Compositional Formal Analysis for Concurrent Object-Oriented Languages

Doctoral Dissertation by

Thi Mai Thuong Tran

Submitted to the
Faculty of Mathematics and Natural Sciences at the University of Oslo
for the degree Philosophiae Doctor in Computer Science

Date of submission: 30.11.2012
Date of public defense: 11.06.2013

Research group for Precise Modeling and Analysis
Department of Informatics
University of Oslo
Norway

Oslo, May 2013

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1359

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

Concurrency is a ubiquitous phenomenon in modern software ranging from distributed systems communicating over the Internet to communicating processes running on multi-core processors and multi-processors. Therefore modern programming languages offer ways to program concurrency effectively. Still, writing correct concurrent programs is notoriously difficult because of the complexity of possible interactions between concurrent processes and because concurrency-related errors are often subtle and hard to reproduce, especially for safety-critical applications. This thesis develops and formally investigates different static analysis methods for various concurrency-related problems in concurrent object-oriented programming languages to guarantee the absence of common concurrency-related errors, hence contribute to the quality of concurrent programs.

Aspects covered by our analyses involve lock-based concurrency, transaction-based concurrency, resource consumption and inheritance. In the lock-based setting, using explicit locks in a non-lexical scope to protect critical regions might be the source of aliasing problems or misuse of locks. In the transaction-based model, a similar problem of misuse of non-lexical transactions can happen in the same way as the misuse of explicit locks. Furthermore, for the purpose of checking conflicts and supporting rollback mechanisms, additional storage are required to keep track of changes during transactions' execution which can lead to resource consumption problems. So it is useful to investigate different analysis methods to achieve safe concurrent programs. In open systems, the combination of inheritance and late-binding problematic, e.g., replacing one base class by another, seemingly satisfying the same interface description, may break the code of the client of the base class. This thesis also investigates an observable interface behavior of open systems for a concurrent object-oriented language with single-class inheritance where code from the environment can be inherited to the component and vice versa.

All analyses developed in this work are formulated as static type and effect systems resp. an open semantics based on a type and effect system. As usual for type and effect systems, they are formalized as derivation systems over the syntax of the languages, and thus compositional. In all cases, we prove the correctness of the analyses. When based on a rigorous formal foundation, the analyses can give robust guarantees concerning the safety of the program. To tackle the complexity of large and distributed applications, we have insisted that all the analysis methods in this thesis should be compositional. In other words, our analyses are scalable, i.e., a composed program should be analyzed in a divide-and-conquer manner relying on the results of the analysis of its sub-parts, so they are useful in practice.

Acknowledgments

My PhD period has been so far the most exciting, enjoyable, and memorable part of my life. I have learnt and grown so much both in knowledge and life. I was given a lot of opportunities to study and do research. I have met many interesting people and learnt so much from them.

Now, at the end of my thesis time, first of all I would like to thank my main supervisor, Martin Steffen. I am greatly thankful for being his student. His broad and deep knowledge has opened a new world for me in the field of formal methods. His love and passion for research have influenced me a lot and kept me always motivated during 4 years of my PhD. It is known that during a PhD, anyone will at some point reach a low-point of motivation and enthusiasm. And I was not an exception. Sometimes I really wanted to give up, but when I looked at him, I said to myself "shame on me", because I have not seen any moment where he is bored with research, even when he is exhausted and could not move his body anymore but if one asks him a research question, he will suddenly become energetic again. From him, I have learnt things from basic to deep levels, how to appreciate one's own effort, how to be open-minded to new ideas, how to be more professional in working, how to think and do research independently. From him, I have learnt to be precise, concrete, and very honest when doing research. I could never expect anything better than that.

As a leader of the group, Olaf Owe is a great supporter both in research and personal life. Olaf created all possibilities for me to access research resources like supporting me to go to conferences and schools where I met so many interesting people and my eyes were opened for new things. I always got back with so much passion for research. He always tried to share and find solutions for problems and difficulties I had to face from time to time. He is also the person who taught me how to keep a good balance between life and work. Also Einar Broch Johnsen has been very helpful and supportive for me. I benefited very much from his clear vision and direction in research, from his deep knowledge and his strict attitude in doing research and in writing papers (and in not tolerating bad coffee). Moreover, I would like to thank all my colleages for being always cheerful and nice to me, and for providing a stimulating working environment. We have been together as a family with all joy and sharing.

Last but certainly not least, I thank my parents who have given me the opportunity to take a good education, who have done beyond their ability to provide me all needed support for my education. I remember, they even volonteered to wash dishes for me and do chores so that I could have a little 5 more minutes to read a book. For me, this gift of time, love, education, and trust is priceless and nothing can be compared to that. They are always my big motivation for everything I have done for my education as well as my life.

Contents

Part I

Overview

CHAPTER 1

Introduction

1.1 Motivation

Software applications are today integrated in all aspects of life from simple applications like children's toys to complex ones, such as safety-critical systems, industrial automation, air traffic control, hospital diagnostic X-ray systems, telecommunications, and software for financial markets. The more complex the systems are, the more high-level programming languages are needed. Programming languages have evolved enormously: modern functional languages are influenced by the early language LISP and modern object-oriented languages are influenced by Simula. For hardware applications, early low-level programming languages which provide little or no abstraction from a computer's instruction set architecture are adequate, e.g., machine-dependent and assembly languages. With the dramatic increase of large and complex software applications, it is, however, impossible to work directly with low-level layers of the architecture using low-level programming languages. Those languages are simple, but difficult to use because the programmer has to remember numerous technical details. Hence, high-level programming languages, such as object-orientation with machine-independent semantics featuring data abstractions and structuring mechanisms, are needed. They make program development simpler and more understandable, and hence they are widely used in practice. Reasons for the growing popularity of high-level programming langugages include encapsulation, re-use, isolation, and structuring.

Moreover, with the rapid development of multi-cores and multi-processors, concurrent high-level programming languages and different models of concurrency are needed to take advantages of multi-processors. Today there are three main concurrency models: lock-based, transaction-based and actor models based on active objects. Some examples of popular concurrent languages following the lock-based model are: Ada [38], CC^{++}[76], an integration of C^{++}[183] and Concurrent C, and Java [82]. For the transaction-based model, Clojure [42] is a dialect of the Lisp programming language supporting STM and running on the Java Virtual Machine, Common Language Runtime, and JavaScript engines; Transactional Featherweight Java [110] is a recent, advanced transactional languages featuring multi-threaded and nested transactions. Example languages based on the Actor model are: Scala [158], Erlang [24] and Creol [119].

Even in high-level object-oriented programs, shared memory concurrency may sometimes cause serious errors which can not be detected by normal testing at the later phase of the software development cycle. Especially, accidents caused by software errors in critical systems can be catastrophic in terms of economic damage and loss of human lives. Software applications have been responsible for many serious accidents in safety-critical systems. Some of the most widely cited software-related accidents in safety-critical systems involved the Therac-25 [128], a computerized radiation therapy machine supporting a multi-tasking environment with concurrent access to shared data. The six accidents caused by massive overdoses using this system resulted in serious injuries and even deaths. There were a number of causes, but concurrent programming errors played an important part. Race conditions between different concurrent activities in the control program resulted in *occasional* erroneous control outputs. Furthermore, the nature of the errors caused by faulty concurrent programs made it difficult to recognize that there was a problem.

It was not because engineers had not taken care in building and testing the system. Actually, before the Therac-25 was deployed, the engineers did use many testing and safety-critical techniques. Moreover, after one accident, the manufacturer tried to reproduce the condition which occurred at the treatment by exhaustively testing the system. When the test did not return any error, they concluded that a hardware error caused the accident, and implemented a solution based on that assumption. It was declared that the system was several orders of magnitude safer, but accidents did not cease. In the end, the error was detected by chance when a rare interaction combination resulting in the mentioned race condition appeared. One may conclude that the designers of the Therac-25 software seemed largely unaware of the principles and practice of concurrent programming. They did not use methods reliable enough to verify the design as well as the implementation of the system to prevent such errors. Concurrency errors such as race conditions or deadlocks can not be detected easily by testing the implementation due to the potential state explosion and arbitrary interferences between concurrent components. Formal methods provide a more suitable approach for assuring the correct functioning of concurrent software by detecting errors early in the design phases with the help of systematic search and analysis techniques.

Detecting concurrent errors is in general challenging. Finding the causes and re-producing the situations in which the errors occurred are even more demanding. Errors in concurrent software systems might not be located in one particular line of code inside a specific component, but by a combination of factors or interactions between them. When software systems become bigger and more complex, the number of combinations of interactions between system components can increase exponentially, and even infinitely. It means that exhaustively testing all cases or all combinations to reproduce the situation is impossible. For certain systems, it is unrealistic to test exhaustively in practice. One needs techniques to analyze and verify such systems to help preventing software errors before the systems are executed in practice, i.e., one needs to develop models and analyze these.

A methodology to build a large complex software system from software components systematically is needed. The concept of software components was proposed by M.D.McIlroy [140]. Components in a broad view can be seen as independent building blocks which provide services to the outside and can be combined with other components to construct larger systems. For example, a component can be a hardware driver, a bricklayer or wall in a house, or a function or procedure in a software program. To be widely applicable to different machines and users, software components are provided in categories as black boxes based on their properties and functionalities. With the advent of the Internet, communication, parallelism, and distribution play a more and more dominant role. Developing large and complex systems, especially for distributed systems often requires cooperation between people located at different places and having different specialities. That cooperating process requires principles and techniques to produce high-quality software components systematically and efficiently.

However, analyzing the global behavior of complex systems is very difficult. The *divide-and-conquer* technique which hierarchically decomposes programs into smaller ones is a powerful principle to master the complexity of large software programs: it allows to reduce the development and verification problem of a program to that of its constituent sub-programs by starting from its top-level specification and verifying that specification on the basis of the specifications of those sub-programs without any knowledge of their interior construction. The components of a system can be composed in different ways wrt. the supported features of the language used to program the system. It is important to provide a guarantee that the whole composed system functions correctly based on the analysis results of its individual parts, because it can be the case that a safe program does not always consist of safe sub-programs, or similarly when composing a program from safe sub-programs, the resulting program may not be safe. For these reasons, we need compositional techniques which enable us to compose a large system from verified subsystems without (much) additional verification effort or redoing all the work done for sub-systems. Compositional techniques are required to guarantee that the composition of independent and verified sub-programs is done properly without incurring extra costs. In software development, the principle of compositionality describes how components can be composed and what characteristics or constraints they have to follow to reduce the cost and errors while building large and critical software systems. The semantics of a programming language is *compositional* if the meaning of a composite expression is determined by the meanings of its constituent expressions. An analysis or a verification method is *compositional* for a programming language if the analysis or verification of a composite program is built on the analyses or verification of its constituent programs. In software analysis and verification, compositionality is an important property of program correctness methods because it allows a bottom-up method: Verifying whether a program meets its specification is verified on the basis of specifications of its constituent components only, without additional need for information about the interior construction of those components.

In this thesis, our analyses are concerned with concurrent object-oriented programming languages. Such languages contain references as well as different forms of concurrency, namely shared variable concurrency within objects and method calls, which make the analyses challenging.

1.2 Research goals

It is generally accepted that concurrent programs are notoriously hard to get right. Despite their obvious usefulness, there is a lack of analysis methods to identify concurrency errors, in particular for advanced constructs. Therefore the overall goal of this thesis is *to guarantee the absence of common concurrency-related errors and hence contribute to the quality of concurrent programs*. To achieve that goal, this thesis will address the following questions:

1. How can we statically guarantee safe use of non-lexical transactions for languages supporting multi-threaded and nested transaction model?

2. How can we statically guarantee safe use of resources in a transaction-based model with implicit synchronization.

3. How can we statically guarantee safe use of explicit non-lexical locks for languages featuring dynamic creation of objects, threads, locks and lock aliasing, such as in the Java

language?

4. How can we capture observable behaviors of open systems in the presence of inheritance and late-binding? How can we present enough information in the interface specification to capture all possible behaviors of the systems, but at the same time exclude impossible interactions from outside?

In order to answer these questions, the work in this thesis develops different analysis methods for concurrent object-oriented programming languages based on the lock-based model, the transaction-based model, and the actor model. Moreover, the developed analysis methods have to meet the following requirements:

Formality and rigor: To give reliable guarantees, the analyses must be described in a formal mathematical manner. Besides, the studied languages and their semantics should be specified rigorously and concisely. Of course, the analyses are required to give correct results wrt. the given semantics. This relationship is known as soundness which must be formally established by means of proofs.

Scalability: To efficiently deal with large and complex systems, the methods must be scalable. Consequently they must be compositional and make use of abstractions. To achieve compositionality, a system must be characterized as an open system, i.e., as interacting with an unknown environment and hiding its internal details.

Usability: To be accepted in practice, the analyses must provide their guarantees with as little user involvement as possible, even more so, since the analyses will be based on formal theories and advanced concepts. Therefore it is essential to automatize the analyses, i.e., they must be based on decidable theories. Again that requires the analyses are based on abstractions of the concrete formally specified program behaviors.

Concurrency: Concurrency comes in many different flavours and modern programming languages use a diversity of mechanisms to express concurrent computations. Our goal is to analyze relevant properties and to investigate prominent synchronization mechanisms, including lock-based synchronization as well as the transaction-based model. Since modern languages include shared variable concurrency and message passing communication, both should be investigated as well. As for message passing, we will cover synchronous method calls as in Java's multi-threading concurrency model as well as asynchronous method calls as in languages based on active objects.

1.3 Outline

In the following chapters, I briefly give some background about concurrency and object orientation to introduce the context for the work reported in the research papers in Part II. Chapter 2 reviews some main characteristics and principles of object-oriented programming languages, the concept of concurrency, the principles of concurrent programming as well as the combination of concurrency and object-orientation in programming languages. Moreover, different communication models of concurrency and synchronization mechanisms are also discussed in the chapter. Chapter 3 discusses different problems related to concurrency, such as safety and liveness and various approaches in the literature to solve them. In Chapter 4, I describe our design choices for the syntax and semantics of our core calculus as well as our analyses based on type and effect systems to solve specific concurrency problems. I give a short overview over various analyses and problems dealt with in all work of this thesis in Chapter 5. Chapter 6 gives an evaluation of the contribution of this thesis towards the goals stated in this chapter. Chapter 7 conclude the thesis with an overview of the contributed research papers and their main contribution.

CHAPTER 2

Concurrency and object-oriented programming languages

In this chapter, we first recall in Section 2.1 problems and difficulties inherent in concurrent programming and mention practices which help alleviating common pitfalls. Furthermore, we discuss common synchronization and communication mechanisms. Afterwards in Section 2.2, we mention language features and structuring mechanisms found in object-oriented languages, and discuss a number of object-oriented languages. Finally, Section 2.3 discusses design issues when combining concurrency and object orientation.

2.1 Concurrency

2.1.1 Concurrent programming

Concurrent applications are notoriously difficult to get right, i.e., to design, program, test, and verify. Two reasons can be identified for that. One is that parallel executions gives rise to a typically enormous number of different schedulings and interferences which are difficult to grasp and to analyze. The resulting non-determinism, moreover, makes errors hard to reproduce which make testing of concurrent programs challenging. The second reason is that interesting properties of concurrent programs and respectively their violations are non-local in nature, i.e., the violations of properties can not be pin-pointed to a single line of code responsible for the error. Examples of such global concurrency-related bugs are race conditions and deadlocks. In the book "Java Concurrency in Practice" [79], the authors identify shared mutable states as the root of the mentioned difficulties and give general guidelines to write good concurrent code, which can be summarized as follows:

Immutability: It is not a good idea to have mutable states, as all concurrency issues boil down to coordinating access to mutable states. The less mutable states, the easier it is to ensure thread safety, i.e., absence of concurrency-related errors when used concurrently. Immutable objects are automatically thread-safe and can be shared freely without protecting it by locking or making defensive copying.

Isolation: for mutable states, one can restrict interference in the following ways:

- **Thread locality:** Avoid sharing mutable data in the first place. Working on thread-local or process-local data obviously avoids interference problems.
- **Data encapsulation:** Encapsulation is crucial to manage the complexity of concurrent programs. For instance, encapsulating data within objects together with a suitable synchronization mechanism makes it easier to preserve their invariants.

Protection: Use synchronization (mutexes) to control accesses to a mutable shared data by multiple threads.

These general guidelines will be reflected in the design of the calculi used in our work as follows: Our calculi are characterized by a strict separation between thread-local variables and instance variables, i.e., fields. The latter are mutable and allocated on the heap, thus in general shared among threads. Another form of isolation is inherent in the transactional setting where each thread operates on a local copy of its data. All our calculi are object-oriented, and objects are the units of encapsulated data. A protection mechanism we investigate is lock-based. Immutability as protection against interference plays no role in our investigations. We use, however, local variables in a single-assignment manner, i.e., each variable gets assigned a value at most once. This standard form of representation simplifies the formulation of the analyses without sacrifying expressivity.

2.1.2 Synchronization mechanisms

Essentially, concurrent programs are difficult to program, test, debug, and verify because:

- concurrency bugs do not manifest themselves predictably as they involve multiple processes competing for shared resources
- concurrency bugs are often detected at the worst possible time may lead to a poor performance and potentially state space explosion.
- different shared resources have different properties and act heterogeneously.

A concurrent system needs to make sure that each process acts correctly in the sense of being in the right place at the right time to take advantage of what resources are available at any given moment. A simplified illustration of a typical situation in a concurrent system is depicted in Figure 2.1. Without loss of generality, we assume that the system has more than two processes running concurrently on a machine with two processors. Since there are two processors, only two processes can run simultaneously. The operating system must periodically allocate CPU time to different processes and swap them in and out of the CPUs so that all the processes can make progress.

Another reason for swapping in and out processes from the processors is to reduce unnecessary waiting time. There is an enormous discrepancy in the speeds of operation of peripheral or I/O devices. If a program is single-threaded, the processor remains idle while it waits for a synchronous I/O operation to complete. In a multi-threaded program, the peripheral devices can operate independently of the central processor, allowing the application to still make process during the blocking I/O. For example, assume that the processes 1 and 2 are now occupying the processors 1 and 2 respectively. Assume further that process 1 is trying to access some of the I/O devices. As known, I/O devices are much slower to access than registers and

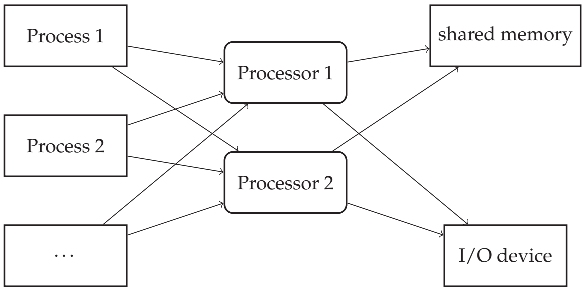


Figure 2.1: Concurrent processing architecture

RAM situated inside CPU. Moreover, at one time there is only one value that can physically be written to a memory address, and only one part of a hard disk that can physically be accessed.

These factors pose two main problems:

- How to avoid unnecessary waiting time of a CPU when it needs to wait for a slower resource, such as main memory or an I/O device.
- What to do when at least two CPUs want to access a shared resource simultaneously which could lead to unexpected results, called data race problems. Data races arise when at least two processors or processes want to write different values to a memory address, or one writes and the other one reads. The same situation applies to a single-processor machine with multiple processes because the processes are usually swapped in and out of the available processor under interrupt, which can be described by Figure 2.1 when we leave out the processor 2.

A simple solution to the first problem is that while one process is waiting on a slow resource, it is swapped out of processor 1 and another process that requires the CPU is swapped in to run. For the second problem, the question of which process gets the right to access the memory first is determined by software or hardware algorithms, called schedulers. Schedulers and synchronization mechanisms are needed to coordinate simultaneous accesses to shared resources in concurrent programs (both in small-scale multiprocessing systems and in distributed systems). Thread (or process) synchronization is the application of particular mechanisms to ensure that two concurrently executing threads or processes do not execute specific portions of a program at the same time. There is at most one thread being executed at a time, any other thread trying to access the same data must wait until the first thread finishes. So synchronization can be used to guarantee thread-safe programs.There are two focuses of synchronization:

- Explicit synchronization is a way of providing synchronization facilities in terms of languages primitives or functions. Concurrent programming languages are said to support explicit synchronization if they provide synchronization constructs at the user-level where users can use those constructs to take control over how concurrent accesses act on a shared data in order to avoid unpredicted results. In high-level programming languages (e.g, Java), these facilities include mutual-exclusion locks, semaphores, and notifications. Each facility guarantees that when synchronization has been achieved, all the side effects of another thread are visible, at least up to the point where that other thread last released the synchronization facility. On the one hand, explicit parallel programming can be useful if skilled programmers can take advantage of explicit synchronization primitives to produce very efficient code. On the other hand, it could cause errors, as programming with explicit parallelism is often difficult and can be misused by unskilled programmers. However, analyses for explicit synchronization programs are normally more simple than implicit ones, where synchronization points are not represented syntactically.

- Implicit synchronization: In contrast to explicit synchronization, implicit synchronization is not (fully) represented in the source code by user-provided language primitives. For example, invoking `notifyAll` on an object in Java is a synchronization command affecting all threads waiting on that object without specifying which threads are meant, i.e., leaving them implicit. Synchronization is taken care of by the underlying semantics. Without worrying about with whom exactly to communicate or synchronize, this may simplify the programming task, but may reduce the control users have over the parallel execution of the program, resulting sometimes in less-than-optimal efficiency. The fact that users may not be aware of when and which threads are being scheduled, can make debugging more difficult. Furthermore, static analyses, which are often syntax-directed, become more complex.

