ends. Hence, if the commit succeeds, the changes of all tasks are committed, and if the commit fails, the changes of no task are committed. If a conflict occurs in one task during its execution, all other tasks in the transaction are aborted as well, and the transaction, as a whole, restarts. As such, developers can use transactional futures as a mechanism to coordinate tasks by encapsulating several tasks in a transaction. Encapsulating tasks in a transaction makes them an atomic (indivisible) unit.

We also observe that the notions of isolation and parallelism are now decoupled. There is no longer a one-to-one mapping between tasks and transactions, instead more fine-grained parallelism becomes possible. When introducing transactional tasks, the parallelism is increased while the granularity of transactions remains the same.

No semantic transparency

Transactional futures are *not* semantically transparent (cf. Section 2.3.2 on page 18): wrapping `fork` around an expression in a transaction changes the semantics of the program. As explained in Section 4.1, this was already the case for non-transactional futures that start their own transaction: they also do not necessarily evaluate to the same result every time.

Violating semantic transparency is a necessary compromise to achieve our goal of executing tasks in parallel. If a transactional task were semantically transparent, its effects on the transactional state would need to be known at the point where it is created, before the parent task can continue. Therefore the child task and its parent would need to be executed sequentially. Instead, we opt to give up on full determinacy as a necessary compromise to achieve efficient parallel execution of tasks, and instead guarantee 'only' intratransaction determinacy.

Nonetheless, we argue that we maintain the "easy parallelism" of futures. First, intratransaction determinacy ensures that the order in which the instructions of the transactional tasks are interleaved does not affect the result. Second, transactional futures have a straightforward and consistent semantics of how the transactional effects of tasks are composed. Each task can modify the transactional state, but its effects become visible in a single step only when it is joined, and conflicts are resolved deterministically.

Transactional tasks are like nested transactions

A transactional task makes a read-only snapshot of the transactional state upon its creation, and stores its modifications in a local store. This mirrors closely how a regular transaction (conceptually) creates a read-only copy of the transactional heap at its creation and stores its modifications in a local store. We could say that, while it is not syntactically shown, a transactional task starts a 'nested transaction'. This similarity should provide a familiar semantics to developers. The differences with nested transactions in the existing literature, particularly our different join semantics, are discussed in Section 4.5.

Non-transactional futures maintain their semantics

Futures that are created outside a transaction have the same semantics in our model as before (in L_f): these *non-transactional tasks* cannot access or modify the transactional state directly. They can of course still create a transaction themselves and modify the transactional state indirectly, as discussed earlier in Section 4.1.

4.5 Related Work

We describe four categories of related work: nested and parallel transactions, Java Transactional Futures, deterministic concurrency models with shared memory, and models that allow parallel tasks to safely access shared memory.

4.5.1 Nested, Parallel, and Nested Parallel Transactions

There is a wide range of work on nested, parallel (or "multithreaded"), and nested parallel transactions. We provide an overview and delve deeper into nested parallel transactions.

Nested transactions

Nested transactions are transactions created in the context of another transaction [Moss 1981, Beeri et al. 1989, Moss and Hosking 2006, Moravan et al. 2006, Ni et al. 2007]. They were already discussed in Section 3.3.3 (page 64), where we explained the difference between open and closed nesting. Nested transactions commit independently from their parent, and therefore can fail separately. Thus, when a nested transaction encounters a conflict, only a portion of the work needs to be retried, potentially improving the performance of large transactions. In contrast to transactional futures, nested transactions do not execute in parallel: they correspond to nested `atomic` blocks and not the nesting of `fork` in `atomic`. They merely improve performance by limiting the cost of a retry (which our approach does as well), not by introducing parallelism.

Internally parallel transactions (or multithreaded transactions)

Transactions in which multiple threads are spawned are **multithreaded transactions** [Haines et al. 1994]. They occur when libraries that use threads are called in a transaction. They work like the 'naive' combination we described in Section 4.2: unlike transactional futures, threads in a transaction do not run within their parent's transactional context. To access transactional memory, a new transaction must be created, but this new transaction is independent from its parent and thus does not roll back when the parent does, breaking serializability.

Moore and Grossman [2008] provide two variations of the `spawn` construct that create two sorts of threads when used in a transaction: **internally parallel** and **on-commit** threads. Internally parallel threads run within the transaction's context, so they can access its state, but there is no isolation between the threads. Therefore, race conditions are possible between internally parallel threads of the same transaction. On-commit threads are executed after the transaction commits. They do not break the isolation of the transaction, but they are limited: as they run outside the transaction's context, they cannot access the transactional state. (A similar technique called atomic deferral has been introduced by Zhou et al. [2017].)

Dabrowski et al. [2013] define the **Atomic Fork Join** language which mixes atomic sections and fork/join parallelism. Dabrowski et al. [2015] formalize this notion of serializability for transactions with internal parallelism: when threads are spawned within a transaction, they are not seen as 'interfering threads' and are allowed to conflict. An atomic section is isolated towards the rest of the program, but inside an atomic section several threads may run at the same time.

Nested parallel transactions

Transactional Featherweight Java combines nested and multithreaded transactions [Vitek et al. 2004]. When a transaction spawns a thread, the new thread inherits the transactional environment of its parent, like the internally parallel threads of Moore and Grossman [2008]. This can lead to race conditions, which are avoided by starting a new, nested transaction in the child thread. When a nested transaction commits, its changes are written to its parent's write set, similar to transactional futures. However, conflicts between child and parent are explicitly forbidden: the value of a variable in a child must be the same as its value in the parent.

Nested parallel transactions (NPTs) are nested transactions that run in parallel [Agrawal et al. 2008, Barreto et al. 2010, Baek et al. 2010, Volos et al. 2009]. They are very similar to transactional futures, differing mostly in how conflicts *within* a transaction are handled. We call these **intratransactional conflicts** and study them in more detail here. To clarify the differences between both models, we use the examples from Agrawal et al. [2008].

Listing 4.7a shows a program in which four threads update a shared variable `x`, increasing it with 1, 10, 100, and 1000. There is no synchronization in this example, so race conditions are possible: every thread executes a read followed by a write on `x`, but `x` may have been updated by another thread in the mean time. This program therefore has 11 different possible outputs, listed in the table shown in Listing 4.7.

NPTs resolve intratransactional conflicts using the traditional serializability of transactions: when two nested transactions conflict, one of both will roll back and retry. Listing 4.7b adds transactions to the previous program so that three threads use a transaction while one does not. Using NPTs, this program has two possible outputs: 1111 when all write operations succeed or 0111 when the thread without a transaction overwrites its sibling.

The same code can run using transactional futures. It will always produce the same output: 1010 when using the default conflict resolution function (which takes the value from the child), or 1111 (arguably the only expected result) when using a conflict resolution function designed for sums. Note that, when using transactional futures, it does not matter whether threads start a new transaction (nested `atomic`) or not; they are part of the encapsulating transaction in any case. A programmer can thus never forget an `atomic`; everything is safe by default.

The advantage of NPTs is that they use the same, familiar semantics to resolve conflicts both inside and between transactions. However, they do not guarantee determinacy, as it is unpredictable which transactions will roll back. In contrast, transactional futures resolve conflicts differently depending on whether they occur inside a transaction or between transactions. *Intra*transactional conflicts (i.e. inside a transaction) are resolved deterministically using conflict resolution functions, thus guaranteeing intratransaction determinacy, and without causing a rollback. *Inter*transactional conflicts (i.e. between transactions) are resolved using the traditional rollback mechanism, guaranteeing serializability but not determinacy.

Transactional futures thus expose two conflict resolution models. This is a result of the fact that they are a combination of two concurrency models with different properties: futures guaranteeing determinacy and transactions guaranteeing serializability. Using the transactional futures presented in this dissertation, developers can choose which model is applicable for each part of their application, and they can combine and nest both in each other.

NPTs and transactional futures have different performance characteristics. Both models allow the same amount of parallelism. In an application without intratransac-

```
(def x (ref 0))
(let [a (fork (ref-set x (+ @x 1)))
      b (fork (ref-set x (+ @x 10))
              (let [c (fork (ref-set x (+ @x 100)))
                    d (fork (ref-set x (+ @x 1000)))]
                (join c)
                (join d)))]
  (join a)
  (join b))
```

(a) A program in which four threads modify a shared variable. There is no synchronization in this example. (This is a translation of Figure 1 from Agrawal et al. [2008], with their construct `parallel { a } { b }` changed to the corresponding `fork` and `join` constructs.)

```
(def x (ref 0))
(atomic                                                          ;t0
  (let [a (fork (atomic (ref-set x (+ @x 1))))                   ;t1
        b (fork (atomic                                          ;t2
                  (ref-set x (+ @x 10))
                  (let [c (fork        (ref-set x (+ @x 100)))    ;no transaction
                        d (fork (atomic (ref-set x (+ @x 1000))))]  ;t3
                    (join c)
                    (join d))))]
    (join a)
    (join b))
```

(b) Four threads modify a shared variable, three use transactions and one does not. (This is Figure 3 from Agrawal et al. [2008].)

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	#
(a) no sync		•		•		•	•	•		•	•	•		•	•	•	11
(b) NPT								•								•	2
(b) TF default											•						1
(b) TF custom																•	1

(c) Possible outputs of the program in (a) without synchronization, and the program in (b) when using Nested Parallel Transactions (NPT), Transactional Futures (TF) with the default conflict resolution function, and Transactional Futures using the custom conflict resolution function `(fn [p c o] (+ p (- c o)))`. The last column sums the number of possible outputs.

Listing 4.7: How conflicts are resolved without transactions, using nested parallel transactions, and using transactional futures.

tional conflicts, both models operate equivalently. However, in our motivating example (the Labyrinth application) such conflicts frequently occur: the parallel expansion of the breadth-first search causes conflicts on overlapping cells. In this application, NPTs would cause frequent rollbacks of the inner transactions, essentially sequentializing them, detrimental to performance. Transactional futures resolve these conflicts without a rollback. In fact, NePaLTM is an implementation of NPTs that always uses mutual exclusion locks on nested transactions, sequentializing them, exactly to improve performance as these conflicts occur often [Volos et al. 2009].[5] Hence, in applications with many intratransactional conflicts, we expect transactional futures to perform better. We look at the performance of transactional futures for the Labyrinth application in Section 9.2.

A final difference is that transactional futures allow a more fine-tuned conflict resolution, for example in Labyrinth where the shortest path (minimum) is preferred. However, they rely on the developer for an appropriate conflict resolution function. The example shown in Listing 4.7 also demonstrated how the conflict resolution function affects the end result.

Coordinated nested transactions (or coordinated sibling transactions)

Last, **coordinated sibling transactions** extend nested parallel transactions with combination operators. Ramadan and Witchel [2009] criticize NPTs for their conflict resolution model: NPTs will only run in parallel if they do not conflict, i.e. if they are independent, but "the fact that the work was put into the same transaction to start with makes complete independence unlikely". Instead, they allow sibling transactions to be composed with three operators: OR, AND, and XOR. Sibling transactions always return either success or failure.

- The OR operator has the same semantics as normal nested transactions: siblings are executed in parallel and can succeed or fail independently. Any sibling that fails is retried.
- The AND operator is used when the siblings should either all succeed or all fail. When one sibling fails, the others are aborted too.
- The XOR operator is used for speculative parallelism: the effects of the first sibling that succeeds are committed. When one sibling succeeds, the others are aborted. This is useful to parallelize search algorithms for instance, where as soon as a result is found by one thread the others can be aborted.

[5] Volos et al. [2009] say "Previous work by Agrawal et al. [2008] has shown that such a design is complex and its efficient implementation appears to be questionable." and "We initially considered an alternative design where transactional concurrency control is used at all nesting levels. However, we eventually abandoned it in favor of the one [in which all nested transactions are sequentialized.]"

Coordinated sibling transactions make it possible to express different kinds of parallelism inside a transaction, but arguably introduce quite complex constructs in doing so.

4.5.2 Java Transactional Futures

Independently from our work, Zeng et al. [2015, 2016] developed **Java Transactional Futures** (JTFs). JTFs start from the same observation as we did: by combining futures and transactions, intratransaction parallelism can be exploited. They also encounter the same problems: how can a consistent view be ensured and how should conflicts be resolved, when multiple futures that run in the same transaction access the same transactional variables? However, their solution is different to ours: JTFs opt to maintain the semantic transparency of futures.

We describe the fork and join semantics of JTFs, discuss its properties and performance, and then highlight the differences with our transactional futures using an example.

Fork The fork semantics of JTFs is essentially the same as that of our transactional futures. When a future is forked, it is automatically wrapped in an atomic block and evaluated as a child transaction. Both the child and parent run in new transactional contexts, which are forked off the original context. The child runs in the future; the parent is said to be running in its *continuation*; and the original context is referred to as the snapshot.

Join On the other hand, the join semantics of JTFs is very different from ours. JTFs maintain the semantic transparency of futures. Therefore, after the future and continuation have finished their execution, the changes from the future are applied to the snapshot first, and then followed by the changes of the continuation. If the future and continuation conflict, the continuation must roll back and retry.

Properties JTFs provide intratransaction determinacy, just like our transactional futures: given the state of the transactional memory at the start of the transaction, it can only have one result. In contrast to our transactional futures, JTFs maintain the semantic transparency of futures too though. The result of a transaction is therefore the same as if it ran without futures.

Performance The choice to maintain semantic transparency has an effect on performance. When a future and its continuation conflict, the continuation needs to roll back and retry. In our motivating example, the Labyrinth application, these kinds of conflicts are frequent: in every layer of the breadth-first search algorithm, many cells are expanded in parallel, and but as these cells often are neighbors, many of the newly expanded neighbors are likely to overlap. Because our transactional futures resolve these conflicts instead of rolling back, we expect performance for the Labyrinth application to be better using our transactional futures. (We evaluate the Labyrinth application in Section 9.2 and observe that conflicts have a large impact on performance.) Our transactional futures forsake semantic transparency to avoid rollbacks in the transaction.

Example and comparison We compare our transactional futures, JTFs, and NPTs using the example in Listing 4.8. It contains two transactional variables, `a` and `b`, initialized to 0 and 1 respectively. A future is created in which `a` is set to 2 (line 4). At the same time, in the original thread, `b` is set to the current value of `a` (line 5). Afterwards the future is joined back into the original thread. Figure 4.9 depicts the execution trace for this program using the three models. The three models differ in which value of `a` is seen by the original thread on line 5, and how conflicts are resolved on line 6:

- Our transactional futures (Figure 4.9a) execute both threads in isolation, so the original thread will never see the updated value of `a` and `b` is always set to 0 on line 5. On line 6, the new value of `a` is merged into the original thread, and no conflicts occur as we do not consider read–write "conflicts". The end result is always `a` = 2 and `b` = 0, guaranteeing determinacy and avoiding rollbacks in the transaction, but breaking semantic transparency.
- JTFs (Figure 4.9b) also evaluate both threads in isolation, so the original thread sets `b` to 0 on line 5 in its first attempt. The results from the future are merged before those of the continuation. When the continuation attempts to merge its changes back into the main thread, a conflict occurs, and the continuation rolls back and retries. In the second attempt, `b` is set to the new value of `a`: 2. The end result is always `a` = 2 and `b` = 2, guaranteeing determinacy and semantic transparency, but introducing rollbacks in a transaction, thus increasing execution time.
- NPTs (Figures 4.9b and 4.9c) serialize the two nested transactions, leading to two possible results. Either the future commits first, followed by the original thread: in this case the original thread encounters a conflict and retries, eventually resulting in `a` = 2 and `b` = 2. This execution is equivalent to JTFs. Otherwise, the original thread commits first, followed by the future: this does not cause a conflict, and results in `a` = 2 and `b` = 0. NPTs do not guarantee determinacy or semantic transparency, but do uphold serializability between the nested transactions.

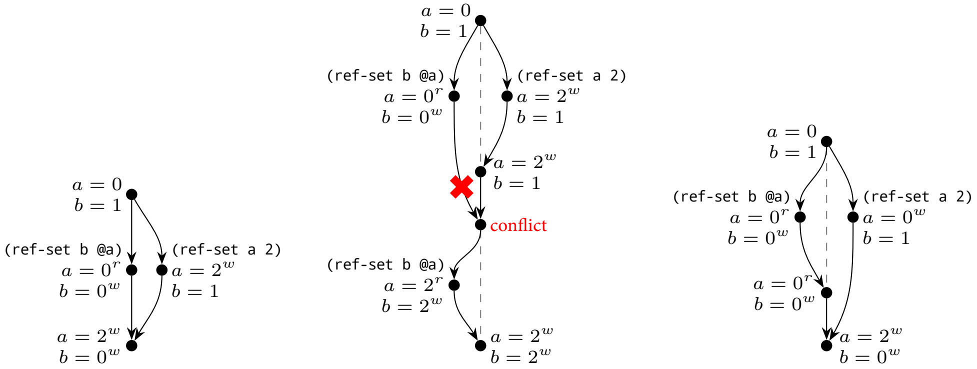
A summary of the properties of the three models is shown in Table 4.10.

4.5.3 Concurrent Revisions and Worlds: Deterministic Shared Memory

In this section, we describe two models that provide deterministic access to shared memory. The semantics of `fork` and `join` in transactional futures is inspired by both.

```
(def a (ref 0))
(def b (ref 1))
(atomic
  (let [f (fork (atomic (ref-set a 2)))]
    (ref-set b @a)
    (join f)))
 (print @a @b)
```

Listing 4.8: Example that demonstrates the different join semantics of our TFs, JTFs, and NPTs. (The underlined atomic may be elided when using JTFs and our TFs.)



(a) The only result using our TFs.

(b) The only result using JTFs; also one of two results using NPTs.

(c) Second of two results using NPTs.

Figure 4.9: Execution traces of Listing 4.8. Read effects are indicate with the subscript ^{r}, write effects with ^{w}. Using our TFs the result is always $a = 2$, $b = 0$ (a), using JTFs the result is always $a = 2$, $b = 2$ (b), and using NPTs the result is either $a = 2$, $b = 0$ (c) or $a = 2$, $b = 2$ (b).

	TF	JTF	NPT
Serializability of top-level transactions	✓	✓	✓
Intratransaction determinacy	✓	✓	✗
Semantic transparency	✗	✓	✗
No rollback in transaction	✓	✗	✗

Table 4.10: Properties of our Transactional Futures (TF), Java Transactional Futures (JTF), and Nested Parallel Transactions (NPT).

Concurrent Revisions are a model for task parallelism with shared memory by Burckhardt et al. [2010]. The semantics of transactional futures was inspired by Concurrent Revisions. They introduce concurrent tasks (called revisions) that can share memory using *versioned variables*. When a task is forked, a conceptual copy is made of the versioned variables; when a task is joined, the modified variables are merged into the joining task. This is identical to how transactional variables behave *inside* a transaction in our model. Furthermore, our conflict resolution mechanism is based on that of Concurrent Revisions, which also relies on deterministic conflict resolution functions for each variable.

The difference with our work that we ensure serializability *between* the transactions and determinacy *in* the transactions, while Concurrent Revisions provide determinacy for the complete program. Transactional futures thus allow two concurrency models to be combined: deterministic futures and non-deterministic but serializable transactions.

Burckhardt et al. [2010] note that, as Concurrent Revisions never require a rollback, they are suited for programs in which conflicts are likely. This is also why we opt for this kind of conflict resolution *in* a transaction: we reason that conflicts between tasks that are put together in one transaction are likely. Top-level transactions are usually unlikely to conflict.

A similar model is provided by **Worlds** [Warth et al. 2011], in which the program state is reified as a *world*. The world can be forked into a child world, a conceptual copy of all program state. The state in a child world can be updated, and eventually committed back (merged) into its parent world. As such, worlds also behave similarly to the transactional variables *in* a transaction and to Concurrent Revisions. However, the Worlds model does not provide parallelism: a child world does not run in parallel with its parent. Instead, Worlds are used as a mechanism to 'undo' changes made to the program state. Consequently, when a child world is merged into its parent there will be no conflicts, as the parent has not changed in the mean time. In the case of transactional futures and Concurrent Revisions, it is possible for the parent to have changed as well, and a form of conflict resolution between parent and child is needed.

4.5.4 Parallel Tasks With Safe Access to Shared Memory

We list related work that provides safety guarantees on access to shared memory from parallel tasks.

Kogan and Herlihy [2014] consider a system in which method calls on shared objects run asynchronously, returning a future. Different threads can call methods on the same object concurrently, and as these methods may have side effects, this can result in unexpected interleavings of their side effects. The authors define three correctness criteria on the order of the interleavings: **strong, medium, and weak futures serializability**. Strong futures linearizability corresponds to semantic transparency: concurrent calls execute as if they were sequential. The authors note that, while this property is easy to reason about, it rules out many interesting optimizations. Medium and weak futures linearizability allow the effect of a future to occur at any moment between its creation and its join. The intratransaction determinacy of transactional futures could arguably be seen as a form of medium futures serializability.

Deterministic Parallel Java enforces safe access to shared memory by using an effect system [Bocchino et al. 2009b], and can thus guarantee determinism. The heap is partitioned into regions, and a list of effects is attached to each method to indicate which regions it reads from and writes to. At compile time, these effects are checked to prevent unsafe accesses. Heumann et al. [2013] extend Deterministic Parallel Java to allow a more flexible **tasks with effects** model. The compiler checks whether the effects correctly describe the memory accesses in each task, and at run time, an 'effect-aware' scheduler ensures that no tasks with interfering effects run at the same time.

Æminium is a language in which every method is annotated with the permissions it needs on shared memory it accesses: unique (exclusive), immutable, or shared access [Stork et al. 2009, 2014]. By default, code is executed concurrently; the use of effects limits concurrency where necessary to guarantee the absence of race conditions. Æminium thus allows parallel tasks to access shared memory safely.

Otello allows parallel tasks to access shared memory, while still running the tasks in isolation [Zhao et al. 2013]. To this end, it introduces *assemblies*, which consist of a task and the set of shared objects it owns. When two assemblies conflict, one is reexecuted after the other has finished. However, while Otello reexecutes code, it does not provide transactions and as such does not guarantee serializability.

4.6 | Summary

In this chapter, we examined on the one hand the creation of transactions in futures, yielding parallel transactions, and on the other hand the creation of futures in transactions, which we call transactional futures.

Creating a transaction in a task is standard behavior in languages that support transactions; it is a prerequisite to run transactions in parallel. It inevitably breaks the determinacy of futures, but maintains the isolation and progress guarantees of transactions.

Creating a future in a transaction is more complicated. In existing languages and frameworks, this is either not allowed or it breaks the isolation of the transactions. Hence, we introduced **transactional futures**, a safe combination of both models. Using transactional futures, each newly created task is part of the encapsulating transaction's context, with access to its state. **Transactional tasks** can observe the changes that occurred before they were created, while they make their modifications in a private local store. When they are joined, their changes become visible in their parent task.

In contrast to a naive combination, transactional futures maintain the **isolation** and **progress** guarantees of transactions. The determinacy that futures normally provide for the whole program is broken, however, it is replaced by a (weaker) guarantee on **intratransaction determinacy**. Developer can thus rely on determinacy in a transaction and isolation between transactions when they combine futures and transactions.

Transactional futures provide some additional useful properties. First, all tasks of a transaction are coordinated so that the tasks in a transaction behave as a single atomic block. Second, transactional futures provide a familiar semantics, as they behave like a form of nested transactions. Third, non-transactional futures maintain their semantics.

In Chapter 7, we will formalize the semantics of transactional futures. In Chapter 9, we will demonstrate the performance benefits of transactional futures. We demonstrate that transactional futures can exploit *more fine-grained parallelism within a transaction*, leading to better performance. Furthermore, we see that by tweaking the number of transactions that run in parallel and the number of tasks created in each transaction, we can lower the chance of conflicts, providing a further boost to performance. We conclude that the conflict resolution of transactional futures makes them especially suitable for applications in which 'intratransactional conflicts' are frequent.

5

Transactional Actors: Communication Between Transactions

This chapter examines the combination of transactions and actors. In Section 5.1, we explain the reasons for combining both models: transactions can be used in actors to protect synchronous access to shared mutable state, and actors can be used in a transaction to distribute and coordinate work that can be executed in parallel. We also establish the problems that occur: the isolated turn principle of actors and the isolation guarantee of transactions can be broken in a naive combination (shown in Table 5.1). In Section 5.2, we introduce transactional actors: a combination of transactions and actors that provides the same constructs as its constituent models, but also defines their semantics when they are combined. We systematically consider each construct of both models and how they can be combined. We list the guarantees provided by transactional actors in Section 5.3 and discuss to which actor models transactional actors apply in Section 5.4. Finally, we compare transactional actors with related approaches in Section 5.5.

We first introduced transactional actors in Swalens et al. [2017].

5.1 | Motivation and Problem Statement

In this section, we motivate the combination of actors and transactional memory. On the one hand, it can be useful to add transactional memory to an actor system, to

→in↓	Future	Transaction	Actor
Future	Nested futures (Section 3.3.3) Det	Parallel transactions (Section 4.1) ~~Det~~ Iso Pro	Communication in future (Section 6.1) ~~Det~~ ~~ITP~~ DLF
Transaction	Parallelism in transaction (Sections 4.2–4.4) ~~Det~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

■ **Guarantees**

Transactions:	Iso	Isolation	Pro	Progress
Actors:	ITP	Isolated Turn Principle	DLF	Deadlock Freedom

Table 5.1: Combinations of transactions and actors.

share state between actors that is protected using transactions (Section 5.1.1). On the other hand, it can be useful to send messages to actors from within a transaction, to distribute work to actors that is coordinated using transactions (Section 5.1.2).

Further, we see that a naive combination of actors and transactional memory can cause problems. As described in Chapter 2, actors guarantee the isolated turn principle and deadlock freedom, while transactions provide guarantees on isolation and progress. We will see that combining the two models naively breaks these guarantees (Section 5.1.3).

5.1.1 Using Transactions in an Actor, to Safely Share Memory

In Section 3.1, we established that introducing shared memory in an actor system can be useful and occurs in practice, based on the results of a study by Tasharofi et al. [2013]. Here, we show that doing so naively has several disadvantages. De Koster et al. [2016a] distinguish two types of actor systems: pure and impure systems; we discuss both separately. Figure 5.2 summarizes the techniques and their disadvantages discussed in this section.

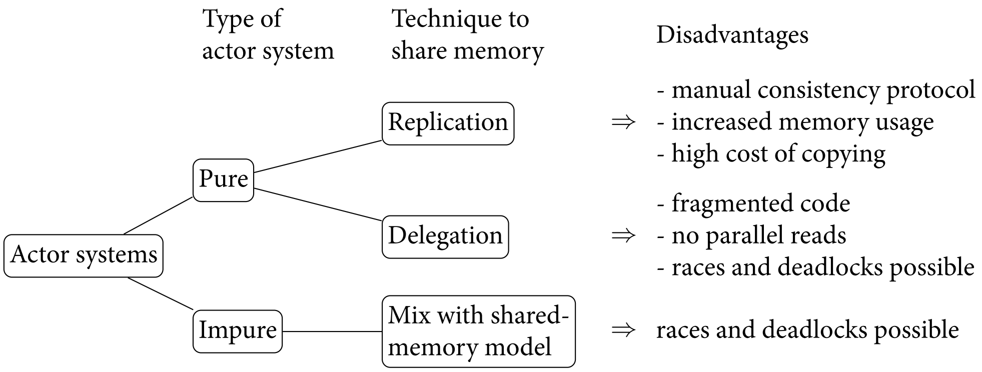


Figure 5.2: Existing techniques to share memory in actor systems and their disadvantages.

Pure actor systems are actor systems that enforce strict isolation between the actors: the state of an actor is fully encapsulated and can only be accessed using asynchronous communication with that actor. Erlang is an example of such a system; other examples are listed in Figure 5.3. These are often languages, which enforce these constraints by construction. (A notable exception is Kilim, a library that enforces isolation using static analysis [Srinivasan and Mycroft 2008].) Thanks to the strict isolation, the developer benefits from strong safety guarantees: low-level data races are prevented by design.

Because pure actor systems enforce isolation, developers cannot directly use shared memory in their program. De Koster et al. [2016a] describe two patterns that are used to represent shared state in these systems instead:

- The first pattern consists of *replicating* the shared state across the different actors that need it. Each actor can read the state directly; writes have to be propagated to each replica. There are several disadvantages to this approach: a consistency protocol is required, replication increases memory usage, and the cost of copying may be too high. De Koster et al. [2016a] found that this is a rarely chosen option.
- The second pattern is to encapsulate the shared state in a separate, *delegate* actor. Each actor can read and write to the state using asynchronous messages. This approach also suffers from several disadvantages: code is fragmented and needs to be written in continuation-passing style, no parallel reads are possible, and race conditions and deadlocks can still occur due to the unexpected interleaving of read and write ('getter' and 'setter') messages.

We conclude that, because pure actor systems enforce strict isolation, they maintain the guarantees of the actor model, but representing shared state in them is complex and error prone. The difficulty of preventing race conditions and deadlocks is pushed entirely to the developer.

Pure actor languages

- Erlang
  http://www.erlang.org
  [Armstrong 2007]
- SALSA
  http://wcl.cs.rpi.edu/salsa/
  [Varela and Agha 2001]
- E
  http://erights.org
  [Miller et al. 2005]
- AmbientTalk
  http://soft.vub.ac.be/amop/
  [Dedecker et al. 2006]
- Pony
  https://www.ponylang.org
  [Clebsch et al. 2015]

Pure actor libraries

- Kilim *(for Java)*
  http://www.malhar.net/sriram/kilim/
  [Srinivasan and Mycroft 2008]
- haskell-actor *(for Haskell)*
  https://hackage.haskell.org/package/actor
  [Sulzmann et al. 2008]

Impure actor languages

- D
  https://dlang.org

Impure actor libraries

- Scala Actors
  [Haller and Odersky 2007]
- Akka *(for Java and Scala)*
  https://akka.io
- Orleans *(for C#/.NET)*
  https://dotnet.github.io/orleans/
- Quasar *(for Java)*
  https://github.com/puniverse/quasar
- Jetlang *(for Java)*
  https://github.com/jetlang
- ActorFoundry *(for Java)*
  http://osl.cs.illinois.edu/software/actor-foundry/
- Pulsar *(for Python)*
  https://github.com/quantmind/pulsar

Many more actor systems can be found at https://en.wikipedia.org/w/index.php?title=Actor_model&oldid=824674140#Actor_libraries_and_frameworks

Figure 5.3: An overview of some pure and impure actor languages and libraries.

Impure actor systems do not enforce strict isolation. Most actor libraries for mainstream languages are impure, as they extend a language that allows shared memory. Scala's actors, for instance, do not prevent developers from using the underlying shared-memory model of Scala. (More examples are listed in Figure 5.3.) In these systems, developers can combine actors with shared memory when that is the most natural or efficient solution.

As we already mentioned in Section 3.1 (page 45), Tasharofi et al. [2013] performed a study of 15 actor programs written in Scala and found that 80% mix the actor model with another concurrency model. In 6 out of 15 cases (40%), developers circumvent the actor model in the places where it is not a good fit to their problem. In some cases, developers introduce shared memory, which they subsequently protect using locks.

The disadvantage of this approach is that the guarantees of the actor model are lost: *the assumptions behind the isolated turn principle are broken*, hence low-level data races become possible.

As we will see in the rest of this chapter, this broken guarantee can be reintroduced by carefully combining actors with transactions, as transactions guarantee isolation. Thus, using transactional memory, memory can safely be shared between actors.

5.1.2 Using Actors in a Transaction, to Distribute and Coordinate Work

Not only are transactions useful to protect access to shared state between actors, conversely, actors are also useful to coordinate work between transactions. We demonstrate how actors can be used to distribute and coordinate work from within a transaction using a travel reservation system. This example is inspired by the Vacation benchmark from the STAMP suite [Minh et al. 2008].

The code of the example is shown in Listing 5.4. Its input consists of a number of customers, who want to reserve two flights, a hotel room, and a car (lines 14–16). These items are stored in transactional memory, so that multiple customers can reserve them in parallel (lines 1–12). Each reservation consists of three steps: the customer looks for available flights, cars, and hotel rooms; reserves them; and updates his record (lines 39–44).

In the original Vacation benchmark, there are a configurable number of worker actors, over which the customers are evenly distributed. However, we believe better performance may be achieved by processing the items that form a reservation in separate actors. (We confirm this performance claim in Chapter 9.) Hence, we create a variation of Vacation, in which the workers send the reservations of the individual items to one of a configurable number of 'secondary' worker actors.

The code then looks as shown in Listing 5.5. However, using traditional actors and STM, this code does not work as expected! In the transaction, four messages are sent, and afterwards the transactional variable c is updated (lines 5–10). If another transaction updates the same variable, this causes a conflict, and the transaction will be aborted. When the transaction retries, the messages are sent again. This is the spurious retries problem of Section 3.2.3 (page 53). The problem is that when a transaction aborts, the messages it sent are not rolled back. As a result, in our example multiple items can be reserved for the same customer.

We observe that *communicating with actors in a transaction breaks its isolation guarantee*: the result of a parallel execution is no longer equal to the result of a serial execution.

```
(def flights [(ref {:id "AC855"  :price 499
                    :orig "London" :dest "Vancouver"
                    :available ["1A" "1B" …] :occupied []})
              …])
(def rooms   [(ref {:id 101      :price 100
                    :location "Vancouver" :beds 5
                    :available ["2017-10-01" …] :occupied []})
              …])
(def cars    [(ref {:id "ABC123" :price 42
                    :location "Vancouver" :seats 5
                    :available ["2017-10-01" …] :occupied []})
              …])

(def customers [(ref {:id 0 :orig "London" :dest "Vancouver" :n 3
                      :start "2017-10-22" :end "2017-10-27" :password nil})
                …])

(defn get-cheapest [items]
  ; Returns the cheapest item from items.
  (first (sort-by (fn [item] (:price @item)) items)))

(defn reserve-flight [orig dest date n-seats]
  ; Finds the cheapest flight from orig to dest on date, and reserve n-seats.
  (let [; Filter flights matching the given criteria, with sufficient available seats
        filtered (filter (fn [f] (and (= (:orig @f) orig)
                                      (= (:dest @f) dest)
                                      (= (:date @f) date)
                                      (>= (count (:available @f)) n-seats)))
                         flights)
        cheapest (get-cheapest filtered)]
    ; Move seats from :available to :occupied
    (ref-set cheapest (occupy cheapest n-seats))))

; Functions reserve-room and reserve-car are similar to reserve-flight.

(defn process-customer [c]
  ; Process a customer: find and reserve two flights (outbound and return), a car, and a room,
  ; and generate a password.
  (atomic
    (reserve-flight (:orig @c) (:dest @c) (:start @c) (:n @c))
    (reserve-flight (:dest @c) (:orig @c) (:end @c) (:n @c))
    (reserve-room   (:dest @c) (:n @c) (:start @c) (:end @c))
    (reserve-car    (:dest @c) (:n @c) (:start @c) (:end @c))
    (ref-set c (assoc @c :password (generate-password)))))
```

Listing 5.4: The Vacation benchmark. (Code has been modified for clarity.)

```
(def customer-behavior
  (behavior [id] [c]
    ; Process a customer: find and reserve two flights (outbound and return), a car, and a room
    ; (all in worker actors), and generate a password.
    (atomic
      (send (rand-nth secondary-workers) :flight (:orig @c) …)
      (send (rand-nth secondary-workers) :flight (:dest @c) …)
      (send (rand-nth secondary-workers) :room   (:dest @c) …)
      (send (rand-nth secondary-workers) :car    (:dest @c) …)
      (ref-set c (assoc @c :password (generate-password))))))

(def reserve-behavior
  (behavior [id] [type & args]
    (case type
      :flight (atomic (apply reserve-flight args))
      :room   (atomic (apply reserve-room   args))
      :car    (atomic (apply reserve-car    args)))))
```

Listing 5.5: An adapted version of the Vacation benchmark, with secondary worker actors.

5.1.3 Problem Statement

We establish that combining actors and transactions can be useful, but leads to broken guarantees (shown in Table 5.1):

- When two or more actors can synchronously access shared, mutable state, **the isolated turn principle is broken**. Consequently, race conditions that were prevented by the actor model can reappear.
- When sending messages in a transaction, **the transaction's isolation guarantee is broken**. Sending messages is a side effect that is not rolled back when the transaction is.

These issues complicate combining both concurrency models in a single application, as the guarantees that developers expect are no longer true. These problems *hinder composability*: when one model is most suitable for one component of the application, and another model fits another component, the developer cannot safely use both. They also *hinder reusability*: including a library that uses one model in an application that uses another may lead to incorrect results.

5.2 Transactional Actors

The problems described in the previous section, breaking the isolated turn principle of actors and the isolation of transactions, occur because the semantics of the combination of two concurrency models is not well defined. In this section, we solve this by defining transactional actors. Transactional actors provide the same constructs as the original actor model and the STM model, described in Chapter 2, but also define a meaningful semantics when their constructs are combined.

We systematically consider how the constructs of each model can be nested in the other. The two models each provide a concurrent construct that contains nested code: `behavior` for actors and `atomic` for transactions. Hence, we study the following combinations: (These correspond to the two cells combining transactions and actors from Table 5.1.)

Transaction in Actor

`behavior` containing:

- `atomic` ①
- `ref` ②
- `deref` ②
- `ref-set` ②

Actor in Transaction

`atomic` containing:

- `behavior` ③
- `spawn` ④
- `send` ④
- `become` ④

We distinguish four categories:

① Embedding `atomic` in `behavior` may execute a transaction in the actors that are spawned with that behavior. This transaction is bound to the current actor.

② Manipulating transactional state using `ref`, `deref`, or `ref-set` follows the canonical semantics of transactions: these actions are only allowed when a transaction is running, and operate within its context. No special semantics is needed when this occurs in an actor.

③ A behavior can be defined in a transaction. The behavior is separate from the transaction in which it is defined: when an actor is spawned with the behavior, at a later time, it no longer has access to the transaction in which the behavior was defined. Hence, a behavior can be defined in a transaction as it can anywhere else, and no special semantics is needed.

④ `spawn`, `send`, and `become` enclose a side effect. Therefore, when they occur in a transaction, their effect should become part of the transaction.

We now discuss these four cases in more detail.

5.2.1 Using Transactional Memory in an Actor

A transaction can run in an actor, inside `atomic` ①. Similar to a thread-based transactional system, where each thread can have one active transaction, here too, each actor can have one transaction active at a time. Manipulating transactional state ② works within the context of the transaction that is active in the current actor. Therefore, transactions run in actors as they do in thread-based systems. This occurs in Listing 5.4, when the function `process-customer` is called in an actor and executes a transaction.

5.2.2 Manipulating Actors in a Transaction

While defining a behavior in a transaction (3) might seem strange, it does not pose any particular difficulties, as it is an idempotent operation. Defining a behavior is similar to defining a function: the code it contains is not executed when it is defined, but at a later time, in a new actor. A behavior can refer to variables in its lexical scope; it is essentially a closure. It does not have access to its encapsulating transaction: while it can refer to transactional variables in its lexical scope, any operations on them must be protected with a new transaction.

Spawning an actor, sending a message, or becoming a new behavior (4) are actions with a side effect. Using these in a transaction requires special semantics, so that they can roll back. Transactional actors make these operations part of the transaction in which they occur.

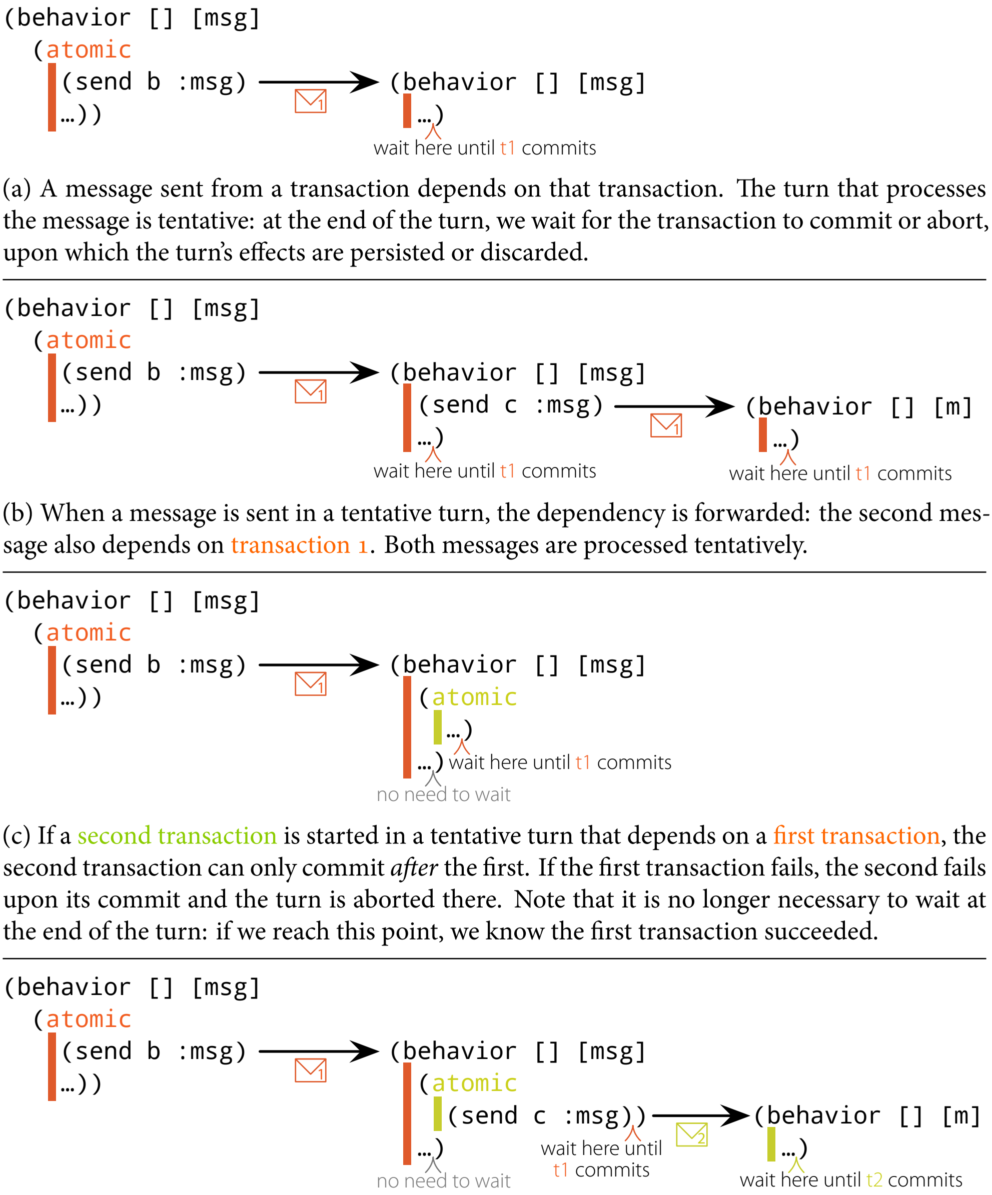
Regular transactional systems use two techniques to incorporate side effects into a transaction. The first technique is to *pessimistically delay* the side effect until it is certain the transaction will commit successfully. For example, an update to a transactional variable is only visible locally at first, the global update is delayed until the transaction commits. In case the transaction aborts, the effect is discarded. The second technique is to *optimistically* perform the side effect immediately, but *roll back* the effect if the transaction aborts, using a compensating action. For example, in transactional systems with open nesting [Ni et al. 2007], one transaction can be nested in another, and commit separately. If the outer transaction aborts, the inner transaction needs to roll back, which relies on the developer specifying a compensating action.

We handle the side effect of each actor construct with the appropriate technique:

spawn Spawning a new actor in a transaction is delayed until the transaction commits. Spawning an actor is a costly operation: memory is allocated for the new actor and its inbox, a thread is created, and the actor's execution then starts on that thread. Hence, taking these three steps and rolling them back if the transaction aborts is not a good idea: each attempt of the transaction would incur this cost over and over again. Delaying the operation until the transaction commits ensures that this cost is only paid once.

become Become is delayed by construction, even in the original actor model: its effect only takes place upon the start of a new turn. As a transaction cannot span multiple turns, it will always be committed before the effect of `become` becomes visible. Hence, it does not matter whether we delay or roll back `become`: both have the exact same cost and result.

send When a message is sent in a transaction, we optimistically send the message immediately to the receiving actor, possibly needing to roll back its effects if the trans-



```
(behavior [] [msg]
  (atomic
    (send b :msg)
    …))
```

```
(behavior [] [msg]
  …)
```

1
wait here until t1 commits

(a) A message sent from a transaction depends on that transaction. The turn that processes the message is tentative: at the end of the turn, we wait for the transaction to commit or abort, upon which the turn's effects are persisted or discarded.

```
(behavior [] [msg]
  (atomic
    (send b :msg)
    …))
```

```
(behavior [] [msg]
  (send c :msg)
  …)
```

```
(behavior [] [m]
  …)
```

1
wait here until t1 commits
1
wait here until t1 commits

(b) When a message is sent in a tentative turn, the dependency is forwarded: the second message also depends on transaction 1. Both messages are processed tentatively.

```
(behavior [] [msg]
  (atomic
    (send b :msg)
    …))
```

```
(behavior [] [msg]
  (atomic
    …)
  …)
```

(c) If a second transaction is started in a tentative turn that depends on a first transaction, the second transaction can only commit *after* the first. If the first transaction fails, the second fails upon its commit and the turn is aborted there. Note that it is no longer necessary to wait at the end of the turn: if we reach this point, we know the first transaction succeeded.

```
(behavior [] [msg]
  (atomic
    (send b :msg)
    …))
```

```
(behavior [] [msg]
  (atomic
    (send c :msg))
  …)
```

```
(behavior [] [m]
  …)
```

(d) When a message is sent in a transaction in a tentative turn, the message depends on this transaction, and not the encompassing turn. The third actor will only proceed when transaction 2 commits, which can only happen when transaction 1 committed as well.

Figure 5.6: Different cases of messages sent in transactions, and their dependencies. Each behavior runs in a different actor. The orange and green lines indicate 'tentative' sections of code: either a transaction (in an atomic block), or a turn that depends on a transaction. A message's dependency is indicated through its color and number (1 or 2).

action aborts. Sending a message is not expensive: it only consists of putting a message in the receiver's inbox (although this requires taking a lock). Immediately sending the message increases parallelism though: the receiver can already process the message before the sender's transaction has completed.

However, this implies that a message may now need to be retracted: when the transaction it was sent in aborts, the message and its effects need to be 'unsent'. We say that messages sent from within a transaction have a *dependency* on the transaction. There are now two types of messages: those sent outside a transaction have no dependency and are **definitive**, those sent within a transaction have a dependency and are **tentative**.

This has an impact on the receiver of a tentative message, as illustrated in Figure 5.6a. When an actor takes a tentative message from its inbox, the turn that processes it also becomes tentative: the message is processed, but the effects it causes should not be persisted yet. Even though this turn is not a transaction, it executes in the same 'tentative' manner, as its effects can roll back. When a **tentative turn** ends, the actor waits until the transaction on which it depends has committed. After a successful commit of its dependency, the actor can continue to its next turn, and we say the turn was successful. If its dependency aborts, the tentative turn fails and its effects are discarded. The actor then processes the next message in its inbox as if nothing happened.

Now that turns can be tentative and may roll back just like transactions, we need to look at which actions with a side effect can occur in this context, as they can roll back too. First, let us look at the constructs of actors. When `become` and `spawn` are used in a tentative turn, their effects are handled as above when they appeared in a transaction: their effect is delayed until the turn is successful, or discarded if it fails. On the other hand, `send` in a tentative turn immediately sends a message, but forwards the current dependency with it, so that its receiver will also depend on the original transaction (Figure 5.6b).

Second, a tentative turn may contain another transaction (as in Figures 5.6c and 5.6d). In other words, a first actor is executing a first transaction, in which it sends a message to a second actor, and when the second actor processes this message, it starts a second transaction. We say that the second transaction depends on the first. There are two serializations of these two transactions: either the first transaction commits before the second, or vice versa. However, because the second transaction is executed as a result of the first, the only valid serialization is the one in which the second transaction is preceded by the first. Therefore, the second transaction needs to wait before it commits, until the transaction it depends on has committed.

The different cases described in this section are summarized in Table 5.7. There are three execution contexts: (1) in a transaction, (2) out a transaction but in a tentative

	Not in a transaction		In a transaction
	Definitive turn	**Tentative turn**	
`behavior`	As before		
`become`	Delayed until end of turn		
`spawn`	Immediate	Delayed (pessimistic)	Delayed (pessimistic)
`send`	Immediate	Immediate (optimistic), dependency forwarded	Immediate (optimistic), dependency on tx
`atomic`	Immediate	Immediate (optimistic), but wait before commit	Immediate (optimistic) with closed nesting
`ref`, `deref`, `ref-set`	Not allowed	Not allowed	As before

Table 5.7: How the constructs on actors and transactions are executed in the three different contexts. Constructs in gray work as before, blue indicates the side effect is pessimistically delayed until the end of the tentative section (turn or transaction), green indicates the side effect is optimistically performed immediately but rolled back on conflict.

turn, and (3) out of a transaction in a definitive turn. We will use the term **tentative section** to refer to both transactions and tentative turns: portions of the code in which `spawn` and `send` are executed 'tentatively'.

5.3 Properties of Transactional Actors

Transactional actors guarantee isolation of the transactions, low-level race freedom, and deadlock freedom. We describe these here and discuss them more formally in Section 7.3 of Chapter 7.

5.3.1 Isolation of the Transactions

Transactional actors maintain the isolation guarantee of their transactions. In a naive combination of transactions and actors, the operations on actors inside a transaction cause side effects, thereby breaking isolation (as in the Vacation example in Section 5.1.2). Transactional actors incorporate these operations into the transaction, so that they succeed or fail depending on whether the transaction commits or aborts, as explained in Section 5.2.2. For other actors and transactions, all effects inside a transaction thus appear to take place at the moment the transaction commits, maintaining the isolation guarantee of the transactional system (serializability, opacity, or snapshot isolation).

As was illustrated in Figure 5.6c, when a (first) transaction causes a second transaction to start in another actor, the effects of the second transaction will only occur if and when the first transaction succeeded *and* the second transaction has no conflicts. These effects also occur in a single atomic step, and this always happens after the dependency committed. Hence, serializability is maintained, but there is only one valid serialization: the transaction with a dependency must be proceeded by its dependency.

Effects of a tentative turn that does not contain a transaction (these are the actors it spawned and its become construct) also only occur after the dependency succeeded. Again, serializability is maintained, as the order in which effects appear to other actors and transactions is equivalent to a serial execution.

5.3.2 Low-Level Race Freedom (Replacing the Isolated Turn Principle)

Transactional actors break the isolated turn principle: the 'isolation' assumption behind the isolated turn principle, defined in Section 2.5.3 on page 38, was that there is no shared state, which is obviously no longer true after transactional memory is introduced. For instance, when a turn contains a transaction followed by other code, the effects of the transaction already become visible before the turn has finished, and therefore the turn is no longer 'isolated'. (The two other assumptions – consecutive and continuous message processing – are upheld by transactional actors.)

However, transactional actors still prevent races, as all accesses to shared memory are protected by transactions. While the actor model guaranteed freedom from low-level races by prohibiting shared memory, transactional actors allow shared memory but require it to be encapsulated in a transaction. This extends the actor model with safe, shared memory.

Traditionally, actors guarantee a consistent view of the memory during a turn, as the only memory that can be accessed synchronously is the private memory of the current actor, which cannot be modified by another actor. Transactions guarantee a consistent view of the memory during a transaction: even when other transactions modify the memory, the transaction still sees the values that existed when it started. Transactional actors combine both: during a turn, the actor has a consistent view of its private memory, and during a transaction, it has a consistent view of the shared memory.

It is this property we call **freedom from low-level races**: it is not possible to introduce a race on the private memory of an actor within a turn, or on shared memory within a transaction. Race conditions on the *private* memory of actors can only occur because of a bad interleaving of turns, as they can in the original actor model; race conditions on *shared* memory can only occur because of a bad interleaving of transactions, as they can in the original transactional model. At the 'level' of turns and transactions, races are impossible.

In summary, using transactional memory, developers can reason about their code at the level of transactions; using the actor model, they can reason at the level of turns, and transactional actors enable developers to reason at the level of transactions *and* turns. They do not need to care about how the individual instructions within turns and transactions are interleaved, only about how turns and transactions as blocks are interleaved.

5.3.3 Deadlock Freedom

Transactional actors maintain the deadlock freedom offered by transactions and actors. This is not immediately obvious. To substantiate this, we need to look at the constructs that can block, and show that they cannot lead to a deadlock. We introduced blocking in two instances: at the end of a tentative turn, and at the end of a transaction in a tentative turn. In both cases, the current actor is blocked until the transaction on which it depends commits or aborts. Could this lead to a deadlock? In other words, could it happen that a transaction tx_a is waiting for another transaction tx_b to finish, and vice versa – a cyclical dependency between transactions?

First, two observations:

1. When a message with a dependency on tx_a is received, the turn that processes that message becomes tentative, depending on tx_a. *A dependency is thus always introduced when at the start of a turn.*
2. Each transaction is fully encapsulated by a turn. When a turn starts, no code is running so no transaction is active. If a transaction is started later in the turn, it will always be committed before the end of the turn, because its `atomic` block is fully encapsulated by the turn. Hence, no transaction is active when a turn starts or ends, so transactions cannot span turns. As dependencies are only introduced when a new turn starts, *no dependency will ever be introduced while a transaction is active.*

We can now see why **dependencies can never be cyclical**: a transaction or tentative turn can only depend on a transaction that started *before* it started. When a message is received with a dependency on tx_a, a dependency upon tx_a is introduced at the *start* of the turn that processes the message (observation 1). This message was sent from within transaction tx_a, which was therefore necessarily already running before the message was received. An inverse dependency is impossible: tx_a is already running or finished, and cannot acquire a new dependency while it is running (observation 2). This entails that dependencies always point to older transactions, and that time defines an order on dependencies. No cyclical dependencies can exist, so no deadlocks can occur.

5.3.4 Conclusion

In conclusion, transactional actors maintain the isolation of transactions, and provide freedom from low-level races in transactions and turns as a replacement for the isolated turn principle. Further, they maintain the deadlock freedom offered by transactions and actors. Using transactional memory, developers can reason about their code at the level of transactions; using the actor model, they can reason at the level of turns. Transactional actors enable developers to reason at the level of transactions *and* turns.

5.4 | Applicability to Other Actor Models

In this work, we build upon an actor model based on the work of Agha [1985]. One might wonder whether our solution also applies to other actor models. De Koster et al. [2016b] define four 'families' of actor models: Classic Actors, Active Objects, Processes, and Communicating Event Loops. We use a Classic Actors model in this dissertation. In this section, we explain that transactional actors can also be applied to actor models within the Active Objects and Communicating Event Loop families, but not to those within the Processes family.

Processes Actor models in the Processes family represent an actor as a process that runs from start to completion. They provide an explicit `receive` statement, which the programmer must manually call in a loop to process each message. This is in contrast to the Classic Actors we use, in which each actor has a behavior that is automatically executed for each message, and no explicit statement exists to fetch a message from the inbox. Erlang is an example of a language in the Processes family.

As explained in the previous section, the deadlock freedom guaranteed by transactional actors hinges on the fact that transactions cannot span turns. In our model, when an actor receives a message, no transaction can be active in that actor. This ensures that transactions cannot acquire new dependencies while they are running and therefore avoids cyclical dependencies. This is a result of the syntax: each turn is defined as a behavior, and an `atomic` block can therefore not span across multiple turns.

This is not true for actor models within the Processes family: the explicit `receive` statement can appear anywhere in the code, even in the dynamic extent of an a transaction. When a message is received while a transaction is running, it is no longer possible to guarantee both isolation and deadlock freedom.

```
Customer = ref {1, to_process, none}. % {customer id, reservation status, reserved car | none}
Car = ref {123, free}.                % {car id, id of customer that reserved it | free}
Log = ref [].                         % list of log messages

start_customer_actor() ->
    register(customer_actor, self()), % register self under name customer_actor
    atomic
        {CustomerId, Status, _} = deref Customer,
        car_actor ! {self(), reserve_car, CustomerId},
        receive
            {car_reserved, CarId} ->
                refset Customer {CustomerId, processed, CarId},
                refset Log ["Customer processed" | deref Log]
        end
    end.

start_car_actor() ->
    register(car_actor, self()), % register self under name car_actor
    atomic
        {CarId, _} = deref Car,
        receive
            {Sender, reserve_car, CustomerId} ->
                refset Car {CarId, CustomerId},
                Sender ! {car_reserved, CarId},
                refset Log ["Car reserved" | deref Log]
        end
    end.

spawn(start_customer_actor, []).
spawn(start_car_actor, []).
```

Listing 5.8: Example program containing `receive` in `atomic`, leading to a deadlock. This program is written in a fictitious extension of Erlang with transactional actors. (In Erlang, variables start with a capital (e.g. `Customer`), while symbols start with a small letter (e.g. `reserve_car`).)

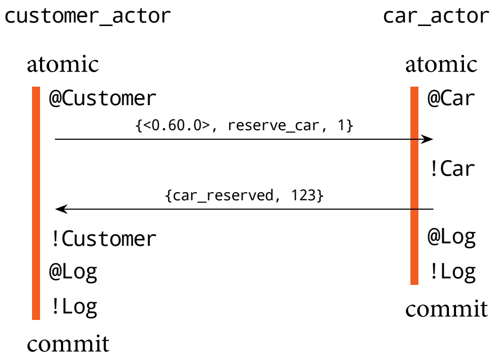


Figure 5.9: The effects of the program in Listing 5.8. There are two actors each containing a transaction. The notations @X and !X refer to read and write operations on a transactional variable X. The arrows indicate the messages that are exchanged between both actors.

We illustrate this using the example in Listing 5.8, which is written in a fictitious extension of Erlang with transactional memory. This example implements a car reservation system, in which a 'customer actor' sends a message to a 'car actor' to reserve a car. It works as follows. The two actors are spawned and each start a transaction. The actors exchange two messages, as illustrated in Figure 5.9. First, the customer actor sends a message to reserve a car (reserve_car) to the car actor. This introduces a dependency from the car actor's transaction on the customer actor's transaction. A reply to confirm the reservation (car_reserved) is sent back from the car actor to the customer actor. This introduces a dependency from the customer actor's transaction on the car actor's transaction. Thus, this leads to cyclical dependencies.

It is not possible to find a semantics that guarantees both serializability and deadlock freedom in this program:

- If we require isolation to be guaranteed, a deadlock will occur, as follows. Requiring serializability means this program must be equivalent to either (1) serially executing the first transaction followed by the second, or (2) vice versa. In the first case, the first transaction will wait forever for the reply of the second. In the second case, it is the second transaction that will wait forever for the message from the first.
- If we require deadlock freedom, both transactions must run simultaneously and they must both commit. However, both transactions modify the same variable Log, leading to a conflict. They cannot both commit without breaking the isolation guarantee.

Hence, it is impossible to come up with a semantics that guarantees both serializability and deadlock freedom. This is a result of allowing messages to be received *while* a transaction is active.

Active Objects and Communicating Event Loops The two other actor families, Active Objects and Communicating Event Loops, lack an explicit receive statement. Actor models in these families execute a method of an object in response to a message. If we extend these languages with transactional actors, the syntax will also not allow these methods to be encapsulated in an `atomic` block, and thereby deadlock freedom can be maintained.

Note that Active Objects and Communicating Event Loops typically allow mutable state, unlike Classic Actors. In a Classic Actors model, the internal state of an actor can only be changed using `become`, of which the effect is only visible on the next turn. Even if `become` occurs during a transaction, the transaction will have finished before the next turn starts, so it is possible to roll back its effect. In an Active Objects or Communicating Event Loops model, the internal state of an actor is modified with assignments on its object(s). To guarantee the isolation of the transaction, such assignments may not appear in a transaction. This could be enforced by the type system, as in Haskell.

In conclusion, we observe that whether transactional actors can be applied to an actor system depends on how the interface of an actor is defined. Classic Actors define an actor's interface by means of a behavior; Active Objects and Communicating Event Loops define them as methods on an object. In both cases, we can guarantee that no transaction is running when a turn is started, and the properties of transactional actors can be guaranteed (possibly with help from a type system). On the other hand, Processes introduce an explicit `receive` statement that may be used anywhere in the code, even nested in a transaction. In that case, it is not possible to guarantee both isolation and deadlock freedom.

5.5 Related Work

There is extensive related work on communication between transactions and on shared memory in actors. We provide an overview here.

Communication between transactions

Our work on transactional actors is not the first to advocate communication between transactions. We discuss three existing techniques.

Using **Transactions with Isolation and Cooperation** [Smaragdakis et al. 2007], a transaction can temporarily 'suspend' its atomicity and isolation, so that data can be exchanged with another transaction. To this end, the construct `Wait` is introduced; e.g. when `Wait(x > 0)` is called in a transaction, the transaction is suspended until x is positive. The variable x is read outside the context of the current transaction, so without isolation. A type system keeps track of such operations and requires them to be nested in an explicit construct, preventing unintentional mistakes. This construct can be used to introduce communication between transactions, but it breaks their isolation guarantee.

Similarly, Luchangco and Marathe [2011] extend transactional memory with **Transaction Communicators**, a special type of object through which transactions can communicate. Access to a communicator must be encapsulated in a `txcommatomic` block, which must be nested in a regular transaction. Again, the isolation of the transaction is broken, but doing this unintentionally is prevented by requiring an explicit construct. This system also introduces dependencies between transactions: when a communicator is read by transaction *a* after it was written to by transaction *b*, transaction *a* depends on *b* to commit successfully. Cyclical dependencies are possible, and lead to a deadlock if both transactions are guaranteed to always abort, e.g. when they both write to the same transactional variable.

Finally, Lesani and Palsberg [2011] introduce **Communicating Memory Transactions**: a combination of transactional memory with channel-based message passing. In an `atomic` block, the constructs `send` and `receive` can be used to communicate over a channel. This introduces a dependency from the receiving transaction on the sending transaction, very similar to our transactional actors. In case of cyclical dependencies, all transactions in the cluster will *attempt* to commit at the same time, while maintaining serializability. If no valid serialization exists, the program is invalid.

Communicating Memory Transactions maintain serializability, but cannot handle programs with cyclical dependencies. In Listing 5.10, we implement the example from Listing 5.8 using Communicating Memory Transactions. The same problem we described in the previous section occurs: while isolation is maintained, a deadlock occurs due to a cyclical dependency. Communicating Memory Transactions allow such a program to be written, even though it is invalid.

From these existing techniques, transactional actors adopt the idea of dependencies between transactions, allowing serializability to be guaranteed. However, deadlocks due to cyclical dependencies are avoided: there can never be a cycle in a dependency chain, as messages can only be sent in a transaction, and not received (see Section 5.4). Transactional actors thus build upon these techniques to provide a system with all desirable properties. This is summarized in Table 5.11.

Shared memory in actors

Our work is also not the first to consider how to safely share mutable state between actors.

```
customer := {1, to_process, none};
car := {123, free};
log := [];
ch := newChan;  // create a channel

let start_customer_process()
    atomic
        {customerId, status, _} = customer;
        ch send {self(), :reserve_car, customerId};
        {:car_reserved, carId} := ch receive;
        customer := {customerId, :processed, carId};
        log := [”Customer processed” | log]

let start_car_process()
    atomic
        {CarId, _} = car;
        {sender, :reserve_car, customerId} := ch receive;
        car := {carId, customerId};
        ch send {:car_reserved, carId};
        log := [”Car reserved” | log]

{ start_customer_process() } || { start_car_process() } // run both in parallel processes
```

Listing 5.10: The example from Listing 5.8 implemented using Communicating Memory Transactions. The syntax `ch` **send** `x` sends the message `x` over the channel `ch`; in the other process, `y := ch` **receive** receives it and assigns it to `y`. Messages are sent synchronously: both processes must rendez-vous to pass the message.

	Isolation	Deadlock freedom
Transactions with Isolation and Cooperation	✗	✗
Transaction Communicators	✗	✗
Communicating Memory Transactions	✓	✗
Transactional Actors	✓	✓

Table 5.11: Comparison between techniques for communicating transactions.

De Koster et al. [2016a] extend the actor model with **domains**, containers that can be accessed from multiple actors. Access must be encapsulated in a `when_shared` or `when_exclusive` block, the former gives shared read-only access while the latter gives exclusive write access. The code in these blocks is executed asynchronously to prevent deadlocks.

Sharing actors [Lesani and Lain 2013] also share state between a single writer actor and multiple reader actors, by replicating the data. Sharing actors encode the replication pattern discussed in Section 5.1.1. When the (single) writer updates the shared data, a message is sent to the readers to update their copies.

Morandi et al. [2014] separate active processors (similar to actors) into their executing thread and their data. They introduce **passive processors**, which consist of only the data without the thread. Active processors can access this data by assuming the identity of a passive processor, giving them exclusive access to that processor's data.

Last, Pony [Clebsch et al. 2015] allows memory to be shared between actors, using **deny capabilities** to statically guarantee there is only one writer to a shared memory location and thereby preventing races. Similarly, Encore [Brandauer et al. 2015] uses capabilities to allow memory to be shared between active objects. Encore plans to support capabilities for both pessimistic and optimistic concurrency, although at the time of writing the exact semantics have not been defined.

All of these approaches introduce shared state in actor systems, but they only allow one writer per container at a time. To allow concurrent but safe access, it is important to split data correctly over these containers. For the Vacation example described in Section 5.1.2, this is not evident: how should the flights be split, so that multiple customers can access all of them consistently at the same time? By using transactional memory, our approach allows multiple actors to write to shared memory, without requiring data to be split into containers: the transactional system ensures consistency.

Transactional communication

Transactors [Field and Varela 2005] encapsulate changes to actors' local state and their communication in transactions. **Communicating transactions** [de Vries et al. 2010] also coordinate distributed processes using transactions. Both use transactions to ensure that state that is distributed over multiple actors or processes can be updated consistently, but do not share memory. Finally, using **transactional events** [Donnelly and Fluet 2006] a set of send and receive operations can be encapsulated in a transaction that ensures they are either all executed or none are. This is used to implement communication patterns such as three-way rendez-vous. Again, there is no notion of shared memory. In these three languages, the transactional operations are the sending and receiving of messages, not reading and writing shared memory locations.

5.6 | Summary

In this chapter, we explored the combination of actors and transactions. We motivated why combining them is useful. On the one hand, actors can use transactional memory to safely share memory. On the other hand, transactions can send and receive messages to distribute and coordinate work that can be executed in parallel. However, through an example, we saw that a naive combination of actors and transactions breaks the guarantees that those models provide when used separately, breaking the assumptions of the developer.

We introduce **transactional actors**: a combination of transactions and actors that provides the same constructs as its constituent models, but also defines their semantics when they are combined. We systematically considered each construct to ensure it has the expected semantics.

Transactional actors realize this by sending messages *tentatively*. When a message is sent in a transaction, a dependency on that transaction is attached to the message. The receiver of the message processes it tentatively: it optimistically processes the message, but ensures that all effects of the message are undone if its dependency aborts.

As a result, transactional actors maintain the isolation of transactions and they guarantee deadlock freedom. Moreover, they ensure freedom from low-level races: developers can reason about their program at the level of transactions and turns, and do not need to care how the individual instructions within turns and transactions are interleaved.

In Chapter 7, we will formalize the semantics of transactional actors. In Chapter 9, we will demonstrate the performance benefits of transactional actors using the Vacation program. We will see that transactional actors can increase performance by distributing transactions over multiple actors, introducing more fine-grained parallelism and lowering the chance and cost of conflicts.

6

Chocola: a Language That Unifies Futures, Transactions, and Actors

This chapter unifies the three concurrency models we examined in this dissertation into one framework, called Chocola. In Section 6.1, we first consider the third combination we have not discussed yet: futures and actors. Using an example, Section 6.2 shows how all three concurrency models can be unified into one framework: Chocola. Next, in Section 6.3, we summarize the guarantees of Chocola and compare them to the guarantees of naive combinations of the models.

6.1 | Combining Futures and Actors

In the previous two chapters, we extensively discussed the combination of transactions with futures and actors respectively. The third combination – futures and actors – poses fewer problems, as we will discuss in this section. Table 6.1 illustrates the combinations of these two models.

Combining actors and futures can be useful. As we saw in Section 3.1 (page 45), in a study of 15 Scala programs that use actors, 12 (80%) combine these with futures and/or threads [Tasharofi et al. 2013]. On the one hand, futures can be introduced in an actor to process a turn in parallel: this is **intra-actor parallelism** (the lower-left cell of Table 6.1). On the other hand, actors can be used in a program with futures

→ in ↓	Future	Transaction	Actor
Future	Nested futures (Section 3.3.3) Det	Parallel transactions (Section 4.1) ~~Det~~ Iso Pro	Communication in future (Section 6.1) ~~Det~~ ~~ITP~~ DLF
Transaction	Parallelism in transaction (Sections 4.2–4.4) ~~Det~~ ~~Iso~~ Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) ~~Iso~~ Pro ~~ITP~~ DLF
Actor	Parallelism in actor (Section 6.1) Det ~~ITP~~ DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ DLF	Actors (Section 3.3.3) ITP DLF

■ **Guarantees**

Futures: Det Determinacy

Actors: ITP Isolated Turn Principle DLF Deadlock Freedom

Table 6.1: Combinations of futures and actors.

to introduce communication between parallel tasks (the upper-right cell of Table 6.1). Furthermore, when a program using one model includes a library that uses the other, the models are combined implicitly.

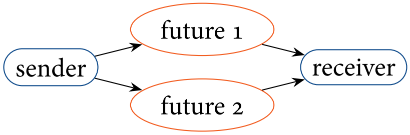
We examine the effect of the combinations on the properties of each model: determinacy of futures (Section 6.1.1), deadlock freedom of actors (Section 6.1.2), and the isolated turn principle of actors (Section 6.1.3).

6.1.1 Determinacy of Futures

When futures are created in an actor, but they do not contain any operations on actors, the future remains equivalent to its serial elision and therefore determinacy is guaranteed (the lower-left cell of Table 6.1). However, when certain actor constructs (`send` and `become`) are used in a future, determinacy can be broken (the upper-right cell of Table 6.1). We discuss the four constructs provided by actors:

`send` Listing 6.2a demonstrates that using `send` in a parallel task can break determinacy. In the example, two futures are forked that both send a message to a second

```
(def sender
  (behavior [] []
    (fork (send receiver "hello"))
    (fork (send receiver "world"))))
```

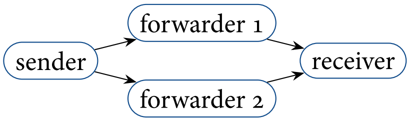


(a) Nesting send in fork leads to non-deterministic results: which of the two messages "hello" or "world" is sent first can vary between executions.

```
(def forwarder
  (behavior [] [a msg]
    (send a msg))) ;forward message

(def forwarder1 (spawn forwarder))
(def forwarder2 (spawn forwarder))

(def sender
  (behavior [] []
    (send forwarder1 receiver "hello")
    (send forwarder2 receiver "world")))
```



(b) The same issue can occur when only using actors: when two messages follow different routes from sender to receiver, there are no guarantees on their arrival order.