2.1.3 Communication mechanisms

In concurrent computing, programs are designed as interacting processes which may be executed in parallel. They can be interleavingly executed on a single processor, or in parallel on a set of processors that may be distributed across a network. The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different concurrent processes, and coordinating access to shared resources. A number of different methods for implementing concurrent programs has been developed, such as implementing each sequential execution as an operating system process, or implementing the processes as a set of threads within a single operating system process. There are two basic models for inter-process communication:

Shared Memory: In computing, shared memory is memory that may be simultaneously accessed by multiple programs. Read and write operations are used to exchange information. Shared memory is an efficient means of passing data between programs because the programs can directly access shared memory locations and change their values. Shared memory is used in several concurrent programming languages, such as Java and C# as an indirect concurrency mechanism in which concurrent components communicate by using shared variables. But allowing simultaneous accesses to shared data can cause unexpected errors, such as race conditions [152]. To control concurrent access to shared data, synchronization mechanisms are used. They can be of the form of semaphores [61], a useful abstraction for controlling accesses to a common resource by multiple processes where the value of the semaphore says how many units of a particular resource are available; or they can be of the form of monitors [98], an object intended to be used safely by more than one thread with its methods executed with mutual exclusion, i.e, at each point in time, at most one thread may be executing any of the monitor's methods. They can use locks to mutually assure exclusive access, i.e, no two processes or threads can be in a common critical section at the same time [62]. For example, the underlying model of Java is based on shared memory and uses lock-based synchronization to guarantee mutual exclusion.

A shared memory system is relatively easy to program since all processes or processors share a single view of data on a single address space. Synchronization and communication in shared memory are distinct. Synchronization operations (e.g., locks and barriers), in addition to read and write operations, need to be provided to assure mutual exclusions. Communication of data between processes and objects in shared memory is asynchronous as tasks share a common address space (i.e., a sender does not wait for a receiver to get the data) and implicit (it is hiddenly done under reading and writing operations) which allows users to concentrate on the issues related to parallelism by relieving them of the details of the inter-process communication. Hence program development can often be simplified. In addition, that all processes and objects communicate via reading and writing to the same address space helps avoid redundant copies and helps provide high-performance access to shared information between multiple processes.

Despite its advantages, such as efficient means of sharing data between programs and reducing the overhead of communicating, one of the known disadvantages of the shared memory is the potential of data races. A data race is a situation where at least two processes try to access a shared data at the same time and at least one of them is writing, which could lead to non-determinist results. Furthermore, shared memory is less powerful and does not scale well, as the communicating processes are running on the same machine, and care must be taken to avoid issues such as data races or the problem of cache coherence. Checking cache coherence must be performed to guarantee that the change in content of one cache is made visible to other processes, which may use the data, to avoid incoherent data. Seeing inconsistent views of the same memory space by processes could lead to unexpected behaviors. Moreover, controlling data locality is hard to understand and beyond the control of the average user. For example, to handle fine-grained locks correctly is challenging as users need to remember which lock is attached to which region and what is the order over locks to avoid deadlocks. Users often lose an overview of locks when dealing with a complex system using many locks at the same time. Using coarse-grained locks can be a solution, but in many cases it can harm performance of the system because of redundant waiting. This makes shared memory programming difficult as far as synchronization is concerned. As far as distributed systems are concerned, many processes accessing the same memory could cause bottlenecks as well as task switching overhead, and thus decrease the performance of the whole system.

Message passing: Message passing, another form of communication used in parallel computing, object-oriented programming, and inter-process communication, does not share common memory. In this model, processes or objects communicate directly with each other in some form of points-to-points connection by sending and receiving messages which contain both data and synchronization in a single unit. This form of communication is well adapted for loosely connected processors (as in a distributed system), and for client-server topologies. The exchange of messages may be carried out either synchronously letting the sender be blocked until the message is received, or asynchronously letting the sender continue its work without waiting for a receiver, which often involves a buffer for intermediate steps. Asynchronous message passing is, however, considered to be more advanced than synchronous because: first of all, synchronous communication can be built on top of asynchronous communication by ensuring that the sender always waits for an acknowledgement message from the receiver to continue; secondly, in the asynchronous model, the sender delivers a message to the receiver without waiting for the receiver to be ready, hence it does not block the whole system and therefore increases the system performance. Message passing is used in Actor models, e.g, Scala, Erlang, and various process calculi like CSP [99] and CCS [144].

Message passing employs an explicit communication mechanism. Programmers are in charge of coordinating all communications events via explicit sends and receives. This can make message passing programming difficult when communication is complex. Synchronization and communication are unified in message passing. In the message passing model, each process or object uses its own local memory during computation; no shared memory is needed, therefore there is no danger of data races. Communication between processes is done by sending and receiving synchronous or asynchronous messages. Multiple tasks can reside on the same physical machine as well as on different machines which makes message passing scalable and more suitable for distributed systems. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation. Supporting both synchronous and asynchronous makes the message passing model more flexible since a shared memory programming model (i.e, a single address space) can be implemented using the message passing architecture; messages are used to keep data values consistent to all processes. Besides, featuring asynchronous message passing helps to reduce redundant waiting and to improve performance of concurrent programs.

As observed, both models have their advantages and disadvantages. Depending on the particular application domains, either of the models is used. One tendency is to combine both shared memory and message passing in general-purpose multi-processor systems.

2.2 Object orientation

Object orientation is one of the most popular current programming language paradigms. Centered around the notion of objects, they offer a combination of structuring and abstraction mechanisms which have proven useful and flexible in practice. Objects combine data, the object fields or instance variables, and code, its methods. Objects can be dynamically created at run-time and are allocated on the heap. In the mainstream of object-oriented languages, such as Simula [54], Smalltalk [80], C^{++} [183], and Java [82], they are created as instances of classes. Furthermore, the following structuring principles play an important role in object-oriented languages:

Abstraction: Abstraction in very general terms means reducing complexity when describing or understanding a phenomenon, for instance by ignoring irrelevant details. In computer science, abstraction is used to describe not just one concrete piece of data or code, but a "collection" of data or programs, by ignoring details of their implementation, or by grouping their commonalities together while keeping other aspects abstract to be instantiated later when specializing the abstraction, for instance by parameter passing. Thus, a class can be seen as an abstraction of objects, i.e., of all its instances abstracting away from the object identity and potentially the arguments of the constructors. Another important abstraction mechanism in programming languages is procedural or functional abstraction. In object-oriented languages, methods are procedural abstractions. Describing objects which combine data and code, classes can be seen as a form of abstract data types. In connection with inheritance, also a super-class can be seen as an abstraction of all its sub-classes since, when designed so that it contains fields and methods common to its sub-classes and inherited to them. In concurrent languages, the notion of threads or processes is an important abstraction which will be described later in Section 2.3.

When abstractly describing programs and their properties, the concept of *interface* is crucially important. Ignoring internal details and capturing only relevant information about a system, interface descriptions allow to reduce and factor out implementation details. Furthermore, interfaces are the key to a compositional description of a system. The notion of interface is rather general and depending on the programming language and properties to be captured, interface descriptions can be of different forms and of different levels of details. For example, in software engineering, a top-down development process could proceed by starting from very high-level, abstract descriptions in the requirement phase to low-level ones when implementing concrete algorithms. Protocols are typically designed in a layered manner where higher layers of the protocols are built upon the lower ones using only the services provided at the interfaces of the lower layers. In programming languages, a very common form of interfaces are types which can be seen as abstractions of the values a program can evaluate to. In object-oriented languages, such as Java, interfaces play the role of types of objects. To be precise, also classes in Java for instance do not only play the role of providing the implementation of its instances, but also the role of their interface or type.

Encapsulation and hiding: Encapsulation protects the integrity of a component by hiding its internal structure, and thus preventing different parts of the program from interfering with each other, for instance setting the internal data into an inconsistent state. Such protection is in particular essential in a concurrent setting. Encapsulation enables to ensure that data structures and operators are used as intended, and by limiting the inter-dependencies between software components, it helps to reduce system complexity, and thus increases robustness. Encapsulation plays also an essential role in object-oriented systems. It can be used to restrict access to the fields of an object preventing unwanted manipulations from outside. In languages like Java or C++, programmers can control the level of privacy or hiding via keywords like public, private, protected and so on. Other object-oriented languages disallow direct access to the object's internal data from the outside altogether so that it can be accessed only via their methods. In this thesis, addressing compositional analyses, we throughout assume fields to be instance-private, i.e., accessible only via methods, never directly from outside.

Inheritance and reuse: Inheritance, a very important concept in object-oriented programming languages, is a mechanism for code-reuse where a subclass inherits fields and methods of its base or super-classes. Typically it is also connected to the notion of sub-typing where an instance of a derived class can be safely used in a context where an instance of a super-class is expected. Supporting inheritance as a code re-use mechanism, for sub-typing, or both depends on a particular programming language [49]. There are mainly two forms of class inheritance:

- Single inheritance, originally proposed in Simula, means that each class has at most one super-class, i.e., the class hierarchy forms a tree. For example, C# and Java support single inheritance between classes. In those languages, however, one interface can have multiple super-interfaces and a class can implement multiple interfaces.

- Multiple inheritance in contrast means that a class can inherit from more than one super-class. Languages supporting multiple inheritance include C++, Common Lisp, Perl, and Python. Although multiple inheritance is more powerful and allows more flexible re-use of code from different classes, it increases complexity and may cause ambiguity in case of conflicting inherited features.

Despite of its accepted usefulness, inheritance can lead to problems. One is known as the fragile base class problem [141]. In particular, the combination of inheritance and concurrency is problematic and this phenomenon is sometimes called the inheritance anomaly [139, 122, 142]. In our work in Chapter 11, we consider single inheritance in a concurrent object-oriented language and for open systems.

2.3 Object orientation and concurrency

The wide-spread adoption of multi-core processors has led to an increasing demand of concurrent software that can exploit the performance benefit by utilizing more than just a single core. As the basic unit of concurrent programs is the process, a program with only one thread can run at most one processor at a time. On an n-processor system, a single-threaded program wastes $(n - 1)$ available CPU resources. On the other hand, programs with multiple active threads can run concurrently on multiple processors. When well-designed, multi-threaded programs can enhance throughput by utilizing available processor resources more effectively. In high-level programming languages, the concept of concurrency is captured by the notion of processes or threads which are units of concurrent execution. As discussed above, object-oriented languages offer different abstraction mechanisms to structure the code appropriately and ease the task of programming. How to combine the object-oriented abstractions with processes or threads can be done in different ways. The most important design decision is whether the objects as units of code and data coincide with the units of parallelism [21]. Keeping these two concepts separate leads to a design using multi-threading and synchronous method calls as in Java. The threads are running in parallel, sharing the fields of the objects which may be protected by the encapsulation and synchronization mechanisms of Java. The alternative is to make the unit of data and code at the same time the unit of concurrency. This model is also known as the *Actor model* where objects encapsulate both state and active behaviour into a single unit. Since objects thus have an activity of their own, communicating with other objects via asynchronous method calls or exchanging messages, they are called *active objects* and can be seen as processes in concurrent programs. Many object-oriented programming languages nowadays support concurrency. Java of course is one successful example of such languages based on multi-threading concurrency. The model based on active objects is represented by languages, such as Erlang [24], Scala [158], and Creol [119].

In conclusion, concurrent object-oriented programming is a successful way to specify and develop parallel programs offering abstraction mechanisms that reflect the way programmers design, analyze and reason about the problems. More concretely:

- the combination of parallelism and object orientation allows users to specify individual objects and their concurrent executions by means of threads locally or remotely on physically remote processing elements.
- communications between objects can be abstracted and defined by synchronous and asynchronous message passing concepts or method calls. Synchronizing concurrent tasks is done by synchronization mechanisms in each particular language.
- abstract data types, classes of data structures that have similar behavior or similar semantics, are used to separate design decisions from implementation issues. This allows

efficient solutions for the problems to be parameterized and reused since it abstracts away from placements of tasks onto processors.

CHAPTER 3

Inheritance and concurrency: A correctness perspective

In this section, we describe different problems caused by concurrency and also in connection with object orientation.

3.1 Synchronization and concurrency

Programming concurrency is known to be difficult and error-prone. In particular, it can lead to new kinds of errors which do not even exist in sequential programs. Typical examples are deadlocks and data races in shared variable concurrency. One can classify these errors depending on whether they violate safety or liveness properties.

3.1.1 Safety

Safety is a class of properties which states that "something bad never happens" [125]. Therefore, a safety violation is a situation where undesired or erroneous behaviors occur, often caused by incorrect inter-process synchronization. Race conditions are the simplest example of such errors. A data race happens when two threads access a shared piece of data simultaneously and at least one of them performs a write access. Without a proper synchronization, this may lead to unexpected and non-deterministic results, possibly leaving the memory in an inconsistent unsafe state.

Consider the following example:

Listing 3.1: Interference

```
x initial value = 1;

Thread 1: x++; print x; // x = 2;

Thread 2: x++; print x; // x = 2;
```

The simple code fragments show that there are two threads trying to increment the same variable's value simultaneously without synchronization. The problem is that the operation $x++$ is not atomic, i.e., with some unexpected timing, two threads could execute that code and terminate in a state where x has value 2 which is unwanted.

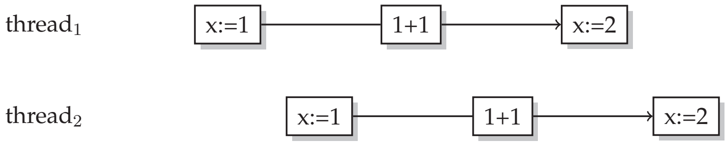


Figure 3.1: Unwanted execution of Example 3.1

Figure 3.1 shows how that happens. The non-atomic increment operation ++ includes three separate steps: reading the value from x the memory, adding 1 to x and then writing the new value back to the memory. As thread 1 and 2 can arbitrarily interleave, it is possible for them to read the value 1 at the same time, then both adding 1 to it. As a consequence the same value is written back by both threads. Erroneous calculations and race conditions look harmless in a program like this, but may nonetheless lead to serious failures in real programs (cf. the Therac-25 incidents mentioned earlier). Since they occur only under particular schedulings, they are very difficult to detect and reproduce in real applications. In practice, with modern multi-core architectures with different levels of caches, weak memory models, and with complex compiler optimizations, race conditions may occur unexpectedly even when using seemingly correct synchronizations, leading to unintuitive outcomes (cf. [164, 170])

Whereas data races are ultimately a consequence of a lack of synchronization, deadlocks [63, 47], another notorious class of concurrency-related errors, are caused by too much synchronization. A deadlock is a situation in which processes wait forever in a cycle for each other's resources without releasing their own resources. The dining philosophers problem is a classical illustration of deadlocks [61, 99]. A common source of deadlocks is the synchronization mechanism of locks, i.e., the resources involved in the deadlock are the locks. Since the lack of locks may lead to a data race and too many locks may lead to deadlocks, it is clear that finding a correctly synchronized program is challenging. Using too many locks also reduces the parallelism in a program, thus decreasing its performance. The traditional way of using locks in Java is via the synchronized command respectively using synchronized methods. This leads to a lexically-scoped discipline of locking and unlocking.

In version 5.0, Java introduced explicit locks to give programmers more freedom by allowing non-lexical use of locks. However, more freedom in programming means more opportunities for errors, and misuse of explicit locks in Java also causes safety problems. The code in Listing 3.2 describes an instance of the producer-consumer problem using explicit locks in Java where a safety problem occurs. In this example, the lock l is taken one time, but released two times. This causes an exception in Java when the method `put()` of Producer is called. The problem is that the programmer misused the lock l, i.e, releasing l without owning it because after the first `unlock` command, l is already free.

Listing 3.2: Unlocking error

```
...
private final ReentrantLock l;
class Producer implements Runnable {
  ...
  public void put(Integer key, Boolean value) {

    l.lock(); // 1 time lock
    try { collection.put(key, value);
        ...
```

```
          l.unlock();
      } finally {
                l.unlock();
            } // 2 times unlock
    ...
  }
```

Now if we change the code by replacing the *l*. unlock right before the statement finally by *l*. lock as in Listing 3.3, this leads to a starvation situation. The reason is that the producer needs to release *l* two times to free the lock, but it does only one *l*. unlock instead before it exits the method put. This means the consumer is starving while waiting for the lock *l*.

Listing 3.3: Hanging locks

```
  ...
  private final ReentrantLock l;
  class Producer implements Runnable {
    ...
    public void put(Integer key, Boolean value) {
      l.lock();                 // 1 time lock
      try { collection.put(key, value);
      ...
      l.lock();                 // 2 times lock
      } finally { l.unlock(); } // 1 time unlock
    ...
  }
```

3.1.2 Liveness

A liveness property says that something "good" eventually happens [125] and another kind of unwanted behavior caused by concurrency is liveness violation. Often a prerequisite to achieve liveness in a system is fairness. A scheduler is considered to be fair if it does not systematically neglect to schedule a process indefinitely even if the process is in principle able to proceed. Under fairness assumptions, each enabled process will be eventually chosen. Liveness is an abstract property concerning the progress of a system. It is often the case that a safety violation leads to a liveness problem. For example, a deadlock can potentially lead to a liveness problem because when a program gets into a stuck state, the deadlocked process(es) can not proceed. The starved consumer in Listing 3.3 is another example of a safety violation which results in a liveness problem: without releasing the lock *l* when terminating, the producer prevents the consumer from proceeding.

Safety and liveness are desired properties of any system, but it can be very tricky to achieve them both. Nonetheless a correct system must be safe and live at the same time, and sacrificing one for the other is not an option: a program that does not do anything is trivially safe, but useless and obviously not live. Sacrificing safety does not make sense either since one would accept erroneous programs. In general, adding synchronizations using locks is helpful in achieving safety by avoiding race conditions. On the other hand it may compromise liveness by introducing deadlocks, and furthermore it may affect performance negatively.

Above we have discussed some concurrency problems, classified into safety and liveness violations. A lot of effort is spent on analyzing the causes of those problems, and how to detect and prevent them. Sometimes, the combination of shared data and interference between concurrent processes can lead to data races, misuse of fine-grained locks can lead to deadlocks, or misuse of explicit re-entrant locks can lead to exceptions or starvation as illustrated in Listings 3.2 and 3.3. Moreover, there has been a lot of effort on improving programming languages to support concurrency better by providing suitable mechanisms and abstractions expressing synchronization and communication [30]. For example, a number of concurrency mechanisms are directly supported in concurrent object-oriented programming languages, such as monitors [98], locks and semaphores [63] or transactions [95] which are synchronization constructs to protect the access to the mutable shared resources within a critical section.

In this thesis, we investigate various safety problems for concurrent object-oriented programming languages, in particular we introduce different static analyses based on type and effect systems to prevent certain concurrency errors [134, 118, 136, 194].

3.2 Problems caused by inheritance

A crucial feature in class-based object-oriented programming languages is *class-inheritance*, which allows code reuse and is intended to support incremental program development by gradually extending and specializing an existing class hierarchy. A well-known problem caused by inheritance and late binding, especially in open systems is the fragile base class problem [141, 179, 177, 169]. A base class in an inheritance hierarchy is a (common) super-class, and fragile means that replacing one base class by another, seemingly satisfying the same interface description, may break the code of the client of the base class, i.e., change the behavior of the "environment" of the base class. Consider the following code fragment.

Listing 3.4: Fragile base class

```
class A  {                         class B extends A {
  void add () {...}                  void add () {
  void add2 () {...}                   size = size + 1;
  ...                                  super.add(); }
}                                     void add2 () {
                                        size = size + 2;
                                        super.add2();}
```

The two methods add and $add2$ are intended to add one, respectively two, elements to some container data structure. This completely (albeit informally) describes the intended behavior of A's methods. Class B in addition keeps information about the size of the container. Due to late-binding, this implementation of B is wrong if the $add2$-method of the super-class A is implemented via *self*-calls using two times the add-method, i.e., the statement $super.add2$ will call two times the method add of the class B instead of A which makes the value of $size$ added 4 instead of 2 as intended. With a behavioral interface specification of the methods using standard pre- and post-conditions, the problem is that *nothing* in the interface helps to avoid the problem: the information about the state of the program before and after execution is not detailed enough, for instance in this case, to reflect the effect of the combination of inheritance and late-binding on behaviors of derived classes. The interface specification is too weak to allow to consider the base class as a black box which can be safely substituted based on its interface specification only. Therefore, in the presence of inheritance, a stronger rigorous account of such an interface behavior is needed, in particular it has to reflect the information whether a method is implemented via *self*-calls or not. A formal and precise interface description is as usual the key to formal verification of open programs as well as a formal foundation for black-box testing.

On the one hand, interface specifications should contain enough information to reason about the behaviors of a program, for instance, they need contain intermediate steps in a concurrent setting, or as in the example of the fragile base class problem information about self-calls is relevant. On the other hand, for the sake of compositionality, the specification should not depend on internal details of an implementation; it should reflect only information which can be seen at the interface of objects or components and the interactions between them. This property is known as observability. For clarification, let us consider a very simple example in Java (cf. Listing 3.5), to get an intuition of the observational set-up. The example shows two classes, one is called P (for program, sometimes also called component) and one is called O (for observer). The example assumes that the program P creates the observer. The observer observes that the method m is being called, and then prints success. Of course, it could additionally observe that the number 42 is passed as argument using an appropriate conditional expression.

Listing 3.5: Program and observer

```
public class P {  // program/component
    public static void main(String[] arg) {
        O x = new O();
        x.m(42);  // call to the instance of O
    }
}

class O {       // external  observer/environment
    public void  m(int x) {
      ...
      System.out.println("success");
    }
}
```

So in this case whether the method m is called or not is observable. One can say that a particular information is observable if it leads to a difference in behavior in its environment. Now let us come back to the fragile base class example in Listing 3.4. From an observational point of view, the only thing that counts is the interaction with the environment (or observer) and whether this interaction leads to observable reactions in the environment. If done properly, it ultimately allows compositional reasoning, i.e., to infer properties of a composed system from the interface properties of its sub-constituents without referring to further internal representation details. A representation-independent, abstract account of the behavior is also necessary for compositional optimization of components: only when showing the same external behavior one program can replace another without changing the interaction with any client code. We investigate formally observability in the presence of inheritance in Chapter 11 (cf. also Section 5.4).

Furthermore, the coexistence of concurrency and inheritance causes additional problems as studied in [20, 121]. One of those problems is known as the inheritance anomaly [139, 122]. It basically says that the two concepts of inheritance and concurrency do not work well together. In other words, a program suffers from *the inheritance anomaly* when using inheritance as a useful mechanism for code-reuse breaks due to the presence of concurrency and corresponding control mechanisms. This code re-writing process can be so painful that the advantage of code-reuse of inheritance is no longer practical. Let us take a look at the following Java-based pseudo-code implementation of the classic bounded buffer example, where the inheritance anomaly occurs [142]

Listing 3.6: BoundedBuffer

```
public class BoundedBuffer{

    ...
    synchronized public Object get() throws Exception {
        while ("buffer is empty"){ wait(); }
        ...
        notifyAll();
        return ...;
    }
    synchronized public void put(Object c) throws Exception {
        while ("buffer is full") { wait(); }
        ...
        notifyAll();
    }

 }
```

The bounded buffer provides the methods `get` and `put` to respectively remove and insert an element. In a concurrent setting, to prevent clients from removing an element from an empty buffer or putting an element into a full buffer, the methods are guarded by the conditions:

`while ("buffer is empty"){ wait();}`

and `while ("buffer is full"){ wait();}` respectively.

Moreover, to guarantee mutual exclusion ensuring that all clients have a consistent view of the buffer, those methods are marked as `synchronized`. Mixing behavioural and synchronization code in class `BoundedBuffer` causes complications, when one wants to reuse the implementation of the bounded buffer in sub-classes, for instance by deriving a subclass, say `HistoryBuffer`, that behaves like its super-class, except that it has an additional method `gget`. The method `gget` behaves as `get` but it can not be called immediately after `get`, which is represented by an additional variable `afterGet` initially set to true. In order to do that, the method `gget` needs to keep track of whether or not the last method to be executed was a `get` or not. As a consequence, we have to redefine all inherited methods (i.e., `get` and `put`) as illustrated in Listing 3.7 which leads to unneccessarily complex code.

Listing 3.7: The class HistoryBuffer

```
public class HistoryBuffer extends BoundedBuffer{

    boolean afterGet = false;
    ...
    synchronized public Object gget() throws Exception {
        while ("buffer is empty" || afterGet){ wait(); }
        afterGet = false;
        ...
    }
    synchronized public Object get() throws Exception {
        Object o = super.get();
        afterGet = true;
        return o;
    }
    synchronized public void put(Object o) throws Exception {
        super.put(o);
        afterGet = false;
    }

}
```

However, the specific difficulties to combine inheritance with late binding or concurrency depend on the particularities of the language. In this thesis, we just concentrate on the problem of inheritance and late binding with concurrency without further investigations on solving the anomaly problem.

3.3 Analyses for concurrent object-oriented languages

In the previous section we have discussed some concurrency problems, in connection with object-oriented programming languages. Next, we present some techniques which have been developed to deal with concurrency-related errors.

It is widely agreed that concurrent programs are inherently much more complex and difficult to understand than sequential ones. There are a number of reasons for that: first of all, arbitrary interleavings of concurrent threads or processes introduce non-determinism. As a consequence for testing, different executions of the same test can produce different results. Thus, it is hard to reproduce and find bugs, because a particular bug may only exhibit itself in one or a few of overwhelmingly many interleavings of a program. Secondly, different interactions and schedulings lead to an enormous size of the state space, often exponentiali n the number of processes, a phenomenon known as the state space explosion problem. This makes a full exploration of the state space unfeasible in practice. Furthermore, the flexibility and expressivity of modern object-oriented languages featuring pointers or references with aliasing, shared variables, callbacks, dynamic object and lock creation, all in the presence of concurrency make analyses challenging. In particular, the interactions between language features, e.g., between synchronization mechanisms and concurrency, or between inheritance and concurrency causes many problems as discussed previously which are hard to detect.

There is a variety of different analysis techniques to detect concurrency bugs. In general, one distinguishes dynamic and static techniques. The dynamic ones analyze the running program while the static ones analyze the code at compile-time. Both techniques have complementary advantages and drawbacks. Responsible software engineering often relies on both to assure quality of software.

An advantage of dynamic approaches including testing and run-time monitoring, is that working with running programs, they are immune from spurious results, i.e., if an error is detected, it is an actual error in the programs. The drawbacks of dynamic techniques typically result from the large state space to cover. Sources of the large state space are, as mentioned, different interleavings in concurrent executions, but also the fact that the input of programs or routines can be drawn from a large or even infinite domain. In general, an exhaustive and precise exploration of all behavior is impossible, which means testing can in practice almost never assure correctness of programs, it can only dectect errors. The challenge, therefore, is to obtain an adequate coverage of behavior thus increasing the chance of spotting errors. The size of modern software requires in any case very many test cases, therefore testing in general will use adequate tools and techniques to automatically generate enough test cases. Since as said, testing can nonetheless not fully guarantee correctness, the remaining bugs are often found late or by customers which may be very expensive. Testing and dynamic analyses are not in the scope of this thesis, for further material about testing software see for instance [149].

An alternative to dynamic techniques are static analyses where compile-time information is used to analyze properties of (concurrent) programs. To guarantee absence of errors of a program, a static analysis should not overlook cases of incorrect behavior, i.e., it needs to cover all potential error cases. Aiming at an automatic analysis and since in general, it is undecidable, whether or not a concrete error occurs at run-time, static analyses always use over-approximation. This may lead to spurious results, such as false positives reporting errors which do not correspond to actual behavior. Too many false errors may render an analysis useless in practice. Therefore accuracy is an important factor, i.e., the rate of false alarms and spurious results should be minimal. Other important factors are scalability, i.e., the complexity of the analysis should grow in a managable manner in the size of the program being analyzed, and related to that efficiency. Designed to capture a specific class of errors, for instance, absence of null pointer exceptions, uninitialized variables and many others, of course a static analysis can not guarantee the overall correctness of a program and absence of all errors in general, just the absence of the errors it was designed to capture.

Many different approaches and tools have been investigated and developed to analyze programs at compile-time including model checking [45, 26], and abstract interpretation [51]. Many successful and efficient static analysis methods are based on type and effect systems [154, 22, 155, 153]. A type is a static abstraction of set of values, for instance, the type of booleans abstracts away from concrete values true and false, and an interface in object-oriented languages describes all objects supporting the interface's methods. The corresponding type system prevents illegal applications of operations to values, for instance, prevents errors caused by attempting to calculate true plus false, or invoking a method on an object it does not support.

Whereas traditional types are abstractions of program *values*, effects are abstractions of its *behavior*. In static analysis based on type and effect systems, type judgements are augmented with behavioural information to capture essential static information about a program's potential dynamic behaviour. This is in particular useful for analyzing concurrent programs [22] where one is interested primarily in errors caused by incorrect behavioral interactions. For concurrent programs, numerous different analyses based on type and effect systems have been developed, for instance for deadlocks freedom [34, 184, 12], data-race freedom [36], and liveness [151]. Typically, type and effect systems are defined as syntax-directed, i.e., compositional over the structure of programming languages, and consequently efficient and scalable. Another advantage is that they are often defined formally assuring the absence of errors in a rigorous manner. Examples of static analysis tools for concurrency bugs which have proved themselves successful in practice are JLint [114], Chord [111], Java PathFinder [91], Coverity [31], and FindBugs [71]. A detailed comparison between different static analysis tools wrt. the effectiveness of detecting actual concurrency bugs and the rate of false positives can be found in [40, 123].

The analyses in this thesis are based on static type and effect systems. For more detailed information about the design of the systems developed in this work, see the next chapter.

CHAPTER 4

Compositional analyses for concurrent object-oriented languages

In this section, we start by presenting an overview of the formalization of the syntax and semantics of the calculi most of our work is based on (cf. Section 4.1 and 4.2). In Section 4.3, we come back to the concept of compositionality and discuss compositional analyses, in particular for type and effect systems and for problems related to concurrency. In Section 4.4, we describe type and effect systems as a well-known approach to formal analysis, in particular in the form needed in our analyses.

4.1 An object-oriented core calculus

Our formal techniques for different problems in concurrent object-oriented models are based on (variations of) a core calculus. Having a precise, succinct syntax and well-defined semantics allows us to focus on the nature of the problems and give a precise solution. By choosing a core language judiciously, we can leave out unnecessary details to focus attention on issues of interest which simplifies our type and effect systems without losing generality. Such an approach, to concentrate on a core calculus and introduce additional features as syntactic sugar, was proposed by Landin in the context of the λ-calculus [126]. In this way, adding new features into the calculus, and analyses for them, are orthogonal to existing ones.

Java is a concurrent, object-oriented programming language which is widely used in practice and well-studied in research. The core calculus in this thesis is inspired by Featherweight Java (FJ) [105]. FJ is an object-oriented core language originally introduced to study typing issues related to Java, such as subtype polymorphism and inheritance, type casts, etc. A number of extensions have been developed for other language features, so FJ is today a generic name for Java-related core calculi. Following that direction, when investigating different aspects of concurrent object-oriented models, we base our calculi on FJ. In contrast to the original FJ proposal, however, we largely ignore sub-typing and type casts, as orthogonal to the issues at hand, but include imperative features such as destructive field updates, furthermore concurrency and concurrency control mechanisms. In particular, we investigate pessimistic concurrency control based on re-entrant locks as well as optimistic ones using transactions. In the latter case, we use a variant of FJ known from the literature as Transactional Featherweight Java (TFJ) [110].

We use three variations of FJ in the thesis; their syntax is shown in Table 4.1. Despite slightly different syntax representation, all the calculi we consider share the following features:

- class-based object-orientation,
- thread-local variables with single assignment,
- instance-local mutable fields, and
- inter-object communication via method calls.

For the first two calculi, statically a program consists of a set of class definitions and one initial thread/process. The syntax of the third calculus is inspired less by FJ, but more by the various object calculi and ultimately the π-calculus. The representation of classes and objects is therefore syntactically slightly different. In particular the calculus contains a ν-binding construct to express dynamic scoping of names.

The most significant differences in features as well as in the underlying models between the three calculi are summarized in Table 4.2. We categorize the concurrency-related differences into the syntactical differences, i.e., which constructs are used for concurrency control; semantics-related differences of how to achieve concurrency control and finally differences in how to introduce concurrency into the language.

As said above, our goal is to study various aspects of concurrency. Calculus 1 is based on *transactions*, an alternative concurrency mechanism to conventional lock-based mechanisms. There we follow [110] by extending FJ with multi-threaded and nested transactions. Syntactically, two constructs `onacid` and `commit` are added to the calculus for opening and committing a transaction. Semantically, to allow an optimistic form of concurrency control, a log is used to record changes inside one transaction. The log is a local view of the transaction on the global memory, and each transaction will work on its own local memory without interference from other transactions. Conflicts will be checked at commit time when the transactions make updates to the main memory. All considered calculi allow dynamic creation of concurrent activities. In calculus 1, new threads can be created using the command `spawn`. In Java, this corresponds to instantiating a thread class or to instantiate an object implementing the `Runnable` interface and calling the `run` method afterwards. In calculus 2, when dealing with pessimistic concurrency control based on *locks* [117, 118], we syntactically extended FJ by supporting four lock operators, `new L` for dynamically creating a new lock, `lock` and `unlock` for acquiring and releasing a lock, which corresponds to the synchronization with *explicit* locks as in Java 5.0. Additionally, `trylock` is a non-blocking variant of lock acquisition using conditional statetments: if the corresponding lock is available, it is taken, otherwise the invocation returns false. Moreover the locks are re-entrant as in Java which means that a thread can take a lock more than one time and correspondingly release the lock the corresponding number of times it has been taken. As far as thread creation is concerned, calculus 1 and 2 are identical. Unlike the first two calculi, calculus 3 does not use explicit lock handling nor transaction handling, i.e., in the calculus there is no user-syntax corresponding to `onacid` and `commit`, respectively `lock` and `unlock`, to achieve mutual exclusion. Mutual exclusion is achieved using locks implicitly, but in contrast to the other two calculi method bodies are executed under mutual exclusion by default, sometimes known as *automatic mutual exclusion* [107, 1]. The user can, however, release the lock temporarily using `suspend`(n), also known as *yield* in other languages. An important feature of calculus 3 is that it uses a different communication model, based on asynchronous

Calculus 1 ([134, 138]): multi-threaded and nested transactions; non-lexical scope; synchronous method calls.			
P	$::=$	$\mathbf{0} \mid P \parallel P \mid p\langle e\rangle$	processes/threads (process id p)
L	$::=$	$\mathsf{class}\ C\{\vec{f}:\vec{T}; K; \vec{M}\}$	class definitions
K	$::=$	$C(\vec{f}:\vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\}$	constructors
M	$::=$	$m(\vec{x}{:}\vec{T})\{e\} : T$	methods
e	$::=$	$v \mid x \mid v.f \mid v.f := v$	expressions
	$\mid$	$\texttt{let}\ x : T = e\ \texttt{in}\ e$	
	$\mid$	$v.m(\vec{v})$	synchronous calls
	$\mid$	$\texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{new}\ C(\vec{v}) \mid \texttt{spawn}\ e$	
	$\mid$	$\texttt{onacid} \mid \texttt{commit}$	transaction constructs
v	$::=$	$b \mid r \mid ()$	values (reference r, basic value b)

Calculus 2 ([117, 118]): re-entrant, explicit locks; exceptions; non-lexical scope; synchronous method calls			
P	$::=$	$\mathbf{0} \mid P \parallel P \mid p\langle t\rangle$	processes/threads (process id p)
D	$::=$	$\mathsf{class}\ C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T}; \vec{M}\}$	class definitions
M	$::=$	$m(\vec{x}{:}\vec{T})\{t\} : T$	methods
t	$::=$	$\texttt{stop} \mid \texttt{error} \mid v$	
	$\mid$	$\texttt{let}\ x{:}T = e\ \texttt{in}\ t \mid \texttt{error}(E)$	threads
e	$::=$	$x \mid t \mid v.f := v \mid v.f$	expressions
	$\mid$	$v.m(\vec{v})$	synchronous calls
	$\mid$	$\texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{new}\ C(\vec{v}) \mid \texttt{spawn}\ t$	
	$\mid$	$\texttt{new L} \mid v.\ \texttt{lock} \mid v.\ \texttt{unlock}$	lock expressions
	$\mid$	$\texttt{if}\ v.\ \texttt{trylock then}\ e\ \texttt{else}\ e$	(L = type/class of locks)
	$\mid$	$\texttt{throw}\ E \mid \texttt{try}\ e\ cb\ \texttt{finally}\ e$	exceptions E
cb	$::=$	$\epsilon \mid \texttt{catch}\ E > e;\ cb$	
v	$::=$	$b \mid r \mid ()$	values

Calculus 3 ([11]): Inheritance and sub-typing; active objects; asynchronous method calls			
C	$::=$	$\mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[\![O]\!]$	component
	$\mid$	$\underline{n[O, \texttt{lock}]} \mid \underline{n\langle t\rangle}$	
O	$::=$	n, M, F	object
M	$::=$	$l = m, \ldots, l = m$	methods
F	$::=$	$l = f, \ldots, l = f$	fields
m	$::=$	$\varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
f	$::=$	$v \mid \bot_n$	field
t	$::=$	$v \mid \texttt{stop} \mid \texttt{let}\ x{:}T = e\ \texttt{in}\ t$	thread
e	$::=$	$x \mid t \mid v.l() \mid v.l() := v$	expressions
	$\mid$	$n@l(\vec{v})$	asynchronous calls
	$\mid$	$\texttt{if}\ v = v\ \texttt{then}\ e\ \texttt{else}\ e$	
	$\mid$	$\texttt{if}\ \mathit{undef}(v.l())\ \texttt{then}\ e\ \texttt{else}\ e$	
	$\mid$	$\texttt{new}\ n \mid \texttt{claim}\ @(n, n) \mid \texttt{suspend}(n)$	
	$\mid$	$\underline{\texttt{get}\ @n} \mid \underline{\texttt{grab}\ (n)} \mid \underline{\texttt{release}\ (n)}$	run-time expressions
v	$::=$	$n \mid ()$	values (n is an object/thread name)

Table 4.1: The core calculi

method calls where there is no blocking at a sender's side when making a call to a receiver. Instead of blocking when calling a method, the caller asynchronously continues executing while the body of the called method is being executed by a new thread. The identity of the new thread also represents a reference to the result of the method's execution after termination.

	Synchronization	Mutex mechanism	Dynamic thread creation
Calculus 1	`onacid` | `commit`	transactions	spawn
Calculus 2	`new L` | `lock` | `unlock`	re-entrant locks	spawn
Calculus 3	`claim` $@(n,n)$ | `suspend`(n)	binary locks	asynchronous calls

Table 4.2: Comparison of syntax of the calculi

Thus the thread identifier also serves as a so-called future reference where the caller may get the method's result back on request as soon as it is available by using the claim operation. A thread claiming a future not only tries to get the result back, but also releases its lock in case the result is not yet there. In this model, each asynchronous method call creates a new thread whose identity at the same time represents the future to obtain the result eventually. Unlike the setting with Java style multi-threading, no activity ever ”leaves” an object because each activity in a different object is carried out by a thread which is created by the asynchronous method calls and thus different from the original thread. The object's boundary encapsulating its states is at the same time also the boundary for the units of concurrency, i.e., the method bodies under execution. Due to mutual exclusion, at most one method body is active at any given time. Another consequence of this model is that any call-back will be executed with a different thread, therefore there is no need for re-entrant locks and active objects achieve mutual exclusion using simple binary locks. This loosely coupled model based on active objects is advantageous when aiming at a compositional description. We are in particular interested in treating a set of classes as units of composition, therefore a compositional description profits from the fact that the units of concurrency are “encapsulated” inside the active objects which are in turn described by classes. We will use this model of loosely coupled objects to achieve compositionality when investigating a behavioural specification of concurrent programs with *inheritance* in an open setting.

4.2 Operational semantics

Semantics is important for all programming languages because it defines the meaning of grammatically correct programs written in a specific programming language. Operational semantics describes how a program construct is executed by means of a transition system containing a set of rules to express relations between program configurations. To allow rigorous arguments and tools which give provable guarantees, one needs a precise semantics. Formal semantics is the field concerned with the mathematically-based study of rigorously specifying the meaning, or behavior, of programs and programming languages. It is rigorous and thus can reveal ambiguities and subtle complexities based on mathematical reasoning. It provides an important tool to reason about many aspects of the behavior of programs and programming languages, ranging from implementation, analysis to verification. It allows language designers to specify rigorously what a language is supposed to do in the same way that software engineers are required to specify in a rigorous manner what the software artifact is supposed to do.

There are many approaches to formal semantics, but here we just name three of them: denotational semantics, operational semantics, and axiomatic semantics. Denotational semantics

[172] formalizes the meanings of programming languages by defining programming language constructs as mathematical objects as, say, functions that map input to output (e.g., partial functions). Thus only the effect is of interest, not how it is obtained. Axiomatic semantics [173] defines the meaning of a program by expressing specific properties of the effect of executing the constructs as assertions about the program state. The assertions are logical statements — predicates over variables, where the variables define the state of the program. This kind of semantics, using logical axioms and rules as examplified in Hoare logic in the form of pre- and post-predicates of statements. An important tenet going through this semantics again is compositionality, i.e., the semantics of a program should be built out of the semantics of its constituents. For example, the meaning of the expression $f(e_1, e_2)$ is defined in terms of the semantics of its sub-components f, e_1 and e_2. This is, however, difficult to achieve, especially in modern languages where the combination of concurrency and shared variables may cause interdependences between sub-components of an expression or their executions can effect each other. For example, assuming that $f(e_1, e_2)$ is parallel composition in a concurrent program, i.e., $f(e_1, e_2) = e_1 \parallel e_2$, evaluating e_1 and e_2 can be done concurrently; however due to interleaving between them, the execution of one of them might affect the other because of shared data causing their semantics to be defined in terms of each other. Or one execution can be terminated by exceptions caused by the other. Moreover, in many cases to obtain compositionality in the semantics of concurrency, one has to take many factors into account apart from variables defining the state of the program, and hence, the semantical formulas become very complex. One different form of semantics is operational semantics, which interprets a valid program as sequences of computational steps. These sequences then are the meaning of the program. Operational semantics of a program can be seen as an abstract machine or transition system where expressions are given meaning by the transitions they induce on states of the machine. In particular, operational semantics captures how a program is executed. The concept of operational semantics was for the first time introduced in [189].

One can distinguish two types of operational semantics: natural semantics, or big-step semantics, which describe how the overall results of one execution is obtained and small-step semantics, which formally describe how the individual steps of a computation take place in a computer-based system. Structural operational semantics was introduced by Gordon Plotkin in the form of a small-step semantics [161] as a logical means to define operational semantics. A small-step semantics defines the behavior of a program in terms of a set of transition relations, i.e., it takes the form of a set of inference rules which define the valid transitions of a composite piece of syntax in terms of the transitions of its components.

For the concurrent setting, a small-step semantics is more suitable because it reflects changes of every single step during execution, therefore taking interference into account. So we have employed small-step structural operational semantics to specify the meaning of our calculi. There are several reasons for that. First of all it is easier for concurrency than a denotational semantics because one just has to define the meaning of every single step of a program. Second, for the sake of compositionality, it allows us to separate local steps (per thread) and global steps (involving more than one) when describing the semantics of the calculi in our papers. Finally, when proving soundness of the type and effect systems, one can employ a well-founded, standard inductive proof technique, namely subject reduction.

For the sake of cleanness and compositionality, in this thesis we always present our semantics and analyses at two levels: local and global. This clean manner helps to distinguish between local variables and shared variables. To have an overview of our semantics rules, we sketch our transition derivations as follows: let σ represent a state of a program, normally σ is of the form of a mapping from program references and variables to their values. In our setting, we strictly separate thread-local variables and references located on the heap. The heap is a mapping from references to object states (and locks when part of the language). Since local variables are handled in a functional manner, where each variable is bound to a value only ones ("single assigment") via the let-construct, they are dealt with by a substitution-based rewriting relation and thus not present in σ. In this thesis, we represent a configuration of a given program e as a pair of its state σ and the program, we often write $\sigma \vdash e$. So the operational semantics in this thesis is described as a transition system containing a set of transitions between configurations of programs and presented at two levels:

Local semantics This level deals with the evaluation of *one* single *expression* and reduces configurations of the form $\sigma \vdash e$. Thus, local transitions are of the form

$$\sigma \vdash e \rightarrow \sigma' \vdash e' \,, \tag{4.1}$$

where e is one expression and σ is the heap. Note that for programs written in our lock-based calculi, σ represents the mutable states of the program and is shared between all threads. It is a finite mapping from references to objects or locks although local semantics deals only with expressions in a local thread. However in our transaction-based calculi, σ refers to a *local environment* of each thread, written E instead of σ in semantical rules, recording changes to only the local variables of the thread and is *not shared* between threads. At the local level, the commands only concern the current thread and consist of reading, writing, invoking a method, and creating new objects.