Listing 6.2: Nesting send in fork leads to non-determinism (a), but non-determinism can also occur when using the actor model individually (b).

actor. The messages can arrive in two orders ("hello" "world" or "world" "hello"), non-deterministically.

Breaking determinacy for this combination is inevitable. When futures – a deterministic model – are combined with any non-deterministic model, their determinacy will always be broken. We argue that this is no problem as long as this only occurs in those places where the programmer explicitly writes the constructs of the non-deterministic model: it is clear to the programmer that determinism is no longer guaranteed when a construct of a non-deterministic model, here send, is used. The behavior for the example using futures is similar to the behavior that occurs when two actors are spawned that each send a message, as shown in Listing 6.2b. This program uses only actors and *also* has a non-deterministic result. As illustrated, both programs lead to a similar structure at run time and have a non-deterministic result, whether futures are used or not.

become Listing 6.3a demonstrates another problem, this time when using become in a parallel task. Our semantics of actors defined that, when multiple become statements occur in a turn, it is the one that is executed last that determines the next behavior of the actor. In this example, this in not deterministic: which become statement is executed last depends on how the two tasks are scheduled at run time.

In this case, we can retain determinism though: we will say that *the* become *state-*

```
(def counter
  (behavior [i] []
    (let [a (fork (become counter 1))
          b (fork (become counter 2))]
      (join a)
      (join b))))
```

(a) Nesting become in fork leads to non-deterministic results: what will the next behavior of this actor be?

```
(def behavior-0
  (behavior [] []
    (let [a (fork (spawn behavior-a))
          b (fork (spawn behavior-b))]
      (send (join a) :msg)
      (send (join b) :msg))))
```

(b) Nesting spawn in fork does not break (observable) determinacy.

Listing 6.3: Nesting become and spawn in fork.

ment of the future that is joined last is the one that takes effect (so in the example the next value of i will be 2). Hence, the result of Listing 6.3a is the same as if the futures were elided. Essentially, we consider become as an effect of a future, and when the future is joined, the effect is merged into its parent. This is similar to our solution for combining transactions and futures from Chapter 4. There, each transactional future stores its effects on the transactional memory locally, and a future's transactional effects are merged into its parent when the future is joined. Here, each future stores the effect of its become locally, also merging the effect when it is joined.

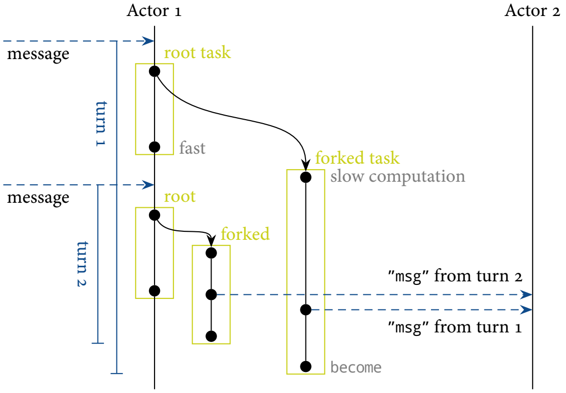
spawn Nesting spawn in a future does not break determinacy: it spawns a new actor, but it is impossible to *observe* the order in which actors were spawned. This is a result of the fact that the only way to interact with another actor is through *asynchronous* messages using send. In Listing 6.3b, the order in which both actors are spawned at run time may vary, and the order in which they process their message may vary, but this is a result of the asynchronous nature of messages in the actor model. It is impossible to observe which of the two actors was spawned first.

behavior As behavior does not cause a side effect, using it in a future maintains determinacy.

6.1.2 Deadlock Freedom of Actors

Combining actors and futures preserves deadlock freedom. There is only one blocking construct, join, which waits for a future to resolve. A deadlock cannot occur, as this would require cyclical dependencies between futures. Even when combined with actors, futures still form a spawn tree which does not contain any cycles (as in Section 3.3.3, page 63): every turn starts with one 'root' task which can spawn tasks and forms a spawn tree per turn.

```
(def escape-example
  (behavior [i] []
    (fork ; This task ‘escapes’ its turn
      (slow-computation)
      (send actor-2 ”msg”)
      (become escape-example 1))
    (fast-computation)))
```



Listing 6.4: This actor creates a task in which a message is sent and the actor’s behavior is changed. Meanwhile, the root task may finish its work and continue to the next turn. The task therefore ‘escapes’ the turn: this is the escaping task problem. In which order are the messages sent, and what happens when the become is reached? The isolated turn principle is violated.

6.1.3 Isolated Turn Principle of Actors

A naive combination of actors and futures breaks the isolated turn principle. As explained in Section 2.5.3 (page 38), the isolated turn principle is a consequence of three restrictions of the actor model: isolation, consecutive message processing, and continuous message processing. We discuss the effect of using futures in an actor on each:

Isolation prohibits shared memory; this remains the case when futures are introduced.

Continuous message processing prohibits the use of blocking operations to guarantee that a turn always runs to completion uninterrupted. While futures introduce the blocking construct `join`, this operation is always guaranteed to complete because the futures in a turn form a spawn tree, as explained in the previous section.

Consecutive message processing requires that an actor must process its messages one by one, without interleaving turns. This requirement can be broken by introducing futures. We call this the **escaping task problem**.

Listing 6.4 illustrates the escaping task problem. An actor creates a task which is never joined by the root task. As a result, the root task finishes its work and proceeds to the next turn while the child task is still running. The two turns overlap, interleaving the processing of two messages, thus violating the isolated turn principle. Here, this leads to two unexpected results: (1) the child task sends a message that can arrive *after* messages sent in the next turn, and (2) the become in the escaped task can still change the behavior of the actor *after* the next turn has already started, with the old behavior.

Fortunately, the isolated turn principle can be reintroduced using a simple requirement: any future created in an actor must be joined before the turn ends. This ensures that only one task (the root task) is running when the turn ends, and that the side effects of all futures created in the turn have occurred and all their effects have been merged into the root task. All tasks end when the turn ends, so no more effects can take place during the next turn. Thus, an actor can exploit internal parallelism using futures while maintaining the isolated turn principle.

We believe this requirement is not overly restrictive: it only applies when a future is forked in a turn but its result is never used in that turn. Furthermore, a similar requirement existed for transactional futures: we required that all futures created in a transaction were joined before the transaction ends. Here, we require that all futures created in a turn are joined before the turn ends. Thus, both techniques provide a consistent model to the developer.

6.1.4 Conclusion

The combination of actors and futures breaks determinacy when messages are sent in futures, but as a non-deterministic construct is used explicitly, this is clear to the developer. Other problems can be avoided after two adjustments to the semantics:

- Chocola allows futures to be forked in an actor, enabling *intra-actor parallelism*, but requires that all futures are joined in the turn in which they were created. The tasks thus form a spawn tree within each turn, and each task will be finished and joined before the turn ends. This ensures that the isolated turn principle is maintained.
- A `become` in a parallel task is a side effect that is part of its future. When it is joined, the effect is propagated up to the task performing the join. Eventually, the effect reaches the 'root task' of the turn. Later `become`s still overwrite earlier ones. This ensures determinacy is maintained in this case (but not in general when combining both models).

The combination of actors and futures thus provides a familiar semantics that maintains the guarantees of the separate models wherever possible, after only minimal changes.

6.1.5 Related Work

Imam and Sarkar [2012] combine actors with the async–finish model (AFM), which is similar to futures. We compare their combination to ours. The AFM provides two constructs: (1) `async` starts a new asynchronous task, possibly delivering its end result to a future (it is similar to our `future` construct), and (2) `finish` encapsulates several async constructs and waits until they have all finished before proceeding. Their motivations for combining these two models are similar to ours: by using the AFM in an actor, messages can be processed in parallel, while by using operations on actors in the AFM, these operations can be coordinated.

When tasks that are spawned in an actor are encapsulated by a finish block, this model is similar to ours: all tasks run fully within their encapsulating turn. However, Imam and Sarkar [2012] also allow a task to escape the actor in which it is spawned. In that case, race conditions are avoided by prohibiting these tasks from modifying the internal state of the actor.

When actors are spawned in tasks that are encapsulated by a finish block, these actors must start, execute, and terminate before the encapsulating finish block proceeds. Thus, this mechanism can be used to coordinate actors: spawning multiple actors in a finish block ensures they must all terminate before the program proceeds. This is not possible in our system, as we do not have a similar finish construct and our actors do not terminate (when they no longer receive messages, they remain idle forever). In our system, a similar coordination mechanism can be implemented by passing messages between the actors.

6.2 Chocola: Composable Concurrency Language

In this section, we present Chocola, the composable concurrency language: a unified framework of futures, transactions, and actors that maintains the semantics and guarantees of its constituent models whenever possible. We discuss how the various concepts from the previous chapters are unified in Chocola (Section 6.2.1). Next, we demonstrate how the three models are combined in an example (Section 6.2.2).

6.2.1 Chocola's Linguistic Concepts

In Chocola, a program starts as a single (main) actor containing a single (root) task that evaluates the code. The program can then spawn actors, fork futures, and create transactions. We summarize the main concepts and constructs:

Actors

- An **actor** is an entity that runs concurrently with other actors. It has an address, an inbox, and a current behavior. The **inbox** is a queue of messages. Actors are created using spawn, which is given an initial behavior.
- A **behavior** defines how an actor acts when receiving a message. It contains code, created using the construct behavior, and a list of parameters. The current actor's behavior can be changed using become.

- A **message** is a list of values that can be sent to an actor using `send`, which appends it to the actor's inbox. When a message is sent from within a transaction or a tentative turn, it is **tentative** and has a dependency on a transaction; otherwise it is **definitive**.
- An actor consecutively processes each message sitting in its inbox, by evaluating its current behavior with that message. This process is called a **turn**. A turn that is the result of a tentative message is a **tentative turn**.
- In a transaction and in a tentative turn, `spawn` and `become` are delayed; we call these **effects on actors** that are gathered to be executed later.

Transactions

- A **transaction** is a section of the code that can access shared memory. It is encapsulated in an `atomic` block.
- The shared memory is represented using **transactional variables**. These can only be manipulated in transactions, using `ref`, `deref`, and `ref-set` to create, read, and write them. (Outside a transaction, these constructs raise an error.)
- A **transactional context** is a structure that contains data related to the current transaction: (1) its **snapshot**: a (conceptual) copy of the transactional memory before the transaction started, (2) its **local store**: the modifications the transaction made to the transactional memory, and (3) the **effects on actors** that occurred during the transaction and will be executed if and when the transaction commits. Note that it is not the transaction itself but its tasks that each contain a transactional context.

Futures

- A **task** is a section of the program that runs concurrently with the rest of the program. It can be created using `fork`, which returns a future. A **future** is the placeholder for the result of its corresponding task. This value can be retrieved using `join`, which blocks until the task has finished and then returns its result.
- When a task is forked in a transaction, we say it is a **transactional task**, which has an associated **transactional future**. Each transactional task has a transactional context, which it adopted from its parent when it was forked and which is merged into its parent when it is joined. Tasks outside transactions are **non-transactional**.

> Chocola does not introduce new syntactical constructs, nor does it change the semantics of its constituent models when used separately. The novelty of Chocola is that it defines the semantics of the constructs of these concurrency models when they are combined with one another.

When to use futures or actors for parallelism inside a transaction

Both futures and actors introduce parallelism. When parallelizing a program with transactions, one might wonder whether to use transactional futures or transactional actors. At first glance, both might seem like similar mechanisms: they can both be used to parallelize the internals of a transaction. However, both mechanisms function quite differently and have different use cases:

- A transactional future runs completely *within* the context of the transaction it was created in. It has its own copy of the heap on which it acts in isolation, but its changes will always be joined back into its parent later. Hence, the modifications to transactional memory that occur in different transactional tasks of the same transaction will be committed at the same time.
- In contrast, transactional actors make it possible to 'escape' a transaction. When a message is sent in a transaction, it carries a dependency on that transaction. However, the turn that is executed as a result of the message, while dependent on the transaction, does not run within the original transaction's context. A second transaction can be started in this turn, and this second transaction will depend on the first, but both run in isolation and have their own copy of the heap, and both commit separately.

These differing semantics are a result of the different use cases of the two models. Futures are used to speed up a deterministic calculation. Thus, when they appear in a transaction, they remain part of the transaction while locally enabling parallelism. On the other hand, actors represent separate components in the application that occasionally communicate using messages. When a message is sent in a transaction, the sender signals to another actor that it must act, but the receiver runs separated from the sender. Which technique to use therefore depends on the intention: to locally parallelize a calculation, use futures; to communicate with a separate component, use actors.

6.2.2 Example

In Chapter 5, we introduced a flight reservation system that combined actors and transactions, based on the Vacation benchmark from the STAMP benchmark suite. In Listing 6.5, this example is extended with futures to increase its parallelism.

This program contains two types of actors. First, a set of *customer actors* receive messages from each customer and process their reservations, using `customer-behavior`. A reservation consists of a transaction in which two flights, a hotel room, and a car are reserved, and the customer's password is generated (lines 31–36). For each of the four items, a message is sent to a set of *reservation actors*, with the behavior `reserve-behavior` (lines 20–25). The reservation actors reserve an item, for instance a

```
(def flights …)
(def rooms   …)
(def cars    …)
(def customers …)

(defn parallel-filter [f xs]
  ; split xs into partitions
  ; fork a task for each partition
  ; in each task, filter the partition
  ; join the tasks and merge their results
  )

(defn reserve-flight [orig dest date n-seats]
  ; Finds the cheapest flight from orig to dest on date, and reserve n-seats.
  (atomic
    (let [filtered (parallel-filter (fn [f] …) flights)
          cheapest (get-cheapest filtered)]
      (ref-set cheapest (occupy cheapest n-seats)))))

(def reserve-behavior
  (behavior [id] [type & args]
    (case type
      :flight (apply reserve-flight args)
      :room   (apply reserve-room   args)
      :car    (apply reserve-car    args))))

(def customer-behavior
  (behavior [id] [c]
    ; Process a customer: find and reserve two flights (outbound and return), a car, and a room
    ; (all in worker actors), and generate a password.
    (atomic
      (send (rand-nth secondary-workers) :flight (:orig @c) …)
      (send (rand-nth secondary-workers) :flight (:dest @c) …)
      (send (rand-nth secondary-workers) :room   (:dest @c) …)
      (send (rand-nth secondary-workers) :car    (:dest @c) …)
      (ref-set c (assoc @c :password (generate-password))))))
```

Listing 6.5: Code snippet of a flight reservation program that combines futures, transactions, and actors.

flight (lines 13–18), by filtering the items using the customer's criteria (e.g. a flight's origin and destination), finding the cheapest item, and reserving that item. This is protected using a transaction to ensure that an item cannot be reserved multiple times. The difference between this example and the one in the previous chapter is that items are filtered in parallel: the list of items is partitioned and each partition is filtered concurrently, each partition returning a future (lines 6–11).

Thus, this application combines the three models:

- It uses *actors* to concurrently process requests from different customers and for different items. A message-passing model naturally matches this use case: customers that want to initiate a reservation send a message to an actor, in an event-driven way.
- Two sections of the code access shared memory and must therefore be protected using *transactions*. First, when processing a customer we must ensure either all items are reserved or none. Second, we must prevent items from being reserved multiple times. Transactions ensure safe access to shared memory.
- *Futures* are used to exploit parallelism in deterministic operations, here filtering a list. Futures guarantee that the parallel version is equivalent to its serial elision.

Figure 6.6 visualizes this program. We show two actors: one customer actor and one reservation actor. (There might be more of each type, to process requests concurrently.) In the customer actor, a transaction is started in which four messages are sent and a transactional variable is modified (corresponding to lines 32–36 in Listing 6.5). The reservation actor starts a transactions to reserve the requested item, and in this transaction the function `parallel-filter` creates a number of subtasks that return futures.

The example demonstrates how the concepts introduced throughout this dissertation work together. The messages sent in the first transaction are tentative, so that they only succeed if the encapsulating transaction commits successfully, using the transactional actors from Chapter 5. In the second transaction, several futures are forked: these are transactional futures with access to the encompassing transactional context, as in Chapter 4. Finally, each turn of each actor consists of a root task that may fork and join subtasks, as in Section 6.1.

This application also illustrates how Chocola can be used to implement cloud applications that process large amounts of data from many users. When web clients connect to the application we create a customer actor for each, and the client's requests are converted to messages that are sent to the customer's actor. Each actor thus processes incoming requests independently. When access to shared data is required, this can be implemented safely using transactions. Finally, 'Big Data' is processed in parallel using futures. This maps onto the web's typical three-tier design, in which the front-end

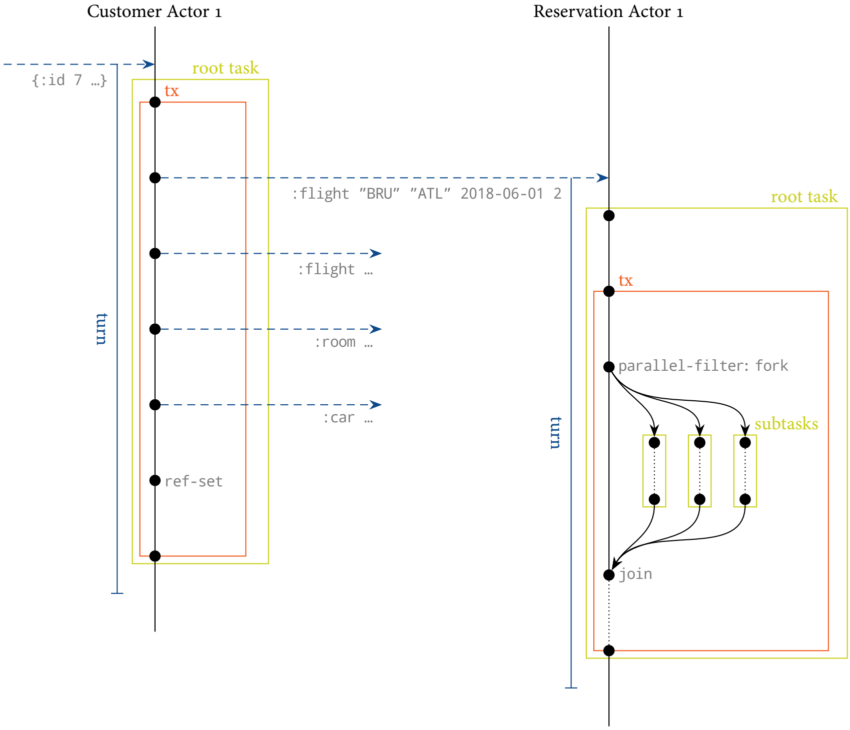


Figure 6.6: Diagram illustrating the actors, futures, and transactions created for the program in Listing 6.5.

that communicates with users uses actors, the application logic is parallelized using futures, and the data back-end is stored in transactional memory.

6.3 Guarantees of Chocola

Table 6.7 summarizes the guarantees of Chocola and compares them to the guarantees of naive combinations of its constituent models (as discussed in Section 3.3 and first shown in Table 3.9 on page 62). We revisit each guarantee.

Determinacy and Intratransaction Determinacy When used separately, futures guarantee determinacy. Chocola sometimes breaks this guarantee. We distinguish two cases: when futures are the outer model (i.e. transactions or actors are used in a future, the first row in the table) or the inner model (i.e. futures are used in a transaction or actor, the first column).

- When futures are the outer model, determinacy is no longer guaranteed (first row in the table). This is inevitable: non-deterministic models introduce non-determinism, even when used in a deterministic model. However, as we argued in Sections 4.1 and 6.1, this is not unexpected because the developer must explicitly use a construct from a non-deterministic model to break determinacy, and it is only in those places that determinacy is broken.
- In contrast, when using futures as the inner model, determinacy is expected (first column in the table). For instance, when a library that uses futures is embedded in a program that uses another model, the developer of the library still assumed determinacy. Using futures in another future (Section 3.3.3) or in an actor (Section 6.1) maintains determinacy. In a naive combination of futures and transactions determinacy is broken, which is why we introduced *intratransaction determinacy*: an alternative property that guarantees determinacy within each transaction (Section 4.4).

Isolation Transactions guarantee a form of isolation, such as serializability, opacity, or snapshot isolation. When they are used in a future or actor, this guarantee is maintained (second column in the table). On the other hand, when a future is forked or a message is sent within a transaction, a naive combination may break isolation (second row in the table). Chocola ensures isolation remains guaranteed even for these problematic combinations by incorporating any side effects into the transaction (discussed in Chapter 4 and 5). Chocola implements snapshot isolation, which requires that:

- Throughout a transaction, read operations get a consistent view of the memory, referred to as its snapshot. In Chocola, all tasks spawned in a transaction share this snapshot.

→ in ↓	Future	Transaction	Actor
Future	Nested futures (Section 3.3.3) Det	Parallel transactions (Section 4.1) ~~Det~~ Iso Pro	Communication in future (Section 6.1) ~~Det~~ ITP DLF
Transaction	Parallelism in transaction (Sections 4.2–4.4) ~~Det~~ → ITD Iso Pro	Nested transactions (Section 3.3.3) Iso Pro	Communication in transaction (Chapter 5) Iso Pro ~~ITP~~ → LLRF DLF
Actor	Parallelism in actor (Section 6.1) Det ITP DLF	Shared memory in actor (Chapter 5) Iso Pro ~~ITP~~ → LLRF DLF	Actors (Section 3.3.3) ITP DLF

Guarantees

Futures:	Det	Determinacy		
Transactions:	Iso	Isolation	Pro	Progress
Actors:	ITP	Isolated Turn Principle	DLF	Deadlock Freedom
Tx futures:	ITD	Intratransaction Determinacy		
Tx actors:	LLRF	Low-level Race Freedom		

Legend

Det	Maintained guarantee, even in a naive combination
~~Det~~	Broken guarantee, inevitable even in Chocola
Iso	Broken in a naive combination but maintained in Chocola
~~Det~~ → ITD	Broken guarantee Det has been replaced with guarantee ITD in Chocola

Table 6.7: Guarantees of the combined models.

- A transaction only succeeds if none of its writes conflict with any writes by other transactions that committed since the snapshot. In Chocola, all writes by all tasks that were spawned in the transaction are committed at the same time, making it possible to satisfy this requirement. Furthermore, if the transaction aborts, any messages sent during the transaction and their effects are rolled back.

Thus, Chocola guarantees safe access to shared memory in all circumstances.

Isolated Turn Principle and Low Level Race Freedom The isolated turn principle guarantees that, once a turn started, it will always run to completion, in isolation. This allows developers to reason at the level of turns: it does not matter in which order the individual instructions of different turns are interleaved, only how the turns as blocks are interleaved. The isolated turn principle assumed no shared memory and no internal parallelism, which is obviously no longer true in Chocola; it is therefore impossible to maintain. Instead, Chocola provides Low Level Race Freedom: a guarantee that there can be no races *within* a turn or a transaction (introduced in Section 5.3). This property provides a similar guarantee to developers, allowing them to reason at the 'level' of turns and transactions. Again, it does not matter in which order the individual instructions within a turn or a transaction are interleaved, only how turns and transactions are interleaved as larger blocks.

Progress and Deadlock Freedom Transactions guarantee progress and actors guarantee deadlock freedom. Even when these models are combined with others, these guarantees are maintained. This is a result of the fact that Chocola only contains one blocking operation, `join`, which always completes because futures form a spawn tree without cycles (discussed in Sections 4.4.1, 5.3.3, and 6.1.2).

6.4 Conclusion

Chocola is a language that combines futures, transactions, and actors while maintaining their semantics and guarantees whenever possible. In the previous chapters, we presented the various concepts at its base; in this chapter we combined them into one framework. The following chapters focus on Chocola's formal semantics (Chapter 7), its implementation (Chapter 8), and its performance benefits (Chapter 9).

7

PureChocola: an Operational Semantics

This chapter presents PureChocola: a formal operational semantics of Chocola. We start, in Section 7.1, by describing its syntax and how to model its program state. In Section 7.2 we list all reduction rules, specifying the constructs of futures, transactions, and actors. In Section 7.3, we show how Chocola's guarantees can be inferred from the formal semantics. We created an executable implementation of PureChocola using PLT Redex, and in Section 7.4 we explain how we used it to verify isolation and intratransaction determinacy. Finally, in Section 7.5 we list the differences between the formal semantics of PureChocola and the actual implementation of Chocola.

7.1 | Syntax and Program State

This section defines the syntax of PureChocola (Section 7.1.1), how its program state is modelled (Section 7.1.2), and the evaluation contexts (Section 7.1.3). Next, it defines some helper functions to extract elements from the program state (Section 7.1.4) and two special operators $|$ and $+=$ that merge effects on transactional memory and actors (Section 7.1.5).

7.1.1 Syntax

Throughout this dissertation, we did not introduce any new syntactical constructs; we merely defined their semantics in certain (new) contexts. Thus, the syntax of the combination of the three models simply consists of their separate syntaxes (as described

$c \in$ Constant	::=	$\texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{""} \mid \texttt{"a"} \mid \ldots$	
$x \in$ Variable			
$f \in$ Future			
$r \in$ TVar			
$a \in$ Address			
$b \in$ BehaviorDef	::=	$\texttt{behavior}\ [\overline{x}_{\text{beh}}]\ [\overline{x}_{\text{msg}}]\ e$	Behavior definition
$v \in$ Value	::=	c	
	$\mid$	x	
	$\mid$	$\texttt{fn}\ [\overline{x}]\ e$	Anonymous function
	$\mid$	f	Future
	$\mid$	r	Transactional variable
	$\mid$	a	Address of actor
	$\mid$	b	Behavior
$e \in$ Expression	::=	v	
	$\mid$	$(e\ \overline{e})$	Function application
	$\mid$	$\texttt{if}\ e\ e\ e$	
	$\mid$	$\texttt{let}\ [x\ e]\ e$	
	$\mid$	$\texttt{do}\ \overline{e};\ e$	
	$\mid$	$\texttt{fork}\ e$	Fork a future
	$\mid$	$\texttt{join}\ e$	Join a future
	$\mid$	$\texttt{atomic}\ e$	Transaction
	$\mid$	$\texttt{atomic}\star\ e$	In transaction (intermediate state)
	$\mid$	$\texttt{ref}\ e$	Create a TVar
	$\mid$	$\texttt{deref}\ e$	Read a TVar
	$\mid$	$\texttt{ref-set}\ e\ e$	Write to a TVar
	$\mid$	$\texttt{spawn}\ e\ \overline{e}$	Spawn an actor
	$\mid$	$\texttt{become}\ e\ \overline{e}$	Become a behavior
	$\mid$	$\texttt{send}\ e\ \overline{e}$	Send a message
	$\mid$	$\texttt{self}$	Address of current actor

Figure 7.1: The syntax of PureChocola.

throughout Chapter 2) combined. The complete syntax of PureChocola is listed in Figure 7.1.

7.1.2 State

The program state of PureChocola fuses the program states of the separate models. Table 7.2 recaps how the program state was represented in the semantics of the separate models in Chapter 2. Figure 7.3 defines the program state and its elements for PureChocola. The program state consists of all actors and tasks that have been spawned up to this point and three shared data structures. We discuss each.

	Program state	**Main elements**
Futures	$\langle \mathrm{T} \rangle$ (tasks)	$\mathsf{task} ::= \langle f, e \rangle$
Transactions	$\langle \mathrm{T}, \tau, \sigma \rangle$ (tasks, transactions, heap)	$\mathsf{task} ::= \langle f, e, \mathsf{n}^? \rangle$ $\mathsf{tx} ::= \langle \circ, \overleftarrow{\sigma}, \overleftarrow{e}, \delta \rangle$
Actors	$\langle \mathrm{A}, \mu \rangle$ (actors, inboxes)	$\mathsf{act} ::= \langle a, e, \mathsf{beh} \rangle$

Table 7.2: Reminder of the most important parts of the semantics of the three separate models described in Chapter 2.

Actors An actor consists of:

- a: the unique address of the actor.
- $f^?_{\mathrm{root}}$. Previously, an actor contained the expression it was evaluating. Now, however, an actor can contain multiple concurrent tasks, each of which stores the expression it is evaluating. Therefore, now when an actor processes a message, it creates one root task and stores the associated future instead. This task may fork more tasks, which are associated with this actor but not explicitly stored in it. Between turns, when the actor is idle, this is $\bullet$.
- beh: the next behavior of the actor. Like before, this consists of the code that defines the behavior (b) and the values for its internal state ($\overline{v}$).
- $\mathsf{n}^?_{\mathrm{dep}}$: when an actor is processing a message with a dependency, this will contain the identifier of the transaction on which the message depends. In that case, the actor is in a tentative turn; in a definitive turn (and between turns) this value is $\bullet$.

Tasks Tasks are extended to contain information about the effects they have on other futures, actors, and transactions. This information is needed to verify correctness and to execute delayed effects. A task contains:

- f: the future associated with the task.

Program state	p	$::=$	$\langle \mathrm{A}, \mathrm{T}, \mu, \tau, \sigma \rangle$
Actors	$\mathrm{A} \subset \mathsf{Actor}$		
Tasks	$\mathrm{T} \subset \mathsf{Task}$		
Inboxes	$\mu : \mathrm{Address} \rightharpoonup \overline{\mathsf{Message}}$		
Transactions	$\tau : \mathsf{TransactionNumber} \rightharpoonup \mathsf{Transaction}$		
Transactional heap	$\sigma : \mathrm{TVar} \rightharpoonup \mathrm{Value}$		
Actor	$\mathsf{act} \in \mathsf{Actor}$	$::=$	$\langle a, f^?_{\mathrm{root}}, \mathsf{beh}, \mathsf{n}^?_{\mathrm{dep}} \rangle$
Task	$\mathsf{task} \in \mathsf{Task}$	$::=$	$\langle f, a, e, \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^? \rangle$
Transaction	$\mathsf{tx} \in \mathsf{Transaction}$	$::=$	$\langle \circ, \overleftarrow{e} \rangle$
Spawned and joined futures	$\mathrm{F_s}, \mathrm{F_j} \subset \mathrm{Future}$		
Effects on actors	eff	$::=$	$\langle \overrightarrow{\mathrm{A}}, \overrightarrow{\mathsf{beh}}^? \rangle$
Transactional context	ctx	$::=$	$\langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\mathrm{tx}} \rangle$
Message	$\mathsf{msg} \in \mathsf{Message}$	$::=$	$\langle a_{\mathrm{from}}, a_{\mathrm{to}}, \overline{v}, \mathsf{n}^?_{\mathrm{dep}} \rangle$
As before:			
Behavior	$\mathsf{beh} \in \mathsf{Behavior}$	$::=$	$\langle b, \overline{v} \rangle$
Snapshot, local store	$\overleftarrow{\sigma}, \delta : \mathrm{TVar} \rightharpoonup \mathrm{Value}$		
Transaction id	$\mathsf{n} \in \mathsf{TransactionNumber}$		
Transaction state	$\circ$	$::=$	$\triangleright \mid ✓ \mid ✗$

Figure 7.3: The representation of the program state and its elements in the semantics of PureChocola.

- a: the actor in which the task is running. A task always runs within one actor.[1]
- e: like before, the expression that is currently being evaluated by the task. When this has been reduced to a single value v, the task has finished.
- $\mathrm{F_s}$: the futures forked by this task. This contains the direct children of the task, which we require to be joined before the task finishes.
- $\mathrm{F_j}$: the futures joined in this task. This contains both the futures that were directly joined in this task, as well as those joined by another task that was joined into this one. Hence, we can say that the effects of these tasks have been incorporated into the current task.
- eff: this contains this task's delayed **effects on actors**. These effects are gathered and will only be executed at the end of the turn, when it is sure they do not need to be rolled back. There are two kind of effects: spawned actors, which are gathered in $\overrightarrow{\mathrm{A}}$, and the result of `become`, whose effect is stored in $\overrightarrow{\mathsf{beh}}^?$ (optional, only present if a `become` occurred).
- $\mathsf{ctx}^?$: when a transaction is active, this structure contains the effects of this task that occurred within the transaction. Thus, this structure represents the **transactional**

[1] Although its future can be passed to other actors.

context in which the task runs. When this is present, the task is transactional; for non-transactional tasks it is $\bullet$. It consists of:

- n: the identifier of the transaction in which this task is running.
- $\overleftarrow{\sigma}$: the snapshot at the start of this task. For the root task of a transaction, this is a copy of the heap. For tasks forked during the transaction, this is the snapshot that represents the transactional state at that point (as in Section 4.3.1, page 75).
- δ: the local store of changes made to the transactional variables in this transactional task only.
- $\mathsf{eff}_{\mathsf{tx}}$: the effects on actors that occurred in this task during the transaction, containing spawned actors and the effect of `become`. These effects will only be triggered when the transaction commits.

Effects on actors can be stored in two locations: in the task (eff) and in its transactional context ($\mathsf{eff}_{\mathsf{tx}}$). When such effects occur inside a transaction, they need to be stored in the transactional context so that they can be rolled back if the transaction aborts. When these effects occur outside a transaction but in a tentative turn, they need to be stored in the task, and may need to be rolled back when the turn ends. Keeping these effects in two places is a result of the fact that there are two kinds of 'tentative' sections that may need to roll back: transactions and tentative turns.

Shared data structures Three data structures can be accessed from multiple tasks:

- The inboxes of the actors μ, containing a list of messages for each actor. Messages are dequeued from the front by the receiving actor and enqueued at the back by any actor that sends a message.
  Messages can now contain a dependency ($\mathsf{n}^?_{\mathrm{dep}}$) that refers to the transaction in which they were sent. Tentative messages have such a dependency; definitive messages contain $\bullet$.
- A mapping τ of transaction identifiers to transactions. When a transaction starts, an entry is added. When it commits, the entry is modified to keep track of the success or failure of the transaction, by the root task of the transaction. Other actors access this structure when they process a tentative message to verify the state of its dependency.
  A transaction now only contains two elements: its state $\circ$ and the expression $\overleftarrow{e}$, contained within its `atomic` block, that is restored when the transaction must retry. Previously it also contained a snapshot and local store, but these data structures moved to the task(s) that run in the transaction.
- The transactional heap σ, which contains the last committed value of the transactional variables. This can be read and modified by any task, but only in a transaction, as before.

7.1.3 Evaluation Contexts

The program evaluation context $\mathcal{P}$ is defined below. It can choose an arbitrary task, and use the term evaluation context $\mathcal{E}$ to find the active site in the term. The definition of $\mathcal{E}$ is simply the combination of the three definitions for the separate models from Chapter 2, which we will not repeat here.

$$\mathcal{P} ::= \quad \langle \mathrm{A}, \mathrm{T} \uplus \langle f, a, \mathcal{E}, \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^? \rangle, \mu, \tau, \sigma \rangle$$

7.1.4 Helper Functions on the Program State

Figure 7.4 defines three helper functions that extract elements out of the program state:

- $\text{actor-tasks}(\mathrm{T}, a)$ returns all tasks in the actor with address a.
- $\text{actor-txs}(\mathrm{T}, a)$ returns the identifiers of all transactions that are active (in a task) within actor a. (Tasks in which no transaction is active do not match the pattern and are thus ignored.)
- $\text{tx-futs}(\mathrm{T}, \mathsf{n})$ returns the futures of all tasks spawned within the transaction with identifier n.

$$\begin{aligned}
&\text{actor-tasks}(\mathrm{T}, a) = \{\mathsf{task} \mid \mathsf{task} = \langle f, a, e, \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^? \rangle \in \mathrm{T}\} && \text{(tasks in actor } a\text{)}\\
&\text{actor-txs}(\mathrm{T}, a) = \{\mathsf{n} \mid \langle f, a, e, \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff_{tx}} \rangle \rangle \in \mathrm{T}\} && \text{(id's of tx's in actor } a\text{)}\\
&\text{tx-futs}(\mathrm{T}, \mathsf{n}) = \{f \mid \langle f, a, e, \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff_{tx}} \rangle \rangle \in \mathrm{T}\} && \text{(futures of tasks in tx n)}
\end{aligned}$$

Figure 7.4: Helper functions to extract elements out of the program state's set of tasks T.