Global semantics This level involves the behaviors of more than one thread in the global context which we formalize as *global* steps, i.e., steps which concern more than one thread. A program under execution contains one or more processes running in parallel and each process is responsible for executing one thread. A global configuration consists of the shared heap σ and a program P containing a "set" of processes and is of the form:

$$\sigma \vdash P \,, \tag{4.2}$$

where σ contains the "passive" data part of the program whereas P contains the "active" part and is given by the following grammar:

$$P \quad ::= \quad \mathbf{0} \quad | \quad P \parallel P \quad | \; p\langle t \rangle \qquad \text{processes/named threads} \tag{4.3}$$

$\mathbf{0}$ represents the empty process, $P_1 \parallel P_2$ the parallel composition of P_1 and P_2, and $p\langle t \rangle$ a process (or named thread), where p is the process identity and t the thread being executed. A thread t in our calculi is of the form: $\texttt{let}\ x : T = e\ \texttt{in}\ t$, which is a representation of a sequential composition. Another reason for our choice of using separate syntax for expressions e and threads t is that we can simplify our operational semantics by removing the evaluation context. The binary $\parallel$-operator is associative and commutative with $\mathbf{0}$ as neutral element. Furthermore, thread identities are *unique*. That way, P can also be viewed as a finite mapping from thread names to threads. With global configurations as given in equation 4.2, global transitions are of the form:

$$\sigma \vdash P \rightarrow \sigma' \vdash P', \text{ or } \sigma \vdash P \rightarrow \mathit{error} \tag{4.4}$$

where P and P', σ and σ' are the *programs* respectively the shared heaps before and after the step. Basically, a program P consists of a number of threads running in parallel, where each thread corresponds to one expression, whose evaluation is described by the local rules. In the lock-based setting, σ is the same in both local and global level as it is shared between threads. However, when formalizing transactions we use one local environment E for each thread name p because the tenet of transactional models is that there is no shared data between threads, each thread works on their own local memory. This means, σ or σ' in the transactional setting is a set of $p{:}E$-bindings where p is the name of a thread and E is its corresponding local environment and $\sigma \vdash P \rightarrow \mathit{error}$ is a transition where error is a special configuration representing an error state.

4.3 Compositionality

A system description is compositional if a property of the system is established based on the properties of its sub-systems alone. Of course, to have a compositional property is a precondition to get a compositional verification method, which means the composed property or correctness of the whole system follows the verified properties of the sub-components. To analyze and verify large and complex systems, a compositional approach is crucial. This allows a *divide-and-conquer* approach and thus is the key to scalability. Ideally, the analysis needs to be based on a rigid mathematical foundation, thus giving a high confidence in the analysis' results. Compositional analyses consider parts of a larger program as black boxes and rely only on their specification respectively interface information to verify the larger program. Users of components just need to know its observable behavior at the interface without further information of their internal implementations or machine-dependent details. Compositional specification techniques using assertional predicates over only their observable behavior have a number of advantages [58]. In particular for concurrent and distributed systems, compositional reasoning techniques have been developed, known as assumption-commitment [120] and rely-guarantee [146] paradigm.

In a simple setting, the specification of a component can contain information about interfaces or services and the exchanged values to guarantee the compatibility of the component as in component-based software or web-services. But the interface specification could also take a more complex form, such as assertions about *behavioral* properties of the component as in software verification. Assume that we have two components C_1 and C_2 such that φ_1 and φ_2 are two predicates or assertions about their behavior. We write $C_1 :: \varphi_1$ and $C_2 :: \varphi_2$ to express C_1 and C_2 satisfy properties φ_1 and φ_2 respectively. A compositional rule to reason about the composition of C_1 and C_2 can be specified as follows:

$$\frac{\vdash C_1 :: \varphi_1 \qquad \vdash C_2 :: \varphi_2 \qquad \varphi = f_{\otimes}(\varphi_1, \varphi_2)}{\vdash C_1 \otimes C_2 :: \varphi} \tag{4.5}$$

where $\otimes$ is a syntactic composition operator to combine C_1 and C_2. It can be interpreted as any binary syntax construct of the given programming language. It basically says that the composed system $C_1 \otimes C_2$ satisfies the property which is asserted by $f_{\otimes}(\varphi_1, \varphi_2)$.

As a prominent example in software verification, compositionality is also characteristic for Hoare logic [97]. Let us take a look at a Hoare rule for the sequential composition $C_1; C_2$ of programs C_1 and C_2:

$$\frac{\{\vdash P\}\, C_1\, \{Q\} \qquad \vdash \{Q\}\, C_2\, \{R\}}{\vdash \{P\}\, C_1; C_2\, \{R\}},$$

where P, Q, and R are assertions about the states, namely about the states before C_1, between C_1 and C_2, and after C_2, respectively. For partial correctness, the triple $\{P\}\ C_1\ \{Q\}$ says that if C_1 starts in a state satisfying the pre-condition P and it terminates, then the final state of C_1 will satisfy the post-condition Q.

Hoare logic works well for sequential programs, but can be applied to concurrent ones, as well. The above rule, for instance, is unsound for shared variable concurrency if used naively, since the state may change between C_1 and C_2 due to interference from other processes not mentioned in the rule. To get a sound version of such a rule in the presence of concurrency, the pre- and post-conditions must be able to capture the possibility of other processes interfering. Consequently, the program states must contain enough information about other processes, in particular at which program points they are. However, pre- and post-conditions formulated adequately based on this additional information can be very complex. However, this should not be considered as a defect of Hoare logic, it just reflects that without appropriate concurrency control mechanisms shared variable concurrency results in complex behavior. Some of the type and effect systems in this thesis can be seen as being based on a form of Hoare style pre- and post-specifications. In particular, our effects will specify the state of shared locks and certain aspects of the state of transactions respectively. In this particular setting, the above rule of sequential composition *is* sound: in the case of transactions, because they are executed in isolation, and in the case of locks, because they are protected against interference. This freedom of interference does not apply to all investigated problems and models of concurrency in the thesis.

As just seen, a compositional specification of program behavior in the presence of concurrency is challenging. Another aspect which is difficult to capture in a compositional manner and which is relevant to (not only) object-oriented languages, are pointers or references and aliases. In a language without aliasing, the following simple "rule of constancy" is important for modular reasoning:

$$\frac{\{P\}\ C\ \{Q\}}{\{P \wedge R\}\ C\ \{Q \wedge R\}}\ \mathit{mod}(C) \cap \mathit{fv}(R) = \emptyset.$$

It expresses the obvious fact that a property R is invariant when executing C if C does not modify variables mentioned in R. Unfortunately, this rule is *unsound* in the presence of aliasing, since program C may change an alias of a variable mentioned in R. That this rule breaks in the presence of aliasing shows that modular reasoning is significantly more complex with references. To reason about the heap and aliasing, Peter O'Hearn and John C. Reynolds invented *separation logic* [167, 159], where the rule of constancy is replaced by the following rule, known as the frame rule of separation logic:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \mathit{mod}(C) \cap \mathit{fv}(R) = \emptyset.$$

The crucial modification is that the logical conjunction of the rule of constancy is replaced by a more complex operator, called separating conjunction, which expresses that P and Q conjunctively hold, but for different, separated parts of the heap. The introduction of this new operator enables local reasoning in the presence of aliasing, therefore provides more compact specifications and proofs, where specifications and proofs concentrate on the portion of memory heap used by a program component, and not the entire global state of the system. Parts of this thesis address references and aliasing, in particular related to the dynamic lock creation.

However, in our compositional analysis, we do not need to assume that different lock variables refer to separate parts of the heap, and which therefore may be aliases.

To sum up, compositional reasoning analysis is not easy to achieve in a concurrent setting. There are many cases where the formulation of the problem at hand, the inherent complexity of the semantics of a program construct such as implicit synchronization, or the tight coupling of the processes concerned, prevents a practical compositional solution. This applies especially to concurrent object-oriented programming languages. For parallel composition, achieving compositional analyses becomes difficult as specifying properties of a systems can be much more complicated due to exponential interaction between concurrent components and interleaving semantics. Moreover, specifying assertional formulas is often very complex because they need to capture different information in the interface, depending on the semantics of parallel composition and used synchronization mechanisms. Besides that also the kind of properties being analyzed and abstractions used influence the formulation of the effect part.

In this thesis, we investigate various compositional, static analyses in particular wrt. parallel composition. We treat compositionality in different concurrency models: lock-based or transaction-based, synchronous or asynchronous message passing in closed systems as well as open systems whose environment is not known in advance.

4.4 Type and effect systems

4.4.1 Types and effects

Intuitionistic or constructive logic is different from classical logic in its definition of the meaning of a statement being true. In classical logic, all well-formed statements are either true or false, even if we do not have a proof of either. In constructive logic, a statement is true if and only if there is a constructive proof that it is true, and is wrong only if there exists a proof that it is wrong. Constructive type theory internalizes that idea of intuitionistic logic, i.e., there exists an isomorphism between propositions and types, usually called the Curry-Howard isomorphism: a proposition is identified with the type of its proofs. There is an identification between the following two predicates: the proposition A has a proof M in constructive logic and a program M has a type A in programming languages. So type theory is a constructive theory which can be automatized and supported by software tools.

Types are used in programming languages to restrict the underlying syntax so that only meaningful expressions are allowed. This allows many programming errors to be detected by the compilers. It guarantees that if a program has passed a static type system of the compiler, then the type of that program needs no run-time checks since it is assured to be of correct type. In other words, well-typed programs can not go wrong. Typically, that is proven in a form of induction and breaks down into "well-typed programs don't exhibit an error now, resp. in the next step" and "a reduction step preserves well-typedness" (the latter also known as subject reduction). Type systems are sets of type derivation rules. Compilers use type systems to prevent unexpected operations due to misuse of a value. For example: to work with numbers, only mathematical operations are allowed on them, or with strings, operations are concatenating two strings, appending and so on. But to add a number and a string is a type error. The process of verifying and enforcing the constraints of types is called type checking. It may take place either at compile-time, called static checking, or at run-time, called dynamic checking.

Dynamic type checking: dynamic type checking is performed at run-time as opposed to at compile-time. Dynamic type checkers generally associate run-time objects with their type information. Dynamic type checkers will result in a run-time type error during execution when a value has an unexpected type. For example, the run-time system will throw an exception if an integer variable is instantiated with a string value. Detecting program bugs at the later phase of the development cycle, i.e., at run-time, could cause serious damages. Moreover, the detected errors returned by dynamic type checkers might be non-deterministic as there may be only some particular execution paths where the misuse occurs, but not in some other paths depending on the input values or the scheduling. Furthermore, run-time checking may affect negatively the efficiency of the program. The results from dynamic checking are, however, more precise compared to static checking since it is based on the actual values. In practice, dynamic and static type checking are often used in combination to increase efficiency and precision.

Static type checking: static type checking is a limited form of program verification, it allows many program errors to be caught early in the development cycle. Static type checkers evaluate the program code at compile time, but are able to guarantee that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks at run-time. That may also help program execution to be more efficient (e.g. faster or more memory efficient) by omitting run-time type checks and enabling other optimizations (e.g. detecting dead-code, backward tracking). Because they evaluate type information during compilation and therefore lack some information only available at run-time, static type checkers are conservative in terms of over-approximation, i.e., some programs might be rejected by static type check even though they are actually well-behaved at run-time. For instance, a program may be rejected as ill-typed, because a static analysis cannot determine which branch in a conditional will be executed. However, conservative approximation can be advantageous where exhaustive checking is sometimes impossible due to a large or infinite state space, especially in the concurrency setting where interactions between threads may lead to state space explosion. In that circumstance, run-time checking might not detect the problem as a type error only reveal itself in a particular execution path. Nonetheless, since it is a symbolic analysis, static type checkers can detect type errors in all possible executions of the program as the state space to be checked at compile-time is finite in the size of the program .

Evaluating an expression in a program sometimes does not result only in a value but also an effect. For example, an exception raised when evaluating a division operator of a number and zero is considered as an effect; or that a value of a variable is changed while it is read by a thread is also a side-effect. In computer science, an effect system is a formal system which describes the computational effects of computer programs, such as side effects. A type and effect system is typically an extension of a type system where the type part is supplemented with an effect part to reason about computational effects of a program. It can be used to provide compile-time checking of the possible effects of the program.

4.4.2 Formalization of type and effect systems

In this thesis, we propose different static analyses based on type and effect systems to deal with various kinds of concurrency errors which are reflected in the different forms of effects and corresponding derivation systems. Type specifications presented in this thesis are divided in two groups: pre- and post-conditions and assumption-commitment framework. Most of our analyses are based on pre- and post-condition, but when introducing an open semantics for open programs, we go for an assumption-commitment framework because as we argue later in the paper [9] pre- and post specification is too week to capture behaviors of concurrent programs in open systems with the presence of inheritance. Each of the frameworks strengthens compositional reasoning.

For the sake of a clean presentation, our type and effect systems are presented at two levels: a thread local level and a global level. At the local level, the derivation system deals with expressions and judgments are of the form:

$$\Gamma; \Delta_1 \vdash e : T :: \Delta_2 \tag{4.6}$$

where we use Γ to capture typing information and Δ effects. The judgements are interpreted as follows: Under the type assumptions Γ, an expression e is of type T and has an effect which is the change from Δ_1 to Δ_2.

At run-time, expressions do not only contain variables but also references as values. They are stored in the heap σ. To prove preservation of well-typedness under reduction ("subject reduction") we need to be able to check also the well-typedness of configurations at run-time. Hence we extend the type and effect judgment from equation (4.6) to

$$\sigma; \Gamma; \Delta_1 \vdash e : T :: \Delta_2. \tag{4.7}$$

In all cases, the typing contexts (or type environments) Γ contain the type assumptions for variables, i.e., they are finite mappings from variables to their types. The effect part captures different properties of our calculi serving different purposes. In this thesis, when dealing with lock or transaction handling, we use Δ to represent *local lock or transaction environments* which capture lock or transaction information. They are finite mappings from either locks or transactions to their values which say how many times a lock is taken or how deep a transaction is nested. Those values we later call lock or transaction balances. Note that locks and transactions are re-entrant and nested, so we use a natural number to represent the lock or transaction status, i.e., they can be taken one or many times.

At the global level, we formalize judgments of the form

$$\sigma; \Gamma \vdash P : ok\ , \tag{4.8}$$

where P is given by equation (4.3). Moreover, we assume that Γ is well-typed and σ is well-formed, i.e., $\sigma \vdash ok$. A process $p\langle t\rangle$ is well-typed if its code t is. Although well-typedness is checked per process, our static analysis is compositional, hence we can conclude that a parallel composition is well-typed if all sub-configurations are.

4.4.3 Formal proofs of soundness

A derivation system is sound if only true formulas can be derived by its derivation rules. In our setting, the formulas to be derived are the type and effect judgements. The intended meaning, i.e., the underlying semantics, of a judgement of the form $\sigma; \Gamma; \Delta_1 \vdash e : T :: \Delta_2$ (cf. Equation (4.7)) is as follows relative to the typing assumption Γ over the free variables of e:

1. if e terminates, the resulting value is a member of the set of values of type T.

2. if e terminates when starting in a state satisfying Δ_1, the ending state satisfies Δ_2.

Thus, the typing as well as the effect part are interpreted as partial correctness assertions over the expression e. The key to soundness of a typing system is known as subject reduction. For the typing part, the overall soundness argument involves the following three steps:

1. the initial configuration is well-typed because we consider only well-typed programs.
2. Well-typedness is preserved under an arbitrary steps of the operational semantics. This property is known as subject reduction.

Starting from a well-typed initial configuration as a base case and with subject reduction as induction case, the first two steps guarantee that all reachable configurations are well-typed. This allows to avoid checking well-typedness at run-time, i.e., the programming language is statically typed. Note that the well-typedness of all reachable configurations as a consequence of the first two steps does not guarantee soundness of the systems in the sense mentioned above. For instance, a "type system" which would assign any type to any expression satisfies that all reachable configurations are well-typed. Since in our partial correctness interpretation, the typing judgements express that the resulting value of an expression with a given type is an element of the domain described by the type, the third step in the soundness proof is as follows:

3. The typing rules for values are obviously true, i.e., sound.

The preservation of the effect part of the judgements requires some adaptations. In contrast to the typing part which describes the potential results when evaluating an expression, i.e., its value, the effect part captures aspects of what happens *during* the evaluation. That means that reduction steps change the effect, i.e., we can not literally expect preservation of the effect part under steps of the operational semantics.

Assume that we are given a judgement in the form of a pre- and post-specification $\sigma_1; \Gamma; \Delta_1 \vdash e_1 :: \Delta$ and the state σ_1 is abstractly described by the pre-condition Δ_1. If e_1 does a step reducing to e_2 thereby changing the state to σ_2, also the effect for e_2 can be derived in the effect system. However, the pre-condition has now changed to, say Δ_2 which abstractly describes the changed state σ_2. This is shown schematically in Figure 4.1. Considering the pre- and post-conditions of the form Δ as abstract states and the change from Δ_1 to Δ_2 as abstract semantic steps, subject reduction can be understood as simulation relation between the concrete steps of the operational semantics and the change of the corresponding effects, in this case in particular the pre-conditions.

$$\begin{tikzcd}
\Delta_1 \arrow[r] \arrow[d] & \Delta_2 \arrow[d] \\
\sigma_1; \Gamma; \Delta_1 \vdash e_1 :: \Delta \arrow[r] & \sigma_2; \Gamma\Delta_2 \vdash e_2 :: \Delta
\end{tikzcd}$$

Figure 4.1: Subject reduction for effects

In this thesis, we prove soundness of different type and effect systems wrt. the operational semantics of the corresponding calculus using subject reduction in the general form as explained. To prove subject reduction, one starts by assuming: 1) well-typedness of a given expression and 2) one reduction step. Both assumptions are justified by a respective derivation, i.e., a derivation in the type and effect system respectively a derivation using the rules of the structural operational semantics. Note that both derivation systems are defined structurally over the syntax of the expressions. In all cases, the proof proceeds by induction on the derivation of the operational semantics. In the chosen syntactic representation of the calculi based on a-normal forms (or administrative normal forms) [74], many of the deduction rules of the structural operational semantics are actually axioms, i.e., pure rewriting rules using top-level rewriting. In this representation, for instance, the syntax is "restricted" in the following form: instead of writing for a method call $e_1.m(e_2)$, the call needs to be written in expanded form as `let` $x_1 = e_1$ `in` (`let` $x_2 = e_2$ `in` $x_1.m(x_2)$). That makes the order of evaluation explicit; in this case that the callee expression e_1 is evaluated before the argument expression e_2. Obviosuly, in an imperative setting where the expressions may have side-effects, the evaluation order matters. This is in contrast to purely functional calculi where the order of evaluation does not matter as far as a potential end result is concerned, thanks to confluence. Obviously, the more general and conventional expression syntax, which we would expect for any *concrete* syntax, can always be transformed into an equivalent a-normal form as intermediate representation, thereby making it easy to *fix* one evaluation order and thus making reduction (per thread) deterministic. To achieve a deterministic evaluation order working directly on general expression (not in a-normal form) an alternative would be rewriting rules based on *evaluation contexts* which can be used to identify the place of the next redex in an expression. Coming back to the inductive proof of subject reduction: As the operational semantics is given as rewriting rules (as opposed to more general SOS-rules with non-trivial axioms) the induction is mostly by case distinction between different rewriting rules. Often, to prove well-typedness of the expression after the reduction step, requires to prove well-typedness of sub-expressions of the original expression. This typically follows from inverting the last rule in the derivation justifying well-typedness of the expression in assumption 1). As it is often the case in inductive proofs, the challenge lies in formulating appropriately the induction hypothesis, i.e., finding the exact formulation of the type system and the abstraction used therein, not in carrying out the actual induction steps.

To summarize: the challenge in achieving a sound static analysis lies in obtaining the following three goals at the same time: 1) compositionality, 2) precision, and 3) soundness. Without compositionality, the analysis is guaranteed not to scale for large programs, therefore not usable in practice. Without precision, compositionality and soundness can trivially be achieved by overly abstracting all details and ultimately rejecting all programs as potentially erroneous. Of course without soundness, it is pointless to formally analyze programs. Achieving all three goals in a satisfactory manner requires human ingenuity. Even if removing interface information by sacrificing precision typically makes a compositional description easier to achieve, this is, however, not uniformly the case. A well-known example is the fact that for parallel processes with shared variables, interface specifications consisting of pre- and post-specifications of the processes' variables are not compositional whereas refining the information into traces of states may lead to a compositional account. Indeed, representing the behavior of a component in full detail as interface information also results in a compositional interface description. Of course this contradicts the goal that it should be possible to calculate automatically the interface description. To capture the exact behavior without any abstraction in the interface would lead to an undecidable, but compositional interface theory. This would contradict the goal of this thesis that the analyses should be automatized to be useful in practice.

CHAPTER 5

Concurrency-specific problems and their analyses

In this section, we give a short overview over various analyses and problems we solve in the different settings described in Section 4.1. This overview is presented in Section 5.1 to Section 5.4 for the different papers which are contained later in the main part of the thesis. This section aims at giving an overall perspective of all work in this thesis, but does not provide all details and concrete solutions corresponding to each paper. For details, the reader needs to consult Part II where all the corresponding papers are included. A short schematic overview of problems and solutions is also shown in Table 5.1.