7.1.5 Operations to Merge Effects

Figure 7.5 defines three operations that will prove useful when tasks are joined and their effects need to be merged.

- $\overrightarrow{\mathsf{beh}}^?_1 \mid \overrightarrow{\mathsf{beh}}^?_2$ (read "behavior 1 otherwise behavior 2") combines two optional behaviors: it returns the first if it exists, otherwise the second.
- $\mathsf{eff}_1 \mathrel{+}= \mathsf{eff}_2$ merges effects on actors from two tasks: the sets of spawned actors are joined and the behaviors are combined (preferring the second over the first if both exist). This operation will occur when a task is joined into another.
- $\mathsf{ctx}_1 \mathrel{+}= \mathsf{ctx}_2$ merges the transactional context of a second ('child') task into a first ('parent') task, which occurs when a transactional task is joined into another. We define that:
  - The transaction identifiers need to be the same: a transactional task can only be merged by another task in the *same* transaction.

- The first task is performing the join, so it keeps its snapshot.
- The local store of the second is added to the first, solving conflicts by preferring the version in the second task. In PureChocola, we will not consider custom conflict resolution functions (as in Section 4.3.2). Instead, we always use the default conflict resolution function, which prefers the value from the child task over that of the parent.
- The effects on actors of the second are added to the first. In case of conflicting becomes, the one from the second task is preferred.

$$\overrightarrow{\mathsf{beh}}^{?}_{1} \mid \overrightarrow{\mathsf{beh}}^{?}_{2} = \begin{cases} \overrightarrow{\mathsf{beh}}^{?}_{1} & \text{if } \overrightarrow{\mathsf{beh}}^{?}_{1} \neq \bullet \\ \overrightarrow{\mathsf{beh}}^{?}_{2} & \text{otherwise} \end{cases}$$

$$\langle \vec{\mathrm{A}}_1, \overrightarrow{\mathsf{beh}}^{?}_{1} \rangle \mathrel{+}= \langle \vec{\mathrm{A}}_2, \overrightarrow{\mathsf{beh}}^{?}_{2} \rangle = \langle \vec{\mathrm{A}}_1 \cup \vec{\mathrm{A}}_2, \overrightarrow{\mathsf{beh}}^{?}_{2} \mid \overrightarrow{\mathsf{beh}}^{?}_{1} \rangle$$

$$\langle \mathsf{n}, \overleftarrow{\sigma}_1, \delta_1, \mathsf{eff}_1 \rangle \mathrel{+}= \langle \mathsf{n}, \overleftarrow{\sigma}_2, \delta_2, \mathsf{eff}_2 \rangle = \langle \mathsf{n}, \overleftarrow{\sigma}_1, \delta_1 :: \delta_2, \mathsf{eff}_1 \mathrel{+}= \mathsf{eff}_2 \rangle$$

Figure 7.5: The operator | (“otherwise”) combines optional behaviors; += merges effects on actors or transactional contexts.

7.2 Reduction Rules

We can now describe the reduction relation → of PureChocola. We start with the base language (Section 7.2.1), and then define the operations on futures (Section 7.2.2), transactions (Section 7.2.3), and actors (Section 7.2.4). These rules are modifications – sometimes small, sometimes large – of those from Chapter 2.

7.2.1 Base Language

$$\underline{\text{congruence}|_{\mathrm{c}}}$$
$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[e], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?} \rangle, \mu, \tau, \sigma \rangle$$
$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[e'], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?} \rangle, \mu, \tau, \sigma \rangle$$
$$\text{if } e \rightarrow_{\mathrm{b}} e'$$

As before, a congruence rule allows the base language (the functional calculus from Chapter 2) to be used in any context, whether in or out a transaction and whether in a definitive or a tentative turn.

7.2.2 Futures

We define how to fork and join a future.

$$\begin{array}{l}
\underline{\mathrm{fork}|_{\mathrm{c}}}\\
\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathsf{fork}\ e], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle, \mu, \tau, \sigma\rangle\\
\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[f_\star], \mathrm{F_s} \cup f_\star, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle \mathbin{\dot\cup} \mathsf{task}_\star, \mu, \tau, \sigma\rangle\\
\quad \text{with } f_\star \text{ fresh}\\
\qquad\quad \mathsf{ctx}^{?}_\star = \begin{cases} \bullet & \text{if } \mathsf{ctx}^{?} = \bullet \\ \langle \mathsf{n}, \overleftarrow{\sigma} :: \delta, \varnothing, \langle \varnothing, \bullet\rangle\rangle & \text{if } \mathsf{ctx}^{?} = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \langle \overrightarrow{\mathrm{A}}, \overrightarrow{\mathsf{beh}^{?}}\rangle\rangle \end{cases} \qquad \begin{array}{r}\text{(outside transaction)}\\ \text{(in transaction)}\end{array}\\
\qquad\quad \mathsf{task}_\star = \langle f_\star, a, e, \varnothing, \mathrm{F_j}, \langle \varnothing, \bullet\rangle, \mathsf{ctx}^{?}_\star\rangle
\end{array}$$

fork As before, forking a future creates a new task to evaluate the given expression. In contrast to regular futures, the task can now be transactional if it was forked in a transaction. In that case, it will have a transactional context, containing:

- a reference to the transaction using its id n;
- a snapshot corresponding to the transactional state at the current point, which is $\overleftarrow{\sigma} :: \delta$;
- an empty local store; and
- no effects on actors.

A new task's set of spawned tasks ($\mathrm{F_s}$) is empty, as it has not spawned any tasks. On the other hand, its joined tasks $\mathrm{F_j}$ are copied from its parent: when these are joined again, their effects should not be applied again.

join There are two rules to join futures: when joining a future for the first time within the same actor, and when joining a future subsequently or within another actor.

$$\begin{array}{l}
\underline{\mathrm{join}_1|_{\mathrm{c}}}\\
\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathsf{join}\ f_\star], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle \mathbin{\dot\cup} \mathsf{task}_\star, \mu, \tau, \sigma\rangle\\
\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[v], \mathrm{F_s}, \mathrm{F_j} \cup \mathrm{F_j^\star} \cup f_\star, \mathsf{eff}_+, \mathsf{ctx}^{?}_+\rangle \mathbin{\dot\cup} \mathsf{task}_\star, \mu, \tau, \sigma\rangle\\
\ \ \text{where } \mathsf{task}_\star = \langle f_\star, a, v, \mathrm{F_s^\star}, \mathrm{F_j^\star}, \mathsf{eff}_\star, \mathsf{ctx}^{?}_\star\rangle \hfill \text{(same actor)}\\
\qquad\ \text{if } f_\star \notin \mathrm{F_j} \hfill \text{(first join)}\\
\qquad\quad \mathrm{F_s^\star} \subseteq \mathrm{F_j^\star} \hfill \text{(must have joined its children)}\\
\quad \text{with } \mathsf{eff}_+ = \mathsf{eff} \mathrel{+}= \mathsf{eff}_\star\\
\qquad\quad \mathsf{ctx}^{?}_+ = \begin{cases} \bullet & \text{if } \mathsf{ctx}^{?} = \bullet \text{ and } \mathsf{ctx}^{?}_\star = \bullet \\ \mathsf{ctx}^{?} & \text{if } \mathsf{ctx}^{?} \neq \bullet \text{ and } \mathsf{ctx}^{?}_\star = \bullet \\ \mathsf{ctx}^{?} \mathrel{+}= \mathsf{ctx}^{?}_\star & \text{if } \mathsf{ctx}^{?} \neq \bullet \text{ and } \mathsf{ctx}^{?}_\star \neq \bullet \end{cases} \qquad \begin{array}{r}\text{(both non-transactional)}\\ \text{(non-tx'al into transactional)}\\ \text{(both transactional)}\end{array}
\end{array}$$

$$\begin{array}{l}
\underline{\mathrm{join}_2|_{\mathrm{c}}}\\
\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathsf{join}\ f_\star], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle \mathbin{\dot\cup} \mathsf{task}_\star, \mu, \tau, \sigma\rangle\\
\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle \mathbin{\dot\cup} \mathsf{task}_\star, \mu, \tau, \sigma\rangle\\
\ \ \text{where } \mathsf{task}_\star = \langle f_\star, a_\star, v, \mathrm{F_s^\star}, \mathrm{F_j^\star}, \mathsf{eff}_\star, \mathsf{ctx}^{?}_\star\rangle\\
\qquad\ \text{if } f_\star \in \mathrm{F_j} \hfill \text{(subsequent join)}\\
\qquad\ \text{or } a \neq a_\star \hfill \text{(different actor)}
\end{array}$$

When a future from the same actor (all occurrences of a are equal) is joined for the first time ($f_\star \notin \mathrm{F_j}$), its effects should be merged into the task performing the join.

Concretely, this means its effects on actors (eff) and the transactional state (ctx) are merged. We do this using the operator $+=$ defined in the previous section.

Merging a transactional task into a non-transactional task is not allowed: the transactional task has effects on transactional state that the non-transactional task cannot handle. The opposite, merging a non-transactional task into a transactional task, is no problem: the non-transactional task does not have any side effects, so it can simply be resolved to its value.

Further, we require that the task $f_\star$ has joined all of its children ($\mathrm{F}_\mathrm{s}^\star \subseteq \mathrm{F}_\mathrm{j}^\star$). As a result, the effects of its children have been merged into $f_\star$ and are now present in its eff and ctx, allowing us to simply merge these effects. If the task $f_\star$ has not joined all its children, no rule is applicable; in the actual implementation of Chocola an error is raised.

The rule $\mathrm{join}_2|_\mathrm{c}$ simply resolves to the future's value without merging effects. This occurs in two cases:

- In the case of subsequent joins: the effects are already present and do not need to be merged again.
- In the case of joining a future from a different actor. This can occur when an actor sends a future to another actor in a message. Merging the effects of the task $f_\star$ from actor $a_\star$ into the task f of actor a is not desirable: effects on actors should not be 'transferred' between actors as this could cause them to be duplicated. Instead, these effects will be merged into the parent of $f_\star$, which exists in the same actor $a_\star$, as a result of the rule that requires parent tasks to join all of their children (a condition on the rules $\mathrm{join}_1|_\mathrm{c}$ and $\text{turn-end}|_\mathrm{c}$).

7.2.3 Transactions

The rules handling transactions are shown in Figure 7.6. We describe their changes compared to Section 2.4.5 from Chapter 2 (page 31). The most significant changes are to the commit rules, to take into account that in a tentative turn the current transaction depends on another transaction.

atomic, ref, deref, ref-set Rule $\mathrm{atomic}|_\mathrm{c}$ has been modified to store the transactional state in ctx instead of directly in the task. The rules $\mathrm{atomic}_\mathrm{tx}|_\mathrm{c}$, $\mathrm{ref}|_\mathrm{c}$, $\mathrm{deref}|_\mathrm{c}$, and $\text{ref-set}|_\mathrm{c}$ work in almost exactly the same way as before.

commit The rules handling a commit have been modified to take into account whether the current turn is definitive or tentative. Figure 7.7 summarizes which commit rule applies in which situation. Before a transaction can successfully commit, in rule $\mathrm{commit}_\checkmark|_\mathrm{c}$, the current turn must either have no dependency or its dependency must have succeeded ($\mathsf{n}^?_\mathrm{dep} = \bullet$ or $\tau(\mathsf{n}^?_\mathrm{dep}) = \langle \checkmark, \overleftarrow{e} \rangle$). This ensures that changes that

$$\underline{\text{atomic}|_{\text{c}}}$$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{atomic}\ e], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \bullet\rangle, \mu, \tau, \sigma\rangle$$

$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{atomic}\star\ e], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}\rangle, \mu, \tau[\mathsf{n} \mapsto \langle \triangleright, e\rangle], \sigma\rangle$$

$$\text{with } \mathsf{n} \text{ fresh}$$

$$\mathsf{ctx} = \langle \mathsf{n}, \sigma, \varnothing, \langle \varnothing, \bullet\rangle\rangle$$

$$\underline{\text{atomic}_{\text{tx}}|_{\text{c}}}$$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{atomic}\ e], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}\rangle, \mu, \tau, \sigma\rangle$$

$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[e], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}\rangle, \mu, \tau, \sigma\rangle$$

$$\underline{\text{ref}|_{\text{c}}}$$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{ref}\ v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau, \sigma\rangle$$

$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[r], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta[r \mapsto v], \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau, \sigma\rangle$$

$$\text{with } r \text{ fresh}$$

$$\underline{\text{deref}|_{\text{c}}}$$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{deref}\ r], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau, \sigma\rangle$$

$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[(\overleftarrow{\sigma} :: \delta)(r)], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau, \sigma\rangle$$

$$\underline{\text{ref-set}|_{\text{c}}}$$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{ref-set}\ r\ v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau, \sigma\rangle$$

$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta[r \mapsto v], \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau, \sigma\rangle$$

$$\underline{\text{commit}_{\checkmark}|_{\text{c}}}$$

$$\langle \mathrm{A} \mathbin{\dot\cup} \mathsf{act}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{atomic}\star\ v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{e}\rangle], \sigma\rangle$$

$$\rightarrow \langle \mathrm{A} \mathbin{\dot\cup} \mathsf{act}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}_{+}, \bullet\rangle, \mu, \tau[\mathsf{n} \mapsto \langle \checkmark, \overleftarrow{e}\rangle], \sigma :: \delta\rangle$$

$$\text{where } \mathsf{act} = \langle a, f_{\text{root}}, \mathsf{beh}, \mathsf{n}^{?}_{\text{dep}}\rangle$$

$$\text{if } \forall r \in \mathrm{dom}(\delta) : \sigma(r) = \overleftarrow{\sigma}(r) \qquad \text{(no conflicts)}$$

$$\forall f_{\star} \in \text{tx-futs}(\mathrm{T}, \mathsf{n}) : f_{\star} \in \mathrm{F_j} \qquad \text{(all futures spawned in the tx must have been joined)}$$

$$\mathsf{n}^{?}_{\text{dep}} = \bullet \text{ or } \tau(\mathsf{n}^{?}_{\text{dep}}) = \langle \checkmark, \overleftarrow{e}\rangle \qquad \text{(in a definitive or a successful tentative turn)}$$

$$\text{with } \mathsf{eff}_{+} = \mathsf{eff} \mathrel{+}= \mathsf{eff}_{\text{tx}}$$

$$\underline{\text{commit}_{✗}|_{\text{c}}}$$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{atomic}\star\ v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff}_{\text{tx}}\rangle\rangle, \mu, \tau[\mathsf{n} \mapsto \langle \triangleright, \overleftarrow{e}\rangle], \sigma\rangle$$

$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{atomic}\ \overleftarrow{e}], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \bullet\rangle, \mu, \tau[\mathsf{n} \mapsto \langle ✗, \overleftarrow{e}\rangle], \sigma\rangle$$

$$\text{if } \exists r \in \mathrm{dom}(\delta) : \sigma(r) \neq \overleftarrow{\sigma}(r) \qquad \text{(a conflict occurred)}$$

$$\forall f_{\star} \in \text{tx-futs}(\mathrm{T}, \mathsf{n}) : f_{\star} \in \mathrm{F_j} \qquad \text{(all futures spawned in the tx must have been joined)}$$

$$\underline{\text{commit}_{\bullet}|_{\text{c}}}$$

$$\langle \mathrm{A} \mathbin{\dot\cup} \mathsf{act}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\texttt{atomic}\star\ v], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}\rangle, \mu, \tau, \sigma\rangle$$

$$\rightarrow \langle \mathrm{A} \mathbin{\dot\cup} \mathsf{act}', \mathrm{T}', \mu, \tau', \sigma\rangle$$

$$\text{where } \mathsf{act} = \langle a, f_{\text{root}}, \mathsf{beh}, \mathsf{n}_{\text{dep}}\rangle$$

$$\text{if } \tau(\mathsf{n}_{\text{dep}}) = \langle ✗, \overleftarrow{e}\rangle \qquad \text{(in a failed tentative turn)}$$

$$\text{with } \mathsf{act}' = \langle a, \bullet, \mathsf{beh}, \bullet\rangle \qquad \text{(reset actor to idle state)}$$

$$\mathrm{T}' = \mathrm{T} \setminus \text{actor-tasks}(\mathrm{T}, a) \qquad \text{(abort and remove all tasks in this turn)}$$

$$\tau'(\mathsf{n}) = \begin{cases} \langle ✗, \texttt{nil}\rangle & \text{if } \mathsf{n} \in \text{actor-txs}(a) \\ \tau(\mathsf{n}) & \text{otherwise} \end{cases} \qquad \begin{array}{r}\text{(abort all transactions in this} \\ \text{turn, including current)}\end{array}$$

Figure 7.6: Rules concerning transactions.

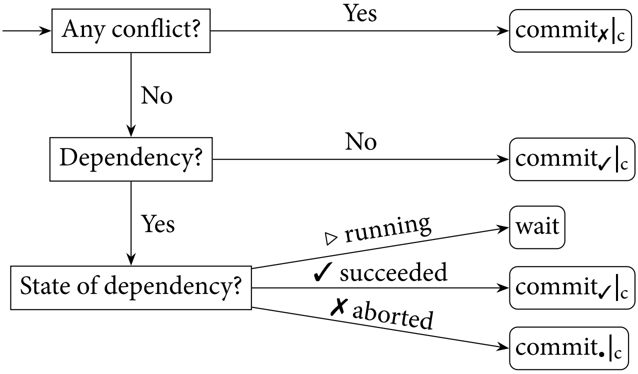


Figure 7.7: This diagram illustrates which commit rules applies in which scenario.

are the result of a tentative message are only written to the transactional heap until it is sure the dependency has succeeded. In $\text{commit}_{\text{✗}}|_{\mathsf{c}}$, no such condition is necessary: when a conflict is detected the transaction will not write any changes to the transactional heap anyway. We also added the rule $\text{commit}_{\bullet}|_{\mathsf{c}}$, which matches when the current transaction runs in a tentative turn whose dependency failed ($\mathsf{n}_{\mathsf{dep}} \neq \bullet$ and $\tau(\mathsf{n}_{\mathsf{dep}}) = \langle \text{✗}, \overleftarrow{e} \rangle$). Then, the following happens:

- The current turn is abandoned and the actor returns to an idle state. Any changes that occurred in the current turn are thus discarded, as they are the result of an invalid message.
- All tasks that were active in this turn are aborted and removed.
- This transaction and any other transactions that are active in this turn are marked as aborted in the map τ. This is necessary because these transactions may have sent tentative messages, which are now invalid. Note that these transactions will not have committed yet, as they must also wait for the dependency; they are either still active ($\triangleright$) or they have aborted (✗). (We can safely set their expression to `nil`, as they will never retry anymore.)

No rule applies when the dependency is still running, i.e. $\mathsf{n}_{\mathsf{dep}} \neq \bullet$ and $\tau(\mathsf{n}_{\mathsf{dep}}) = \langle \triangleright, \overleftarrow{e} \rangle$. As a result, the reduction of the current task will be stuck until its dependency either commits or aborts, at which point either $\text{commit}_{\text{✓}}|_{\mathsf{c}}$ or $\text{commit}_{\bullet}|_{\mathsf{c}}$ applies and the current task can proceed.

Additionally, the rules $\text{commit}_{\text{✓}}|_{\mathsf{c}}$ and $\text{commit}_{\text{✗}}|_{\mathsf{c}}$ have two other modifications:

- All tasks that were forked in a transaction must have been joined before the commit. This ensures that all effects have been merged into the root task of this transaction, and can therefore be applied atomically. Programs that do not adhere to this are invalid; in the actual implementation an error is raised then.

- Upon a successful commit, the effects on actors (eff) that occurred in the transaction are merged into the task (rule $\text{commit}_{\checkmark}|_c$). If the transaction aborts, they are discarded (rule $\text{commit}_{\times}|_c$).

7.2.4 Actors

The operations involving actors, shown in Figure 7.8, need to take into account whether a transaction or tentative turn is active, or neither. Table 5.7 from Chapter 5 (page 104) described how each operation works in each of three contexts: in a transaction, outside a transaction in a tentative turn, and outside a transaction in a definitive turn. The reduction rules likewise differentiate between these cases.

self The rule $\text{self}|_c$ is unmodified, simply returning the address of the current actor.

spawn spawn creates a new actor, but this actor is not immediately active (it is not added to the set A in the program state). Instead:

- If no transaction is active, the new actor is stored in the effects eff of the current task. This occurs both in definitive and tentative turns. The actor will become active if and when the current turn ends successfully (rule $\text{turn-end}|_c$). Note that this is different from the implementation in Chapter 8, where the actor becomes active immediately in definitive turns.
- If a transaction is active, the new actor will be stored in the effects eff_{tx} of the transaction. If the transaction commits successfully, these effects will be merged into the effects of the task upon commit. These will eventually be executed at the end of the turn. If the transaction aborts, the effects are discarded.

The inbox is created immediately though, as it should be able to receive (possibly tentative) messages immediately. (If eventually the actor is discarded, we do not remove the inbox, although it is no longer used and could be cleaned up.)

become become similarly distinguishes these two cases. If no transaction is active, the effect is stored in the task; otherwise, it is stored in the transaction. Again, the effects in a transaction are merged into the task when the transaction commits; and the effects become active if and when the current turn ends successfully.

send As explained in Chapter 5, send always immediately sends the message, but possibly it is tentative. This is indicated through an additional parameter $\mathsf{n}^?_{\text{msg}}$, which refers to the current transaction in a transaction, to the current dependency in a tentative turn, or is $\bullet$ in a definitive turn. This corresponds to the three cases described in Table 5.7 (page 104). Hence, messages without a dependency ($\bullet$) are definitive, those with a dependency are tentative.

$\underline{\text{self}|_{\text{c}}}$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathtt{self}], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle, \mu, \tau, \sigma\rangle$$
$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[a], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle, \mu, \tau, \sigma\rangle$$

$\underline{\text{spawn}|_{\text{c}}}$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathsf{spawn}\ b_\star\ \overline{v}], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle, \mu, \tau, \sigma\rangle$$
$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[a_\star], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}', \mathsf{ctx}'\rangle, \mu[a_\star \mapsto []], \tau, \sigma\rangle$$

with $a_\star$ fresh

$$\mathsf{act}_\star = \langle a_\star, \bullet, \langle b_\star, \overline{v}\rangle, \bullet\rangle$$

$$\begin{cases} \text{if } \mathsf{ctx}^{?} = \bullet: & \mathsf{ctx}' = \bullet & \text{(outside transaction)} \\ & \mathsf{eff}' = \mathsf{eff} \mathrel{+}= \langle \mathsf{act}_\star, \bullet\rangle & \\ \text{if } \mathsf{ctx}^{?} = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff_{tx}}\rangle: & \mathsf{ctx}' = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff_{tx}} \mathrel{+}= \langle \mathsf{act}_\star, \bullet\rangle\rangle & \text{(in transaction)} \\ & \mathsf{eff}' = \mathsf{eff} & \end{cases}$$

$\underline{\text{become}|_{\text{c}}}$

$$\langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathsf{become}\ b_\star\ \overline{v}], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle, \mu, \tau, \sigma\rangle$$
$$\rightarrow \langle \mathrm{A}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathtt{nil}], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}', \mathsf{ctx}'\rangle, \mu, \tau, \sigma\rangle$$

$$\text{with} \begin{cases} \text{if } \mathsf{ctx}^{?} = \bullet: & \mathsf{ctx}' = \bullet & \text{(outside transaction)} \\ & \mathsf{eff}' = \mathsf{eff} \mathrel{+}= \langle \varnothing, \langle b_\star, \overline{v}\rangle\rangle & \\ \text{if } \mathsf{ctx}^{?} = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff_{tx}}\rangle: & \mathsf{ctx}' = \langle \mathsf{n}, \overleftarrow{\sigma}, \delta, \mathsf{eff_{tx}} \mathrel{+}= \langle \varnothing, \langle b_\star, \overline{v}\rangle\rangle\rangle & \text{(in transaction)} \\ & \mathsf{eff}' = \mathsf{eff} & \end{cases}$$

$\underline{\text{send}|_{\text{c}}}$

$$\langle \mathrm{A} \mathbin{\dot\cup} \mathsf{act}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathsf{send}\ a_{\text{to}}\ \overline{v}], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}, \mathsf{ctx}^{?}\rangle, \mu[a_{\text{to}} \mapsto \overline{\mathsf{msg}}], \tau, \sigma\rangle$$
$$\rightarrow \langle \mathrm{A} \mathbin{\dot\cup} \mathsf{act}, \mathrm{T} \mathbin{\dot\cup} \langle f, a, \mathcal{E}[\mathtt{nil}], \mathrm{F_s}, \mathrm{F_j}, \mathsf{eff}', \mathsf{ctx}'\rangle, \mu[a_{\text{to}} \mapsto \overline{\mathsf{msg}} \cdot \mathsf{msg}], \tau, \sigma\rangle$$

where $\mathsf{act} = \langle a, f_{\text{root}}, \mathsf{beh}, \mathsf{n}^{?}_{\text{dep}}\rangle$

with $\mathsf{msg} = \langle a, a_{\text{to}}, \overline{v}, \mathsf{n}^{?}_{\text{msg}}\rangle$

$$\mathsf{n}^{?}_{\text{msg}} = \begin{cases} \mathsf{n_{tx}} & \text{if } \mathsf{ctx}^{?} = \langle \mathsf{n_{tx}}, \overleftarrow{\sigma}, \delta, \mathsf{eff_{tx}}\rangle & \text{(in transaction)} \\ \mathsf{n}^{?}_{\text{dep}} & \text{if } \mathsf{ctx}^{?} = \bullet \text{ and } \mathsf{n}^{?}_{\text{dep}} \neq \bullet & \text{(in tentative turn)} \\ \bullet & \text{otherwise} & \text{(definitive)} \end{cases}$$

$\underline{\text{receive}|_{\text{c}}}$

$$\langle \mathrm{A} \mathbin{\dot\cup} \langle a, \bullet, \mathsf{beh}, \bullet\rangle, \mathrm{T}, \mu[a \mapsto \langle a_{\text{from}}, a, \overline{v}_{\text{msg}}, \mathsf{n}^{?}_{\text{dep}}\rangle \cdot \overline{\mathsf{msg}}], \tau, \sigma\rangle$$
$$\rightarrow \langle \mathrm{A} \mathbin{\dot\cup} \langle a, f_\star, \mathsf{beh}, \mathsf{n}^{?}_{\text{dep}}\rangle, \mathrm{T} \mathbin{\dot\cup} \mathsf{task}, \mu[a \mapsto \overline{\mathsf{msg}}], \tau, \sigma\rangle$$

$$\text{with} \quad \begin{array}{ll} f_\star \text{ fresh} & e_\star = \text{apply-behavior}(\mathsf{beh}, \overline{v}_{\text{msg}}) \\ \mathsf{task} = \langle f_\star, a, e_\star, \varnothing, \varnothing, \mathsf{eff}, \bullet\rangle & \mathsf{eff} = \langle \varnothing, \bullet\rangle \end{array}$$

$\underline{\text{turn-end}|_{\text{c}}}$

$$\langle \mathrm{A} \mathbin{\dot\cup} \langle a, f_{\text{root}}, \mathsf{beh}, \mathsf{n}^{?}_{\text{dep}}\rangle, \mathrm{T} \mathbin{\dot\cup} \mathsf{task}_{\text{root}}, \mu, \tau, \sigma\rangle$$
$$\rightarrow \langle \mathrm{A} \mathbin{\dot\cup} \langle a, \bullet, \mathsf{beh}', \bullet\rangle \mathbin{\dot\cup} \overrightarrow{\mathrm{A}}', \mathrm{T} \mathbin{\dot\cup} \mathsf{task}_{\text{root}}, \mu, \tau, \sigma\rangle$$

where $\mathsf{task}_{\text{root}} = \langle f_{\text{root}}, a, v, \mathrm{F_s}, \mathrm{F_j}, \langle \overrightarrow{\mathrm{A}}, \overrightarrow{\mathsf{beh}}^{?}\rangle, \bullet\rangle$

if $\mathrm{F_s} \subseteq \mathrm{F_j}$ (all futures spawned in the turn must have been joined)

$$\text{with} \begin{cases} \text{if } \mathsf{n}^{?}_{\text{dep}} = \bullet \text{ or } \tau(\mathsf{n}^{?}_{\text{dep}}) = \langle ✓, \overleftarrow{e}\rangle: & \mathsf{beh}' = \overrightarrow{\mathsf{beh}}^{?} \mid \mathsf{beh} & \text{(definitive, or success-} \\ & \overrightarrow{\mathrm{A}}' = \overrightarrow{\mathrm{A}} & \text{ful tentative turn)} \\ \text{if } \tau(\mathsf{n}^{?}_{\text{dep}}) = \langle ✗, \overleftarrow{e}\rangle: & \mathsf{beh}' = \mathsf{beh} & \text{(failed tentative turn)} \\ & \overrightarrow{\mathrm{A}}' = \varnothing & \end{cases}$$

Figure 7.8: Rules concerning actors.

Receive (start of turn) The rule $\text{receive}|_c$ is triggered when an actor takes a message from its inbox. As before, it binds the parameters of the expression in the behavior to the respective values in the internal state and the message. A root task is created to evaluate this expression and its future is stored in the actor. If the message is tentative, its dependency is copied to the actor, causing the turn to be tentative as well. ($\mathsf{n}^?_{\text{dep}}$ can be • too, in which case the turn is definitive.)

End of turn Finally, the rule $\text{turn-end}|_c$ is evaluated when the turn ends, i.e. when its root task has been reduced to a single value. The actor moves to an idle state (its root future is replaced with •) and loses its dependency. We require that the root task has joined all its children. In turn, these children must have joined their children (required by rule $\text{join}_1|_c$). (In contrast to the formal semantics, the implementation raises an error if this is not the case.) As a result, the effects of all tasks that were created in this turn have been merged and are present in the root task's eff. The turn can now succeed or fail:

- A turn succeeds when it is definitive (its dependency is •) or when it is tentative and its dependency committed. In that case, its delayed effects can be executed: its behavior is updated (if necessary) and newly spawned actors become active.
- A turn fails when it is tentative and its dependency has aborted. Then, its delayed effects do not become visible: its behavior remains the same and the actors it spawned are discarded.

If the dependency is still executing (▷), this rule will not be applicable. Hence, the reduction of this actor becomes stuck until the dependency finishes, at which point it will either succeed or fail. In the implementation, an actor executing a tentative turn will wait at the end of the turn until its dependency either commits or aborts.

7.3 Guarantees

In this section, we show how the guarantees of Chocola can be inferred from the formal semantics. We will not give formal proofs in this dissertation (these are considered future work), but instead we argue slightly more informally how the guarantees are embedded in the formal semantics.

7.3.1 Intratransaction Determinacy

Intratransaction determinacy states that, given the initial state of the transactional heap, a transaction will always have the same result, assuming that all conflict resolution functions are determinate. In this section, we show that the semantics of PureChocola complies with this property.

Reduction of a transaction A transaction starts when the expression `atomic` e is encountered (and no transaction is running yet), at which point the rule $\text{atomic}|_{\text{c}}$ triggers. This rule replaces the expression with `atomic`$\star$ e. Next, e will be reduced, possibly forking new tasks. The transaction commits or aborts when this expression has been fully reduced to a value, when either of the rules $\text{commit}_{\checkmark}|_{\text{c}}$ or $\text{commit}_{\times}|_{\text{c}}$ applies. (We can safely ignore transactions that are aborted when their turn failed due to $\text{commit}_{\bullet}|_{\text{c}}$, as these transactions do not have any result.)

The reduction of the transaction's expression e to its final value v can occur along different paths: when the transaction contains multiple tasks, their instructions can be interleaved in different orders, non-deterministically. In PureChocola, this non-determinism is introduced in the definition of the program evaluation context $\mathcal{P}$, in Section 7.1.3: when reducing a program state, any task that can be reduced may be chosen to be reduced next.

To prove intratransaction determinacy, we need to show that any valid ordering of the instructions of the transaction's tasks always leads to the same effects. A transaction has three kinds of effects: (1) its final value after it is fully evaluated, (2) its effects on transactional memory and (3) its effects on actors. We consider each separately.

Its final value The final value of a transaction is not influenced by the order in which the instructions of its tasks are scheduled. This is a result of the fact that each task runs in isolation and only reads and writes to its own snapshot and local store. An instruction from one task can thus never influence the result of an instruction of a different task, and hence each task always has the same result. Different interleavings of the same instructions therefore still lead to the same result.

Its effects on transactional memory The effects of a transaction on transactional memory are also always deterministic. Each task runs in isolation and gathers its effects in its local store. When it is joined, its effects are joined into its parent, using conflict resolution functions that we assume to be determinate.

Furthermore, the order of the joins is fixed. We require the programmer to explicitly join each task into its parent, hence, this order is explicitly prescribed in the code. (If the programmer forgets to join a task, an error is raised. This also does not depend on the order in which tasks are scheduled and is therefore deterministic too.)

In the end, the root task will contain all effects of all tasks, merged deterministically. These are then applied in a single step during the commit. Thus, the effects of a transaction on transactional memory do not depend on the order in which its instructions are interleaved, and are therefore determinate.

Burckhardt and Leijen [2011] prove determinacy for Concurrent Revisions, a related technique that introduces concurrent tasks that can access versioned variables, providing a model similar to our transactional futures (described in Section 4.5.3, on page 87). A similar technique can be applied here.

Its effects on actors A transaction can have effects on actors using `spawn`, `become`, and `send`. These effects are determinate:

- The order in which actors are spawned is not observable, and therefore cannot cause observable non-determinism.
- When `become` is encountered in a transaction, its effects are stored in the current task (rule $\text{become}|_c$). When the task is joined, these effects are merged into its parent and conflicts are resolved deterministically by preferring the value of the child (rule $\text{join}_1|_c$ and the definition of $\mathsf{eff}_1 \mathrel{+}= \mathsf{eff}_2$). Eventually, these effects will wind up in the root task and be committed, similar to the changes to transactional memory in the local store.
- Given its initial state, a transaction will always send the same messages, as the `send` statements do not depend on the order in which the transaction's tasks are scheduled. Note that the *order* in which these messages are sent is actually non-deterministic: for instance, when two transactional tasks each send a message to the same actor, the two messages can arrive in any order. Only the messages and their values are determinate, not the order in which they are sent.

Conclusion PureChocola guarantees intratransaction determinacy: the effects of a transaction – its final value and its effects on transactional memory and actors – only depend on the state of the transactional heap at the start of the transaction. This makes it easy to reason about transactions: developers do not need to take into account how the instructions of tasks inside a transaction are interleaved, as each task runs in isolation.

7.3.2 Low-Level Race Freedom

Low-level race freedom states that race conditions can only occur due to bad interleavings of turns or transactions, not due to bad interleavings of the individual instructions within turns or transactions. We show how these low-level races are prevented in PureChocola.

Races occur when different threads access shared memory in an incorrect way. In PureChocola, there are two places in which data is stored: an actor's private memory and the shared transactional memory. Both are protected from races:

Actors' private memory is stored in their behavior. Access to this memory is safe, because it is only read and modified by the actor itself. It is read once when a turn starts (the apply-behavior function in the rule $\text{receive}|_c$), ensuring that all tasks in the turn have a consistent view throughout the whole turn. The private memory can be modified using `become`, but those changes will only be visible in the next turn. Even when multiple tasks update the memory, the values of the task that is joined last are stored, independent of how their instructions are interleaved (as can been seen when the effects are merged in rule join$_1|_c$).

The transactional memory is shared, but accesses to this memory are protected using transactions. In the reduction rules, a consistent view is ensured by making a copy of the transactional memory at the start of the transaction ($\overleftarrow{\sigma}$). Atomic updates are ensured by storing updates during the transaction in a local store δ, and only applying them upon a successful commit (rule commit$_{\checkmark}|_c$). (Remember that the actual implementation does not create copies, but instead ensures this property using the MVCC algorithm.)

7.3.3 Deadlock Freedom of Actors

Blocking operations are encoded in the semantics as rules with conditions that may not be satisfied immediately. Thus, one task's reduction path is blocked until another task has reached a certain state. There are three such cases: when joining a future (rule join$_1|_c$), at the end of a turn (rule turn-end$|_c$), when committing a transaction (rule commit$_{\checkmark}|_c$ and commit$_{\bullet}|_c$), and when receiving a message (in rule receive$|_c$). The actor model guarantees deadlock freedom, meaning that once a turn starts, it always runs to completion; hence, we should check whether the first three cases do not break this property. The fourth case may be blocked forever; this is even expected when an actor has processed all its messages.

When joining a future The rule join$_1|_c$ requires the task that is joined to have been reduced to a single value. It can therefore only trigger after this task has completed. This could potentially cause deadlocks when two tasks are waiting for each other to finish. However, this can never happen: the tasks in an actor form a spawn tree, just like they did when used outside actors (as described Section 3.3.3 on page 63), and therefore they never contain a cycle. This condition will thus always be satisfied eventually. The spawn tree within a turn can be derived from the semantics, as the actor stores a reference to its root task and each task stores a reference to its children (in its set of spawned futures F_s).

At the end of a turn An actor can block at the end of a tentative turn, waiting for its dependency to be resolved. This is encoded in the rule turn-end$|_c$ by checking whether the dependency has succeeded or failed ($\tau(\mathsf{n}^?_{\mathrm{dep}}) = \langle \checkmark, \overleftarrow{e}\rangle$ or $\langle \text{✗}, \overleftarrow{e}\rangle$). While the transaction is still running ($\tau(\mathsf{n}^?_{\mathrm{dep}}) = \langle \triangleright, \overleftarrow{e}\rangle$), none of both conditions is satisfied, so this rule is not applicable, nor is any other rule. It is only when the dependency commits that this rule applies and the actor proceeds.

In Section 5.3.3, on page 106, we discussed why transactional actors cannot cause a deadlock due to cyclical dependencies. We repeat the observations made there and relate them to the formal semantics:

1. *Dependencies are always introduced at the start of a turn:* the only rule in which an actor's dependency goes from • to n_{dep} is in receive$|_\text{c}$, when a message with a dependency is received.
2. No transaction is running when a turn starts: no task is active in the begin state of the rule receive$|_\text{c}$. Hence, *dependencies are never introduced while a transaction is active.*

Here we see again why cyclical dependencies are impossible. When an actor blocks at the end of a tentative turn (rule turn-end$|_\text{c}$), its dependency n_{dep} was introduced at the start of that turn (rule receive$|_\text{c}$, observation 1). This dependency came from a message, which was sent in the transaction with identifier n_{dep} (rule send$|_\text{c}$). That transaction was therefore already running before the turn that depends on it. An inverse dependency is impossible: the transaction n_{dep} is already running or finished, and cannot acquire a dependency on later transactions (observation 2). Hence, cyclical dependencies are impossible, so deadlocks cannot occur.

When committing a transaction In a tentative turn, a transaction will wait to commit until its dependency is resolved. This is encoded in the rules commit$_{\checkmark}|_\text{c}$ and commit$_{\bullet}|_\text{c}$: if there are no conflicts and the dependency is still running ($\tau(\mathsf{n}^?_{\text{dep}}) = \langle \triangleright, \overleftarrow{e} \rangle$), no rule is applicable so the reduction is stuck until the dependency commits.

No deadlocks occur because the dependency will always commit eventually. The same reasoning given above applies here: cyclical dependencies are impossible because the waiting transaction is always more recent than the transaction on which it is waiting and dependencies are never introduced in a running transaction. The dependency will therefore commit eventually, at which point the current transaction can commit.

7.3.4 Isolation of Transactions: Snapshot Isolation

Transactional systems guarantee a form of isolation; in Section 2.4.3, we chose to provide snapshot isolation for Chocola. This property is still present in the semantics of PureChocola, in a quite direct way.[2] When a transaction is started, a copy of the heap is stored in its root task as a 'snapshot' $\overleftarrow{\sigma}$. All changes to the transactional memory in a transaction are stored in a separate 'local store' δ. When the transaction attempts

[2] Remember that here, PureChocola does not match the actual implementation of Chocola. The algorithm we actually use to implement these semantics, Multiversion Concurrency Control, was described in Section 2.4.4 (page 27) and provides the same guarantees.

to commit, it verifies correctness by comparing the snapshot with the current heap, and then applies all changes at once to the transactional heap. This follows exactly the definition of snapshot isolation (see Section 2.4.3, page 25), which required that (1) a transaction sees a consistent view of the memory (the snapshot), and (2) a transaction can only commit if none of its updates conflict with any concurrent updates made since the snapshot.

Even when multiple tasks are forked in a transaction, isolation is maintained. Each task has its own local store, and when it is joined into its parent, its local store is merged with that of the parent (rule $\text{join}_1|_c$). By requiring that all tasks that are created in a transaction have been joined into its root task before the transaction ends, we ensure that all changes of all tasks have been merged into the root task's local store (condition $f_\star \in F_j$ in rule $\text{commit}_{\checkmark}|_c$). These are then applied to the heap in an atomic step (update to σ in $\text{commit}_{\checkmark}|_c$).

The same is true for the effects on actors. `spawn` and `become` are delayed and stored as a task's effects (eff_{tx} structure in a task's transactional context ctx, modified in the rules $\text{spawn}|_c$ and $\text{become}|_c$). When a task is joined, its effects are merged into its parent (rule $\text{join}_1|_c$). Again, this will cause all effects of all tasks to have been merged into the root task's effects. These effects are then applied atomically when the transaction commits, or discarded when it aborts.

Messages in a transaction are not delayed, but sent immediately. A dependency is attached to the message (rule $\text{send}|_c$), causing the turn that processes this message to become tentative (rule $\text{receive}|_c$). When a tentative turn ends, its effects are only applied if its dependency committed successfully; while on failure, the effects are discarded (rule $\text{turn-end}|_c$). This ensures that the effects of messages sent in a transaction only become visible if the transaction succeeds, thus maintaining isolation.

7.3.5 Progress of Transactions: Deadlock Freedom

Algorithms that implement transactional systems can guarantee a form of progress. We chose to implement the Multiversion Concurrency Control (MVCC) algorithm, as explained in Section 2.4.4 (page 27), which guarantees deadlock freedom. Deadlock freedom of transactions ensures that, when two transactions conflict, one is delayed so that another can make progress. This guarantee is a property of the implementation algorithm; it is not visible in the semantics. Chocola still uses MVCC and thus maintains the property.

While this property guarantees progress when two transactions conflict, deadlocks might also occur within a single transaction: using `join` in a transaction could potentially cause deadlocks when two tasks are waiting for each other to finish. However, the tasks in a transaction form a spawn tree, just like they did when used outside transactions, and no cycles occur. The spawn tree can be derived from each task's set of spawned futures F_s.

7.4 Mechanical Verification of Isolation and Intratransaction Determinacy

We have created an executable implementation of parts of PureChocola using PLT Redex.[3] PLT Redex is a language and framework in which formal semantics can be written down and executed [Felleisen et al. 2009]. As argued by Klein et al. [2012], creating an executable version of a semantics often helps to find mistakes and to verify its properties. We have used PLT Redex to explore and visualize reductions of test programs and we have specified several test programs and their expected reductions, which are verified automatically.

We have not implemented the full formal semantics of PureChocola in PLT Redex. Instead, we focused on some crucial parts: we implemented each of the three separate models and transactional futures.

For transactions, we created two implementations: one in which each transaction is executed in a single atomic step (corresponding to the serialized version of the program) and one in which steps from different transactions are interleaved (and commits can therefore succeed or abort and transactions can roll back). We can verify isolation by reducing programs with both implementations and checking that they lead to the same end results. We tested this by systematically generating programs conforming to different patterns (e.g. two transactions executed in sequence, two parallel transactions that do not conflict, two parallel transactions that conflict).

We implemented transactional futures in PLT Redex to verify the guarantees of intratransaction determinacy and isolation when transactions and futures are combined. Again, we systematically test certain patterns (e.g. no conflicts, conflicts between the tasks within a transaction, conflicts between transactions, and both).

We demonstrate a reduction using the example in Listing 7.9. This example is slightly convoluted, to demonstrate that the guarantees hold even in a complex scenario. The program creates two transactions that modify a shared transactional variable `r`. The two transactions each create two tasks internally, which concurrently modify the variable. In the first transaction, the first task adds 1 to `r` and the second task adds 2. Intratransaction determinacy requires that, because the second task is joined last, the result of this transaction must always add 2 to `r`, never 1. Likewise, the second transaction must always add 4 to `r`.

There are two serializations of this program:

[3] Available online at https://soft.vub.ac.be/~jswalens/chocola.

- If transaction 1 precedes transaction 2, transaction 1 returns 2 (= 0 + 2) and transaction 2 returns 6 (= 2 + 4). The end result of the program is 8 (= 2 + 6).
- If transaction 2 precedes transaction 1, transaction 2 returns 4 (= 0 + 4) and transaction 1 returns 6 (= 4 + 2). The end result of the program is 10 (= 6 + 4).

Figure 7.10 contains a trace of this program generated using PLT Redex. It shows how different interleavings lead to different intermediate states, but in the end they collapse into two terminal states, corresponding to the two possible serializations of this program. This is only possible because (1) thanks to isolation there are only two serializations, and (2) thanks to intratransaction determinacy each transaction only has one result given an initial state. Using PLT Redex, we can verify isolation and intratransaction determinacy for any program, by checking that the number of terminal states for a program corresponds to the number of possible serializations.

7.5 Differences Between PureChocola and Chocola

We list the differences between our formal semantics PureChocola and the actual implementation Chocola:

- PureChocola is built around a base language, which is not specified here. Chocola is built on top of Clojure. The base language of PureChocola is limited to the functional subset of Clojure: functions with side effects are not considered.
- In PureChocola, the conflict resolution function of transactional variables is always assumed to be the default one, which prefers the value in the joined task over the value in the task performing the join. While it is possible to add support for custom conflict resolution functions to PureChocola, we omitted this functionality to simplify the formalization.
- PureChocola implements snapshot isolation in a direct way: a snapshot is taken at the start of each transaction's attempt and verified during commit. Chocola instead relies on MVCC to guarantee snapshot isolation. One notable consequence is that in Chocola, a transaction can abort early when a conflict is detected while the transaction is still running, while in PureChocola, conflicts are only ever detected at the end of the transaction.
- The rule commit$\bullet|_c$ specifies how a turn is aborted when, at the end of a transaction in a tentative turn, it appears the dependency aborted. In PureChocola, all tasks in the current turn are immediately stopped and removed and any transaction in the current turn immediately aborts. In Chocola, this is implemented using an exception: the current transaction throws an exception, which bubbles up through the spawn tree of the current turn, until it eventually reaches the root task. (This exception cannot be caught by the programmer.) The root task then

```
(let [(r (ref 0))
      (f (fork
           (atomic
             (let [(x (fork (ref-set r (+ @r 1))))
                   (y (fork (ref-set r (+ @r 2))))]
               (join x)    ; ⇒ r = original + 1
               (join y)    ; ⇒ r = original + 2
               @r))))      ; ⇒ returns original + 2
      (g (fork
           (atomic
             (let [(x (fork (ref-set r (+ @r 3))))
                   (y (fork (ref-set r (+ @r 4))))]
               (join x)    ; ⇒ r = original + 3
               (join y)    ; ⇒ r = original + 4
               @r))))]     ; ⇒ returns original + 4
  (+ (join f) (join g)))
```

Listing 7.9: Example program in which two transactions both create two transactional tasks. Each of the four tasks modifies the shared transactional variable r. Despite a multitude of possible interleavings of the instructions of the four tasks, this program only has two possible outputs (8 or 10), corresponding to the two serializations of the transactions.

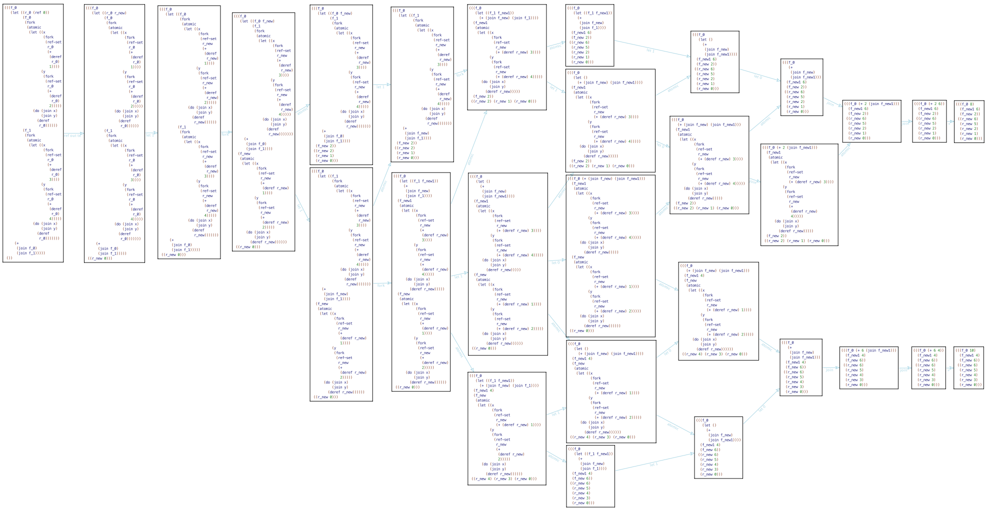


Figure 7.10: Trace of the evaluation of the program in Listing 7.9, generated using PLT Redex with an executable version of the semantics. (Transactions are evaluated in a single step, corresponding to their atomicity.) The details of each intermediate state are not important, the important thing to note is that during the execution of the program many different interleavings are possible, but that in the end there are only two possible terminal states, corresponding to the two serializations of the two transactions.

aborts. Hence, in PureChocola all tasks seem to abort at exactly the same moment, while in Chocola they abort at different moments.

- In PureChocola, even in a definitive turn `spawn` is delayed until the end of the turn. In Chocola it happens immediately in a definitive turn.
- PureChocola expects programs to satisfy some conditions: a future, a transaction's root task, and a turn's root task must all join all their children. When these conditions are not satisfied in Chocola, an exception is raised. In PureChocola, we did not make these errors explicit.

7.6 | Conclusion

In this chapter, we presented PureChocola, a formalization of the semantics of Chocola. We described how Chocola's guarantees can be inferred from this formalization. Additionally, we created an executable implementation of parts of this formalization, to mechanically verify the isolation and intratransaction determinacy guarantees.

8

An Implementation of Chocola

We have implemented Chocola as a fork of Clojure.[1] In this chapter, we first describe 'standard' implementations of the separate models: futures and transactions as found in Clojure in Sections 8.1 and 8.2 respectively, and our simple implementation of actors in Section 8.3. Next, we describe how Chocola modifies these to support transactional actors in Section 8.4 and transactional futures in Section 8.5. Further, Section 8.6 lists the parts of Clojure that Chocola is compatible with.

Clojure is partially implemented in Clojure itself, and partially in Java. A brief description of Clojure is given in Appendix B. Clojure uses Java's concurrency primitives extensively, including futures, thread pools, and atomic integers; we do not discuss their internal implementation here.

Throughout this chapter, we use the syntax and semantics of Chocola. In some cases our syntax and semantics differ slightly from those of standard Clojure; Appendix D enumerates these differences. Note also that the code in all listings in this chapter has been abridged, modified, and documented for clarity.

8.1 | Futures

Clojure's futures use Java's thread pools and futures internally. `fork` and `join` are implemented as follows:

[1] It is available online at http://soft.vub.ac.be/~jswalens/chocola/. Chocola is open source, under the Eclipse Public License.

`fork` `fork` is a Clojure macro that translates `(fork body)` into `(future-call (fn [] body))`.[2] `future-call` submits this function to a thread pool, returning a Java `Future`[3].

Every task corresponds to a thread on a cached thread pool[4]. In a cached thread pool, threads are reused: when a task is finished, its thread remains active and will be reused for new tasks. This decreases the time needed to create new threads, at the cost of higher memory usage.

`join` When a task finishes, its future is resolved to the final value. `(join f)` calls the `Future`'s `get` method, which waits for the future to be resolved and then returns its result.

8.2 Transactions

Clojure implements transactions using the MVCC algorithm, which we already discussed in Section 2.4.4 on page 27. In this section, we sketch how Clojure implements this algorithm. We aim only to describe the gist of the algorithms and their implementation, therefore many details have been elided.

Clojure implements transactional variables as `Ref`s. An overview of the class `Ref` is given in Listing 8.1. Transactions are represented by the class `Transaction`, shown in Listings 8.2 and 8.3.

There is a global, shared clock, implemented using the atomic integer `lastPoint`, a static field of `Transaction`. The current transaction is stored in the thread-local variable `Transaction.CURRENT`.

A transaction's data structures A transaction contains several data structures to store its modifications: `sets`, `commutes`, and `ensured` contain the refs modified using `ref-set`, `commute`, and `ensure` (the last two are Clojure-specific and not discussed in this dissertation), while `vals` contains the accompanying values. Together these correspond to the local store δ of our operational semantics.

Transactional variables and operations Each ref contains a limited history of previous values and the time at which they were set. These values are called `TVals`. Thus, transactions can read an older version of a ref even after it has been overwritten, preventing conflicts. Further, refs contain a read–write lock that allows multiple transactions to read the ref but only one to write to it.

[2] Everywhere we write `fork`, Clojure uses `future`.

[3] https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html

[4] https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool()

```
 1 public class Ref extends ARef implements IFn, Comparable<Ref>, IRef {
 2     // A TVal is one entry in the history of this ref
 3     static class TVal {
 4         Object val; // Value
 5         long point; // Write point: time at which this value was valid
 6         TVal prior; // TVals form a circular doubly linked list: pointers to the prior...
 7         TVal next;  // ...and next values
 8         // Methods elided
 9     }
10 
11     // The history of the ref: this points to the last one TVal, they form a linked list.
12     TVal tvals;
13     // A read-write lock: transactions that read this ref take a read-lock, those that write take a write-lock.
14     // Hence, only one transaction can write at a time.
15     ReentrantReadWriteLock lock;
16     // Transaction that owns the write-lock.
17     Transaction tx;
18 
19     // Some methods elided
20 
21     // Write to a ref.
22     Object set(Object val) {
23         Transaction t = Transaction.CURRENT.get(); // Get the active tx (elided: otherwise throw an exception)
24         return t.doSet(this, val);
25     }
26     // Functions deref, alter, commute, and ensure are similar.
27 }
```

Listing 8.1: The class `Ref`, which implements transactional variables.

All operations on refs implement a similar pattern. The operation, e.g. `(ref-set r v)`, is first translated to a method call on the ref, e.g. `r.set(v)` (line 22). This method checks whether a transaction is running and forwards the operation to the current transaction, e.g. `tx.doSet(r, v)`. If no transaction is running, an exception is raised.

When reading a ref, `doGet(r)` searches its value first in the transaction's `vals`, which contains its in-transaction value in case it was written to during the transaction. Otherwise, it searches the ref's history for the most recent value before the transaction's read point. If no recent enough value exists, the transaction must retry.

Modifying a ref, in `doSet(r, v)`, adds the ref to `sets` and its value to `vals`, if a write lock can be acquired on the ref. If this lock is held by another transaction, the most recent of both aborts (based on their start point), and will retry when the older transaction has finished. This is an *early abort*: an unavoidable conflict is detected before the transaction has finished, increasing performance by avoiding computations that are doomed to fail. This raises a `RetryException`.

Transaction A transaction `(atomic e)` is translated into `Transaction.runInTx(fn)`, where `fn` corresponds to a function executing `e`. If a transaction is running already, the contents of the nested transaction are simply executed within the context of the outer transaction, implementing closed nesting (see Section 3.3.3, page 64). Otherwise, a new transaction is created and its `run` method is called.

```
public class Transaction {
  // Global, shared, atomic integer containing time. Incremented each time a transaction starts, retries, or commits.
  static AtomicLong lastPoint;

  // The current transaction is stored in a thread-local variable. (If no transaction is active, this is null.)
  static ThreadLocal<Transaction> CURRENT;

  AtomicInteger status;  // RUNNING, COMMITTING, RETRY, KILLED, or COMMITTED.
  long readPoint;   // The read point: only read refs older than this time.
  long startPoint;  // Read point of first attempt. Used when two tx's conflict, to stop the newer one.

  HashMap<Ref, Object> vals;              // In-transaction values of set and commuted refs
  HashSet<Ref> sets;                      // Modified refs
  TreeMap<Ref, List<CFn>> commutes;  // Commuted and ensured refs (not further considered here)
  HashSet<Ref> ensures;

  static class RetryEx extends Error {} // Exception thrown when a transaction should retry

  // Acquire write lock on ref. If the lock cannot be acquired within 100 ms, this causes the transaction to retry.
  void tryWriteLock(Ref ref) { … }
  // Acquire write lock on ref. If a conflict with another transaction occurs, this aborts the most recent of both.
  Object lock(Ref ref) { … }

  Object doGet(Ref ref) {
      if (vals.containsKey(ref)) // This ref was modified in this tx: return that value.
          return vals.get(ref);
      try {
          ref.lock.readLock().lock();
          Ref.TVal ver = ref.tvals;
          do { // Find the latest version of the ref before this tx's read point.
              if (ver.point <= readPoint) return ver.val;
          } while ((ver = ver.prior) != ref.tvals);
      } finally {
          ref.lock.readLock().unlock();
      }
      throw new RetryEx(); // If we reach this point, no old enough version of the ref exists: the tx must retry.
  }

  Object doSet(Ref ref, Object val) {
      if (!sets.contains(ref)) { // If we haven't set the ref before, add it to sets and lock it.
          sets.add(ref);
          lock(ref);
      }
      vals.put(ref, val); // Store the new value.
      return val;
  }
  // doCommute and doEnsure elided

  // Run the given function in a transaction.
  static Object runInTransaction(Callable fn) throws Exception {
      if (CURRENT.get() != null) // If a tx is running already, this is a nested tx.
          // Don't create a new tx, but simply call the function (closed nesting).
          return fn.call();
      Transaction t = new Transaction();
      CURRENT.set(t);
      try { // Run the function in the new transaction.
          return t.run(fn);
      } finally {
          CURRENT.remove();
      }
  }
```

Listing 8.2: The original class Transaction (part 1 of 2).

```
  Object run(Callable fn) throws Exception {
      boolean committed = false;
      Object result = null; // Return value
      // Retry the transaction until it succeeds, or the maximum number of attempts is reached.
      for (int i = 0; !committed && i < RETRY_LIMIT; i++) {
          // Get read point and set status
          readPoint = lastPoint.incrementAndGet();
          if (i == 0) startPoint = readPoint;
          status.set(RUNNING);
          try {
              result = fn.call(); // Execute the transaction's body, store the return value.
              if (status.compareAndSet(RUNNING, COMMITTING)) {
                  commit();
                  committed = true; // End this loop
              }
          } catch (RetryEx retry) { // Swallow the retry exception: a new attempt will start in the next iteration.
          } finally {
              // Clear data
          }
      }
      if (!committed) throw Util.runtimeException(”Transaction failed after reaching retry limit”);
      return result;
  }

  void commit() {
      List<Ref> locked = new ArrayList<Ref>(); // Locked refs
      try {
          for (Ref ref : sets) { // Lock all set refs
              tryWriteLock(ref);
              locked.add(ref);
          } // If this did not throw an exception, we can commit.

          long commitPoint = lastPoint.incrementAndGet(); // Log current time
          for (Map.Entry<Ref, Object> e : vals.entrySet()) { // Commit values of set refs
              Ref ref = e.getKey();
              Object val = e.getValue();
              if (ref.tvals == null) {
                  // Ref doesn't have a history yet: add a TVal with the new value.
                  ref.tvals = new Ref.TVal(val, commitPoint);
              } else if (ref.histCount() < ref.maxHistory) {
                  // Ref has a history, but it's not full yet: add a TVal to the linked list.
                  ref.tvals = new Ref.TVal(val, commitPoint, ref.tvals);
              } else {
                  // History is full: recycle oldest TVal for new value and put it at the front of the circular linked list.
                  ref.tvals = ref.tvals.next;
                  ref.tvals.val = val;
                  ref.tvals.point = commitPoint;
              }
          }
          status.set(COMMITTED); // Update status
      } finally {
          // Unlock locked refs
      }
  }

  // Other methods elided
}
```

Listing 8.3: The original class `Transaction` (part 2 of 2).

Running a transaction consists of repeating a loop of attempts until it either succeeds or the maximal number of attempts is reached (10 000 in Clojure). An attempt consists of three steps. First, a new time point is generated: this is the read point of the attempt. Next, the body of the transaction is executed, which reads and writes to the transaction's data structures. Finally, the transaction attempts to commit. The commit protocol consists of the following steps:

1. Each modified `Ref` is locked if it was not already. (Refs modified using `commute` have not been locked, those with `ref-set` have.)
2. The global clock is increased; the current time is recorded as the commit point.
3. The new values are committed. If a ref's history is full, its oldest entry is deleted.
4. All locks are released.

If a `RetryException` is raised at any point during the transaction or its commit, all of the transaction's data structures are cleared and its locks are released, and a next attempt starts. When this happens due to a write–write conflict on a ref whose write lock is held by another transaction, the next attempt will only start after that transaction has released the lock.

8.3 | Actors

Clojure does not support actors. In this section, we briefly sketch how Chocola extends Clojure with support for (non-transactional) actors. It is a rather simple implementation, meant to demonstrate the semantics of the model but without further optimizations.

Actors Actors are instances of the `Actor` class, shown in Listing 8.4. An actor contains two components: its current behavior and its inbox.

Behaviors An actor's behavior is parameterized over two types of parameters: the internal state of the actor and the values constituting a message (as described in Section 2.5.1). Both lists of parameters are passed to the construct `behavior`, e.g. `(behavior [flights] [orig dest n] e)`. `behavior` is a macro that translates this into nested functions, in the example: `(fn [flights] (fn [orig dest n] e))`. To store its behavior, an actor keeps an instance of the class `Behavior` that combines this nested function and the actor's internal state in its `body` and `args` fields.

Inbox and messages The inbox is a queue of messages, protected using a lock (implemented using Java's `LinkedBlockingDeque`). Each message has a receiver (the actor to which it was sent) and a list of arguments.

```java
public class Actor implements Runnable {
  static class Behavior {
      IFn body;   // Function of function, containing turn’s body
      ISeq args;  // Internal state
      // Methods elided
  }
  static class Inbox {
      LinkedBlockingDeque<Message> q;
      void enqueue(Message message) { q.put(message); }
      Message take() { return q.take(); }
  }
  static class Message {
      Actor receiver;
      ISeq args;
      // Methods elided
  }

  static ThreadLocal<Actor> CURRENT;

  Behavior behavior;
  Inbox inbox = new Inbox();

  Actor(IFn behBody, ISeq behArgs) { behavior = new Behavior(behBody, behArgs); }

  static void doBecome(IFn behaviorBody, ISeq behaviorArgs) {
      Actor.CURRENT.get().become(new Behavior(behaviorBody, behaviorArgs));
  }
  void become(Behavior newBehavior) { behavior = newBehavior; }

  static void doSend(Actor receiver, ISeq args) {
      receiver.enqueue(new Message(receiver, args));
  }
  void enqueue(Message message) { inbox.enqueue(message); }

  static Actor doSpawn(IFn behaviorBody, ISeq behaviorArgs) {
      Actor actor = new Actor(behaviorBody, behaviorArgs);
      actor.start();
      return actor;
  }
  void start() { Agent.soloExecutor.submit(this); }

  // Once a new thread has been created for the actor, it runs this method.
  void run() {
      CURRENT.set(this);
      while (true) { // Continually take message from inbox and apply behavior.
          Message message = inbox.take();
          IFn behaviorInstance = (IFn) behavior.apply();
          behaviorInstance.applyTo(message.args);
      }
  }
}
```

Listing 8.4: The original class Actor.

Operations Each actor operation is translated into a static method call on `Actor`, e.g. `(become b v…)` is translated to `Actor.doBecome(b, [v…])`. All operations are split into two parts: a static method (prefixed with `do`) that is called where the operation appears and a method on the affected actor that executes the effect (`become`, `enqueue`, and `start`). Extracting the effect into a separate method will make it easier later on to delay the effect in a transaction or tentative turn.

The three operations on actors are straightforward:

- `become` updates the behavior of the current actor.
- `send` enqueues a message in the inbox of the receiver.
- `spawn` creates a new actor with the given initial behavior and spawns a new thread that runs the actor.

Running an actor When an actor is spawned, a thread is created that executes its `run` method. This consists of an infinite loop that takes a message from the inbox and calls the behavior's body with the actor's internal state and the message's values.

Future work Garbage collection of actors has not yet been implemented in our current prototype. When there are no more references to an actor, it cannot receive messages anymore. Once its inbox is fully processed, it will keep existing in memory, but it will be idle permanently.

8.4 | Transactional Actors

This section lists the changes that need to be made to the standard implementations of transactions and actors to support transactional actors. Listing 8.5 shows the modifications made to the class `Actor` to supports transactional actors. (The changes to `Transaction` are discussed in the next section.)

Each operation is implemented as defined in Chapter 5:

`become` In a transaction, `become` stores its effect in the current transactional future's `nextBehavior` field. This will only become active if the transaction succeeds.

`send` Messages have a dependency whose value depends on the context: in a transaction, that transaction is the dependency; in a tentative turn but outside a transaction, the dependency of the turn is used; and in a definitive turn it is null.

`spawn` In a transaction, spawned actors are stored in the transactional future's set of `spawned` actors. These actors are only actually started at commit time, if the transaction succeeds. In a tentative turn, spawned actors are stored in the actor, to be started if the turn ends successfully. In definitive turns actors are started immediately.

```
public class Actor implements Runnable {
  // Inner classes Behavior and Inbox are unchanged
  static class Message {
      Actor receiver;
      ISeq args;
      Transaction dependency;  // New, can be null
      // Methods elided
  }
  static class AbortEx extends Error {} // Thrown if the dependency aborted

  // Fields CURRENT, behavior, and inbox are unchanged. New:
  Transaction dependency;  // The current dependency (can be null)
  List<Actor> spawned;     // Actors spawned during this turn, if it is tentative

  // Constructor Actor unchanged

  static void doBecome(IFn behaviorBody, ISeq behaviorArgs) {
      Behavior b = new Behavior(behaviorBody, behaviorArgs);
      if (TransactionalFuture.CURRENT.get()) { TransactionalFuture.CURRENT.get().nextBehavior = b; }
      else                                   { Actor.CURRENT.get().become(b); }
  }
  // Method become unchanged
  static void doSend(Actor receiver, ISeq args) {
      Transaction dep;
      if (TransactionalFuture.CURRENT.get())   { dep = TransactionalFuture.CURRENT.get().tx; }
      else if (Actor.CURRENT.get().dependency) { dep = Actor.CURRENT.get().dependency; }
      else                                     { dep = null; }
      receiver.enqueue(new Message(receiver, args, dep));
  }
  // Method enqueue unchanged
  static Actor doSpawn(IFn behaviorBody, ISeq behaviorArgs) {
      Actor a = new Actor(behaviorBody, behaviorArgs);
      if (TransactionalFuture.CURRENT.get())   { TransactionalFuture.CURRENT.get().spawned.add(a); }
      else if (Actor.CURRENT.get().dependency) { Actor.CURRENT.get().spawned.add(a); }
      else                                     { a.start(); }
      return a;
  }
  // Method start unchanged

  void run() {
      CURRENT.set(this);
      while (true) {
          Message message = inbox.take();
          dependency = message.dependency;  // Can be null. If it not, this is a tentative turn.
          Behavior oldBehavior = behavior;  // Make copy to use in case of roll back
          try {
              IFn behaviorInstance = (IFn) behavior.apply();
              behaviorInstance.applyTo(message.args);
              if (dependency != null) { // This was a tentative turn
                  dependency.waitUntilFinished();
                  if (dependency.status.get() != Transaction.COMMITTED)
                      throw new Actor.AbortEx();
                  for (Actor actor : spawned) // Persist effects
                      actor.start();
              }
          } catch (AbortEx e) { // Roll back on abort
              behavior = oldBehavior;
          } finally {
              // Reset dependency and spawned
          }
      }
  }
}
```

Listing 8.5: The class Actor with support for transactional actors.

Processing a turn (in `run()`) has also changed slightly:

- At the start of a turn, the current behavior is copied.
- At the end of a tentative turn, the actor waits for the dependency to complete. If the dependency succeeds, the delayed spawned actors are started. If it aborts, the current turn is aborted, which means spawned actors are not started and any changes to the behavior are rolled back.

Our implementation is a prototype, designed to be sufficient to demonstrate the benefits of our approach in the next chapter, but without further optimizations. We can think of several potential optimizations. For example, when a message with a dependency is taken from the inbox, we could check whether its dependency has finished already. If it finished and succeeded, the turn does not need to be tentative, and if it aborted, the message can be discarded immediately.

8.5 | Transactional Futures

In this section, we describe which changes need to be made to the implementations of futures and transactions in order to support transactional futures and transactional actors. A new class `TransactionalFuture` implements such futures, and many of the data structures and operations on transactions moved from `Transaction` (shown in Listing 8.6) to `TransactionalFuture` (Listings 8.7 and 8.8).