	Paper #1 [134] Chapter 8	Paper #2 [138] Chapter 9	Paper #3 and #4 [117, 118] Chapter 10	Paper #5 [11] Chapter 11
Concurrency model	Multi-threading	Multi-threading	Multi-threading	Active objects
Synchronization	Transactions	Transactions	Locks	Locks
Problems	Safe commits	Resource estimation	Safe locking	Observable behavior
Challenges	Transactions	Implicit join synchronization	Aliasing, substitution, parameter passing	Inheritance
Analysis	Single-threaded	Multi-threaded	Single-threaded	Multi-threaded
Analysis effects	Pre- and post-conditions	Pre- and post-conditions	Pre- and post-conditions	Traces

Table 5.1: Overview of the papers

Our analyses cover different aspects of concurrency, such as transactions vs. locks and multi-threading vs. active objects. In different sections, we describe four different kinds of problems. Two of them deal with avoiding misuse of concurrency synchronization constructs, namely for transactions on one hand and for locks on the other, which we call *safe commits* and *safe locking*. Another section statically calculates an estimation for resource consumption in a transactional model. The last result formalizes a behavioral interface description of the observable behavior of open components. These problems pose also different technical challenges for static analyses. One distinguishing criterion in that context is whether the analysis is single-threaded or multi-threaded.

Similarly as for the semantics, our typing rules are always presented at two levels: a thread-local one and a global one. An analysis is *single-threaded* if the property being analyzed for the global program follows in a trivial manner from the corresponding analyzed properties of individual threads. Basically, each thread can be analyzed in isolation and considering the problem at the global level does not add further complications. Analyses which are not single-threaded are *multi-threaded*. For example, the two problems assuring safe usages of transactions respectively safe usages of locks are single-threaded in this sense. Writing $\vdash p\langle t \rangle :: ok$ to express that the thread named p executing t safely uses its locks, the rule to assure safe locking on the global level looks as follows:

$$\frac{\vdash p_1\langle t_1 \rangle :: ok \qquad \vdash p_2\langle t_2 \rangle :: ok}{\vdash p_1\langle t_1 \rangle \parallel p_2\langle t_2 \rangle :: ok}$$

which makes the problems single-threaded. Note, however, that it does not mean that the two threads composed in parallel do not interact or synchronize with each other. In our case, they do use locks or transactions. It means that the nature of the analyzed problems is single-threaded. Later we will come back to discuss why safe commits and safe locking are single-threaded in Section 5.1 and Section 5.3, respectively. In general, multi-threaded problems are harder than single-threaded ones. There are, however, other specific challenges connected to each problem. In the simplest analysis dealing with safe commits in a transactional setting, the analysis needs to keep track of the number of `onacids` and `commits` executed in each thread to avoid committing too often or forgetting to commit at the end of a thread.

Changes of the language and of the problem impose additional complications on top of the analysis. In the paper to estimate resource consumption, the problem is no longer single-threaded: due to concurrency, calculating maximal resource consumption for threads running in parallel does not follow in a trivial manner from the resource consumptions of the individual threads. The reason for that is synchronization between threads, in particular the form of synchronization characteristic for the chosen model which corresponds to an implicit form of join synchronization. This is challenging because synchronization is implicit, i.e., without explicit synchronization constructs in the syntax whereas our compositional analysis is syntax-directed. The complication in the safe locking papers does not come from the change of problem —assuring safe locking is more or less analogous to assuring safe usage of transactions— but from the differences in the language, in particular from the fact that locks have explicit identities at the user-level whereas transactions have not. As a consequence, the analysis has to deal with aliasing and passing lock identities as method parameters. To overcome problems in connection with aliasing requires a special form of "resource-aware" substitution which not only takes into account lock identities being substituted, but the values of their lock counters as well. In the last paper, we are interested in the observable behavior of an open system consisting of a set of classes. The particular challenge there is to capture the observable interface behavior in the presence of inheritance, late binding, and overriding.

In all the analyses, the language and the nature of the problem determine also the form of interface information often in terms of an effect needed for a compositional analysis. In all problems except the one dealing with inheritance, the effects are given in the form of pre- and post-conditions. For the resource analysis which is a multi-threaded problem, the pre- and post-conditions are significantly more complex than for the single-threaded problems. The observable interface behavior for the inheritance problem is given in the form of traces. We continue by discussing the individual problems in more detail.

5.1 Safe commits

Transactions are considered to be a high-level, more abstract, and more compositional alternative for concurrency control to more traditional mechanisms, such as locks and monitors. As known from databases, transactions offer valuable safety and failure guarantees: atomicity, consistency, isolation, and durability, or ACID for short. One characteristic difference of transactions compared to locks is a non-blocking behavior. All threads/transactions may run in parallel provided that they guarantee the mentioned ACID properties. As a result, transactional programming languages may make better use of parallelism and resources in concurrent systems, and may avoid also deadlock situations, but may on the other hand cause roll-backs.

How to syntactically capture transactional programming in the language may vary. One option is lexical scoping, e.g., using an *atomic* keyword to mark a block of statements as a transaction where all the code in that region must commit before leaving the transaction. More flexible is non-lexical scoping, where transactions can be started and finished (i.e., committed) freely. One proposal supporting non-lexical scoping of transaction handling is *Transactional Featherweight Java* (TFJ) [110]. The start of a transaction in TFJ programs is marked by the `onacid` keyword and the end by the `commit` keyword. The transactional model of TFJ is quite general. It supports *multi-threaded* and *nested* transactions. Multi-threaded transactions mean that inside one transaction there can be more than one thread running in parallel. *Nesting* of transactions means that a parent transaction may contain one or more child transactions must synchronize as follows: to commit an entire transaction, all child transaction must have committed and the child threads and the thread itself must commit at the same time.

The flexibility of non-lexical use of `onacid` and `commit` comes at a cost: not all usages of starting and committing transactions "make sense". In particular, it is an error to perform a commit without being inside a transaction. For example, by running *main* method in Listing 5.1, m is executed 4 times and we have: `onacid`; `commit`; `onacid`; `commit`; `onacid`; `commit` ; `onacid`; `commit`; `onacid`; `commit`. In this case, the program has no error at run-time, but it will be an error if the main method had another `commit`, for instance *void main(){ n();* `commit`; `commit`;} since the last `commit` is executed outside any transaction. Note that due to non-lexical scope, users can commit a transaction started inside a method outside its scope (e.g, the transaction created by a `onacid` inside method n is committed by a `commit` at the beginning of the method m. Similarly the last transaction created by m is committed by a `commit` in the main method). So checking errors of methods in isolation is incorrect. A method needs to be put in a larger context together with other methods to know whether a `commit` is executed outside a transaction.

Listing 5.1: Example of onacid and commit

```
    void n(){ onacid; m(3); }

    void m(i){
            commit;
            if(i ≤ 0)
            then  onacid;
            else calculate e; onacid; this.m(i-1);
            //e doesn't contain onacid or commit
            }
    void main(){ n();  commit;}
```

Our solution for this problem is to keep track of starting and committing transactions in a static type and effect system to assure that starting and committing transactions is done "properly", in particular to avoid committing when outside a transaction, which we call *commit errors*. To catch commit errors, the system keeps track of the number of `onacids` and `commits`; we refer to the number of `onacids` minus the number of commits as the *balance*. E.g., for an expression e = `onacid`; e_1; `commit`; `commit` where e_1 does not contain any `onacid` or `commit`, the *balance* equals $1 - 2 = -1$. An execution of a thread is *balanced,* if there are no pending transactions, i.e., if the balance is 0 after the thread terminates. The model does not only support non-lexical use of nested transactions, but also multi-threaded transactions which must be reflected in the analysis as well.

Thus to determine the effect in terms of the balance, we need to calculate the balance for *all* threads potentially concerned, which means for the thread executing the expression being analysed plus all threads (potentially) spawned during that execution. From all threads, the one which carries the expression being evaluated plays a special role, and is treated specially. A thread spawned inside a transaction executes inside this transaction, therefore in the static analysis, the balance at the point of spawning will be used as pre-condition when analyzing the spawned thread. Apart from this connection between the parent and child threads, the analysis is basically single-threaded. The judgements of the analysis are of the following form:

$$n_1 \vdash e :: n_2, S \ , \tag{5.1}$$

which reads as: starting with a balance of n_1, executing e results in a balance of n_2. The multi-set S of lock-balances contains the balances for new threads *spawned* by e, but not of the thread executing e itself. The balance for the new threads in S is calculated *cumulatively;* i.e., their balance includes n_1, the contribution of e before the thread is spawned, plus the contribution of the new thread itself. The judgement of method calls in our type and effect system is slightly different: $\vdash m(\vec{x}{:}\vec{T})e :: n_1 \rightarrow n_2, S$. Here we require that the balance of the method itself has the form $n_1 \rightarrow n_2$ where n_1 is interpreted as pre-condition, i.e., it is safe to call the method *only* in a state where the balance is at least n_1. The number n_2 as the post-condition corresponds to the balance after exiting the method, when called with balance n_1 as pre-condition. The pre- and post-condition formalization is needed for two purposes: to allow flexibility of non-lexical scope of transactions where transactions can be opened in one method and closed in another methods as in Listing 5.1, the precondition n_1 is required to assure that at the call-sites the method is only used where the execution of the method body does not lead to a negative balance. And later when dealing with recursive method calls, an additional constraint requiring $n_1 = n_2$ is used as loop invariant to make sure that body of a method must not change the balance to be well-typed as explained below.

Now let us come back to Listing 5.1. First observe that the program shows no commit-errors during run-time. Method m calls itself recursively and the two branches of the conditional in its body both execute one `onacid` each. Especially, method m is called only via method n, after n has performed an `onacid`, i.e., m is called *inside* one transaction. If m were called *outside* a transaction it would result in an error, as the body of m starts by executing a commit-statement. In our effect system, method m can be declared as of effect $1 \rightarrow 1$, which expresses not only that the body of m does not change the balance, but that as a precondition, it must be called *only* from call-sites where the balance is ≥ 1, as is the case in the body of n.

For recursive calls, an effect like $1 \rightarrow 1$ can be interpreted as *loop invariant*: the body of the method must not change the balance to be well-typed. However, not every method needs to be balanced; the non-recursive method n is one example which (together with the call to m) has a net-balance of 1. As an aside: if the method would contain the spawning of a new thread, the recursive execution would lead to an unbounded number of new threads; for the analysis, this does not pose a problem, as it analyses the method body in isolation (without "following" the recursion, as is customary for type analyses).

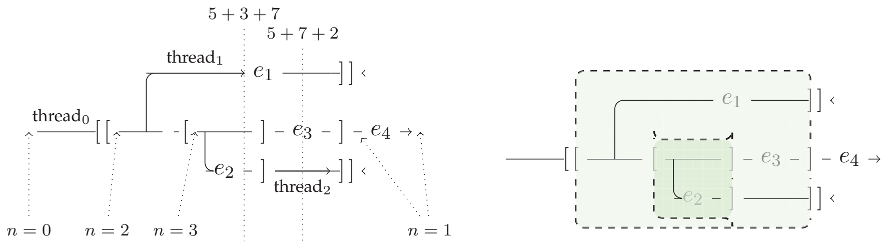
The detail of our type and effect system to prevent *commit errors* is presented in Chapter 8 where we prove the soundness of our analysis using subject reduction and the analysis is compositional.

5.2 Resource estimation

Although transactions based on optimistic concurrency control are a more compositional alternative to conventional concurrency control mechanisms, in particular transaction models based on versioning and roll-back mechanisms can lead to resource consumption problems. In transactional programming, to achieve isolation, each transaction operates via reads and writes on its own local copy of the memory, e.g., a local log is used to record these operations to allow validation or potential rollbacks at commit time. As each transaction operates on its own log of the variables it accesses, a crucial factor in the memory consumption is the number of thread-local transactional memories (i.e., logs) that may co-exist at the same time in parallel threads. Note that the number of logs neither corresponds to the number of transactions running in parallel (as transactions can contain more than one thread) nor to the number of parallel threads, because of the nesting of transactions. A further complication is that parallel threads do not run independently; instead, executing a commit in a transaction may lead to a form of implicit *join synchronization* with other threads inside the same transaction due to the multi-threaded and nested transaction model. Unlike the previous analysis for safe commits, the problem of resource estimation in this setting is multi-threaded. Let us take a look at an example in Listing 5.2. The main expression of thread 0 spawns two new threads 1 and 2. The `onacid`-statement expresses the start of a transaction and `commit` the end. Hence, thread 1 starts its execution at a nesting depth of 2 and thread 2 at depth 3. See also Figure. 5.1a, where the values of n represent the nesting depth of open transactions at different points in the main thread. We often write [and] for starting respectively committing a transaction. Note that e.g. thread 1 is executing *inside* the first two transactions started by its parent thread and that it uses two commits (after e_1) to close those transactions. Important is that parent and child thread(s) commit an enclosing transaction at the same time, i.e., in a form of join synchronization. We call an occurrence of a commit-statement which synchronizes in that way a *joining commit*. Figure. 5.1b makes the nesting of transactions more explicit and the right-hand edge of the corresponding boxes mark the joining commits. E.g., e_2 and e_3 cannot execute in parallel since e_2 is sequentialized by a joining commit before e_3 starts. If the child thread, say in e_1, starts its own transactions (nested inside the surrounding ones), e.g., if $e_1 = [\,;\,[\,;\,[\,;\,]\,;\,]\,;\,]$, then these three commits are no joining commits.

Listing 5.2: Joining commits

```
onacid;                              // thread 0 (main thread)
  onacid;
    spawn (e1;commit;commit);        // thread 1
    onacid;
```



(a) (b)

Figure 5.1: Nested, multi-threaded transactions and join synchronization

```
      spawn (e2;commit;commit;commit);   // thread 2
    commit;
    e3
  commit;
  e4;
```

As always, our goal is a compositional static analysis, in this case a worst-case estimation of memory resource consumption for the sketched execution model. As mentioned, to assure isolation, an important transactional property, each thread operates on a local copy of the needed memory which is written back to global memory when and if the corresponding transaction commits. We measure the resource consumption at a given point by the *number* of logs co-existing at the same time. This ignores that different logs have different memory needs (e.g., accessing more variables transactionally). Abstracting away from this difference, we concentrate on the synchronization and nesting structure underlying the concurrency model. A more fine-grained estimation of resource consumption per log is an orthogonal issue and the corresponding refinement can be easily incorporated. The refinement would need to rely on a conservative estimation of the memory consumption of one log, which in turn depends on the resource consumption per variable used in the transaction, and potentially, dependent on the transactional model, how many times variables are accessed.

To illustrate the concept of resource consumption in our setting, let us take a look at Listing 5.2. Assuming that e_1 opens and closes three transactions, e_2 four, e_3 five, and e_4 six. The resource consumption after spawning e_2's thread and before the subsequent `commit` is at most $15 = 5 + 3 + 7$ (at the left vertical line): the main thread executes inside 3 transactions, thread 1 inside 5 (3 from e_1 plus 2 "inherited" from the parent), and thread 2 inside 7. At the point when thread 0 executes e_3, i.e., after its first commit, the worst case is $14 = 5 + 7 + 2$. Note that e_2 cannot run *in parallel* with e_3 whereas e_1 can: the commit before e_3 synchronizes with the commit after e_2 which sequentializes their execution. Thus e_1 still contributes 5, e_2 contributes only 2, and the main thread of e_3 contributes 7 (i.e, 5 from e_3 and 2 from the enclosing transactions).

To be scalable and thus usable in practice, the analysis must be *compositional*. In our setting, the analysis needs to cope with parallelism and synchronization. In principle, the resource consumption of a *sequential* composition $e_1; e_2$ is approximated by the *maximum* of consumption of its constituent parts. For e_1 and e_2 running (independently) in parallel, the consumption of $e_1 \parallel e_2$ is approximated by the *sum* of the respective contributions. The challenges in our setting are:

Multi-threaded analysis: due to joining commits, threads running in parallel do not necessarily run independently, and a sequential composition `spawn` $e_1; e_2$ does not sequentialize e_1 and e_2. They may synchronize, which introduces sequentialization, and to achieve precision, the analysis must be aware of which program parts can run in parallel and which cannot. Assuming independent parallelism would allow us to analyze each thread in isolation. Such a single-threaded analysis would still yield a sound over-approximation, but be too imprecise.

Implicit synchronization: Compositional analysis is rendered intricate as the synchronization is *not* explicitly represented syntactically. In particular, there is no clean syntactic separation between sequential and parallel composition. E.g., writing $(e_1 \parallel e_2); e_3$ would make the sequential separation of $e_1 \parallel e_2$ from e_3 explicit and would make a compositional analysis straightforward. Here instead, the sequentialization constraints are entailed by joining commits and it is not explicitly represented with which other threads, if any, a particular commit should synchronize.

Thus, the model has neither independent parallelism nor full sequentialization, but synchronization is affected by the nesting structure of the multi-threaded transactions. As usual, the type and effect system will be presented in two levels: a thread-local one and a global one.

On the local level, the judgments of the effect part are of the following form:

$$n_1 \vdash e :: n_2, h, l, \vec{t}, S \ . \tag{5.2}$$

Compared to the judgment of equation (5.1) for safe commits, the form of the local judgment is more complex. The interpretations of the natural numbers n_1 and n_2 are unchanged representing the balances before and after executing e. Since the analysis is aiming at upper bounds on the number of logs, h simply captures the maximal balance during the execution of e. Assuming independent parallelism, i.e., ignoring the implicit join synchronization, the maximal balance per thread captured in h would be adequate to get an approximation for parallel programs. It could be calculated by summing up the maximal resource consumption of each thread. Even in the presence of join synchronization, the analysis would be sound. The remaining information $\vec{t}$ and S in this judgment therefore serves to achieve a precise and compositional analysis for parallel compositions with join synchronization. More precisely, the information S is needed to achieve compositionality wrt. sequential composition and $\vec{t}$ for compositionality wrt. parallel composition:

- The S-part contains information concerning the resource consumption of threads being spawned in e, more precisely their resource consumption *after* e. S needs to be taken into account when considering e in a sequential composition with a trailing expression.
- In contrast, the $\vec{t}$ is needed for compositionality wrt. parallel composition. The $\vec{t}$ is a sequence of non-negative numbers, representing the maximal, overall ("total") resource consumption *during* the execution of e, including the contribution of all threads (the current and the spawned ones) separated by joining commits of the main thread. We call $\vec{t}$ a joining-commit sequence, or *jc-sequence* for short.

At the global level, the key is again to find an appropriate representation of the resource effects which is compositional wrt. parallel composition of threads. Now that more than one thread is involved, the jc-sequences are generalized to *jc-trees* which are basically finitely branching, finite trees where the nodes are labeled by a transaction label and an integer. With t as jc-tree, the judgments at the global level are of the following form:

$$\Gamma \vdash P :: t\,. \tag{5.3}$$

Chapter 9 contains more examples illustrating the type and effect system and the intuition behind jc-sequences and jc-trees. Furthermore, it proves soundness of the analysis using subject reduction.

5.3 Safe locking

Locks are a common synchronization mechanism for concurrency control in many concurrent object-oriented languages. How to syntactically capture concurrent programming in the language may vary. There are two syntactical ways to use locks in Java. The first one is to use lexically-scoped locks via the *synchronized* keyword to protect a shared data within a block. Later, Java 5.0 introduces non-lexical locks to give programmers more flexibility. Users can take a lock l by executing $l.$ `lock` in a block and release the lock by calling $l.$ `unlock` in another block as in Listing 3.2. Misuse of locks in Java, such as to attempt to release a lock without owning it and to takes a lock without releasing it afterwards, could cause safety problems, such as deadlock and even liveness. Java deals with these misuses of locking by throwing a run-time exception. We introduce a static analysis to guarantee absence of certain erroneous use of locks. We call such a discipline safe locking.

Our static analysis is formulated for an extension of Featherweight Java with concurrency and explicit lock support, but without inheritance and type casts. Expressions dealing with locks are of the following syntax:

$$e ::= \ldots\ v.\,\texttt{lock} \mid v.\,\texttt{unlock}\ \ldots$$

The problem with non-lexical scope at a first look seem quite similar to the one for transactions. Generalizing our approach for transactions to lock handling, however, is not straightforward because locks and transactions have behavioral differences relevant for type-based analysis which are summarized in Table 5.2.

	Lock-based setting	Transaction-based setting
non-lexical scope	yes	yes
program level identity	yes	no
re-entrance	yes	no
nested transactions (critical sections)	no	yes
internal multi-threading	no	yes

Table 5.2: Transactions and explicit locks of Java

Both languages have one common feature which is the flexibility of non-lexical scoping. Unlike transactions, locks have identities at the program level which can create aliasing problems since locks are passed around as arguments. Furthermore, locks and monitors in Java are *re-entrant*, i.e., one thread holding a lock can recursively re-enter a critical section or monitor. Re-entrance is not an issue in the transactional setting: a thread leaves a transaction by committing it (which terminates its transaction), hence re-entrance into the same transaction makes no sense. Transactions in the transaction-based setting can be nested. Of course, in Java, a thread can hold more than one lock at a time; however, the critical sections protected by locks do not follow a first-in-last-out discipline, and the sections are not nested as they are independent. For nested transactions, in contrast, a commit to a child transaction is propagated to the surrounding parent transaction, but not immediately further, until that parent commits its changes in turn. Finally, the transaction-based setting allows concurrency within a transaction (supporting multi-threaded transactions), whereas monitors and locks in Java are meant to ensure mutual exclusion. In particular, if an activity inside a monitor spawns a new thread, the new thread starts executing *outside* any monitor, in other words, a new thread holds *no* locks.

Those differences cause the following difficulties for our static analysis of safe locking. A consequence of identities and re-entrance is that our analysis needs to take lock identities into account to keep track of which lock is taken by which thread and how many times it has been taken. Furthermore, the analysis needs to handle dynamic lock creation, aliasing, and passing of locks between threads.

On the local level, the type and effect system uses a judgment similar to the one for safe commits (cf. equation (5.1)):

$$\Delta_1 \vdash e :: \Delta_2 \, . \tag{5.4}$$

Since locks carry an identity, the thread needs to keep track of balances per lock, i.e., how many times a lock is taken or released, not just the global balance as in the transaction setting. In the judgment, the pre- and post-conditions Δ_1 and Δ_2 are mappings from lock variables or references to their values which are the natural number representing the lock status (or the lock balance), and are either 0 in case the lock is marked as free, or n (with $n \geq 1$) capturing that the lock is taken n times by the thread under consideration. Our analysis will guarantee that at no point in time, the lock status or balance goes to minus. In other words, our type and effect system guarantees that our well-typed programs will not result in any run-time errors. As mentioned, the presence of lock identities means that aliasing may become a problem. Two variables are aliases if they refer to the same identity, in our case to the same lock at run-time. There are static analyses which approximate aliasing information which are known as pointer analyses or alias analyses. In particular, in the presence of concurrency, alias analyses may be complicated and imprecise. For a particular property, namely the lock balances, both problems, the aliasing problem and concurrency, can be dealt with smoothly. Concurrency does not pose a major problem in this setting. Despite the fact that locks are shared between threads and that they represent mutable data via lock counters, interference is not a problem since each lock can be held at most by one thread. Therefore, assuring safe locking is a single-threaded problem. As for aliasing, the core observation is as follows:

> given two variables in a thread without knowing whether they are aliases or not, if each variable behaves properly in isolation without leading to a lock error, the same remains true if it turns out at run-time that they are aliases.

In our calculi, the local variables play two different roles, one is as formal parameters of method definitions and the other is as thread-local variables. Variables are user-syntax and at run-time they are bound to, which means *substituted* by, corresponding run-time values. In the case of lock-typed variables, the corresponding run-time values are lock references. The above core observation stipulates that if during run-time two different variables happen to be substituted by the same lock reference, i.e., they become aliases, this does not lead to a locking error.

Slightly simplifying the later judgments, we write $\vdash p\langle t\rangle : ok$ to express that the process p, starting without holding any locks and executing t, has no lock errors. The above observation can then be written as follows:

$$\vdash p\langle t\rangle : ok \text{ then} \vdash p\langle t[l/x_1][l/x_2]\rangle : ok \ .$$

In other words, the ok-judgment is preserved even under aliasing substitution. During the analysis of the code, the judgments are of the form as shown in equation (5.4). In the judgment, the pre- and post-conditions capture the lock balances, both the lock variables and the lock references. Now substitution must, however, reflect the fact that the bindings in the lock environments represent resources. For instance, a Δ of the form $x_1{:}1, x_2{:}1$ specifies that the lock represented by the variable x_1 has a balance of 1 and the same for the variable x_2. If at run-time x_1 and x_2 happen to be aliases, i.e., the variables x_1 and x_2 are substituted by the same lock l, then the result of the substitution must take into account that the resources are to be combined:

$$\Delta' = \Delta[l/x_1][l/x_2] = l{:}(1+1)$$

i.e., l is of balance 2.

The substitution of lock variables by lock references at run-time occurs in two situations, one for parameter passing in method calls and the other when doing a reduction step of a let-construct of the form `let` x:L $= l$ `in` t. The let-construct is in general of the form `let` $x{:}T = e$ `in` t and is a generalization of a sequential composition, in this case of e followed by t. The standard rule of pre- and post-condition reasoning for sequential composition requires that the pre-condition of t equals the post-condition of e. The let-construct, however, does not only represent sequential composition but also introduces a local variable x with scope t. In particular, after t, the post-condition of t can no longer about the balance of x since its scope has ended. Therefore the correct typing rule for the let-construct must in its pre- and post-conditions substitute x by the lock reference to which e evaluates. To be able to do so, the type system must represent this information (in the part $\&v$ below) and the judgment looks as follows:

$$\Delta_1 \vdash e :: \Delta_2\&v \tag{5.5}$$

for lock-typed expressions.

Generalizing the treatment of method calls for safe commits described in Section 5.1, method declarations are of the form:

$$\vdash o.m() : \ \Delta_1 \to \Delta_2, \tag{5.6}$$

basically saying that method m must be called in a context where the actual locks have greater or equal values compared to the required values of those lock parameters specified in its pre-condition Δ_1, again taking into account the resource-aware substitutions described above.

Apart from standard control flow, this work also deals with Java's exception mechanism. The construct for handling exceptions, in its general form, consists of three parts or blocks: The try-part harnesses the code which may raise an exception, one catch-branch is executed if it matches an exception raised in the try-block. The catch-clauses work like a case construct in that *at most* one case-branch is executed and which one (if any) is decided on a first-match policy. Especially, if an exception is thrown in one of the catch-clauses, it cannot be fielded in a subsequent catch-clause of the same try-catch-finally expression. The trailing finally-clause is unconditionally executed, i.e., independent of whether or not an exception is raised and/or caught in the try- and the catch-clauses. As known, exceptions complicate the sequential control flow by introducing non-local "jumps" from the place where an exception is raised to the one where it is caught and handled (or alternatively "falls through"). Not only does this require to over-approximate thrown (and potentially caught) exceptions, but also, the analysis must keep track of the different lock-status at the points where the exceptions may occur. As a consequence, our type and effect system needs to be extended in general to express the possibility of exceptions being thrown. This is expressed by introducing another effect, basically the "set" of potential exceptions raised (and not caught) during the execution of an expression or thread.

We have proved soundness of our analysis for the calculus. The analysis is compositional and can handle aliasing, dynamic lock creation, multi-threading concurrency, and exceptions (cf. Chapter 10).

5.4 Observable behavior

A crucial feature in mainstream object orientation is *inheritance,* which allows code reuse and is intended to support incremental program development by gradually extending and specializing an existing class hierarchy. However, as discussed in Section 3.2, inheritance is not unproblematic as can be seen in the fragile base class problem, and in particular the combination of inheritance and concurreny can be complicated, known as inheritance anomaly. For open systems, the problems become even more complex. With a behavioral interface specification given as method pre- and post-conditions, replacing one super or base class by another satisfying the same interface description may break the code of the client of the super class, i.e., change the behavior of the "environment" of the super class. The problem is that the interface specification is too weak to allow considering the super-class as a black box which can be safely substituted based on its interface specification only. In the open setting, an *open* system is a part of a larger system, which interacts with its environment, and best considered as a black box whose internals are hidden. Such a separation of internal behavior from externally relevant interface behavior is crucial for compositionality.

Our goal is to obtain a formal, compositional behavioral interface description for open systems which *matches* what can be observed by client code in the presence of inheritance and late-binding. This means that observable phenomena must be represented in the interface description. In addition, the interface description should only include possible behaviors, i.e., behaviors generated by *some* actual (well-formed, well-typed) program in the given language. From an observational point of view, only the interaction with the environment (or observer) counts and whether this leads to observable reactions in the environment. A rigorous account of such an interface behavior is the key to formal verification of open programs as well as a formal foundation for black-box testing. It ultimately allows compositional reasoning, i.e., to infer properties of a composed system from the interface properties of its sub-constituents without referring to further internal representation details. A representation-independent, abstract account of the behavior is also necessary for compositional optimization of components: only when showing the same external behavior one program can replace another without changing the interaction with any client code.

To achieve a compositional abstraction in terms of an open and compositional semantics to reason about behaviors of open systems in the setting of missing information about their environments, we need to keep in mind some important factors when defining a formal semantics. E.g., self-calls lead to observable differences in the presence of inheritance and thus are part of the observable behavior. On the other hand, behavior which is impossible can not be included in the open semantics. Therefore, the question of what exactly can be observed from outside a "component" when considering *inheritance* is subtle. An interface interaction happens if a step of the component affects the environment and vice versa. Objects encapsulate their states, and thus the interaction takes the form of method calls and returns, where the control changes from executing component code to environment code (by an outgoing message) and vice versa (by an incoming message). Thus the interface behavior will be given in terms of traces of call and return labels exchanged at the interface, where in our setting component classes can extend those from the environment via inheritance, and vice versa. Writing $Comp \stackrel{t}{\Longrightarrow} \acute{Comp}$, the t denotes the *trace* of interface actions by which component $Comp$ evolves into $\acute{Comp}$, potentially executing internal steps, as well, not recorded in t. Being open, $Comp$ does not act in isolation, but interacts with *some* environment. I.e., we are interested in traces t where *there exists an environment* Env such that $Comp \parallel Env \underset{\bar{t}}{\stackrel{t}{\Longrightarrow}} \acute{Comp} \parallel \acute{Env}$ by which we mean: component $Comp$ produces the trace t and Env produces the dual trace $\bar{t}$, both together "canceling out" to internal steps. Our goal is an open semantics with the environment *existentially abstracted away.* With infinitely many possible environments Env, the challenge is to capture what is common to *all* those environments. This will be done in form of *assumptions* about the environment: the operational semantics specifies the behavior of $Comp$ under certain assumptions Ξ_E about the environment. Following standard notation from logics, we do not write $\Xi_E \parallel Comp$, but rather $\Xi_E \vdash Comp$. Reductions thus will look like

$$\Xi_E \vdash Comp \stackrel{t}{\Longrightarrow} \acute{\Xi}_E \vdash \acute{Comp} \ . \tag{5.7}$$

Such a characterization of the abstract interface behavior is relevant and useful for the following reasons. Firstly: the set of traces according to equation (5.7) is in general more restricted than the one obtained when ignoring the environments altogether. This means, when *reasoning* about the behavior of $Comp$ based on the traces, e.g., for the purpose of verification, more precise knowledge of the possible traces allows to carry out stronger arguments about $Comp$. Secondly, an application for a trace description is black-box testing, in the sense that one describes the behavior of a component in terms of the interface traces and then synthesizes appropriate test drivers from it. Obviously it makes no sense to specify interface behavior which is not possible, at all, since in this case one could not generate a corresponding tester. Finally, and not as the least gain, the formulation gives *insight* into the inherent semantical nature of the language, as the assumptions Ξ and the semantics captures the existentially abstracted environment behavior.

When additionally abstracting away from any concrete component $Comp$ and replacing it by a commitment context, in the symmetric way as abstracting the environment by an assumption context in (5.7), one obtains a formalization of possible interface interactions in the language which we call *legal* traces.

We formalize allowed interface behavior in general. We prove the soundness of the abstractions. Including inheritance influences in subtle ways what is observable, e.g., the observer may override component methods or inherit its own methods to the component which then are rebound by late binding. Capturing the resulting interface behavior accurately complicates the semantics considerably. Several lessons drawn from this work are presented in Chapter 12 where we argue that asynchronous concurrency with the actor model is good for compositionality. The result of full abstraction is a future extension of this work and is useful when proving that when two programs are equal.

CHAPTER 6

Discussion

In this chapter, we return to the research questions of Section 1.2 and we summarize the contributions of the thesis and how its results answer our goals. In addition, we discuss its limitations and future work.

6.1 Summary of the contributions

Our overall goal is to investigate analysis methods for concurrent object-oriented programming languages which can guarantee the absence of common concurrency-related errors, and hence contribute to the quality of concurrent programs. We have developed different static analysis methods for various concurrency-related problems to support higher-quality of code. To tackle the complexity of large and distributed applications, we have insisted that all the analysis methods in this thesis should be compositional. Analyzing and studying the behavior of concurrent programs is known to be demanding because of the complexity of possible interactions between concurrent processes.

As motivated in Chapter 1, the best time to find errors in concurrent programs is at compile-time, when reviewing the source code. Therefore, this thesis has concentrated on static analysis methods, i.e., studying a program without actually executing it. Furthermore, to handle large and even infinite state spaces: the analyses of this thesis are based on abstraction and compositionality. Being based on abstraction which over-approximates the run-time behavior, static analysis may have the advantage of symbolic execution, i.e., it can cover all possible behaviors of a program including those which occur under rare conditions or specific schedulings and which are therefore hard to detect or reproduce. In this thesis, we have investigated static analysis methods based on type and effect systems for different variations of an object-oriented calculus, to prevent common concurrency-related problems occurring in different settings of concurrency. To achieve the overall research goal, we have broken it down into more specific sub-goals. We will now briefly summarize the contributions of the thesis wrt. these goals (cf. also Chapter 7):

1. For the safe use of non-lexical transactions for languages supporting multi-threaded and nested transaction model, we chose TFJ as a formal model. For that calculus, we have developed a static type and effect system in paper #1 [134] that allows to keep track of the nesting depth of the transactions and thus prevent commit errors.

2. For resource estimation, we have developed a static, multi-threaded analysis in paper #2 [138] for the same calculus which not only take the nesting depth into account, but in particular the synchronization model of the language. The concurrency model based on multi-threaded and nested transactions results in an implicit form of synchronization which complicates the syntax-directed analysis.

3. For the safe use of explicit non-lexical locks, we have used a similar variant of Featherweight Java featuring dynamic creation of objects, threads, locks. The static analyses in this case have particularly dealt with lock aliasing and exceptions and are developed in papers #3 and #4 [117, 118].

4. The behavior of open systems in the presence of inheritance and late-binding has been formalized as an observable interface behavior in paper #5 [11]. Since the observing environment can inherit code from the program being observed and vice versa, connectivity information about the heap is part of interface behavior to get a precise presentation. A general description of possible interface behavior is given the form of traces.

In addition, the analyses developed in this thesis also meet the requirements stated in the research goals section as follows:

Formality and rigor

Aiming at formality, we have taken a variant of Featherweight Java with a rigorous semantics as a core calculus. From there we have investigated different extensions of the calculus to study various concurrency aspects and different models of concurrency and synchronization. To give a precise meaning of a program, we use structural operational semantics in all of the papers where the steps of the program are formally defined by a given set of derivation rules. Since all considered languages are concurrent, the derivation rules are given in form of a small-step semantics. In the four papers #1, #2, #3 and #4, we have developed type and effect systems to deal with various concurrency problems related to lock handling and exceptions in Java, transaction handling as well as resource consumption. Paper #5 formally specifies the behavior of open systems, especially in the presence of inheritance. The paper introduces an open semantics in terms of traces for the given language to formally describe the observable behaviors of a component given as a set of classes. In all papers, not only the semantics is formally specified, but the static analyses as well, namely in the form of logical derivation rules. This furthermore allowed to rigorously establish that the static analyses are sound wrt. the run-time behavior as given in the operational rules. The core of these correctness arguments is the preservation of well-typedness under reduction, known as subject reduction.

Scalability

As mentioned before, to be scalable, analyses must be compositional. As it is common, the type and effect systems in this work are syntax-directed. This means there is one typing rule per construct in the language breaking down each expression into its sub-expressions whose analyses are specified by premises of the corresponding rule. Consequently, the complexity of the analyses given in that way is linear in the size of the program code. Depending on problems and language features, achieving a compositional analysis is challenging. The challenge in designing the analyses in the given setting is finding the right level of abstraction, i.e., the right amount of interface information such that the property of the composed system follows from the properties of its subsystems. If a global property of the whole program depends on the synchronization and communication between parallel parts of a program, the corresponding analysis is multi-threaded. Even for the single-threaded analyses in papers #1, #3, and #4 where compositionality wrt. parallel composition is straightforward, other language features and constructs can be difficult to capture compositionally because they involve global aspects. For instance, in paper #3 featuring dynamic lock creation and explicit lock handling, aliasing is an example of such a global aspect characteristic for object-oriented languages. Our contribution here is a specific form of "resource-aware" substitutions which achieves soundness of the analysis even without a global view on potential aliases wrt. lock identities. This yields a compositional analysis.

Paper #2 deals with a multi-threaded problem, namely the estimation of resource consumption. The problem is multi-threaded because a precise estimation of the maximal resource consumption of a current program depends on being able to determine which parts of the code can run in parallel and which not. This of course depends on the specific synchronization and communication model. A compositional account in the chosen transactional setting is complicated by the following two facts: firstly, the complex form of join synchronization based on nested and multi-threaded transactions requires a complicated form of abstraction. Secondly, the synchronization caused by executing a commit is implicit, i.e., the syntax does not express the synchronization partners. This renders the syntax-directed analysis more intricate. The contribution of paper #5 is a formalization of the observable behavior over objects, in particular the influence of inheritance on the specification of interface behavior. The paper includes a compositional non-behavioral type system; the focus, however, is in getting a precise description of possible traces. Allowing inheritance across the boundary of components given as a set of classes makes the structure of the heap partially observable which must be reflected in the observable traces. It is observable in an over-approximate manner whether objects communicating over the component's interface may be connected through the heap or not. The interface behavior therefore contains approximate "connectivity" information. To obtain a compositional account, the observable behavior is formulated using the dynamic scope extrusion mechanism as known from the π-calculus based on ν-binders.

Usability

To be accepted in practice, analysis methods must provide guarantees with as little user involvement as possible. Using appropriate static abstractions of the run-time behavior, the analysis methods in this thesis are decidable and moreover described in such a way that their descriptions lend themselves in a straightforward manner toward implementation. The rule-based specifications of the analyses almost directly specify recursive algorithms working on the abstract syntax trees of the chosen programming languages. Since the rules are syntax-directed, the termination of the corresponding algorithm is guaranteed. Moreover, the corresponding algorithms are efficient and scalable as they are linear-time in the size of the analyzed programs.

For clarification, as far as paper #5 is concerned, the contribution is to provide a theoretical description of infinite sets of possible traces without being concerned with decidability. Furthermore, in paper #4, the part dealing with exceptions makes use of a simple form of sub-typing and the rules include a rule of subsumption. This subsumption rule is not syntax-directed. However, it is straightforward to obtain an equivalent formulation without subsumption which then is syntax-directed.

Concurrency

Concurrency comes in many different flavours and modern programming languages use a diversity of mechanisms to express concurrent computations. Our thesis covers several different prominent mechanisms and analyzes relevant properties which are shown in overview in Table 5.1 of Section 5. As discussed, we cover lock-based vs. transaction-based concurrency control mechanisms, traditional Java multi-threading concurrency vs. active objects, i.e., synchronous vs. asynchronous method calls. Dealing with object-oriented programs involves also shared variable concurrency at least within objects. Another very important feature we investigate is inheritance. Design issues and lessons learnt when analyzing different concurrency models are summarized and discussed in Chapter 12.

6.2 Limitations and future work

In this section, we discuss some limitations of our work, and identify some possible extensions and future work. We consider our focus on a core calculus with a carefully chosen set of constructs not as one of the restrictions. Of course, when dealing with a real language, such as Java, other more complex features need to be covered as well. Many of those features will be orthogonal to, or irrelevant for, the handling of locks and transactions, which the main part of the thesis concentrates. Furthermore, we consider the abstract syntax of the core language where more complex features could be captured by syntactic sugar. In particular, the restriction to specific forms of threads and expressions restricting the syntax, for instance to $v.m(\vec{v})$ instead of $e.m(\vec{e})$, does not restrict the expressiveness of the language. We choose this formulation to simplify the formal definition of the semantics and since it allows to obtain the deterministic evaluation order per thread without the use of evaluation contexts. More importantly, this formulation simplifies the presentation of the type and (in particularly) effect systems without loss of precision. A general restriction for all type and effect analyses is that they do not cover higher-order functions. The motivation for this restriction is that we concentrate on object-oriented languages, such as Java, which do not feature higher-order functions. It could be interesting to investigate how to integrate the analyses in a setting with higher-order functions.

In the current formulation of the type and effect systems, the effect parts are automatically inferred except for one case, namely the case where the user has to specify the interface information for method decorations. It could be interesting and practically valuable if also for this case, no user annotations are needed but are automatically inferred as well. In particular, the work dealing with resource consumption in Chapter 9 uses a coarse measure for resource estimation, namely the number of transactions potentially running in parallel. A more precise estimation should take into account that different transactions have different memory needs. The difference comes from the size of data used in the transaction and how different versions of a memory location is needed during the execution of the transaction. A corresponding refinement of the effect system seems feasible. This will require that also variable access in the calculation are considered in the effect system. Another restriction in that work is concerned with method calls. Currently the analysis requires that the balance at the call-site matches exactly the specified pre-condition. The reason for that restriction is rather technical and it is introduced to simplify the treatment of recursive method calls which lead to infinite resource consumptions. Lifting that restriction so that the method can be called at the call-site with a balance greater or equal than the specified pre-condition seems not as straightforward as done in Chapter 8 and 10. In general, a natural next step is to provide implementations for the analysis methods to study their usefulness in practice. In particular, implementations should allow the analyses to be done without user involvement.

CHAPTER 7

List of the research papers

This chapter gives a short summary of each research paper in Part II of this thesis. The contents of the papers appear as in their original publication, but have been reformatted to fit the layout of this thesis. The contribution of the papers is divided into two parts:

- one part is about static compositional analysis based on type systems to prevent concurrency errors caused by complications of concurrent object-oriented language features. This includes papers #1, #2, #3, and #4.
- The second part (paper #5) investigates an abstract compositional semantics for an asynchronous message passing language. Here the goal is to obtain a clean abstract description based on traces for capturing behaviors of open programs. Paper #6 discusses different design choices in that context.

7.1 *Paper# 1: Safe commits [134]*

Abstract: Transactions are a high-level alternative for low-level concurrency-control mechanisms such as locks, semaphores, monitors. A recent proposal for integrating transactional features into programming languages is *Transactional Featherweight Java* (TFJ), extending Featherweight Java by adding transactions. With support for *nested* and *multi-threaded* transactions, its transactional model is rather expressive. In particular, the constructs governing transactions —to start and to commit a transaction— can be used freely with a *non-lexical* scope. On the downside, this flexibility also allows for an incorrect use of these constructs, e.g., trying to perform a commit outside any transaction. To catch those kinds of errors, we introduce a static type and effect system for the safe use of transactions for TFJ. We prove the soundness of our type system by subject reduction.

7.2 *Paper# 2: Resource estimation [138]*

Abstract: We present an effect-based static analysis to calculate upper and lower bounds on the memory resource consumption in an execution model supporting nested and concurrent transactions. The analysis is compositional and takes into account implicit join synchronizations that arise when more than one thread jointly commit a transaction. Central for a compositional and precise analysis is to capture as part of the effects a tree-representation of the future resource consumption and synchronization points. The analysis is formalized for a concurrent variant of Featherweight Java extended by transactional constructs. We show the soundness of the analysis.

7.3 Paper# 3 and #4: Safe locking [117, 118]

Abstract: Many concurrency control mechanisms have been developed for high-level programming languages such as Java. The original mechanism for multi-threading in Java is lexically scoped. Answering the need for more flexible control protocols, Java 5 introduced non-lexical control mechanisms, supporting lock primitives on re-entrant lock objects.