`Transaction` and `TransactionalFuture`

Transaction The class `Transaction` now only contains one data structure: `futures`, the set of futures forked in the transaction. Additionally, it implements two important methods:

- `run` evaluates the transaction. This works as before except for two changes. First, instead of evaluating its body directly, a root future is created to encapsulate this. All transactional operations thus manipulate the data structures in that future or one of its children. Second, in a tentative turn a transaction will wait for the dependency and only commit if the dependency succeeded.
- `commit`: the transaction attempts to commit after the root future has finished. Because a transactional future only finishes after it has merged all of its children, all changes from all futures will have been merged into the root future before it commits. The commit protocol is almost unchanged, except that it uses the data structures of the root future and takes into account operations on actors.

```
public class Transaction {
    // These fields are unchanged:
    static AtomicLong lastPoint; // Global clock
    AtomicInteger status;        // RUNNING, COMMITTING, RETRY, KILLED, or COMMITTED
    long readPoint;              // The read point
    long startPoint;             // Read point of first attempt

    // Transaction's data structures (vals, sets...) moved to TransactionalFuture.
    // Instead, we only store the futures created in the transaction:
    Set<TransactionalFuture> futures = Collections.synchronizedSet(
        new HashSet<TransactionalFuture>());

    static class RetryEx extends Error {} // Exception thrown when a transaction should retry

    // Run the given function in a transaction.
    static Object runInTransaction(Callable fn) throws Exception {
        if (TransactionalFuture.CURRENT.get() != null) // Already in a transactional future: nested transaction
            return fn.call();
        Transaction tx = new Transaction();
        return tx.run(fn);
    }

    Object run(Callable fn) throws Exception { // Unchanged, except for the green lines
        boolean committed = false;
        Object result = null; // Return value
        // Retry the transaction until it succeeds, or the maximum number of attempts is reached.
        for (int i = 0; !committed && i < RETRY_LIMIT; i++) {
            // Get read point and set status
            readPoint = lastPoint.incrementAndGet();
            if (i == 0) startPoint = readPoint;
            status.set(RUNNING);
            try {
                // Instead of result = fn.call(); we now do this in a new future:
                TransactionalFuture f_root = new TransactionalFuture(this, null, fn);
                result = f_root.call(); // Wait for the future to complete
                Actor.abortIfDependencyAborted(); // In a tentative turn, wait for dependency and possibly abort
                if (status.compareAndSet(RUNNING, COMMITTING)) {
                    commit(f_root);
                    committed = true; // End this loop
                }
            } catch (RetryEx retry) { // Swallow the retry exception: a new attempt will start in the next iteration.
            } finally {
                // Clear data
            }
        }
        if (!committed) throw Util.runtimeException("Transaction failed after reaching retry limit");
        return result;
    }

    void commit(TransactionalFuture f_root) {
        // ...
        // Commit protocol unchanged, but using the data structures of the root future and extended with:
        if (f_root.nextBehavior != null)
            Actor.CURRENT.get().become(f_root.nextBehavior);
        for (Actor actor : f_root.spawned)
            actor.start();
    }

    // Other methods elided
}
```

Listing 8.6: The class Transaction with support for transactional futures and actors.

```
public class TransactionalFuture implements Callable, Future {
    // The current future, stored in a thread-local variable. (If no transaction is active, this is null.)
    static ThreadLocal<TransactionalFuture> CURRENT;

    Transaction tx;  // Transaction for this future
    Future future;   // Java Future that will resolve to this future’s value. (null for root future)
    Callable fn;     // Function executed in this future
    Object result;   // Result of future (return value of fn)

    static class Vals<K, V> { … } // Linked list of hash maps

    Vals<Ref, Object> snapshot;          // vals when this future was created (set in ancestors, read-only)
    Vals<Ref, Object> vals;              // In-transaction values of refs modified in this future
    HashSet<Ref> sets;                   // Modified refs
    TreeMap<Ref, List<CFn>> commutes;  // Commuted and ensured refs (not further considered here)
    HashSet<Ref> ensures;
    List<Actor> spawned;                 // Spawned actors
    Actor.Behavior nextBehavior;         // Last become (null if no become occurred)
    HashSet<TransactionalFuture> merged;  // Futures merged into this one

    // Create a new transactional future. This function runs in the parent; the future that is being created is not yet running.
    TransactionalFuture(Transaction tx, TransactionalFuture parent, Callable fn) {
        this.tx = tx;
        this.fn = fn;
        if (parent == null) { // Root future: snapshot is empty and vals start empty
            snapshot = null;
            vals = new Vals<Ref, Object>();
        } else { // Child future: snapshot = current parent vals, our and the parent’s vals ‘fork’ this
            if (!parent.vals.isEmpty()) {
                snapshot = parent.vals;
                vals = new Vals<Ref, Object>(parent.vals);
                parent.vals = new Vals<Ref, Object>(parent.vals);
            } else { // Optimization: if parent has not set anything, this can point straight to the parent’s ancestor,
                     // and parent can ‘re-use’ its vals. This avoids creating empty vals.
                snapshot = parent.vals.prev;
                vals = new Vals<Ref, Object>(parent.vals.prev);
            }
        }
        synchronized (tx.futures) { tx.futures.add(this); }
    }

    // Spawn future, called for forks: outside transaction regular future, in transaction a transactional future.
    static Future spawnFuture(Callable fn) {
        TransactionalFuture current = TransactionalFuture.CURRENT.get();
        if (current == null) { // Outside transaction
            return Agent.soloExecutor.submit(fn);
        } else { // In transaction
            TransactionalFuture f = new TransactionalFuture(current.tx, current, fn);
            f.future = Agent.soloExecutor.submit(f);
            return f;
        }
    }
```

Listing 8.7: The class `TransactionalFuture` (part 1 of 2).

```
    // Execute future in this thread, and wait for all sub-futures to finish. Called directly for the root future,
    // or in new thread for the others.
    Object call() throws Exception {
        try {
            CURRENT.set(this);
            result = fn.call();
            // Elided: wait for all futures in tx.futures to finish, by calling their get method.
        } finally {
            CURRENT.set(null);
        }
        return result;
    }

    // Join the future. Waits if necessary for the computation to complete, and then retrieves its result.
    Object get() throws ExecutionException, InterruptedException {
        // For a call to child.get(): this = child and CURRENT = its parent.
        future.get(); // Sets field result
        TransactionalFuture.CURRENT.get().merge(this); // Merge child (this) into parent (CURRENT)
        return result;
    }

    Object doGet(Ref ref) {
        Object val = vals.get(ref); // First look in vals (both current local store and ancestors),
        if (val == null) return requireBeforeTransaction(ref); // otherwise in ref's history.
        return val;
    }
    // Methods doSet, doCommute, and doEnsure are unchanged

    // Merge child into this.
    void merge(TransactionalFuture child) {
        if (merged.contains(child)) return; // Already merged

        // vals: add in-transaction-value of refs set in child to parent
        for (Ref r : child.sets) {
            Object v_child = child.vals.get(r); // Current value in child
            Object v_parent = vals.get(r); // Current value in parent: in its vals, else in the ref's history
            if (v_parent == null) v_parent = requireBeforeTransaction(r);
            Object v_original = child.snapshot.get(r); // Original value: in snapshot, else in ref's history
            if (v_original == null) v_original = requireBeforeTransaction(r);

            if (v_parent == v_original) { // No conflict, just take over value
                vals.put(r, v_child);
            } else { // Conflict
                if (r.getResolve() != null) // Call custom conflict resolution function if it was defined
                    vals.put(r, r.getResolve().invoke(v_original, v_parent, v_child));
                else // Default conflict resolution = prefer child
                    vals.put(r, v_child);
            }
        }
        sets.addAll(child.sets); // sets: add sets of child to parent
        spawned.addAll(spawned); // spawned: add actors spawned in child to parent
        if (child.nextBehavior != null) nextBehavior = child.nextBehavior; // Child did become: take it over
        merged.addAll(child.merged); // merged: add futures merged into child to futures merged into parent
        merged.add(child); // Child has been merged now
        // commutes and ensures elided
    }
}
```

Listing 8.8: The class TransactionalFuture (part 2 of 2).

Transactional future The class TransactionalFuture contains the data structures needed in a transactional future and implements both operations on transactional variables (doGet, doSet, doCommute, and doEnsure) and on transactional futures (spawnFuture for fork and get for join). It contains the following data:

- A thread-local variable CURRENT per future. A traditional implementation of transactions uses a thread-local variable per transaction, as every transaction only contains one thread. Here, we need a separate thread-local variable per future.
- A reference to its encapsulating transaction tx.
- Every TransactionalFuture has an associated Java Future future, which will contain its final value.
- When the future is forked, its body is wrapped in a function fn, which is then evaluated in a new thread.
- The variable result will allow other threads to read the value of this future.
- A transactional future contains the same data structures that were previously stored per transaction (vals, sets, commutes, and ensures), as well as its snapshot. The implementation of snapshot and vals, using the class Vals, is discussed in detail below.
- Delayed operations on actors are stored: the set of spawned actors and possibly the nextBehavior passed to become.
- Finally, a transactional future keeps track of the futures it merged.

Reading a transactional variable (doGet) now consists of looking up the ref in vals, which looks both in the local store and snapshot, up the tree of futures. If it cannot be found there, the ref's history is searched. The other transactional operations (doSet, doCommute, and doEnsure) are implemented as before.

Creating and joining a transactional future

Creating a transactional future TransactionalFuture's constructor is called once for the root future, and next when a parent future creates a child. The root future's snapshot is null and its vals starts empty. For other futures:

- The child's snapshot is the parent's current local store.
- The child's local store is an empty wrapper around the parent's local store.
- The parent's local store is modified to be a wrapper around its current local store. This ensures that further modifications in the parent are not visible by the child.

The construct (fork body) is a macro (written in Clojure) that wraps body in a function and calls spawnFuture with this function. Outside a transaction, this executes the body in a new thread as before. In a transaction, this creates a transactional future to execute on a new thread.

Joining a future When `(join f)` appears in a transaction, it is translated to `f.get()`. This method waits until the child has finished (that is, until its Java Future `future` has resolved), and then merges the child into the parent. This consists of merging their data structures:

- The child's local store (`vals`) is merged into the parent. To do so, we look up the current values in child and parent, and the value in the snapshot. In case the parent has not modified its ref, the child's value can simply be taken over. Otherwise, there is a conflict, so the ref's conflict resolution function is called. In case the developer did not specify a custom conflict resolution function, we default to the one that picks the child's value.
- The child's set of modified refs (`sets`) is joined with the parent's.
- The child's `spawned` actors is joined with the parent's.
- If the child did a `become`, its effect is taken over by the parent. (Even if the parent also did a `become`, the child's value is preferred.)
- The child's set of `merged` futures is also joined into the parent.

Finally, the child future is added to the set of merged futures of the parent. This will prevent subsequent joins from merging the child's data structures again.

Implementation of snapshot and local store that avoids duplication

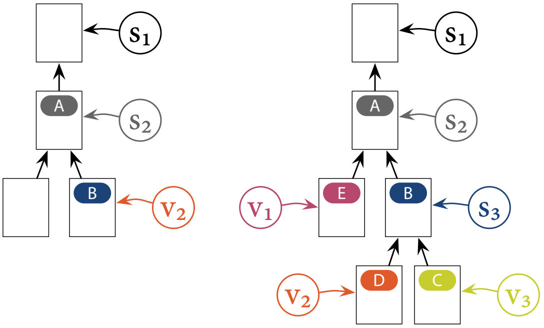
`snapshot` and `vals` are the most frequently used data structures in a transactional future, as they are accessed for each read and write. Conceptually, the snapshot and local store of a task are copied from its parent when it is created. Here we describe how our actual implementation avoids creating duplicates.

Figure 8.9a lists a program that creates three tasks that each modify some refs. Figures 8.9b and 8.9c illustrate how this is stored in memory after lines 4 and 9 respectively. We write s_i and v_i for the snapshots and values of task i. Each data structure consists of a linked list of hash maps. We exploit the fact that snapshots are immutable to share some of these hash maps between the data structures.

The transaction starts with a root task with an empty snapshot s_1 and local store v_1. Line 2 sets the ref gray to A, which is reflected in v_1. Next, a second task is forked, with its snapshot s_2 a duplicate of v_1 and an empty local store v_2. On line 4, the second task sets blue to B. Consequently, after this step, s_1 is empty, v_1 and s_2 both contain A, and v_2 consists of A and B. Figure 8.9b illustrates the data structures of the two tasks at this point. The snapshots are shared between the two tasks: these are immutable structures only used for look-up. The values of both tasks, v_1 and v_2, consist of a linked list that first contains a hash map that stores their private changes and next contains the shared snapshots. When ref B is updated in the second task, it is updated in the first hash map

```
1  (atomic
2    (ref-set gray A)
3    (fork
4      (ref-set blue B)
5        (fork
6          (ref-set green C))
7      (ref-set orange D)
8      …)
9    (ref-set purple E)
10   …)
```

(a) Code example of a transaction with three nested tasks.



(b) Data after line 4.

(c) Data after line 9.

Figure 8.9: In the code example three tasks are created. Each task contains a snapshot, which is immutable throughout the task's lifetime, and values, to which updated values for refs are written. (b) and (c) illustrate how the data structures are stored in memory at different points in time using linked lists of hash maps (each rectangle is a hash map, the arrows link them).

pointed to by v_2. When a ref is read in the second task, we iterate over the linked list pointed to by v_2, up the tree, until the ref is found.

Creating a new task, as on line 5, is now a matter of modifying some pointers. When task 3 is forked by task 2, the node that represented v_2 becomes the snapshot s_3 of the new task, with two empty children to contain the new values of tasks 2 and 3. After line 9, in Figure 8.9c, tasks 2 and 3 have updated their values with D and C respectively. Hence, v_3 now consists of the new values of task 3, the snapshot of task 3, the snapshot of task 2, and the snapshot of task 1. This is as in our operational semantics: reading a ref looks it up first in the current task's local store, and next in the snapshot made when each ancestor was forked.

By representing the `snapshot` and `vals` data structures of a transactional task as a linked list of hash maps, the memory overhead of duplicated entries is eliminated. In exchange, the look-up time slightly increases as we need to iterate over the list of maps. The time to update a value is unchanged: a write happens directly in the first hash map. Forking is a matter of creating two new maps and adjusting an existing pointer. Joining still means copying values and potentially resolving conflicts, as explained earlier.

Additionally, we performed another optimization for a common use case. In many programs it is common to create several tasks immediately after another, for example in a parallel map as in the Labyrinth program of Section 4.2. This leads to a sequence of empty nodes in the tree. Instead of pointing a new child to an empty node, we directly point to the previous non-empty node. This optimization avoids the need to traverse empty nodes on look-ups. It is a safe optimization as non-leaf nodes in the tree are always snapshots and thus immutable.

8.6 | Compatibility with Clojure

As Chocola is implemented on top of Clojure, developers can use every feature of Clojure in Chocola. However, this is not always safe: the guarantees of Chocola can be broken. We list which features of Clojure can be used safely inside a future, transaction, or actor.

✓ The **functional subset** of Clojure, i.e. any built-in function which has no side effect, can safely be used in Chocola. As these functions are deterministic, nonblocking, and do not access shared state, they can be embedded in concurrent futures, transactions, and actors without breaking Chocola's guarantees.

✓ Clojure provides **futures** and **transactions**. When used separately, Chocola provides the same semantics as Clojure. When they are combined, Chocola and Clojure have a different semantics: in Clojure the problems described in Chapter 4 occur while these are solved in Chocola.

✗ Any **other concurrency model** provided by Clojure is incompatible with Chocola, including atoms, agents, promises, channels, and thread-local variables. Using these concurrency models within the three models provided by Chocola can break their guarantees, as was shown extensively in Chapter 3.

✗ In general, any other **functions with side effects** cannot safely be used in Chocola. These include input/output and Clojure's interoperability with Java. Some examples are: the use of non-deterministic input breaks determinacy; mutating a Java `LinkedList` breaks determinacy, isolation, and low-level race freedom; and waiting for the response to an HTTP request can break deadlock freedom.

8.7 | Conclusion

In this chapter we presented a prototypical implementation of Chocola, built on top of Clojure. We described how standard implementations of the three separate models need to be modified to implement transactional futures and transactional actors. In the next chapter, we will use this implementation to benchmark several applications that use transactional futures and actors.

9 Evaluation

In this chapter, we quantitatively measure the performance benefits of using Chocola and qualitatively assess the effort required to use it. We show that programs can exploit parallelism more efficiently using Chocola, as multiple concurrency models can be combined. In Section 9.1, we describe our methodology and experimental set-up. Next, we look at three applications – Labyrinth (Section 9.2), Bayes (Section 9.3), and Vacation2 (Section 9.4) – that use transactions and show that introducing transactional futures or actors improves their performance by exploiting additional parallelism. In Section 9.5, we assess the developer effort required to make these changes.

9.1 Methodology and Experimental Set-Up

In this section, we describe the goal and criteria of our evaluation (Section 9.1.1), how we selected specific benchmarks (Section 9.1.2), and how we transformed them to use Chocola (Section 9.1.3). Furthermore, we specify the hardware and software used for our experiments (Section 9.1.4).

9.1.1 Evaluation Goal and Criteria

The aim of this evaluation is to demonstrate that, in existing programs that use transactions, additional parallelism can be exploited within transactions by introducing transactional futures and actors, without fundamentally changing the design of the program. Hence, we pick a number of existing applications that use transactions, and compare the original application with a transformation of the program that introduces transactional futures or actors.

Our evaluation focuses specifically on programs with transactions. This is because the combination of actors and futures, discussed in Chapter 3, did not pose significant issues and did not require major changes to the semantics of the separate models. The evaluation therefore focuses on applying transactional futures and transactional actors, by introducing futures and actors in programs that already use transactions.

We compare the original and transformed programs using two criteria:

Performance The end goal of introducing additional parallelism in transactions is to increase performance. To gauge the improvement in performance, we compare the total execution time of the transformed program with that of the original. We perform several measurements in which we vary the number of threads, to simulate environments in which a varying number of cores are available. From these results, we calculate the **speed-up**: this is the execution time of a reference result (usually the original program with a single thread) divided by the execution time of the result with *n* threads. The speed-up indicates how much faster the program becomes when using more cores and when applying our techniques. The advantage of plotting speed-up instead of execution time is that it makes it easier to compare results on higher thread counts.

Developer effort Afterwards, in Section 9.5, we qualitatively assess the effort that is required from the developer to use our techniques. We do this by describing which changes were made to introduce futures and actors in these programs. The goal of this qualitative evaluation is to compare the effort of introducing transactional futures and transactional actors in a program with transactions with the effort of introducing (regular) futures and actors in a program without transactions.

9.1.2 Selection of Benchmarks

As said in the previous section, our evaluation starts from existing benchmarks that use transactions. We use the STAMP benchmark suite as a basis: STAMP (Stanford Transactional Applications for Multi-Processing) is a benchmark suite consisting of eight applications that use transactional memory [Minh et al. 2008], commonly used to compare the performance of transactional systems. These applications are based on real-world scenarios in which transactions are used and cover a breadth of application domains. Moreover, they cover a range of characteristics of transactional programs, such as frequent or rare use of transactions throughout the program, long or short transactions, and high or low contention.

Table 9.1 lists three characteristics of the eight applications in the benchmark suite: the average length of the transactions, the average proportion of the program's execution time that is spent in transactions, and the average number of retries per transaction (a measure of contention). These numbers were gathered by Minh et al. [2008] on a simulated 16-core system.

Application	Tx length (mean # of instructions/tx)	Average time in tx	Contention (retries/tx for lazy STM)	Domain
Labyrinth	219,571 █	100% █	0.94 ▂	Engineering
Bayes	60,584 ▇	83% ▇	0.59 ▂	Machine learning
Yada	9,795 ▆	100% █	2.51 ▆	Scientific
Vacation-high	3,223 ▅	86% ▇	0.00 □	Transaction processing
Genome	1,717 ▅	97% █	0.14 ▁	Bioinformatics
Intruder	330 ▄	33% ▃	3.54 █	Security
Kmeans-high	117 ▄	7% ▁	2.73 ▇	Data mining
SSCA2	50 ▂	17% ▂	0.00 □	Scientific

Table 9.1: Characterization of the STAMP applications, abridged from Minh et al. [2008]. These numbers were gathered on a simulated 16-core system, and are color-coded high, medium, low. (This is an extended version of Table 4.4 on page 72.)

To evaluate transactional futures, we are interested in two characteristics in particular. First, the execution time spent in transactions: when most of the program's execution occurs in transactions, we can assume that those transactions execute performance-critical parts of the application. To further parallelize these applications, it will be necessary to introduce parallelism *within* the transactions. Second, we look at the transaction length: long-running transactions may benefit from more fine-grained parallelism, as in these cases the benefits of introducing parallelism may outweigh its costs.

As shown in Table 9.1, five out of the eight applications in the STAMP suite spend a large proportion of time in transactions, three of which have long-running transactions: Labyrinth, Bayes, and Yada. In Sections 9.2 and 9.3, we study Labyrinth (which already featured as the running example of Chapter 4) and Bayes.

We also examined the Yada application. While its transactions have a relatively long execution time, they contain sequential dependencies that make it difficult to parallelize. We could therefore not transform the program to introduce transactional futures without significantly changing the design of the program. (Our transformations are discussed below.)

To evaluate transactional actors, we look at a slightly reduced version of the Vacation benchmark, called Vacation2, in Section 9.4. This application already served as a running example in Chapter 5. It is suited to the use of transactional actors: the actor model naturally encodes the event-driven way in which a travel reservation system processes requests from customers. It is also an application which spends much of its total execution time in transactions, thus it may benefit from offloading parts of these transactions to different actors.

9.1.3 Transformation of Benchmarks to Use Chocola

For these case studies, we first port the STAMP applications from C to Clojure. This translation retains the design and algorithms of the C program. Afterwards, we introduce transactional futures and transactional actors where applicable, by performing the following steps:

1. First, we search for the transaction in which the largest proportion of the program's execution time is spent. In our cases, this is always the transaction that performs the 'main' task of the program.
2. Using profiling tools, we search for the part of the transaction in which most time is spent. In our cases, this is always a loop.
3. We try to parallelize this loop. To determine how to do this, we examine whether there are dependencies between the iterations of the loop, which occur when an iteration uses the result of a previous iteration. There are three cases (illustrated in Figure 9.2):

   - When the iterations are *independent*, we parallelize the loop. This is easy to do: each iteration can be processed in parallel. If the loop contains many short iterations, we first divide the iterations into a smaller number of partitions and then process each partition in parallel. This occurs in the Bayes and Vacation2 benchmarks.
   - When there are *dependencies* between the iterations, but the program follows a *standard algorithm* for which a parallel version exists in literature, we replace the sequential algorithm with its parallel equivalent. This occurs in the Labyrinth benchmark, in which a sequential breadth-first search is changed into a parallel search algorithm.
   - When there are *dependencies* between the iterations and the program uses a *custom algorithm*, we reach a negative result and do not introduce futures or actors. This is the case for the Yada benchmark. We will therefore not look at this application in more detail in the rest of this chapter. Note that this does

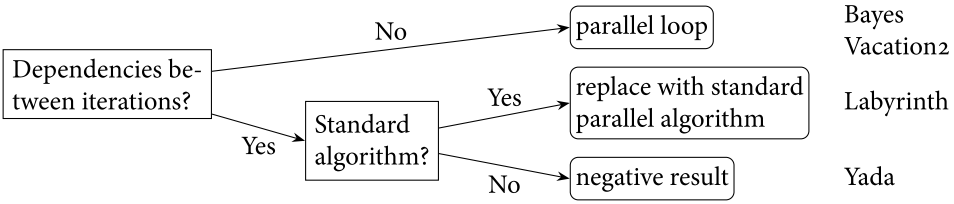


Figure 9.2: This diagram illustrates the steps taken to transform the STAMP benchmarks into a version that uses transactional futures or transactional actors.

not necessarily mean that the loop cannot be parallelized, it might also mean that specific domain expertise is required to parallelize the custom algorithm.

The code of all benchmarks is available online, both the original as well as the transformed version.[1]

In all experiments, the transformed benchmarks are compared with the original benchmark in Clojure. We do not compare to the implementations in C, as the performance characteristics of Clojure and C are so wildly different that they render a comparison meaningless.[2]

Each benchmark in the STAMP suite further has several parameters that can be varied. We always pick fixed, representative values for the parameters, based on the defaults set by Minh et al. [2008]. They are noted for each experiment as we go along.

9.1.4 Hardware and Software Set-Up

We describe the hardware and software used to run our experiments. Due to the timing of the development of these ideas, we implemented transactional futures and transactional actors separately for these experiments. The implementation of transactional futures extends Clojure 1.6 while transactional actors extend Clojure 1.8.

Transactional Futures (Labyrinth and Bayes) The Labyrinth and Bayes experiments were originally reported in Swalens et al. [2016]. They ran on a machine with two Intel Xeon E5520 processors, each containing four cores with a clock frequency of 2.27 GHz and a last-level cache of 8 MB. HyperThreading was enabled, leading to a total of 16 logical threads. The machine has 8 GB of memory. Our transactional futures are built as a fork of Clojure 1.6.0, running on the Java HotSpot 64-Bit Server VM (build 25.66-b17) for Java 1.8.0. On this machine, we used the JVM's G1 garbage collector, which is optimized for multiprocessor machines with a large memory.[3]

Transactional Actors (Vacation2) The Vacation2 benchmark, originally reported in Swalens et al. [2017], ran on a machine with four AMD Opteron 6376 processors, each containing 16 cores with a clock frequency of 2.3 GHz and a last-level cache of 16 MB, resulting in a total of 64 cores. The machine has 128 GB of memory. We implemented transactional actors as a fork of Clojure 1.8.0, running on the OpenJDK 64-Bit Server VM (build 25.131-b11) for Java 1.8.0. This experiment used the JVM's default garbage collector.

[1] https://github.com/jswalens/labyrinth • https://github.com/jswalens/bayes
https://github.com/jswalens/yada • https://github.com/jswalens/vacation2

[2] Just to name a few: a statically typed programming language with manual memory management vs. a dynamically typed language with garbage collection, a program written in an imperative style vs. one written in a functional style, and STM implementations based on different algorithms.

[3] http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html

Performing statistically rigorous performance experiments on virtual machines with just-in-time compilation, such as the Java Virtual Machine in our case, is notoriously difficult [Georges et al. 2007, Barrett et al. 2017]. As we are interested in the execution time of ephemeral benchmark programs that process some input to produce some output, and not the steady-state performance of continually running applications, our experiments all measure the execution time for all or part of the program from start to finish. Each measurement corresponds to a new execution of the program, for which a new instance of the JVM is started.

9.2 | Labyrinth (Transactional Futures)

The Labyrinth benchmark was already introduced in Section 4.2 of Chapter 4. The goal of the application is to connect given pairs of points in a grid using non-overlapping paths. For each pair, a breadth-first search is executed in a new transaction. In Section 4.3, we discussed how each iteration of the breadth-first search can process cells in the grid in parallel, leading to a transformation of the program that performs a parallel search using transactional futures. We further optimized this solution to first distribute the cells into partitions and then process these partitions in parallel.

We ran the experiments on a three-dimensional grid of $64 \times 64 \times 3$ with 32 input pairs. We varied two parameters:

- t: the number of worker threads that process input pairs in parallel, and
- p: the maximal number of partitions created on each iteration of the parallel search (only for the version with parallel search).[4]

All other parameters are the defaults of the STAMP version.

In the original version, only the parameter t influences the amount of parallelism. The maximal ideal speed-up is therefore t: in an ideal case where no transactions fail and the overhead is zero, we can expect a speed-up of maximally t. In the version with parallel search, the parameter p affects the parallelism as well. Each of the t worker threads can create at most p partitions, therefore the maximal number of threads and thus the maximal ideal speed-up in the version with parallel search is $t \times p$.

In Figure 9.3 we measure the speed-up when running the program with several values of t and p. The speed-up is calculated relative to the version with sequential search and only one worker thread ($t = 1$), which takes 27.1 s. For the version that uses sequential search, the number of worker threads is increased (blue line). For the version with parallel search, both the number of partitions (different lines) and the

[4] To minimize the overhead of forking futures, we ensure that each partition contains at least 20 elements.

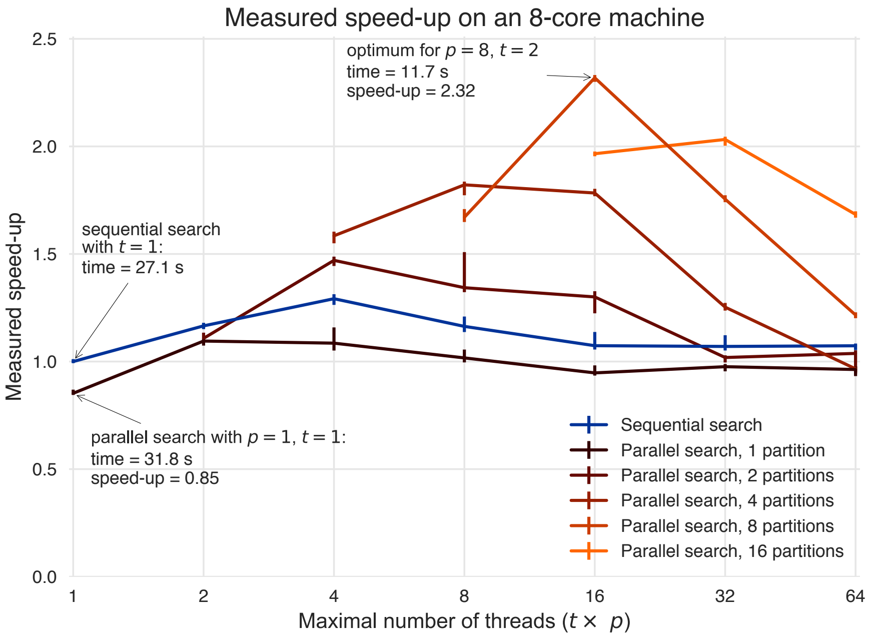


Figure 9.3: Measured speed-up of the Labyrinth application for the version with sequential search (blue line) and parallel search (other lines), as the total number of threads ($t \times p$) increases (logarithmic scale). Each point on the graph is the median of 30 executions, the error bar depicts the interquartile range.

number of worker threads (different points on the same line) are varied. The x axis denotes the maximal number of threads, which is t for the sequential search and $t \times p$ for the parallel search. In an ideal case, the measured speed-up would be equal to the maximal number of threads.

The blue line depicts the results of the original version of the Labyrinth application. Increasing the number of threads causes only a modest speed-up, because they find overlapping paths and consequently need to be rolled back and reexecuted. This is shown in Figure 9.4, which lists the average number of attempts per transaction. If there is only one thread, each transaction executes only once, but as the number of threads increases each transaction reexecutes several times on average. For 16 threads, the average transaction executes 2.10 times, which means it rolls back more than once. This curtails any potential speed-up.

In the version with parallel search, as the parameter p increases, the speed-up improves, for small values of p. Each transaction now spawns p tasks, and consequently each transaction can finish its execution faster. On the tested hardware, an optimal speed-up of 2.32 is reached for $t = 2$ and $p = 8$ (11.7 s absolute time), when

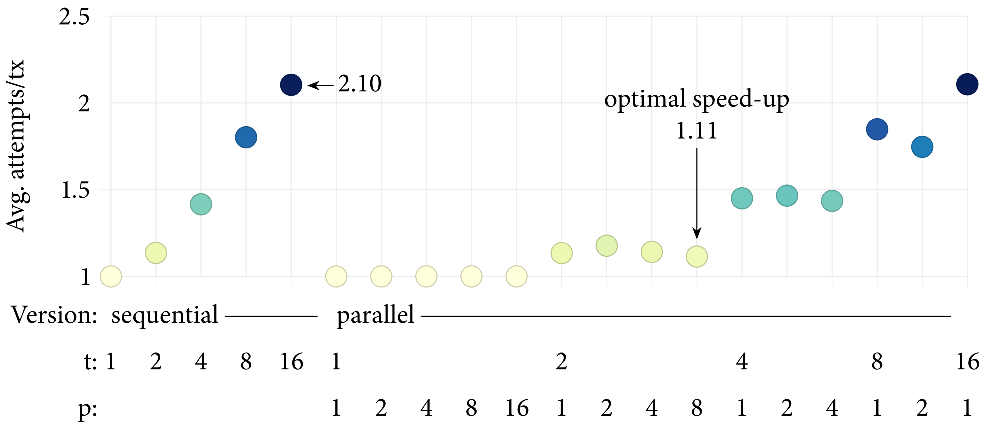


Figure 9.4: Average number of attempts per transaction for different values of t (transactions executing simultaneously) and p (partitions per transaction) for the Labyrinth application.

two worker threads process elements and create up to eight partitions. For this case, the number of conflicts is low: each transaction executes 1.11 times on average (Figure 9.4).

Further increases in p lead to worse results: the additional parallelism does not offset the overhead of forking and joining tasks. Joining transactional futures is expensive for this benchmark, as conflicts between the tasks are likely,[5] and each conflict calls a conflict resolution function (the minimum, as explained in Section 4.3.2).

The parallel search with $p = 1$ (which does not actually search in parallel as only one partition is created) is slower than the sequential search. The version with parallel search with 1 partition and 1 worker thread has an execution time of 31.8 seconds, compared to only 27.1 s for the corresponding version with sequential search. This is due to the difference in used algorithms: the parallel algorithm creates several sets on each iteration to keep track of the work queue, while the sequential algorithm uses one list throughout. The difference between the blue and the light yellow line corresponds to this cost.

These results demonstrate two ways in which transactional futures can increase performance. First, the execution time of each transaction decreases by exploiting parallelism in the transaction. Second, the lower execution time of a transaction also means that the cost of conflicting transactions is decreased, as each attempt takes less time. By varying the two parameters t and p, we can find an optimum between running several transactions simultaneously but risking conflicts (t) and speeding up the transactions internally but with more costly fine-grained parallelism (p).

[5] In the parallel breadth-first search algorithm, a conflict occurs each time two cells expand into a shared neighbor.

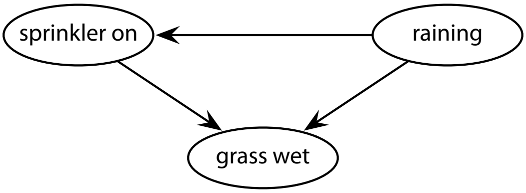


Figure 9.5: Simple example of a Bayesian network consisting of three random variables. The edges represent conditional dependencies: the grass is more likely to be wet if the sprinkler is on and/or if it rains, the sprinkler is more likely to be off if it rains.

9.3 | Bayes (Transactional Futures)

The Bayes application is another benchmark from the STAMP suite, which implements an algorithm that learns the structure of a Bayesian network given observed data [Chickering et al. 1997, Minh et al. 2008]. We describe this application and our transformation, before discussing the performance results.[6]

Original program Figure 9.5 illustrates a Bayesian network: a graph in which the nodes represent random variables (variables that have some probability of being true, e.g. the probability that it rains) and the edges correspond to their conditional dependencies (e.g. if it rains, the probability of the grass being wet is higher). The Bayes application 'learns' such a network given some input data: it starts from a network of variables without dependencies and adds dependencies to maximize its ability to predict the input data.

In the program, each variable of the network is represented as a transactional variable that contains references to its parents and children in the network. Initially, there are no dependencies between the variables. A shared work queue contains the dependencies to insert next, and is initialized to one dependency per variable.

t worker threads process the work queue in parallel: they insert the dependency into the network, and then calculate which dependencies (if any) could be inserted next, appending the best candidate to the work queue. The best candidate dependency is the one that maximizes a score function that calculates the capability of the network to estimate the input data. This is encapsulated in a transaction to prevent two dependencies from being added to the same variable simultaneously.

[6] We ran the Bayes application with the default parameters from the original STAMP version, except: (1) the number of variables was increased from 32 to 48 to increase the number of cores that can be used, (2) the number of records was decreased from 4096 to 512 to decrease the running time of the experiment, and (3) the maximum number of parents per node was increased from 4 to 5 to increase the chance of conflicts.

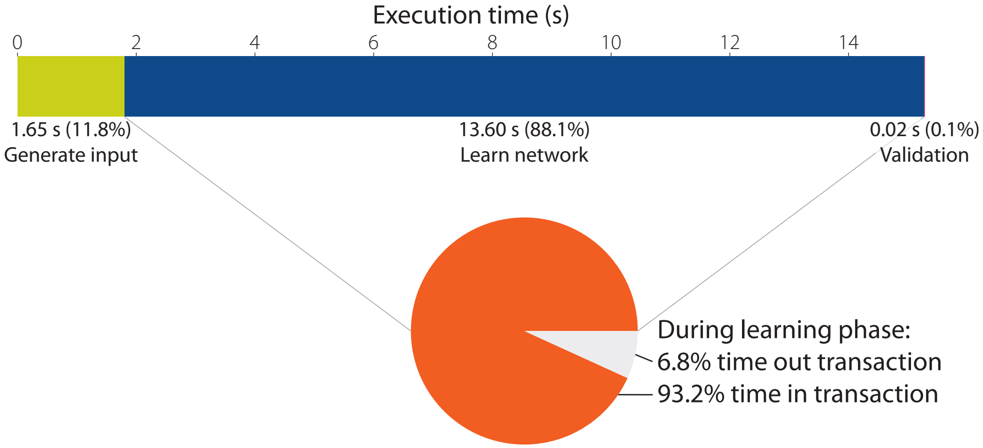


Figure 9.6: Proportion of time spent in different parts of the Bayes application (with $v = 48$).

Dependencies are inserted until the work queue is empty. As more dependencies are discovered, connected subgraphs of dependent variables form in the network.

Our transformation Figure 9.6 illustrates a typical execution of the program. Before the algorithm starts, the application generates the input data, taking 11.8% of the total time. Then, the t worker threads process the work queue in parallel to learn the dependencies, taking 88.1% of the execution time. Finally, the solution is validated, taking just 0.1% of the time. As learning the network takes the most time, we focus on this part only. In that part, 93.2% of the execution time is spent in the transactions that determine the best next dependency. To optimize the program, we therefore focus on this transaction.