These more flexible operators, however, may lead to run-time errors and unwanted behavior, e.g., taking a lock without releasing it, which could lead to a deadlock, or trying to release a lock without owning it. This paper develops a static type and effect system to prevent the mentioned lock errors for a formal, object-oriented calculus which supports non-lexical lock handling and exceptions and which is inspired by Featherweight Java.

Based on an operational semantics, we prove soundness of the type and effect analysis. Challenges in the design of the type and effect system are dynamic creation of threads, objects, and especially of locks, aliasing of lock references, passing of lock references between threads, and re-entrant locks as found in Java. Furthermore, the exception handling mechanism complicates the control-flow and thus the analysis.

7.4 Paper# 5: Observable behavior [11]

Abstract: This paper formalizes the observable interface behavior of *open* systems for a strongly-typed, concurrent object-oriented language with single-class inheritance. We formally characterize the observable behavior in terms of interactions at the program-environment interface. The behavior is given by transitions between contextual judgments, where the absent environment is represented abstractly as assumption context. A particular challenge is the fact that, when the system is considered as open, code from the environment can be inherited to the component and vice versa. This requires to incorporate an abstract version of the heap into the environment assumptions when characterizing the interface behavior. We prove the soundness of the abstract interface description.

7.5 Paper#6: Design issues [135]

Abstract: This paper discusses different choices in the design of object-oriented, concurrent language from the perspective of observability. Observability takes the standpoint that two “program fragments” are observably equivalent if one can be replaced by the other without leading to differences in a larger context. Characterizing the observable behavior of a program fragment is therefore crucial for compositionality.

The choice of language constructs has a big impact on what can be observed, and thus also how well-suited the language is for being composed. In this paper, we concentrate on well-established variants of constructs in object-oriented languages and discuss their influence on the observable semantics. In particular, we discuss classes as units of code, inheritance as the mainstream way of code re-use in class-based, object-oriented languages. For concurrency, we compare the two most prominent ways to combine objects and concurrency: multi-threading as for instance done in Java vs. the active objects or actor paradigm. A final aspect is the influence of the synchronization mechanism of locks and monitors.

7.6 Further papers

In the following, we list further papers which are technically not presented as part of this thesis, or correspond to shorter and preliminary versions of the work reported in this thesis.

Safe commits: The type and effect system assuring safe commits has been presented as an extended abstract at NWPT'09 [182] and at the Young Researchers Forum of MFCS-CLS'10 [131]. A preliminary version has been published as UiO technical report [181].

Resource estimation: The type and effect system estimating resource consumption of transactions has been presented as an abstract at NWPT'11 [137]. A preliminary long version has been published as UiO technical report [136].

Safe locking: The type and effect system assuring safe locking has been presented as an extended abstract at NWPT'10 [115]. A preliminary long version has been published as UiO technical report [116].

Observable behavior: The open semantics for an object-oriented language with inheritance has been presented as an extended abstract at NWPT'11 [10] and at the Young Researchers Forum of Concur'10 [132]. A preliminary version has been published as UiO technical report [9].

Other papers: A different approach to the problem of resource estimation has been published in the proceeding of SSoICT'12 [194]. A comparison of type systems for two different models of concurrency based on locks and on transactions has been published in the proceeding of KSE'10 [133]. Both papers are not part of the thesis.

Part II

Formal analysis for concurrent object-oriented languages

CHAPTER 8

Safe Commits for Transactional Featherweight Java

8.1 Introduction

With CPU speeds and memory capacities ever increasing, and especially with the advent of multiprocessor and multi-core architectures, effective parallel programming models and suitable language support are in need to take full advantage of the architectural advances. Transactions, a well-known and successful concept originating from database systems [191][83], have recently been proposed to be directly integrated into *programming languages.* As known from databases, transactions offer valuable safety and failure guarantees: atomicity, consistency, isolation, and durability, or ACID for short. Atomicity means that the code inside a transaction is executed completely or not at all, consistency that all transactions have the same "view" on shared data, isolation says that when a transaction is running, other transactions cannot interfere, and durability states successfully committed changes are persistent. One characteristic difference of transactions compared to locks is a non-blocking behavior. All threads/transactions may run in parallel provided that they guarantee the mentioned ACID properties. As a result, transactional programming languages may make better use of parallelism and resources in concurrent systems, and may avoid also deadlock situations.

As mechanism for concurrency control, they can be seen as a high-level, more abstract, and more compositional alternative to more conventional means for concurrency control, such as locks, semaphores, monitors, etc. How to syntactically capture transactional programming in the language may vary. One option is lexical scoping, e.g., using an *atomic* keyword, similar to the *synchronized* keyword in Java for lock-handling. More flexible is non-lexical scoping, where transactions can be started and finished (i.e., committed) freely. One proposal supporting non-lexical scoping of transaction handling is *Transactional Featherweight Java* (TFJ) [110]. In the free use of the transactional constructs, it resembles also the way Java 5.0 allows for lock handling (using the lock and unlock methods via the `Lock`-interface). The start of a transaction in TFJ programs is marked by the onacid keyword and the end by the commit keyword. The transactional model of TFJ is quite general and supports *nested* transactions which means a transaction can contain one or more child transactions, which is very useful for composability and partial rollback. Furthermore, TFJ supports *multi-threaded* transactions, i.e., one transaction can contain internal concurrency. To commit an entire transaction, all child transaction must have committed and the child threads and the thread itself must commit at the same

$$\begin{array}{rcll}
P & ::= & \mathbf{0} \mid P \parallel P \mid t\langle e\rangle & \text{processes/threads} \\
L & ::= & \mathsf{class}\ C\{\vec{f}:\vec{T}; K; \vec{M}\} & \text{class definitions} \\
K & ::= & C(\vec{f}:\vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\} & \text{constructors} \\
M & ::= & m(\vec{x}{:}\vec{T})\{e\} : T & \text{methods} \\
e & ::= & v \mid x \mid v.f \mid v.f := v \mid \mathtt{if}\ v\ \mathtt{then}\ e\ \mathtt{else}\ e & \\
 & \mid & \mathtt{let}\ x : T = e\ \mathtt{in}\ e \mid v.m(\vec{v}) & \text{expressions} \\
 & \mid & \mathsf{new}\ C(\vec{v}) \mid \mathsf{spawn}\ e \mid \mathsf{onacid} \mid \mathsf{commit} & \\
v & ::= & b \mid r \mid () & \text{values}
\end{array}$$

Table 8.1: Abstract syntax

time. The flexibility of non-lexical use of onacid and commit comes at a cost: not all usages of starting and committing transactions "make sense". In particular, it is an error to perform a commit without being inside a transaction; we call such an error a *commit error.* In this paper, we introduce a static type and effect system to prevent these errors by keeping track of starting and committing transactions. The static analysis is formulated as a type and effect system [154]. We concentrate on the effect part, as the part dealing with the ordinary types works in a standard manner and is straightforward. See [181] for details.

The paper is organized as follows. After Section 8.2, which recapitulates the syntax and the operational semantics of the calculus, Section 8.3 defines the effect system to prevent *commit errors.* The soundness of the type system relative to the given semantics is shown in Section 8.4. Section 8.5 concludes with related and future work. In particular we draw some parallel to the lock handling in Java 5.

8.2 An object-oriented calculus with transactions

Next we present the syntax and semantics of TFJ. It is, with some adaptations, taken from [110] and a variant of Featherweight Java (FJ) [105] extended with *transactions* and a construct for thread creation. The main adaptations are: we added standard constructs such as sequential composition (in the form of the let-construct) and conditionals. Besides that, we did not use evaluation-context based rules for the operational semantics. We first present the syntax, and afterwards sketch the underlying type system (without the effects). The type system is fairly standard and largely omitted here. For further detail, cf. the technical report [181].

8.2.1 Syntax

FJ is a core language originally introduced to study typing issues related to Java, such as inheritance, subtype polymorphism, type casts. A number of extensions have been developed for other language features, so FJ is today a generic name for Java-related core calculi. Following [110] and in contrast to the original FJ proposal, we ignore inheritance, subtyping, and type casts, as these features are orthogonal to the issues at hand, but include imperative features such as destructive field updates, further concurrency and support for transactions. Table 8.1 shows the abstract syntax of TFJ. A program consists of a number of processes/threads $p\langle e\rangle$ running in parallel, where p is the thread's or process's identifier and e is the expression being executed. The syntactic category L captures class definitions. In absence of inheritance, a class definition class $C\{\vec{f} : \vec{T}; K; \vec{M}\}$ consists of a name C, a list of fields $\vec{f}$ with corresponding list of types $\vec{T}$ (assuming that all f_i's are different, see equation (8.1) below for the syntax of available types), a constructor K, and a list $\vec{M}$ of method definitions. A constructor $C(\vec{f}{:}\vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\}$ of the corresponding class C initializes the fields of instances of that class, these fields are mentioned as the formal parameters of the constructor. We assume that each class has exactly one constructor; i.e., we do not allow constructor overloading. Similarly, we do not allow method overloading by assuming that all methods defined in a class have a different name; likewise for fields. A method definition $m(\vec{x}{:}\vec{T})\{e\} : T$ consists of the name m of the method, the formal parameters $\vec{x}$ with their types $\vec{T}$, the method body e, and finally the return type T of the method. Here the vector notation is used analogously to the vector $\vec{f}$ which presents a list of fields. The vector $\vec{T}$ represents a sequence of types, $\vec{x}$ stands for a sequence of variables. When writing $\vec{x} : \vec{T}$ we assume that the length of $\vec{x}$ corresponds to the length of $\vec{T}$, and we refer by $x_i : T_i$ to the i'th pair of variable and type. For brevity, we do not make explicit or formalize such assumptions, when they are clear from the context.

In the syntax, v stands for values (where we also use u and v_i, u' etc. as syntactic variants), i.e., expressions that can no longer be evaluated. In the core calculus, we leave unspecified basic values b like booleans, integers, ..., so values can be object references r or the unit value $()$. Expressions include variables x and furthermore, the expressions $v.f$ and $v_1.f := v_2$ represent field access and field update respectively. Method calls are written $v.m(\vec{v})$ and object instantiation is new $C(\vec{v})$. The next two expressions deal with the basic, sequential control structures: `if` v `then` e_1 `else` e_2 represents conditions, and the let-construct `let` $x : T = e_1$ `in` e_2 represents sequential composition: first e_1 is evaluated, and afterwards e_2, where the eventual value of e_1 is bound to the local variable x. Consequently, standard sequential composition $e_1; e_2$ is syntactic sugar for `let` $x : T = e_1$ `in` e_2 where the variable x does not occur free in e_2. The let-construct, as usual, binds x in e_2. We write $fv(e)$ for the free variables of e, defined in the standard way. The language is multi-threaded: spawn e starts a new thread of activity which evaluates e in parallel with the spawning thread. Specific for TFJ are the two constructs onacid and commit, two dual operations dealing with transactions. The expression onacid starts a new transaction and executing commit successfully terminates a transaction by committing its effect, otherwise the transaction will be rolled back or aborted. A note on the form of expressions and the use of values may be in order. The syntax is restricted concerning where to use general expressions e. E.g., Table 8.1 does not allow field updates $e_1.f := e_2$, where the object whose field is being updated and the value used in the right-hand side are represented by general expressions that need to be evaluated first. It would be straightforward to relax the abstract syntax that way and indeed the proposal of TFJ from [110] allows such more general expressions. We have chosen this presentation, as it slightly simplifies the operational semantics and the (presentation of the) type and effect system later: [110] specifies the operational semantics using so-called evaluation contexts, which fixes the order of evaluation in such more complex expressions. With that slightly restricted representation, we can get away with a semantics without evaluation contexts, using simple rewriting rules (and the let-syntax). Furthermore, the formulation of the typing rules, especially the effect parts, gets notationally simpler. This form of representation is also known as a-normal form (or administrative normal form) [74]. Of course, this is not a real restriction in expressivity. E.g., the mentioned expression $e_1.f := e_2$ can easily and be expressed by `let` $x_1 = e_1$ `in` (`let` $x_2 = e_2$ `in` $x_1.f := x_2$), making the evaluation order explicit. The transformation from the general syntax to the one of Table 8.1 is standard.

8.2.2 The underlying type system

We first describe the *underlying* type system, i.e., the standard type system for the object-oriented language that assures that actual parameters of a method call match the expected types for that method, that an object can handle an invoked method. The available types are given in equation (8.1).

$$T \;::=\; C \mid B \mid \texttt{Unit} \tag{8.1}$$

In a nominal type system, class names C serve as types. In addition, B represents basic types (left unspecified) such as booleans, integers. Finally, `Unit` is the type of the unit value (). As usual it corresponds also to the empty sequence of types $\vec{T}$, i.e., the input type of a method with an empty list of formal parameters. In general we return () when evaluating expressions which do not return any "meaningful" value, i.e., it is used for expressions evaluated for their side-effect, only.[1]

The type system is given inductively in Table 8.2. For expressions, the type judgments are of the form $\Gamma \vdash e : T$ ("under type assumptions Γ, expression e has type T"). The *type environment* Γ keeps the type assumptions for local variables, basically the formal parameters of a method body and the fields. Environments Γ are of the form $x_1{:}T_1, \ldots, x_n{:}T_n$, where we silently assume the x_i's are all different. This way, Γ is also considered as a finite mapping from variables to types. By $dom(\Gamma)$ we refer to the domain of that mapping and write $\Gamma(x)$ for the type of variable x in Γ. Furthermore, we write $\Gamma, x{:}T$ for extending Γ with the binding $x{:}T$, assuming that $x \notin dom(\Gamma)$.

The rules of Table 8.2 are straightforward and similar to the ones found for other variants of FJ. To define the rules, we need two additional auxiliary functions. We assume that the definitions of all classes are given. As this information is static, we do not explicitly mention the corresponding "class table" in the rules; relevant information from the class definitions is referred to in the rules by $\mathit{fields}(C)$ or $\mathit{mtype}(C, m)$.

The type of a variable is looked up in Γ in rule T-VAR. The unit value has, as mentioned, the type `Unit` (cf. rule T-UNIT). A conditional expression is well-typed with type T, if both branches carry that type and if the conditional expressions is a boolean expression (cf. rule T-COND). To determine the type of a field lookup in rule T-LOOKUP, we use the *fields*-function to look up the types of the fields of the appropriate class. Similarly for methods calls in rule T-CALL, where mtype yields the type of the method as found in the concerned class. For assignments $e_1.f := e_2$, in rule T-UPD, the type of the appropriate field is determined using *fields* as for field access, and furthermore checked that the type of e_2 coincides with it. The type of a let-construct is the type of the last expression e_2 (cf. rule T-LET). A freshly instantiated object carries the class it instantiates as type, and spawning a new thread has a side-effect, only, and returns no particular value of interest, hence spawn carries type `Unit` (cf. rules T-NEW and T-SPAWN). Similarly, the two operations for starting and ending a transaction, onacid and commit, are evaluated for their effect, only, and carry both the type `Unit` (cf. rules T-ONACID and T-COMMIT).

Rule T-METH deals with method declarations, explicitly mentioning the class C which contains the declaration. The body e of the method is type checked, under a type environment extended by appropriate assumptions for the formal parameters x and by assuming type C for the self-parameter this. A class definition class $C\{\vec{f} : \vec{T}; K; \vec{M}\}$ is well-typed (cf. rule T-CLASS), if all $\vec{M}$ are well-typed in C (the premise $\vdash \vec{M} : ok\ in\ c$ of the rule is meant as iterating over all

[1] In a more imperative setting, one would use type `Void` representing the absence of a value. In the chosen representation using the let-construct for local scopes and sequencing, using `Void` is problematic.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{}{\Gamma \vdash () : \mathtt{Unit}}\ \text{T-UNIT} \qquad \frac{\Gamma \vdash e : C \qquad \mathit{fields}(C) = \vec{f} : \vec{T}}{\Gamma \vdash e.f_i : T_i}\ \text{T-FIELD}$$

$$\frac{\Gamma \vdash v : \mathsf{Bool} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T}\ \text{T-COND}$$

$$\frac{\Gamma \vdash e : C \qquad \mathit{mtype}(C, m) : \vec{S} \to T \qquad \Gamma \vdash \vec{e} : \vec{S}}{\Gamma \vdash e.m(\vec{e}) : T}\ \text{T-CALL}$$

$$\frac{\Gamma \vdash e_1 : C \qquad \mathit{fields}(C) = \vec{f} : \vec{T} \qquad \Gamma \vdash e_2 : T_i}{\Gamma \vdash e_1.f_i := e_2 : T_i}\ \text{T-ASSGN}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash e_2 : T_2}{\Gamma \vdash \mathtt{let}\ x : T_1 = e_1\ \mathtt{in}\ e_2 : T_2}\ \text{T-LET} \qquad \frac{}{\Gamma \vdash \mathsf{new}\, C : C}\ \text{T-NEW}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{spawn}\, e : \mathtt{Unit}}\ \text{T-SPAWN} \qquad \frac{}{\Gamma \vdash \mathsf{onacid} : \mathtt{Unit}}\ \text{T-ONACID} \qquad \frac{}{\Gamma \vdash \mathsf{commit} : \mathtt{Unit}}\ \text{T-COMMIT}$$

$$\frac{\vec{x}{:}\vec{S}, \mathsf{this}{:}C \vdash e : T}{\vdash m(\vec{x} : \vec{S})\{e\} : T : \mathit{ok\ in}\ C}\ \text{T-METH}$$

$$\frac{K = C(\vec{f} : \vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\} \qquad \vdash \vec{M} : \mathit{ok\ in}\ C}{\vdash \mathsf{class}\ C\{\vec{f} : \vec{T}; K; \vec{M}\} : \mathit{ok}}\ \text{T-CLASS}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash p\langle e \rangle : \mathit{ok}}\ \text{T-THREAD} \qquad \frac{}{\Gamma \vdash \mathbf{0} : \mathit{ok}}\ \text{T-EMPTY} \qquad \frac{\Gamma_1 \vdash P_1 : \mathit{ok} \qquad \Gamma_2 \vdash P_2 : \mathit{ok}}{\Gamma_1, \Gamma_2 \vdash P_1 \parallel P_2 : \mathit{ok}}\ \text{T-PAR}$$

Table 8.2: The underlying type system

of the class's methods, using T-METH for each individual one). A thread $p\langle e \rangle$ is well-typed, if the expression e it evaluates is (cf. rule T-THREAD). Rules T-EMPTY and T-PAR assure that a program is well-typed if all its threads are.

8.2.3 Semantics

This section describes the operational semantics of TFJ with some adaptations at two different levels: a local and a global semantics. We use let-constructs, a simple rewriting formulation, instead of using *evaluation contexts* of [110]. Second, the operational semantics there uses *labelled* transitions (for technical reasons). For the soundness of the type system, the labels are irrelevant, so we omit them. The local semantics is given in Table 8.3. These local rules deal with the evaluation of *one* single *expression/thread* and reduce configurations of the form $E \vdash e$. Thus, local transitions are of the form

$$E \vdash e \to E' \vdash e' \,, \tag{8.2}$$

where e is one expression and E a *local environment.* At the local level, the relevant commands only concern the current thread and consist of reading, writing, invoking a method, and creating a new object.

Definition 1 (Local environment)**.** *A* local environment *E of type $LEnv$ is a finite sequence of the form $l_1{:}\varrho_1, \dots l_k{:}\varrho_k$, i.e., of pairs of transaction labels l_i and a corresponding* log *ϱ_i. We write $|E|$ for the size of the local environment (number of pairs $l{:}\varrho$ in the local environment).*

Transactions are identified by labels l, and as transactions can be nested, a thread can execute "inside" a number of transactions. So, the E in the above definition is ordered, with e.g. l_k to the right refers to the inner-most transaction, i.e., the one most recently started and commiting removes bindings from right to left. The number $|E|$ of a thread represents the nesting depth of the thread, i.e., how many transactions the thread has started but not yet committed. The corresponding *logs* ϱ_i can, in a first approximation, be thought of as "local copies" of the heap including bindings from references to objects. The *log* ϱ_i keeps track of changes of the threads actions concerning transaction l_i. The exact structure of such environments and the logs have no influence on our static analysis, and indeed, the environments may be realized in different ways (e.g., [110] gives two different flavors, a "pessimistic", lock-based one and an "optimistic" one). Relevant for our type and effect system will be only a number of *abstract properties* of the environments, formulated in Definition 7 later. The operational rules are formulated exploiting the let-construct/sequential composition, and the restricted form of (abstract) syntax. The syntax for the conditional construct from Table 8.1, e.g., insists that the boolean condition is already evaluated (i.e., either a boolean value or value/reference to such a value), and the R-COND-rules apply when the previous evaluation has yielded already true, resp. false. We use the let-construct to unify sequential composition, local variables, and handing over of values in a sequential composition, and rule R-LET basically expresses associativity of the sequential composition, i.e., ignoring the local variable declarations, it corresponds to a step from $(e_1; e); e'$ to $e_1; (e; e')$. Note further that the left-hand side for all local rules (and later the global ones) insists that the top-level construct is a let-construct. That is assured during run-time inductively by the form of the initial thread and the restiction on our syntax. The first two rules deal with the basic evaluation based on substitution and specifying a left-to-right evaluation (cf.R-RED and R-LET). The two R-COND-rules deal with conditionals in an obvious way. Unlike the first four rules, the remaining ones do access the heap. Thus, in the premises of these rules, the local environment E is consulted to look up object references and then *changed* in the step. The access and update of E is given abstractly by corresponding access functions $read$, $write$, and $extend$ (which look-up a reference on the heap, update a reference, resp. allocate an entry for a new reference on the heap). The details can be found in [110] but note that also the $read$-function used in the rules actually *changes* the environment from E to E' in the step. The reason is that in a transaction-based implementation, read-access to a variable may be *logged*, i.e., remembered appropriately, to be able to detect conflicts and to do a roll-back if the transaction fails. This logging may change the local environment. The premises assume the class table is given implicitly where $\mathit{fields}(C)$ looks up fields of class C and $mbody(m, C)$ looks up the method m of class C. So, field look-up in R-LOOKUP works as follows: consulting the local environment E, the $read$-function looks up the object referenced by r; the object is $C(\vec{v})$, i.e., it's an instance of class C, and its fields carry the values $\vec{v}$. The (run-time) type C of the object is further used to determine the fields $\vec{f}$ (using of the object the object referenced by r, *fields* finds the fields of the object referenced by r, and the step replaces the field access $r.f_i$ by the corresponding value v_i. Field update in rule R-UPD works similarly, again using $read$ to look up the objects, and additionally using $write$ to write the value v' back into the local environment, thereby changing E' to E'' (again, the exact details of the function are left abstract). The function $mbody$ in the rule R-CALL for method invocation gives back the method's formal parameters $\vec{x}$ and the method body, and invocation involves substituting $\vec{x}$ by the actual parameters $\vec{v}$ and substituting this by the object's identity r. Rule R-NEW, finally, takes care of object creation, using a fresh object identity r to refer to the new instance $C(\vec{v})$, which has all fields set to $\vec{v}$. The function $extend$ in that rule extends E by binding the fresh reference r to the newly created instance.