Looking at the code, we see that the transaction in question contains a loop that calculates the score for each candidate and then selects the maximum. Each of the iterations of this loop is independent, and can therefore run in parallel in a transactional future. We modified the application to do this (illustrated in Listing 9.7), and will compare this version with a parallel loop to the original version of the benchmark.

```
(atomic
  …
  (for [from-id (range (:n-var adtree))]
      (compute-local-log-likelihood …)))
```

(a) Fragment of the original Bayes benchmark.

```
(atomic
  …
  (parallel-for [from-id (range (:n-var adtree))]
      (compute-local-log-likelihood …)))
```

(b) Bayes with transactional futures.

Listing 9.7: The Bayes application before and after the introduction of transactional futures.

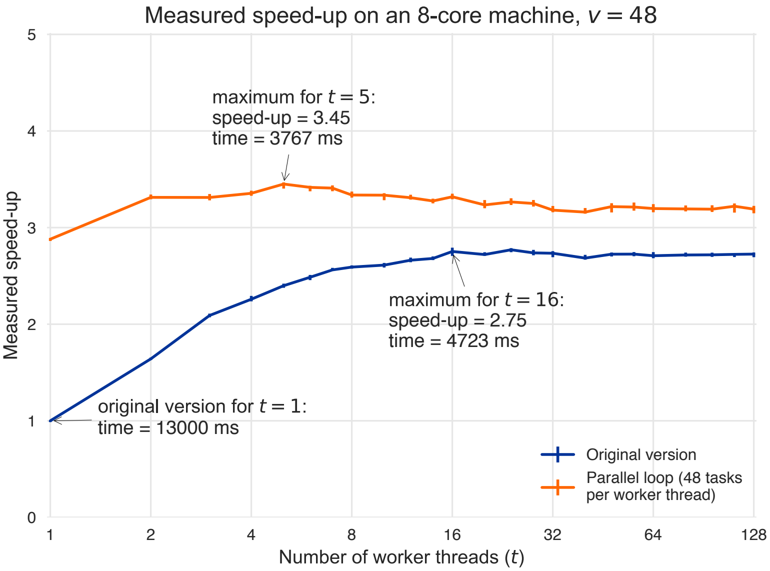


Figure 9.8: Measured speed-up of the learning phase for the Bayes application, as the number of threads increases. The blue line shows the original version. The red line shows the version with a parallel for loop, where each of the (at most) 48 iterations is executed in parallel.

Performance In Figure 9.8, we measure the speed-up of the learning phase as the number of worker threads (t) increases, for a network of 48 variables. The blue line is the original version: t threads process dependencies in parallel. The red line shows the version in which the loop is executed in parallel. Here, in each transaction, up to v transactional tasks run in parallel, where v is the number of Bayesian variables in the network (48 in our experiment). Therefore, the maximal ideal speed-up in the original version is t, while in the version with the parallel loop it is $t \times v$.

The speed-up of the original version (blue line) increases as number of threads increases, up to a speed-up of 2.75 for 16 threads (decreasing the execution time from 13 s for one thread to 4.7 s for 16 threads). After this point, the speed-up plateaus. By examining the execution of the program, we find that even though a larger number of worker threads are created, only a limited number of them actually perform any work. The others are idle as not enough work is available after a certain point in the execution of the program.

In the version with parallel tasks, we see that even when there is only one worker thread processing one transaction at a time ($t = 1$), the parallelization of its internal loop produces a speed-up of 2.88. By increasing the number of worker threads, a maximal speed-up of 3.45 is achieved for 5 worker threads (corresponding to an execution time of 3.8 s). Again, the speed-up reaches a plateau as not enough work is available for all worker threads. However, the reached speed-up is higher than the original version as more fine-grained parallelism is available in each unit of work.

This result demonstrates another benefit of transactional tasks. In the original version, the amount of parallelism corresponded to the number of transactions, which is equal to number of work items. Hence, if at a certain point there are fewer work items than cores in the machine, not all potential parallelism is exploited. By introducing parallelism inside the transactions, we make better use of the available hardware. Even if there is limited work and therefore a limited number of transactions, transactional tasks allow us to make use of more fine-grained parallelism within the transactions.

9.4 | Vacation2 (Transactional Actors)

Our Vacation2 application is inspired by the Vacation benchmark from the STAMP benchmark suite. We add the suffix "2" to clearly indicate that, in contrast to the other benchmarks, we omitted some functionality: in the original STAMP benchmark, customers can be deleted and items can be changed, while Vacation2 does not support this.

The benchmark's input consists of c customers that want to book a holiday. At the start of the program, r flights, r hotel rooms, and r cars are generated – we call these *items* – with a random price and random number of seats/beds (between 100 and 500). Each customer will reserve between one and five seats on two flights, a hotel room, and a car. For each of these four items, the customer will select a subset of q random items, pick the cheapest with sufficient available seats, and book it. Additionally, for each customer a password is generated using a cryptographically secure hash. We ran the experiments with $c = 1000$, $r = 50$, and $q = 10$.[7]

Each item and each customer is stored in a transactional variable. Customers are written to five times: four times to update their bill (for the four items), and once to store their password. Each reservation also writes to four items: two flights, a room, and a car.

In the original benchmark, there are p worker actors. The c customers are evenly distributed so that each worker actor processes c/p customers. Each customer reserves four items and generates a password. This is encapsulated in a transaction, hence, there are c transactions each writing to one distinct customer and four items. This was illustrated in Listing 5.4 on page 98.[8] There will never be a conflict on the customer, as it is distinct for each reservation, but there can be conflicts on the items.

[7] $r = c \times 5/100$ ensures that there are at least as many available seats as requested. $q = 10$ is as in the original STAMP benchmark.

[8] Note that the code in Listing 5.4 has been modified for this text. The full code of the benchmark can be found at https://github.com/jswalens/vacation2.

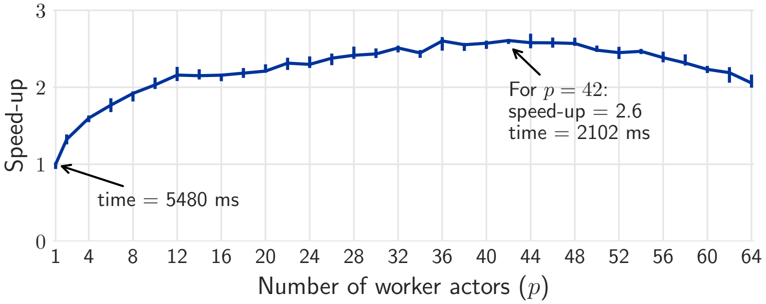


Figure 9.9: Speed-up of original Vacation2 for an increasing number of worker actors (p) on a 64-core machine. Each result is the median of 15 measurements; the error bars depict interquartile ranges.

We transform the original benchmark to parallelize this transaction using transactional actors: the four items will be reserved in separate actors. In the transformed version, next to p primary worker actors, there are s secondary worker actors. As before, the c customers are distributed evenly over the p primary worker actors. However, now each customer's reservation sends four messages to randomly selected secondary worker actors, and then generates the customer's password (as in Listing 5.5 on page 99). The secondary worker actors will look for and reserve an item of the requested type, in a transaction. Hence, in this application, there are c transactions that write to one distinct customer each and send four messages, and $4c$ transactions that write to one customer and one item. In this version, there can be conflicts on the customers as well as the items.

Performance We benchmark both versions, varying p and s, and measure the total execution time. Each variation is repeated 15 times, of which we calculate the median and interquartile ranges. Next, we calculate the speed-up compared to the result for $p = 1$ and $s = 1$, for each experiment separately.

The results of the original version are shown in Figure 9.9. Using a single worker actor, the program runs in 5480 ms. As the number of worker actors increases, the execution time decreases, reaching a minimum of 2102 ms for 42 worker actors. This corresponds to a speed-up of 2.6. On a machine with 64 cores, this speed-up is very limited. This is a result of STM's optimistic concurrency: as the number of transactions that execute in parallel increases, the chance of conflicts and thus the number of retries increases.

Figure 9.10 shows a subset of the results for the version that uses transactional actors. (The full results are shown in Table 9.11.) In this benchmark, both the number of primary and secondary worker actors are varied. The version using one primary and

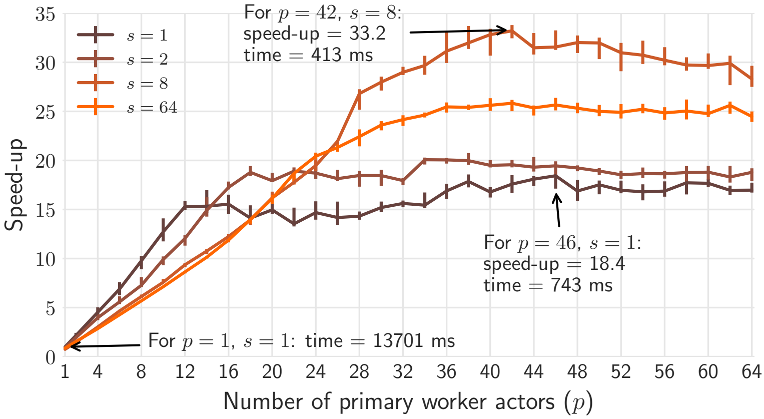


Figure 9.10: Speed-up of version of Vacation2 that uses transactional actors, for increasing numbers of primary (p) and secondary (s) worker actors, again on a 64-core machine. Full results are in listed in Table 9.11.

one secondary worker actor is slower than the original version, at 13 701 ms. However, better performance is achieved when increasing the number of actors.

The black line in Figure 9.10 shows the results using only one secondary worker actor, so customers are processed in parallel but the reservation of individual items is not. Here, a minimal execution time of 743 ms is reached for 46 primary worker actors, a better result than the original version. This is because there are far fewer conflicts: there is only one secondary actor reserving items, so there can never be any conflicts on the items.

By increasing the number of secondary worker actors (the other lines in Figure 9.10), a higher speed-up can be achieved. We see that a maximum speed-up of 33.2 is reached for 42 primary and 8 secondary worker actors, on this machine. At this point, the balance between increased parallelism and a low chance of conflicts is optimal. Using more than 8 secondary worker actors will again lower the performance, due to a higher chance of conflicts. The optimal result for this version corresponds to an execution time of 413 ms, compared to a minimum of 2102 ms for the original version. This indicates that this application benefits from being parallelized in two places, instead of only parallelizing the processing of customers (as in the original version) or only parallelizing the reservation of items (the results of $p = 1$ in Table 9.11), the optimum is found by combining both.

This experiment capitalizes on another benefit of transactional actors as well: they allow a transaction to be split up into multiple transactions with dependencies. Every transaction in the original version was split into one primary and four dependent transactions. If the primary transaction fails, the four dependent transactions fail too.

	Number of primary worker actors (p)																															
Number of secondary worker actors (s)	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62	64
1	1.0	2.2	4.5	6.9	9.8	12.7	15.3	15.3	15.5	14.1	15.0	13.5	14.7	14.2	14.3	15.2	15.6	15.4	16.9	17.9	16.8	17.6	18.1	18.4	16.9	17.5	17.0	16.8	16.9	17.7	17.7	17.0	17.0
2	0.9	1.9	4.0	5.6	7.3	9.9	12.0	14.9	17.3	18.8	17.9	18.9	18.7	18.1	18.5	18.5	17.9	20.1	20.1	20.0	19.5	19.6	19.3	19.5	19.2	18.9	18.5	18.7	18.6	18.8	18.8	18.3	18.8
4	0.8	1.6	3.4	5.0	6.9	8.4	10.2	12.0	14.0	15.8	18.3	20.1	21.3	22.7	23.7	23.9	28.1	28.9	27.7	26.7	27.4	27.7	27.4	26.7	26.6	26.5	25.9	25.4	26.0	26.0	25.1	25.2	24.6
6	0.8	1.5	3.2	4.8	6.6	7.9	9.3	11.4	13.0	14.4	16.1	18.1	20.1	22.3	24.2	29.1	30.3	31.1	31.7	31.7	31.4	32.0	30.7	30.4	30.4	30.6	29.1	29.6	28.8	28.8	28.2	28.7	28.0
8	0.7	1.5	3.0	4.7	6.1	7.6	9.4	10.8	12.2	14.0	16.2	17.8	19.5	22.0	26.8	28.0	29.0	29.7	31.1	32.0	32.8	33.2	31.5	31.6	32.0	32.0	31.0	30.7	30.2	29.7	29.7	29.9	28.3
10	0.8	1.5	3.0	4.6	6.0	7.6	9.3	10.5	12.1	13.5	15.3	17.3	18.8	23.0	25.1	27.9	29.1	31.3	31.5	32.3	32.0	31.8	32.2	31.5	31.2	31.5	31.3	30.0	30.2	30.6	30.8	29.5	29.4
12	0.8	1.4	2.9	4.5	6.1	7.5	8.8	10.3	11.8	13.7	15.4	16.6	19.2	21.1	25.5	27.4	29.2	30.0	30.8	31.4	31.6	31.8	32.0	29.9	29.8	30.6	30.5	30.1	29.2	30.2	28.6	28.8	29.5
14	0.8	1.4	2.9	4.4	5.9	7.4	8.8	10.4	11.7	13.1	14.7	16.8	18.6	22.0	23.7	26.9	28.9	31.1	31.2	31.0	30.8	30.8	30.5	30.4	30.2	30.9	30.1	28.8	29.1	29.5	28.5	28.7	28.5
16	0.8	1.4	2.9	4.4	5.8	7.2	8.9	10.2	11.8	13.2	15.2	16.9	19.4	21.0	25.2	28.2	29.4	29.9	30.2	30.7	30.1	30.5	30.1	30.2	30.2	29.8	29.7	28.8	28.1	28.3	26.8	28.1	26.8
18	0.9	1.4	2.9	4.3	5.8	7.3	8.7	10.3	11.7	13.5	14.9	17.0	18.7	21.0	24.4	26.9	29.4	29.5	30.2	30.5	30.3	30.0	30.6	30.8	29.4	29.1	28.5	28.8	28.7	28.0	27.7	27.0	27.8
20	0.9	1.4	2.8	4.3	5.7	7.2	8.6	10.3	11.9	13.3	15.2	16.8	18.9	21.0	25.7	27.3	28.6	29.2	29.4	30.0	29.9	29.7	29.1	29.2	29.6	29.3	28.3	27.9	27.5	28.1	26.7	27.0	26.8
22	0.9	1.5	2.9	4.3	5.7	7.1	8.4	10.4	11.8	12.9	15.0	17.1	19.4	21.6	25.2	27.1	28.3	28.9	28.9	28.6	28.9	28.9	29.8	29.4	28.5	28.5	28.2	28.0	28.2	27.0	27.1	26.8	27.3
24	0.9	1.5	2.9	4.3	5.7	7.0	8.9	10.2	11.8	13.5	15.0	17.5	20.4	23.3	25.1	26.7	27.8	28.0	28.6	28.7	28.7	29.3	29.2	27.9	27.6	28.0	28.2	28.7	27.0	27.6	26.6	25.9	26.1
26	0.9	1.5	2.8	4.2	5.7	7.3	8.8	10.3	11.8	13.4	15.2	17.3	19.8	22.8	24.5	26.5	28.1	28.5	28.5	28.4	28.3	29.0	28.5	27.8	28.1	28.3	27.8	27.9	26.9	27.4	26.8	26.3	27.3
28	0.9	1.5	2.8	4.3	5.8	7.3	8.7	10.3	11.8	13.3	15.3	16.9	20.0	21.9	25.1	26.4	27.5	28.4	28.2	28.1	28.1	28.2	27.9	27.5	26.7	28.2	27.9	27.1	27.4	26.1	26.5	26.2	25.6
30	0.9	1.5	2.8	4.3	5.9	7.3	8.7	10.2	11.9	13.3	15.3	17.5	20.0	22.5	25.1	26.7	27.0	27.6	28.1	28.3	27.9	28.1	28.0	28.3	28.1	28.5	27.4	27.1	28.3	27.2	26.2	26.5	25.6
32	0.9	1.5	2.9	4.3	5.8	7.3	8.8	10.3	11.7	13.6	15.4	17.9	20.0	23.0	25.7	26.5	26.7	27.3	28.2	27.5	28.1	27.3	27.4	28.1	27.8	27.4	28.0	27.0	26.6	26.0	26.9	26.0	25.8
34	0.9	1.5	2.9	4.3	5.8	7.2	8.8	10.3	11.7	13.5	15.5	17.8	19.9	23.4	24.9	26.1	27.2	27.1	27.2	28.2	27.6	27.8	27.9	27.1	27.3	28.0	26.5	27.5	26.1	26.6	26.7	26.0	25.3
36	0.9	1.5	2.9	4.3	5.8	7.3	8.7	10.3	11.8	14.0	16.1	17.8	21.5	24.0	25.1	26.0	26.5	26.6	26.9	27.2	27.0	27.2	26.8	26.6	26.8	26.8	26.8	25.9	26.6	26.3	25.5	25.2	25.4
38	0.9	1.5	2.9	4.3	5.8	7.2	8.7	10.3	11.9	13.5	15.7	18.0	21.9	23.6	25.3	25.5	26.2	26.5	26.7	27.0	27.7	27.0	27.3	26.9	26.7	26.4	26.9	26.6	26.2	26.1	26.2	26.2	25.8
40	0.9	1.5	2.9	4.3	5.7	7.2	8.7	10.1	12.0	13.9	15.8	18.6	22.0	23.3	24.4	25.5	25.9	27.1	26.5	27.0	26.8	26.8	26.9	26.6	26.5	27.1	26.1	26.3	25.5	26.9	25.4	25.5	25.7
42	0.9	1.5	2.9	4.3	5.7	7.2	8.7	10.1	11.9	13.7	16.4	18.5	20.4	23.3	24.7	24.7	26.1	26.4	26.8	27.1	27.2	26.8	26.8	26.6	26.3	26.5	26.0	26.1	25.8	25.7	25.8	25.1	25.5
44	0.9	1.5	2.8	4.3	5.7	7.2	8.7	10.2	11.9	13.6	16.3	18.1	20.4	22.0	22.8	23.6	24.3	26.0	26.5	26.8	26.2	26.6	27.1	27.3	27.1	26.6	25.7	26.4	25.9	24.9	25.2	25.6	26.1
46	0.9	1.5	2.9	4.3	5.8	7.2	8.7	10.1	11.8	13.3	15.7	17.7	19.6	21.9	22.9	23.4	23.5	24.6	25.8	25.8	27.0	26.5	26.3	26.6	26.3	26.4	25.9	25.3	26.0	26.0	25.5	25.6	25.5
48	0.8	1.6	2.9	4.3	5.7	7.2	8.6	10.1	11.5	13.6	15.6	18.0	20.8	21.6	23.0	23.3	24.2	24.6	24.7	25.8	25.9	25.9	26.1	26.1	25.8	26.4	26.2	26.5	26.0	26.0	25.3	25.8	25.3
50	0.9	1.6	2.9	4.3	5.8	7.2	8.6	10.2	11.6	13.7	15.7	18.3	19.8	21.8	22.7	23.5	23.9	24.5	24.8	25.4	26.7	26.0	26.1	26.1	26.4	26.1	26.1	26.0	25.7	25.4	25.8	24.9	24.9
52	0.9	1.5	2.9	4.3	5.7	7.1	8.6	10.1	11.7	13.6	15.9	18.1	20.2	21.9	22.6	23.3	24.0	24.1	25.7	25.7	25.7	25.8	25.7	26.1	26.1	25.6	26.0	25.6	25.9	25.8	25.2	24.9	25.3
54	0.9	1.6	2.9	4.3	5.7	7.1	8.6	10.1	11.7	13.9	15.8	18.1	20.4	21.3	22.5	22.9	24.1	25.2	25.9	25.7	25.6	25.6	26.1	26.3	25.9	26.0	25.5	25.9	25.5	25.3	25.8	25.1	24.9
56	0.9	1.5	2.9	4.3	5.7	7.1	8.6	10.2	11.8	13.7	16.3	18.2	20.6	21.7	22.4	23.0	23.3	25.0	25.4	25.7	25.9	25.6	26.1	25.5	25.8	25.8	26.3	25.9	25.4	25.3	25.3	25.0	24.2
58	0.9	1.6	2.8	4.3	5.7	7.3	8.7	10.1	12.0	13.6	15.9	17.9	19.7	21.6	22.5	22.9	23.0	24.1	25.1	25.8	25.5	25.5	26.1	25.7	25.2	25.9	25.4	25.3	25.7	25.2	25.4	25.3	24.9
60	0.9	1.5	2.9	4.3	5.7	7.1	8.7	10.3	12.0	13.6	16.1	18.4	19.9	21.3	22.2	23.2	23.9	24.7	25.1	25.8	25.7	25.8	25.5	25.6	26.4	25.4	25.2	25.5	25.3	25.1	24.8	24.9	25.1
62	0.9	1.6	2.9	4.3	5.7	7.2	8.6	10.1	11.8	13.8	16.4	18.2	19.9	21.2	22.2	23.8	24.0	24.9	25.5	25.3	25.4	25.5	25.8	25.9	25.9	26.1	25.1	25.1	25.8	25.3	24.8	24.8	24.7
64	0.9	1.5	2.9	4.3	5.7	7.1	8.6	10.1	11.9	14.0	16.1	18.7	20.4	21.3	22.4	23.6	24.2	24.6	25.5	25.4	25.6	25.8	25.4	25.7	25.3	25.0	24.9	25.2	24.8	25.0	24.8	25.6	24.5

Table 9.11: Full results for speed-up of the version of Vacation2 that uses transactional actors for increasing numbers of primary (p) and secondary (s) worker actors. Higher (better) speed-ups are colored more green.

However, if a dependent transaction fails, this does not abort any other transaction. Using transactional actors, transactions can be split up, lowering the cost of a conflict in a dependent transaction, as only this part needs to retry.

Finally, we note the high overhead of transactional actors: while the original version took 5480 ms when using a single worker actor, the version with transactional actors takes 13 701 ms when $p = 1$ and $s = 1$, hence it is more than twice as slow when using one core. We suspect this is due to our relatively simplistic implementation of actors and we believe further optimizations could improve the performance (some were suggested in Section 8.4 of the previous chapter).

In conclusion, this experiment shows that transactional actors can increase the performance of an application that uses transactions by distributing the transactions over multiple actors, introducing more fine-grained parallelism and lowering the chance and cost of conflicts. (Even with our relatively simple implementation of actors; further optimizations could improve these results.)

9.5 Developer Effort

In this section, we assess the effort required from developers to use Chocola. We focus on the qualitative aspect: we are interested in which changes were necessary to

Benchmark	Added	Removed	Lines of code in original version
Labyrinth	78 (11%)	30 (4%)	682
Bayes	1 (<1%)	1 (<1%)	1248
Vacation2	25 (8%)	17 (5%)	320

Table 9.12: The number of lines of code added and removed to introduce transactional futures or actors in each benchmark.

introduce transactional futures and transactional actors in a program with transactions, and how they compare to the changes necessary to introduce regular futures and actors in a program without transactions. We use the same three applications of the previous three sections and describe the transformations we applied. Table 9.12 summarizes the quantitative aspect of this evaluation: the number of lines of code that were changed for each application.

Furthermore, we note that the Labyrinth and Bayes programs from the STAMP benchmark suite end with a verification phase that checks whether the generated output is correct. All our implementations of these benchmarks also include this verification, to ensure no errors were introduced either when porting from C to Clojure or when introducing futures or actors. (The Vacation2 benchmark is different from STAMP's original Vacation benchmark, so in that case there is no reference implementation to compare against.)

Labyrinth

The original Labyrinth application uses a sequential search algorithm, in a transaction, and consists of 682 lines of code in total. After introducing transactional futures, 30 lines (4%) were removed and 78 lines (11%) were added. (This was illustrated in Listing 4.5 of Chapter 4, on page 73.) Almost all of these changes are a result of swapping the sequential search algorithm with a parallel search algorithm, which is more complex and requires structural changes. Besides that, the developer needs to pick and define a suitable conflict resolution function when initializing the transactional variables that represent the grid.

Thus, we observe that, while several changes are necessary to introduce transactional futures in this application, this effort is similar to the effort required to parallelize any sequential code. These changes would be necessary even outside a transaction, and are not due to specific requirements of our techniques. There is one notable exception: the definition of the conflict resolution function.

Bayes

To transform the original version of the Bayes application into the one that uses transactional futures, actually only one line (out of 1248) had to be changed: the keyword `for` was replaced by `parallel-for`, a macro that uses transactional tasks internally to execute its iterations in parallel. This was shown in Listing 9.7 on page 182. This is possible as each iteration of the loop is independent, exhibiting a type of parallelism sometimes described as 'embarrassingly parallel' [Herlihy and Shavit 2011].

This benchmark thus epitomizes the small developer effort required to introduce transactional futures. While in a naive combination of transactions and futures, read operations on transactional state inside `parallel-for` would not be possible or give inconsistent result, here transactional futures deliver the expected result.

Vacation2

A snippet of the changes between the two versions of this benchmark is shown in Listing 9.13a. (These do not correspond exactly to the actual code, as some comments and logging were elided to include them in this document.) To introduce transactional actors in the main transaction of the original version, the code needs to be changed three places:

- During the initialization of the program, the original version spawns only primary worker actors, while the version with transactional actors spawns secondary worker actors as well (3 lines added).
- The behavior `reserve-behavior` of the secondary worker actors has to be defined, containing the part of the original transaction that reserves a single item (14 lines added, corresponding to lines 10–15 in Listing 9.13b and some additional bookkeeping[9] and comments).
- The transaction in the primary worker actor, in `customer-behavior`, has to be transformed to send messages to the secondary worker actors instead of reserving items directly (17 lines removed and 8 lines added instead, corresponding to the changes in `customer-behavior` in Listings 9.13a and Listing 9.13b and some extra logging).

In total, out of 320 lines of code in the original version, 25 lines (8%) were added and 17 lines (5%) were removed.

To exploit transactional actors in this application, some structural changes were required: a new type of actor was introduced and part of the behavior of the original actors moved to there. However, we observe that transactional actors do not require

[9] Workers communicate with a 'master' actor that tracks which customers has been processed and ends the program when they all have.

```
(def customer-behavior
  (behavior [id] [c]
    (atomic
      (reserve-flight (:orig @c) (:dest @c) (:start @c) (:n @c))
      (reserve-flight (:dest @c) (:orig @c) (:end @c) (:n @c))
      (reserve-room   (:dest @c) (:n @c) (:start @c) (:end @c))
      (reserve-car    (:dest @c) (:n @c) (:start @c) (:end @c))
      (ref-set c (assoc @c :password (generate-password)))))
```

(a) The behavior for primary worker actors in the original Vacation2 benchmark.

```
(def customer-behavior
  (behavior [id] [c]
    (atomic
      (send (rand-nth secondary-workers) :flight (:orig @c) …)
      (send (rand-nth secondary-workers) :flight (:dest @c) …)
      (send (rand-nth secondary-workers) :room   (:dest @c) …)
      (send (rand-nth secondary-workers) :car    (:dest @c) …)
      (ref-set c (assoc @c :password (generate-password))))))

(def reserve-behavior
  (behavior [id] [type & args]
    (case type
      :flight (atomic (apply reserve-flight args))
      :room   (atomic (apply reserve-room   args))
      :car    (atomic (apply reserve-car    args)))))
```

(b) The behavior for primary and secondary worker actors in the adapted benchmark.

Listing 9.13: Code snippets from the original and adapted Vacation2 benchmark. These have been shortened to make them easier to understand, and some bookkeeping, comments, and logging were elided.

the developer to use new constructs, and therefore the same techniques that are used to introduce actors in a sequential program without transactions can also be applied to introduce them in a program with transactions. Hence, using transactional actors developers can reuse their existing knowledge of the models even when they are combined, because the guarantees of the separate models are maintained.

9.6 Conclusions

Based on these experiments, we draw the following conclusions:

- Out of the eight applications in the STAMP benchmark suite, which represent a variety of use cases for transactions, we examined the four applications with the longest average transactions. In three out of four – Labyrinth, Bayes, and Vacation – we found that we could use transactional futures or transactional actors to increase performance, by parallelizing a loop in the longest-running transaction. (For the fourth application, Yada, either no further parallelization is possible or more domain expertise is required to do so.)
- The Labyrinth application spends almost all of its time in transactions that execute a search algorithm, which can be parallelized using transactional futures. This leads to a speed-up thanks to faster (internally parallel) transactions and fewer conflicts.
- The Bayes application spends most of its time in a loop in a transaction that can be trivially parallelized. As there is only limited work available, at a certain point the number of transactions is lower than the number of cores in the machine. Transactional futures allow us to introduce more fine-grained parallelism. This is a matter of changing `for` into `parallel-for`, and increases the maximal speed-up on an eight-core machine from 2.75 for the original version to 3.45 for the version with transactional tasks.
- The Vacation2 application implements an event-based vacation reservation system, to which actors can be applied naturally. We split the transaction of the original application into one smaller primary transaction and four dependent transactions that are distributed over different actors. This improves performance by introducing more fine-grained parallelism and lowering the chance and cost of conflicts.
- As was shown in Table 9.12, in the Labyrinth and Vacation2 applications, about 10% of the code was changed, as the structure of a part of these program needed to be transformed to parallelize the internals of their transactions. For the Bayes application, only a single line was changed, as this application was trivial to parallelize. In each case, the required effort is mostly due to the introduction of additional parallelism, which would be necessary even outside a transaction, and not

due to specific requirements of our techniques. The only exception to this is the definition of the conflict resolution function in the Labyrinth application.

These results demonstrate that Chocola allows developers to improve the performance of their transactional applications with only limited effort. Because Chocola does not introduce any new constructs, developers can now reuse their existing knowledge of the separate models even when they are combined. Moreover, as our implementation is a relatively simple prototype in Clojure, further optimizations could decrease any overheads and improve its performance.

It should be possible to apply transactional futures and actors to other STM systems, such as Haskell or ScalaSTM. In those systems, the studied applications can benefit from parallelism in the transaction as well and the development effort to introduce them should be similar, but depending on the implementation the speed-up may be different.

10 Conclusion

10.1 Summary

Since the introduction of multicore processors, concurrency has been a crucial but difficult aspect of software development. Researchers have created a plethora of concurrency models for programming languages, aimed at different program designs and providing different guarantees. We observe that existing programs and programming languages often combine these concurrency models. We studied three concurrency models and showed that naive combinations can annihilate the guarantees of their constituent models (Chapter 3). Thus, the assumptions of developers are broken and the errors that were prevented by using each separate concurrency model can resurface.

In this dissertation, we studied three concurrency models from three categories in particular:

- **futures**: a deterministic model, therefore guaranteeing determinacy,
- **transactions**: a shared-memory model that guarantees isolation and progress, and
- **actors**: a message-passing model that guarantees the isolated turn principle and deadlock freedom.

We unified these three models into Chocola: a programming language framework that specifies a semantics for their combinations with the aim of introducing additional parallelism while preserving each model's guarantees whenever possible. At the same time, the semantics of each model remains unchanged when used separately.

We started from the pairwise combinations of the three concurrency models:

- First, **transactional futures** are futures created in a transaction with access to the encompassing transactional context (Chapter 4). Transactional futures make it possible to exploit parallelism inside transactions. They ensure isolation and progress of transactions, and replace the determinacy of futures with intratransaction determinacy.

- Second, **transactional actors** combine transactions and actors, making it possible to create transactions in actors, and vice versa, to send messages to actors in transactions (Chapter 5). Our semantics maintains the isolation and progress guarantees of transactions, while guaranteeing low-level race freedom and deadlock freedom for the actors.
- Third, the combination of **futures and actors** only required a few small changes in order to provide a familiar semantics (Section 6.1). We ensure that the isolated turn principle and deadlock freedom of actors are maintained, and that determinacy is maintained within each turn.

Next, we unified the three concurrency models into one programming language framework, called Chocola (Chapter 6).

- We formalized Chocola's operational semantics in PureChocola, which unifies transactional futures and transactional actors (Chapter 7). We described its guarantees, which remain as close as possible to the guarantees of its constituent models. Additionally, we created an executable implementation of parts of PureChocola using PLT Redex.
- We have implemented Chocola by extending Clojure (Chapter 8). Our implementation starts from standard implementations of its three constituent models, which are modified where the models interact. It demonstrates that an efficient implementation is possible, which we used to evaluate the performance benefits of Chocola.
- Using three benchmarks from the commonly used STAMP benchmark suite, we evaluated Chocola (Chapter 9). These applications all use transactions and were extended using transactional futures and actors. By introducing futures or actors additional parallelism can be exploited, leading to better performance without significantly altering the structure of the program. We saw that this requires only a small effort from the developer: the changes required to introduce new concurrency models are similar to the effort required to parallelize any program, as there were only limited changes required due to the interactions between concurrency models.

Hence, Chocola is a unified framework of futures, transactions, and actors, in which each model can not only be used separately but also in combination with the others, while maintaining the guarantees of each model wherever possible. Thus, using Chocola developers can optimally exploit parallelism for little effort, i.e. without drastically changing the architecture of their application.

This dissertation also aims to open the discussion on combining concurrency models. It is the first to study combinations of three models, doing so using three specific models. We hope it will be a stepping stone to future research on combinations of other concurrency models.

10.2 Contributions

Our scientific contributions are the following:

- To the best of our knowledge, this dissertation is the first to **comprehensively and systematically study the combination of three concurrency models**: futures, transactions, and actors. For each combination, we determined which guarantees are broken in a naive, ad-hoc combination.
- This dissertation introduces **transactional futures**: futures created in a transaction with access to the encompassing transactional context. They guarantee intratransaction determinacy and the isolation and progress guarantees of transactions.
- This dissertation introduces **transactional actors**. These make it possible both to create transactions in actors, and vice versa to send messages to actors in transactions. Our semantics maintains the isolation and progress guarantees of transactions, while guaranteeing low-level race freedom and deadlock freedom for the actors.
- We combine all three models into **one unified linguistic framework: Chocola**, consisting of:
  - a **specification of the operational semantics** of Chocola, PureChocola, which we use to demonstrate its guarantees,
  - an **implementation** of Chocola on top of Clojure, and
  - an **evaluation** using three benchmark applications.

The implementation of Chocola, an executable implementation of the semantics of PureChocola, and the three benchmark applications are available at http://soft.vub.ac.be/~jswalens/chocola/.

10.3 Future Work

In this section, we list possible avenues for future research.

Formal proofs of the guarantees of PureChocola In future work, we would like to formally prove the guarantees provided by the operational semantics of PureChocola (described informally in Section 7.3). Ideally, the operational semantics would be implemented using a formal proof system, such as Coq, so that the guarantees can be verified mechanically. Alternatively, we could extend our current implementation of the operational semantics in PLT Redex to support all of Chocola's constructs and verify the guarantees using PLT Redex's randomized testing.

Exploration of different concurrency models Chocola picks one concurrency model from each category in the taxonomy of Van Roy and Haridi [2004]. It would be interesting to explore the possibility of choosing different models within each category. We can think of several existing concurrency models that would pose interesting challenges. For instance:

- Using *Concurrent Revisions* [Burckhardt et al. 2010] instead of transactions as the shared-memory model. Transactional futures currently have a different approach for conflict resolution within a transaction (using conflict resolution functions, like Concurrent Revisions) and between transactions (using abort and retry). By replacing transactions with Concurrent Revisions, the same approach to conflict resolution would apply throughout the program, which may provide a simpler model to the programmer. Concurrent Revisions are deterministic, which at the same time provides a stronger guarantee to the programmer but may limit the programs that can be expressed. It would therefore be interesting to study the effects of their use instead of transactions in Chocola. Moreover, unlike transactions, Concurrent Revisions avoid reexecuting parts of the program, which is the main cause for incompatibilities between the different concurrency models in this dissertation. It may thus be easier to combine other concurrency models with Concurrent Revisions than with transactional memory.
- *Communicating Sequential Processes* (CSP) [Hoare 1978] provide concurrent processes that pass messages over channels. While in the actor model there is a one-to-one mapping between inboxes that receive messages and the actors that process these messages, in CSP there is no such 'restriction': any process can read from any channel it has access to. This gives the programmer more flexibility, but may make the program harder to reason about, as it is no longer possible to reason in 'turns' like in the actor model. Moreover, CSP is a synchronous model, which may pose additional challenges. It would be interesting to study the effects of replacing actors with CSP in Chocola.

This dissertation hence examines only one point in the landscape of combinations of concurrency models. It is a stepping stone that opens the discussion on combinations of concurrency models; future work can further uncover the landscape by examining other combinations.

Decomposition of concurrency models into elementary 'building blocks' The approach taken in this dissertation, in which every pairwise combination of concurrency models is examined one by one, does not scale well to combinations of an increasing number of models. To make studying combinations of a larger number of concurrency models feasible, a possible approach is to decompose each concurrency model into a set of elementary 'building blocks'. Studying combinations of concurrency models then corresponds to studying the combinations of their building blocks.

Additionally, we noticed some properties of certain concurrency models make combinations especially problematic, for instance the presence of constructs that retry or block (as concluded in the case study of Clojure in Section 3.2) or nondeterminism (as we saw when using transactions and actors in futures). On the other hand, other properties facilitate combinations, such as determinism or the absence of side effects. Further exploration of different concurrency models can lead to a list of such problematic and helpful properties. This can then lead to a table of properties that do or do not combine well, for instance, using a model with side effects in a model with retrying operations is a problematic combination.

'Multidimensional' nesting of concurrency models This dissertation examines combinations in which concurrency models are nested pair by pair. One may wonder how these results generalize to situations in which more than two models are nested, e.g. when a future is created in a transaction that runs in an actor. In other words, does the table in Figure 6.7 suffice, or do we need to examine a cube in which all three models are nested in $3 \times 3 \times 3$ possible combinations? Moreover, by nesting the same model multiple times, in theory further complications could arise: is creating a future in a transaction in a future in a transaction (four levels deep), the same as creating a future in a transaction (two levels deep)?

In the combinations of the three models studied in this dissertation, we found that it was sufficient to study nested combinations of only two levels. We then generalized these results to a language with three models, for which the formalization in Chapter 7 specifies a well-defined semantics that works for any level of nesting. In contrast, combinations of other concurrency models may not necessarily generalize in the same way. It would be interesting to explore which properties of concurrency models make 'deeply nested' combinations problematic or not.

Applicability of transactional futures and transactional actors We evaluated Chocola by extending three applications that currently use transactions with futures or actors. In future work, we would like to study their applicability more comprehensively. Using a larger set of programs, we aim to find out where futures, transactions, and actors can be applied, and in which parts of the program they are combined and interact.

Moreover, by applying Chocola to a wider range of programs, we might be able to determine certain patterns that often appear in the code. These could consequently be encoded into library functions or even language constructs. The implementation could also be optimized for these typical use cases.

Comparison of implementation techniques and optimizations The current implementation of Chocola serves only as a starting point. While we aimed for our implementation to be sufficiently mature to use it in our performance evaluation, we still observe many opportunities for optimizations. In our implementation of transactional futures, we presented a technique that makes a trade-off between speed and memory usage (Section 8.5, page 169). Based on real-world usage, we could further refine this trade-off. Our implementation of transactional actors uses a rudimentary implementation of actors. Based on the vast amount of existing actor platforms, many existing optimizations could be applied to our implementation. To assess which optimizations are opportune in real-world applications, we would need to gauge them against a suite of benchmark applications that use Chocola.

Applicability to other programming paradigms Chocola is built around a 'base' language that is purely functional. In future work, we can examine how a different base language affects the guarantees provided by Chocola. For instance, using a language in which side effects (such as input/output) can appear anywhere in the code can break the isolation of transactions. Or, reactive languages inherently have concurrency built in and using them as a base will pose unique problems.

One interesting shortcoming of our base language is the lack of exceptions. The interactions between concurrency models and exceptions have been the topic of previous research (e.g. for futures [Navabi and Jagannathan 2009] and transactions [Harris et al. 2005, 2010]) and can pose challenges due to their non-local control flow, in some cases allowing an exception to 'escape' a future or transaction.

Distribution and fault tolerance Chocola was developed in the context of a single machine executing a parallel program on a multicore processor. An open question is whether the techniques and ideas presented in this dissertation could also be applied in a distributed context, when multiple machines cooperate to execute a program. The actor model is frequently used in distributed contexts, and some related work discussed in Section 5.5 of Chapter 5 focuses on the use of shared memory or transactions in such systems. A possible idea is to explore whether transactional actors can also be applied in a distributed setting.

Two issues are immediately apparent. First, reconciling the use of a shared-memory model such as Software Transactional Memory with a distributed architecture brings challenges in keeping the shared state consistent and available on all machines. Second, distributed architectures are inherently prone to network faults, making fault tolerance indispensable. An open question is how to ensure fault tolerance in a distributed version of Chocola, for example in a scenario where a tentative turn or transaction with a dependency runs on one machine but depends on a transaction on another machine.

A

Notation

We use the following notations for sets, sequences, and mappings.

Sets

$\varnothing$	Empty set
$\mathrm{A} \mathbin{\dot\cup} \mathrm{B} = \mathrm{C} \iff \mathrm{A} = \mathrm{C}\setminus\mathrm{B}$	Disjoint union
$\mathrm{A} \cup a \iff \mathrm{A} \cup \{a\}$	Short-hand for union on singletons

Sequences

$[]$	Empty sequence
$\overline{a}$	(Possibly empty) sequence of a
$f \cdot \overline{r} = \overline{s}$	(De)construction of sequence $\overline{s}$ from/into its first element f and the rest $\overline{r}$

Partial Functions (Mappings)

$\alpha : \mathrm{A} \rightharpoonup \mathrm{B}$	Declaration of partial function (mapping)
$\varnothing$	Empty mapping
$\alpha[a \mapsto b](x) = \begin{cases} b & \text{if } x = a \\ \alpha(x) & \text{otherwise} \end{cases}$	Extension of mapping
$(\alpha :: \beta)(x) = \begin{cases} \beta(x) & \text{if } x \in \mathrm{dom}(\beta) \\ \alpha(x) & \text{otherwise} \end{cases}$	Concatenation of two mappings

Meta-Syntax

Some syntax is pervasive throughout this dissertation:

$x^? ::= x \mid \bullet$ Optional element

We use the notation $x^?$ to indicate an optional element, i.e. an element that can have the ‘empty’ value $\bullet$.

B

A Clojure Primer

In this appendix, we briefly describe Clojure. Clojure is a general-purpose, dynamically typed programming language. It is founded on four principles:[1]

To be a 'modern' Lisp. It has a syntax built on S-expressions, but with convenient shortcuts for common data types (see next section). It also supports macros.

Based on a functional core. Its built-in data types are immutable (see next section) and functions are first-class. However, in contrast to many existing functional programming languages, it is dynamically typed. It also does not enforce purely functional programming everywhere.

Compatibility with an existing platform. Clojure compiles to Java Virtual Machine bytecode. It interoperates with code written in Java: code written in Java can be called from Clojure and vice versa. Hence, the existing Java ecosystem of libraries can be reused.

Designed for concurrency. Clojure supports many different concurrency models, built around a philosophy of immutable values that are encapsulated in 'containers' whose value can change over time.

Below, we describe some common constructs from Clojure.

Syntax and data structures

Table B.1 lists Clojure's syntax for built-in data types. Clojure's collection types (strings, lists, vectors, maps, and sets) are *immutable*: a modification does not mutate the original value, but instead *returns* the new value. For example, `(pop '(1 2 3))` returns

[1] https://clojure.org/about/rationale

Integer	`1`	Vector	`[1 2 ”a”]`
String	`”a”`	Map	`{:a 1, :b 2, :c 3}`
Keyword	`:a`	Set	`#{1 2 ”a”}`
(Linked) list	`'(1 2 ”a”)`	Comment	`; This is a comment`

Table B.1: Clojure’s syntax for common built-in data types.

the new list `'(2 3)`, leaving the original unmodified. Clojure’s collections are implemented using persistent data structures [Driscoll et al. 1989], which implement such immutable data structures efficiently, avoiding duplication.

Variable bindings

New variable bindings can be introduced using `let` or `def`. `let` defines local variables in the lexical scope. These are always immutable. `(def x v)` defines a global variable `x` with value `v`. Global values are also immutable, except in some cases that are not relevant in the context of this dissertation. The construct `(defn f [parameter1 parameter2] body)` defines a function `f`, taking a vector of parameters. This is illustrated in Listing B.2.

```
(let [a 1
      b 2]
  (+ a b))
```

(a) Evaluates to 3.

```
(def a ”hello”)
```

(b) Defines the global variable `a`.

```
(defn factorial [n]
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

(c) Defines the function `factorial`, taking one parameter `n`.

Listing B.2: `let`, `def`, and `defn` in Clojure.

Other common constructs

- `fn` defines an anonymous function, e.g. the following two lines are identical:

```
(defn inc [n] (+ n 1))
(def inc (fn [n] (+ n 1)))
```

- do encapsulates a sequence of statements. It is equivalent to putting `;` between statements in C-like languages.

```
(do
  (println ”Print one thing”)
  (println ”and another”))
```

- `for` loops are syntactic sugar for `map`:

```
(for [x [1 2 3]]
  (* x 2))
(map (fn [x] (* x 2)) [1 2 3])
; both return the vector [2 4 6]
```

- loop and recur are used to implement loops:

```
(loop [n 5]  ; on the first iteration, n is 5
  (if (= n 0)
    (print ” Done!”)
    (do
      (print n)
      (recur (- n 1)))))  ; start the next iteration, with n decremented by 1
; prints 54321  Done!
```

Only tail recursion using explicit loop and recur constructs is optimized. Tail recursive functions are not optimized, as this is not supported by the JVM.

Some syntactical differences with Scheme

For seasoned Scheme users, we list three syntactical differences between Clojure and Scheme:

- Some of Clojure constructs have a different name from Scheme: Scheme's begin is do in Clojure, define is def for variables and defn for functions (shown above).
- Clojure's let takes a single vector of variable–value bindings, instead of a nested list of pairs (as in Listing B.2a).
- Clojure's loop and recur are equivalent to a named Scheme's let. The loop example above translated to Scheme is:

```
(let loop ([n 5])
  (if (= n 0)
    (print ” Done!”)
    (begin
      (print n)
      (loop (- n 1)))))
```

More information

The following websites provide more information on Clojure:

- Clojure's website is at https://clojure.org/ and its documentation at https://clojure.org/reference/documentation.
- A reference sheet of Clojure's built-in functions can be found at http://clojuredocs.org/quickref.

C

Language and Library Support for Concurrency Models

Many programming languages have built-in support for multiple concurrency models. Moreover, when a concurrency model is not built into the language, developers often build libraries instead. These claims are supported by the table below.

	Clojure	Scala	Java	Haskell[b]	C++
Deterministic models					
Futures	✓	✓	✓	[7]	✓
Promises	✓	✓	✓	[7]	✓
Fork/Join	✓[a]	✓[a]	✓		[11]
Parallel collections	✓[a]	✓	✓	[8]	[11]
Dataflow	[1]	[2]	[4]	[9]	
Shared-memory models					
Threads	✓[a]	✓[a]	✓	✓	✓
Locks	✓[a]	✓[a]	✓	✓[c]	✓
Atomic variables	✓	✓[a]	✓	✓	✓
Transactional memory	✓	[3]	[5]	✓	[12][d]
Message-passing models					
Actors	[1]	[4]	[4]	[10]	[13]
Channels	✓	✓	[6]	✓	[14]
Agents	✓				

Legend

✓ Built into the language or its standard library
[i] Available as a library, linked to below

Libraries

We list at least one library, often several more exist:

[1] Pulsar: http://docs.paralleluniverse.co/pulsar, supports actors and dataflow
[2] Ozma: https://github.com/sjrd/ozma
[3] ScalaSTM: https://nbronson.github.io/scala-stm
[4] Akka: https://akka.io, supports actors (Akka Actors) and dataflow concurrency (Akka Streams)
[5] DeuceSTM: https://sites.google.com/site/deucestm
[6] JCSP: https://www.cs.kent.ac.uk/projects/ofa/jcsp
[7] future package for Haskell: https://hackage.haskell.org/package/future
[8] DPH: https://wiki.haskell.org/GHC/Data_Parallel_Haskell
[9] Etage: https://hackage.haskell.org/package/Etage
[10] thespian: https://hackage.haskell.org/package/thespian
hactors: https://hackage.haskell.org/package/hactors
[11] Threading Building Blocks: https://www.threadingbuildingblocks.org
[12] TinySTM: http://www.tmware.org/tinystm.html
[13] C++ Actor Framework: http://actor-framework.org
[14] Boost channels (in its Fiber library): https://www.boost.org

Notes

(a) These models are part of Java and can be used in Clojure or Scala as these languages are built on top of the Java Virtual Machine.
(b) Many of these features are not part of the Haskell language standard per se, but are provided by the standard library of GHC.
(c) Available in the form of `QSem` semaphores, see https://hackage.haskell.org/package/base-4.11.1.0/docs/Control-Concurrent-QSem.html.
(d) While transactional memory is currently available as a library for C++, it is also being considered for inclusion in the C++20 language standard (to be released in 2020). Several proposals have been made, the latest version at the time of writing can be found at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf (May 2015). There is also already experimental support for transactional memory in GCC since version 4.7, released March 2012 (more details at https://gcc.gnu.org/wiki/TransactionalMemory).

D

Semantics of Futures and Transactions in Clojure and Haskell

The syntax and operational semantics defined in Chapter 2 for futures (Section 2.3.5 on page 20) and transactions (Section 2.4.5 on page 31) are deliberately very similar to those offered by Clojure and Haskell, as we aim to demonstrate how the problems described in this dissertation also apply to these programming languages. This appendix details the (minor) syntactic and semantical differences between our formal model and the implementations of Clojure and Haskell.

D.1 | Clojure

Clojure 1.9.0 (released December 8, 2017) differs from the presented syntax in the following ways:

- Clojure encapsulates all forms in parentheses, as S-expressions. Furthermore, it has a slightly different syntax for `let`.
- `fork` and `join` are named `future` and `deref` respectively. That is, `deref` is overloaded for both futures and transactional variables.
- Clojure's (`dosync` $\overline{e}$) is equivalent to our `atomic` (`do` $\overline{e;}$).

The semantics differ only on these two points:

- Clojure allows `ref` and `deref` to be used outside a transaction: Clojure's (`ref` *e*) and (`deref` *e*) are equivalent to our `atomic` (`ref` *e*) and `atomic` (`deref` *e*), but with an optimized implementation.
- Clojure supports `alter`, `commute`, and `ensure`, which are essentially variations of `ref-set` with different performance characteristics.

Finally, Clojure allows futures to be created in a transaction, leading to unexpected results as described in Chapter 4.

D.2 | Haskell

Syntactically, Haskell writes `forkIO`, `atomically`, `newTVar`, `readTVar`, and `writeTVar` for `fork`, `atomic`, `ref`, `deref`, and `ref-set`. The semantics differ in the following ways:

- Haskell has a different syntax for anonymous functions and `let`.
- Haskell's `forkIO` returns a thread identifier, and not a future. Nevertheless, our formalization models the use of transactions in tasks, in Haskell one uses `atomically` in a task created using `forkIO`.
- Moreover, Haskell does not support the `join` operation on thread identifiers. Instead, waiting for a thread and retrieving its result is usually implemented manually using an `MVar`[1], while our language uses futures for this purpose. Hence, porting our solution to Haskell requires adding a `join` operation to Haskell.
- Transactions are encapsulated in the `STM` monad, and the main program is encapsulated in the `IO` monad. This leads to a different semantics of the `do` block, which in Haskell is syntactic sugar for monad sequencing. Haskell's do notation allows monadic binding (<-) and `let` binding, and may require `return`.
- Haskell does not allow multiple `atomically` blocks to be nested: the type signature of `atomically` is `STM a -> IO a`, and a result of type `IO a` cannot be used in an `STM` block.
- Haskell supports `retry` to abort a transaction and `orElse` to compose two alternative `STM` actions.

Haskell does not allow `forkIO` in a transaction: this is prevented by the type system as `forkIO` operates in the `IO` monad and transactions run in the `STM` monad. One could see transactional futures as the addition of `forkSTM` to Haskell, a fork construct that works within the `STM` monad.

[1] As indicated in the documentation of the Control.Concurrent package at https://hackage.haskell.org/package/base-4.8.1.0/docs/Control-Concurrent.html#g:12.

Bibliography

Agha, G. A. (1985). *Actors: a model of concurrent computation in distributed systems*. PhD thesis, Massachusetts Institute of Technology.

Agha, G. A., Mason, I. A., Smith, S. F., and Talcott, C. L. (1997). A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72.

Agrawal, K., Fineman, J. T., and Sukha, J. (2008). Nested Parallelism in Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 163–174.

Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.

Baek, W., Bronson, N., Kozyrakis, C., and Olukotun, K. (2010). Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 253–262.

Baker, H. C. and Hewitt, C. (1977). The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59.

Barreto, J. a., Dragojević, A., Ferreira, P., Guerraoui, R., and Kapalka, M. (2010). Leveraging Parallel Nesting in Transactional Memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10.

Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S., and Tratt, L. (2017). Virtual

Machine Warmup Blows Hot and Cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):52:1–52:27.

Beeri, C., Bernstein, P. A., and Goodman, N. (1989). A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10.

Bernstein, P. A. and Goodman, N. (1981). Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221.

Bieniusa, A. and Thiemann, P. (2011a). Proving Isolation Properties for Software Transactional Memory. In *Proceedings of the 20th European Symposium on Programming*, ESOP'11, pages 38–56.

Bieniusa, A. and Thiemann, P. (2011b). The Semantics of Twilight Transactions. Technical report, Institut für Informatik, Universität Freiburg.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216.

Bocchino, R. L., Adve, V. S., Adve, S. V., and Snir, M. (2009a). Parallel Programming Must Be Deterministic by Default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09.

Bocchino, Jr., R. L., Adve, V. S., Dig, D., Adve, S. V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. (2009b). A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116.

Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E. B., Pun, K. I., Tarifa, S. L. T., Wrigstad, T., and Yang, A. M. (2015). Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, SFM 2015, pages 1–56.

Bright, P. (2017a). AMD's moment of Zen: Finally, an architecture that can compete. https://arstechnica.com/gadgets/2017/03/

amds-moment-of-zen-finally-an-architecture-that-can-compete/, (Online; published March 2, 2017).

Bright, P. (2017b). Intel launches its new precious metal Xeon platform. https://arstechnica.com/information-technology/2017/07/intels-new-xeon-scalable-platform-is-its-most-complex-yet/, (Online; published July 11, 2017).

Bronson, N. G., Chafi, H., and Olukotun, K. (2010). CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*.

Burckhardt, S., Baldassin, A., and Leijen, D. (2010). Concurrent Programming with Revisions and Isolation Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707.

Burckhardt, S. and Leijen, D. (2011). Semantics of Concurrent Revisions. In *Proceedings of European Symposium on Programming*, ESOP '11, pages 116–135.

Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141.

Cavé, V., Zhao, J., Shirako, J., and Sarkar, V. (2011). Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61.

Chakravarty, M. and Keller, G. (2001). Nepal — Nested Data Parallelism in Haskell. In *Euro-Par 2001 Parallel Processing*, pages 524–534.

Chickering, D. M., Heckerman, D., and Meek, C. (1997). A Bayesian Approach to Learning Bayesian Networks with Local Structure. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, UAI'97, pages 80–89.

Clebsch, S., Drossopoulou, S., Blessing, S., and McNeil, A. (2015). Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12.

Coffman, E. G., Elphick, M., and Shoshani, A. (1971). System Deadlocks. *ACM Computing Surveys*, 3(2):67–78.

Dabrowski, F., Loulergue, F., and Pinsard, T. (2013). Nested Atomic Sections with Thread Escape: An Operational Semantics. In *Proceedings of the 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '13, pages 29–35.

Dabrowski, F., Loulergue, F., and Pinsard, T. (2015). A Formal Semantics of Nested Atomic Sections with Thread Escape. *Computer Languages, Systems & Structures*, 42(C):2–21.

De Koster, J., Marr, S., Van Cutsem, T., and D'Hondt, T. (2016a). Domains: Sharing state in the communicating event-loop actor model. *Computer Languages, Systems & Structures*, 45:132–160.

De Koster, J., Van Cutsem, T., and De Meuter, W. (2016b). 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, pages 31–40.

de Vries, E., Koutavas, V., and Hennessy, M. (2010). Communicating Transactions. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 569–583.

Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., and De Meuter, W. (2006). Ambient-Oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254.

Denning, P. J. and Dennis, J. B. (2010). The Resurgence of Parallelism. *Communications of the ACM*, 53(6):30–32.

Dijkstra, E. W. (1965). Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569.

Donnelly, K. and Fluet, M. (2006). Transactional Events. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 124–135.

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. (1989). Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124.

Emerick, C., Carper, B., and Grand, C. (2012). *Clojure Programming*. O'Reilly Media, Inc.

Ennals, R. (2006). Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge.

Farchi, E., Nir, Y., and Ur, S. (2003). Concurrent bug patterns and how to test them. In *Proceedings of International Parallel and Distributed Processing Symposium*, IPDPS '03.

Felleisen, M., Findler, R. B., and Flatt, M. (2009). *Semantics Engineering with PLT Redex*. The MIT Press.

Field, J. and Varela, C. A. (2005). Transactors: A Programming Model for Maintaining Globally Consistent Distributed State in Unreliable Environments. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 195–208.

Flanagan, C. and Felleisen, M. (1995). The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 209–220. ACM.

Geer, D. (2005). Chip makers turn to multicore processors. *Computer*, 38(5):11–13.

Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76.

Godefroid, P. and Nagappan, N. (2008). Concurrency at Microsoft – An Exploratory Survey. Technical report. https://www.microsoft.com/en-us/research/publication/concurrency-at-microsoft-an-exploratory-survey/.

Guerraoui, R. and Kapałka, M. (2008). On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184.

Guerraoui, R., Kapalka, M., and Vitek, J. (2007). STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324.

Haines, N., Kindred, D., Morrisett, J. G., Nettles, S. M., and Wing, J. M. (1994). Composing First-class Transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736.

Haller, P. and Odersky, M. (2007). Actors That Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION'07, pages 171–190.

Halstead, R. H. (1985). MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538.

Harris, T., Larus, J. R., and Rajwar, R. (2010). *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2nd edition.

Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. (2005). Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60.

Harris, T. and Singh, S. (2007). Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 251–264.

Herlihy, M. and Moss, J. E. B. (1993). Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300.

Herlihy, M. and Shavit, N. (2011). *The art of multiprocessor programming*. Morgan Kaufmann.

Heumann, S. T., Adve, V. S., and Wang, S. (2013). The Tasks with Effects Model for Safe Concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 239–250.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245.

Hickey, R. (2012). Clojure Concurrency. Talk given to the Western Mass. Developers Group. Available online at https://www.youtube.com/watch?v=nDAfZK8m5_8.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

Hovemeyer, D. and Pugh, W. (2004a). Finding Bugs is Easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 132–136.

Hovemeyer, D. and Pugh, W. (2004b). Finding Concurrency Bugs In Java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*.

Imam, S. M. and Sarkar, V. (2012). Integrating Task Parallelism with Actors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 753–772.

Jones, S. P., Leshchinskiy, R., Keller, G., and Chakravarty, M. M. T. (2008). Harnessing the Multicores: Nested Data Parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2, pages 383–414.

Karaorman, M. and Bruno, J. (1993). Introducing Concurrency to a Sequential Language. *Communications of the ACM*, 36(9):103–115.

Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S., and Findler, R. B. (2012). Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 285–296.

Kogan, A. and Herlihy, M. (2014). The Future(s) of Shared Data Structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 30–39.

Lea, D. (2000). A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43.

Lee, C. Y. (1961). An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365.

Lee, E. A. (2006). The Problem with Threads. *Computer*, 39(5):33–42.

Lee, J. K. and Palsberg, J. (2010). Featherweight X10: A Core Calculus for Async-finish Parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 25–36.

Lesani, M. and Lain, A. (2013). Semantics-preserving Sharing Actors. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, pages 69–80.

Lesani, M. and Palsberg, J. (2011). Communicating Memory Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 157–168.

Liskov, B. and Shrira, L. (1988). Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 260–267.

Lu, S., Park, S., Seo, E., and Zhou, Y. (2008). Learning from mistakes — A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 329–339.

Luchangco, V. and Marathe, V. J. (2011). Transaction Communicators: Enabling Cooperation Among Concurrent Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 169–178.

Message Passing Interface Forum (1994). *MPI: A Message-Passing Interface Standard.* https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps.

Message Passing Interface Forum (2015). *MPI: A Message-Passing Interface Standard – Version 3.1.* https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

Miller, M. S., Tribble, E. D., and Shapiro, J. (2005). Concurrency Among Strangers. In De Nicola, R. and Sangiorgi, D., editors, *International Symposium on Trustworthy Global Computing*, pages 195–229.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46.

Moore, G. E. (1998). Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85.

Moore, K. F. and Grossman, D. (2008). High-level Small-step Operational Semantics for Transactions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 51–62.

Morandi, B., Nanz, S., and Meyer, B. (2014). Safe and Efficient Data Sharing for Message-Passing Concurrency. In *Proceedings of the 16th International Conference on Coordination Models and Languages*, COORDINATION'14, pages 99–114.

Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. (2006). Supporting Nested Transactional Memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 359–370.

Moss, J. E. B. (1981). *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology.

Moss, J. E. B. and Hosking, A. L. (2006). Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201.

Nash, M. and Waldron, W. (2016). *Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications.* O'Reilly Media, Inc., 1st edition.

Navabi, A. and Jagannathan, S. (2009). Exceptionally Safe Futures. In *Proceedings of the 11th International Conference on Coordination Models and Languages*, COORDINATION'09, pages 47–65.

Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. (2007). Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 68–78.

Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., and Valero, M. (2008). The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-core Environment. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 67–78.

Ramadan, H. E. and Witchel, E. (2009). The Xfork in the Road to Coordinated Sibling Transactions. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '09.

Randall, K. H. (1998). *Cilk: Efficient multithreaded computing.* PhD thesis, Massachusetts Institute of Technology.

Reppy, J., Russo, C. V., and Xiao, Y. (2009). Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 257–268.

Reppy, J. H. (1991). CML: A Higher Concurrent Language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 293–305.

Shavit, N. and Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10(2):99–116.

Smaragdakis, Y., Kay, A., Behrends, R., and Young, M. (2007). Transactions with Isolation and Cooperation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 191–210.

Srinivasan, S. and Mycroft, A. (2008). Kilim: Isolation-Typed Actors for Java. In Vitek, J., editor, *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP'08, pages 104–128.

Stork, S., Marques, P., and Aldrich, J. (2009). Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 933–940.

Stork, S., Naden, K., Sunshine, J., Mohr, M., Fonseca, A., Marques, P., and Aldrich, J. (2014). Æminium: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Transactions on Programming Languages and Systems*, 36(1):2:1–2:42.

Sulzmann, M., Lam, E. S. L., and Van Weert, P. (2008). Actors with Multi-headed Message Receive Patterns. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, COORDINATION'08, pages 315–330.

Surendran, R. and Sarkar, V. (2016). Automatic Parallelization of Pure Method Calls via Conditional Future Synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 20–38.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3).

Swalens, J., De Koster, J., and De Meuter, W. (2016). Transactional Tasks: Parallelism in Software Transactions. In *Proceedings of the 30th European Conference on Object-Oriented Programming*, ECOOP '16, pages 23:1–23:28.

Swalens, J., De Koster, J., and De Meuter, W. (2017). Transactional Actors: Communication in Transactions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems*, SEPS 2017, pages 31–41.

Swalens, J., Marr, S., De Koster, J., and Van Cutsem, T. (2014). Towards Composable Concurrency Abstractions. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software*, PLACES'14.

Tanenbaum, A. S. and Bos, H. (2014). *Modern Operating Systems*. Prentice Hall Press, 4th edition.

Tasharofi, S., Dinges, P., and Johnson, R. E. (2013). Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *Proceedings of the 27th European conference on Object-Oriented Programming*, ECOOP'13, pages 302–326.

Van Roy, P. and Haridi, S. (2004). *Concepts, techniques, and models of computer programming*. The MIT Press.

Varela, C. and Agha, G. (2001). Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34.

Vitek, J., Jagannathan, S., Welc, A., and Hosking, A. L. (2004). A Semantic Framework for Designer Transactions. In *Proceedings of the 13th European Symposium on Programming*, ESOP '04, pages 249–263.

Volos, H., Welc, A., Adl-Tabatabai, A.-R., Shpeisman, T., Tian, X., and Narayanaswamy, R. (2009). NePalTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 291–292.

Warth, A., Ohshima, Y., Kaehler, T., and Kay, A. (2011). Worlds: Controlling the Scope of Side Effects. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 179–203.

Watson, I., Kirkham, C., and Lujan, M. (2007). A Study of a Transactional Parallel Routing Algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 388–398.

Welc, A., Jagannathan, S., and Hosking, A. (2005). Safe Futures for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 439–453.

Yonezawa, A., Briot, J.-P., and Shibayama, E. (1986). Object-oriented Concurrent Programming in ABCL/1. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 258–268.

Zeng, J., Barreto, J., Haridi, S., Rodrigues, L., and Romano, P. (2016). The Future(s) of Transactional Memory. In *Proceedings of the 45th International Conference on Parallel Processing*, ICPP '16, pages 442–451.

Zeng, J., Romano, P., Rodrigues, L., Haridi, S., and Bareto, J. (2015). In Search of Semantic Models for Reconciling Futures and Transactional Memory. In *Proceedings of the 7th Workshop on the Theory of Transactional Memory*.

Zhang, L., Krintz, C., and Nagpurkar, P. (2007). Language and Virtual Machine Support for Efficient Fine-Grained Futures in Java. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 130–139.

Zhang, Y. and Hansen, E. A. (2006). Parallel breadth-first heuristic search on a shared-memory architecture. In *Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, AAAI '06.

Zhao, J., Lublinerman, R., Budimlić, Z., Chaudhuri, S., and Sarkar, V. (2013). Isolation for Nested Task Parallelism. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 571–588.

Zhou, T., Luchangco, V., and Spear, M. (2017). Brief Announcement: Extending Transactional Memory with Atomic Deferral. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 371–373.

Zyulkyarov, F., Gajinov, V., Unsal, O. S., Cristal, A., Ayguadé, E., Harris, T., and Valero, M. (2009). Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 25–34.

A Multi-Paradigm Concurrent Programming Model

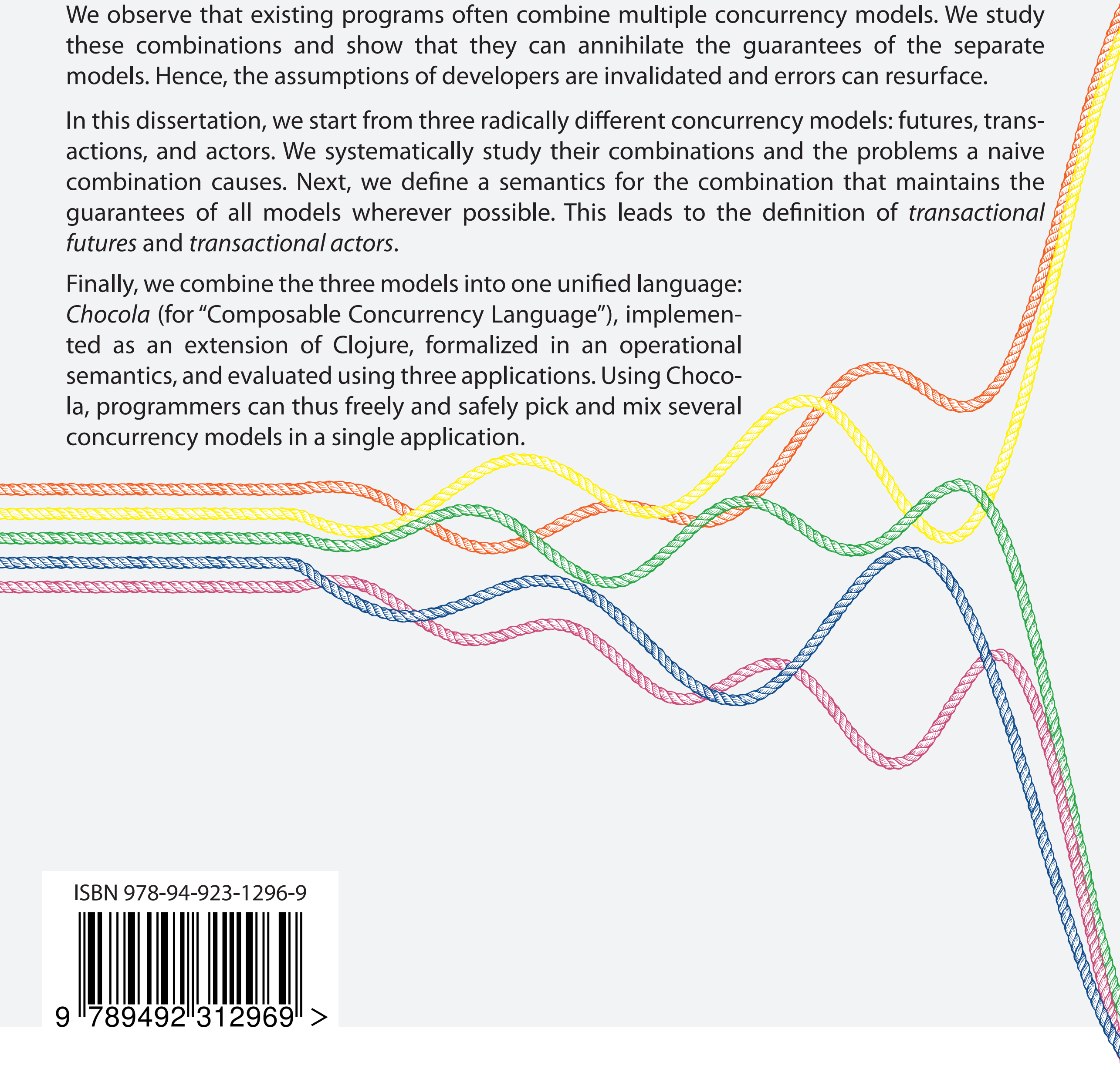
Janwillem Swalens

Since the introduction of multi-core processors, programmers must explicitly use concurrency to make their programs faster. This is notoriously difficult. Hence, programming languages provide concurrency models: techniques that introduce parallelism in a controlled manner, providing guarantees that prevent common errors such as race conditions and deadlocks.

We observe that existing programs often combine multiple concurrency models. We study these combinations and show that they can annihilate the guarantees of the separate models. Hence, the assumptions of developers are invalidated and errors can resurface.

In this dissertation, we start from three radically different concurrency models: futures, transactions, and actors. We systematically study their combinations and the problems a naive combination causes. Next, we define a semantics for the combination that maintains the guarantees of all models wherever possible. This leads to the definition of *transactional futures* and *transactional actors*.

Finally, we combine the three models into one unified language: *Chocola* (for "Composable Concurrency Language"), implemented as an extension of Clojure, formalized in an operational semantics, and evaluated using three applications. Using Chocola, programmers can thus freely and safely pick and mix several concurrency models in a single application.



ISBN 978-94-923-1296-9

9 789492 312969 >