The five rules of the *global* semantics are given in Table 8.4. The semantics works on configurations of the following form:

$$\Gamma \vdash P\ , \tag{8.3}$$

where P is a *program* and Γ is a global environment. Besides that, we need a special configuration $error$ representing an error state. Basically, a program P consists of a number of threads evaluated in parallel (cf. Table 8.1), where each thread corresponds to one expression, whose evaluation is described by the local rules. Now that we describe the behavior of a number of (labeled) threads $t\langle e\rangle$, we need one E for each thread t. This means, Γ is a "sequence" (or rather a set) of $t{:}E$ bindings where t is the name of a thread and E is its corresponding local environment.

Definition 2 (Global enviroment)**.** *A* global environment Γ *of type* $GEnv$ *is a finite mapping, written as* $p_1{:}E_1, \dots p_k{:}E_k$*, from threads names* p_i *to local environments* E_i *(the order of bindings does not play a role, and each thread name can occur at most once).*

So global steps are of the form:

$$\Gamma \vdash P \Longrightarrow \Gamma' \vdash P' \quad \text{or} \quad \Gamma \vdash P \Longrightarrow error\ . \tag{8.4}$$

$$E \vdash \texttt{let}\ x : T = v\ \texttt{in}\ e \rightarrow E \vdash e[v/x] \qquad \text{R-RED}$$

$$E \vdash \texttt{let}\ x_2 : T_2 = (\texttt{let}\ x_1 : T_1 = e_1\ \texttt{in}\ e)\ \texttt{in}\ e' \rightarrow E \vdash \texttt{let}\ x_1 : T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2 : T_2 = e\ \texttt{in}\ e') \qquad \text{R-LET}$$

$$E \vdash \texttt{let}\ x : T = (\texttt{if}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2)\ \texttt{in}\ e \rightarrow E \vdash \texttt{let}\ x : T = e_1\ \texttt{in}\ e \qquad \text{R-COND}_1$$

$$E \vdash \texttt{let}\ x : T = (\texttt{if}\ \texttt{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2)\ \texttt{in}\ e \rightarrow E \vdash \texttt{let}\ x : T = e_2\ \texttt{in}\ e \qquad \text{R-COND}_2$$

$$\frac{read(r, E) = E', C(\vec{v}) \quad fields(C) = \vec{f}}{E \vdash \texttt{let}\ x{:}T = r.f_i\ \texttt{in}\ e \rightarrow E' \vdash \texttt{let}\ x{:}T = u_i\ \texttt{in}\ e}\ \text{R-LOOKUP}$$

$$\frac{read(r, E) = E', C(\vec{v}) \quad write(r \mapsto C(\vec{v}) \downarrow_i^{v'}, E') = E''}{E \vdash \texttt{let}\ x{:}T = r.f_i := v'\ \texttt{in}\ e \rightarrow E'' \vdash \texttt{let}\ x{:}T = v'\ \texttt{in}\ e}\ \text{R-UPD}$$

$$\frac{read(r, E) = E', C(\vec{v}') \quad mbody(m, C) = (\vec{x}, e)}{E \vdash \texttt{let}\ x{:}T = r.m(\vec{v})\ \texttt{in}\ e' \rightarrow E' \vdash \texttt{let}\ x : T = e[\vec{v}/\vec{x}][r/\mathsf{this}]\ \texttt{in}\ e'}\ \text{R-CALL}$$

$$\frac{r\ fresh \quad E' = extend(r \mapsto C(\vec{v}), E)}{E \vdash \texttt{let}\ x{:}T = \mathsf{new}\ C(\vec{v})\ \texttt{in}\ e \rightarrow E' \vdash \texttt{let}\ x = r\ \texttt{in}\ e}\ \text{R-NEW}$$

Table 8.3: Semantics (local)

$$\frac{E \vdash e \rightarrow E' \vdash e' \qquad \Gamma = \Gamma', p{:}E \qquad \mathit{reflect}(p, E', \Gamma) = \Gamma'}{\Gamma \vdash P \parallel p\langle e\rangle \Longrightarrow \Gamma' \vdash P \parallel p\langle e'\rangle} \text{ G-PLAIN}$$

$$\frac{p' \mathit{\ fresh} \qquad \mathit{spawn}(p, p', \Gamma) = \Gamma'}{\Gamma \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathsf{spawn}\, e_1\ \mathtt{in}\ e_2\rangle \Longrightarrow \Gamma' \vdash P \parallel p\langle \mathtt{let}\ x : T = ()\ \mathtt{in}\ e_2\rangle \parallel p'\langle e_1\rangle} \text{ G-SPAWN}$$

$$\frac{l \mathit{\ fresh} \qquad \mathit{start}(l, p, \Gamma) = \Gamma'}{\Gamma \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathsf{onacid}\ \mathtt{in}\ e\rangle \Longrightarrow \Gamma' \vdash P \parallel p\langle \mathtt{let}\ x : T = ()\ \mathtt{in}\ e\rangle} \text{ G-TRANS}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma'', p{:}E \qquad E = E', l{:}\varrho \qquad \mathit{intranse}(l, \Gamma) = \vec{p} = p_1 \ldots p_k \\ \mathit{commit}(\vec{p}, \vec{E}, \Gamma) = \Gamma' \quad p_1{:}E_1, p_2{:}E_2, \ldots p_k{:}E_k \in \Gamma \quad \vec{E} = E_1, E_2, \ldots, E_k\end{array}}{\Gamma \vdash P \parallel \prod_{i=1}^{k} p_i\langle \mathtt{let}\ x{:}T_i = \mathsf{commit}\ \mathtt{in}\ e_i\rangle \Longrightarrow \Gamma' \vdash P \parallel \prod_{i=1}^{k} p_i\langle \mathtt{let}\ x{:}T_i = ()\ \mathtt{in}\ e_i\rangle} \text{ G-COMM}$$

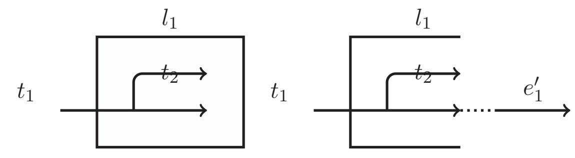
$$\frac{\Gamma = \Gamma'', p{:}E \qquad E = \emptyset}{\Gamma \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathsf{commit}\ \mathtt{in}\ e\rangle \Longrightarrow \mathit{error}} \text{ G-COMM-ERROR}$$

Table 8.4: Semantics (global)

As for the local rules, the formulation of the global steps makes use of a number of functions accessing and changing the (this time global) environment. As before, those functions are left abstract and only later we will formalize abstract properties that Γ and E considered as abstract data types must satisfyin order to achieve soundness of the static analysis. Rule G-PLAIN simply *lifts* a local step to the global level, using the reflect-operation, which roughly makes local updates of a thread globally visible. Rule G-SPAWN deals with starting a thread. The next three rules treat the two central commands of the calculus, those dealing directly with the transactions. The first one G-TRANS covers onacid, which starts a transaction. The start function creates a new label l in the local environment E of thread p. The two rules G-COMM and G-COMM-ERROR formalize the successful commit resp. the failed attempt to commit a transaction. In G-COMM, the label of the transaction l to be committed is found (right-most) in the local context E. Furthermore, the function $\mathit{intranse}(l, \Gamma)$ finds the identities $p_1 \ldots p_k$ of all concurrent threads in the transaction l and which all join in the commit. Note that, by definition of $\mathit{intranse}$, the process p in the premise must be one of the p_i's. In the conclusion of the rule, $\prod_{i=1}^{k}$ builds the parallel composition of the k mentioned processes. In the erroneous case of G-COMM-ERROR, the local environment E is empty; i.e., the thread executes outside of any transactions, which constitutes an error.

In the following, we use the following definitions. Let TrName be the type of transaction labels. Given a local environment E, the function $l : \mathit{LEnv} \rightarrow \mathit{List\ of\ TrName}$ is defined inductively as follows: $l(\epsilon) = \epsilon$, and $l(l{:}\_, E) = l, l(E)$. Overloading the definition, we lift the function straightforwardly to global environments (with type $l : \mathit{TName} \times \mathit{GEnv} \rightarrow \mathit{List\ of\ TrName}$), s.t. $l(p, (p{:}E), \Gamma) = l(E)$. We furthermore define the length of a local environment, written $|E|$, as the number of bindings in E.

The first part of this definition, extracting the list of transaction labels from a local environment E is a straightforward projection, simply extracting the sequence of transaction labels.



(a) Multi-threaded transaction (b) Sequential composition

Figure 8.1: Transactions and multi-threading

As for the *order* of the transactions, as said: the most recent, the innermost transaction label is to the right.

8.3 The effect system

The effect system assures that starting and committing transactions is done "properly", in particular to avoid committing when outside a transaction, which we call *commit errors*. To catch commit errors at compile time, the system keeps track of onacids and commits; we refer to the number of onacids minus the number of commits as the *balance*. For instance, for an expression $e = \text{onacid};\ e_1;\ \text{commit};\ \text{commit}$, the *balance* equals $1 - 2 = -1$. An execution of a thread is *balanced,* if there are no pending transactions, i.e., if the balance is 0. The situation gets slightly more involved when dealing with multi-threading. TFJ supports not only nested transactions, but *multi-threaded* transactions: inside one transaction there may be more than one thread active at a time. Due to this internal concurrency, the effect of a transaction may be non-deterministic. Figure 8.1 shows a simple situation with two threads p_1 and p_2, where p_1 starts a transaction with the label l_1 and spawns a new thread p_2 inside the transaction. An example expression resulting in the depicted behavior of Figure 8.1b is $e_1 = \text{onacid}; \text{spawn}\ e_2; e_1'$, where e_1 is the expression evaluated by thread p_1, and e_2 by the freshly created p_2. In TFJ's concurrency model, to terminate the parent transaction l_1, both p_1 and p_2 must *join* via a common commit. To keep track over the number of open and yet uncommitted transactions, we must take into account that e_2 and the rest e_1' of the original thread are executed in parallel, and furthermore, that when executing e_2 in the new thread p_2, one onacid has already been executed by p_1, namely before the spawn-operation. Hence, we need to keep track of the balance not just for the thread under consideration, but take into account the balance of the newly created threads, as well. Even if a spawning thread and a spawned thread run in parallel,[2] the situation wrt. the analysis is not symmetric. More precisely, the current thread of control plays a specific role when it comes to sequential composition of expressions. Consider the expression $\text{onacid}; \text{spawn}\ e_2; e_1'$. The first expression is depicted in Figure 8.1b, where the "open box" represents the transaction started by the first onacid. Considering the balance for the left of $\text{onacid}; \text{spawn}\ e_2$, the balance for both "threads" after execution amounts to $+1$, i.e., both threads are executing inside one enclosing transaction (assuming that e_2 itself does not start or end any transactions). When calculating the combined effect for $\text{onacid}; \text{spawn}\ e_2; e_1'$, the balance value of onacid is treated differently from the one of e_2, as the control flow of the sequential composition connects the trailing e_1' with onacid, but not with the thread of e_2(indicated by the

[2]Both the original thread and the freshly spawned one run without preference or priority to either one. Indeed, in the semantics, the information which threads is the father of which other thread is not maintained.

dotted line in Figure 8.1b). I.e., the analysis of the sequentially composed expression calculates the *sum* of the balance of onacid and of e_1' cumulatively.

To sum up: to determine the effect in terms of the balance, we need to calculate the balance for *all* threads potentially concerned, which means for the thread executing the expression being analysed plus all threads (potentially) spawned during that execution. From all threads, the one which carries the expression being evaluated plays a special role, and is treated specially. Therefore, we choose a pair of an integer n and a (finite) multi-set S of integers to represent the effect after evaluating an expression as follows:

$$n, S : \mathsf{Int} \times (\mathsf{Int} \to \mathsf{Nat}) \; . \tag{8.5}$$

The integer n represents the balance of the thread of the given expression, the multi-set the balance numbers for the threads potentially spawned by the expression. The multi-set can be seen as a function of type Int $\to$ Nat (the multi-set's characteristic function), and we write $dom(S)$ for the set of elements of S, ignoring their multiplicity. As an example: we use also the set-like notation $\{-3, 1, 1, 2\}$ to represent the finite mapping $-3 \mapsto 1, 1 \mapsto 2, 2 \mapsto 1$ (and all other integers to 0). We write $\emptyset$ for the empty multi-set, $\cup$ for the multi-set union. Note that multi-set union is defined by building the maximum of the characteristic functions, i.e., $\{1, 1, 2\} \cup \{1, 2, 2, 2\} = \{1, 1, 2, 2, 2\}$. As a further operation, we use "addition" and "subtraction" of such multisets and integers illustrated on a small example: $\{-3, 1, 1, 2\} + 5$ gives $\{2, 6, 6, 7\}$. Based on S, we know how many newly created threads with their corresponding balances in the current expression, including threads with the same balance. The judgements of the analysis are thus of the following form:

$$n_1 \vdash e :: n_2, S \; , \tag{8.6}$$

which reads as: starting with a balance of n_1, executing e results in a balance of n_2 and the balances for new threads spawned by e are captured by S. The balance for the new threads in S is calculated *cumulatively*; i.e., their balance includes n_1, the contribution of e before the thread is spawned, plus the contribution of the new thread itself.

The effect system is given in Table 8.5. For clarity, we do not integrate the effect system with the underlying type system of Section 8.2.2. Instead, we concentrate on the effects in isolation. Variables, the unit value, field lookup, and object creation have no effect (cf. T-VAR, T-UNIT, T-LOOKUP, and T-NEW in Table 8.5), so they leave the balance unchanged and since no threads are generated, the multi-set of balances is empty. A field update has no effect (cf. T-UPD), as we require that the left- and the right-hand side of the assignment are already evaluated. In contrast, the two dual commands of onacid and commit have the expected effect: they simply increase, resp. decrease the balance by one (cf. T-ONACID and T-COMMIT). A class declaration (cf. T-CLASS) is correct if the effect of all of its methods corresponds to their declaration, which is looked up in the class table. Rule T-METH deals with method declarations. In this rule, we require that all spawned threads in the method body must have the balance 0 after evaluating the expression e, that the balance of the method itself has the form $n_1 \to n_2$ where n_1 is interpreted as pre-condition, i.e., it is safe to call the method *only* in a state where the balance is at least n_1. The number n_2 as the post-condition corresponds to the balance after exiting the method, when called with balance n_1 as pre-condition. The precondition n_1 is needed to assure that at the call-sites the method is only used where the execution of the method body does not lead to a negative balance (see also the T-CALL-rules below). Rule T-SUB captures a notion of *subsumption* where by $S_1 \leq S_2$ we mean the subset relation on multi-sets.[3] In a let-expression (cf. T-LET), representing sequential composition, the effects are accumulated. Creating a new thread by executing spawn e does not change the balance of the executing thread (cf. T-SPAWN). The spawned expression e in the new thread is analyzed starting with the same balance n in its pre-state. The resulting balance n' of the new thread is given back in the conclusion as part of the balances of the spawned threads, i.e., as part of the multi-set. For conditionals $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ (cf. T-COND), the boolean condition v does not change the balance, and the rule insists that the two branches e_1 and e_2 agree on a balance n'.

For method calls, we distinguish two situations (cf. T-CALL$_1$ and T-CALL$_2$), depending on whether the method being called creates new threads or not. In both cases, the class C in the premise is determined by the type system (cf. the rule T-CALL there) resp. by the combination of the type and the effect system; for clarity we presented the type system first in isolation and we do not repeat the typing part here. In the case of T-CALL$_2$, the multi-set of balances for method m in class C is required to be empty by the third premise of the rule. In that situation, the precondition of the method can be interpreted in a "loose" manner: the current balance n in the state before the call must be *at least* as big as the pre-condition n'_1.[4] If, however, the method may spawn a new thread (cf. T-CALL$_1$), the pre-condition is interpreted *strictly*, i.e., we require $n = n'_1$ (with this equality, T-CALL$_1$ could be simplified; we chose this representation to stress the connection with T-CALL$_2$, where $n > n'_1$). Allowing the loose interpretation also in that situation would make the method callable in different levels of nestings at the caller side; however, only exactly *one* level actually is appropriate, as with concurrent threads inside a transaction, all threads must join in a commit to terminate the transaction (remember also the explanations concerning Figure 8.1). A thread $t\langle e\rangle$ is well-typed (cf. T-THREAD), if the expression has balance 0 after termination, starting with a balance corresponding to the length $|E|$ of the local environment E. We use *ok* to indicate that the thread is well-typed, i.e., without commit-error. This balance in the pre-state corresponds to the level of nesting inside transactions, the thread $p\langle e\rangle$ currently executes in. A program is well typed, if all threads in the system are well-typed (cf. T-PAR). In the conclusion of that rule, by writing Γ_1, Γ_2, we implicitly require that Γ_1 and Γ_2 are disjoint in the sense that no thread name occurs in both Γ's. We illustrate the system with the following two examples: the first one deals with multithreading, and the second one concentrates on method calls.

Example 3. *The following derivation applies the effect system to the expression* $e_1; \mathsf{spawn}(e_2; \mathsf{spawn}\, e_3); e_4 :: n_4, \{n_2, n_3\}$*, when starting with a balance of* 0.

$$\dfrac{\dfrac{}{0 \vdash e_1 :: n'_1, \{\}} \qquad \dfrac{\dfrac{\dfrac{n'_1 \vdash e_2 :: n_2, \{\} \qquad \dfrac{n_2 \vdash e_3 :: n_3, \{\}}{n_2 \vdash \mathsf{spawn}\, e_3 :: n_2, \{n_3\}}}{n'_1 \vdash (e_2; \mathsf{spawn}\, e_3) :: n_2, \{n_3\}}}{n'_1 \vdash \mathsf{spawn}(e_2; \mathsf{spawn}\, e_3) :: n'_1, \{n_2, n_3\}} \qquad n'_1 \vdash e_4 :: n_4, \{\}}{n'_1 \vdash \mathsf{spawn}(e_2; \mathsf{spawn}\, e_3); e_4 :: n_4, \{n_2, n_3\}}}{0 \vdash e_1; \mathsf{spawn}(e_2; \mathsf{spawn}\, e_3); e_4 :: n_4, \{n_2, n_3\}}$$

The derivation demonstrates sequential composition and thread creation with a starting balance of 0 for simplicity. Remember that sequential composition $e_1; e_2$ *is syntactic sugar for* $\texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2$,

[3]The non-structural rule of subsumption makes the system non syntax-directed. To turn it to an algorithm, one would have to disallow subsumption and derive a minimal multiset instead.

[4]To keep the situations where rules T-CALL$_1$ and T-CALL$_2$ apply *disjoint*, we actually require $n > n'_1$, not $n \geq n'_1$ here.

where x does not occur free in e_2; i.e., assume that the expressions $e_1, \ldots e_4$ themselves have the following balances $0 \vdash e_i :: n'_i, \{\}$, which implies (cf. Lemma 5 below):

$$\begin{aligned} n'_1 &\vdash e_2 :: n'_1 + n'_2 = n_2, \{\} \\ n_2 &\vdash e_3 :: n_2 + n'_3 = n_3, \{\} \\ n'_1 &\vdash e_4 :: n'_1 + n'_4 = n_4, \{\} \end{aligned}$$

Example 4. *Assume the following code fragment:*

```
...
void n(){ onacid; m(10); }

void m(i){      commit;
                if     (i ≤ 0)
                then   onacid;
                else   ....; onacid; this.m(i-1); }
void main(){ n();  commit; }
```

First observe that a program using that methods, when being executing shows no commit-errors (concerning the shown fragment of code). Method m calls itself recursively and the two branches of the conditional in its body both execute one onacid *each. Especially, method m is called (in this fragment) only via method n, especially after n has performed an* onacid*, i.e., m is called* inside *one transaction. If m were called* outside *a transaction it would result in an error, as the body of m starts by executing a commit-statement. In our effect system, method m can be declared as of effect $1 \to 1$, which expresses not only that the body of m does not change the balance, but that as a precondition, it must be called* only *from call-sites where the balance is ≥ 1, as is the case in the body of n (cf. also* T-METH *and* T-CALL*). So the declarations of the two shown methods are of the form*[5]

$$n() : \texttt{Unit} \to \texttt{Unit}, 0 \to 1 \quad \textit{and} \quad m(i) : \mathsf{Int} \to \texttt{Unit}, 1 \to 1$$

For recursive calls, an effect like $1 \to 1$ can be interpreted as loop invariant*: the body of the method must not change the balance to be well-typed. However, not every method needs to be balanced; the non-recursive method n is one example which (together with the call to m) has a net-balance of 1. As an aside: if the method would contain the spawning of a new thread, the recursive execution would lead to an unboundenden number of new threads; for the analysis, this does not pose a problem, as it analyses the method body in isolation (whithout "following" the recursion, as is customary for type analyses).* □

Lemma 5. *If $n_1 \vdash e :: n_2, S$, then $n_1 + n \vdash e :: n_2 + n, S$.*

Proof. By straightforward induction over the length of derivation. □

8.4 Soundness of the type and effect system

Next we prove that the type and effect system does what it is designed to do, namely absence of commit errors. The main part of the proof is preservation of well-typedness under reduction, also called *subject reduction.*

Lemma 6 (Subject reduction (local))**.** *Let $n = |E|$. If $n \vdash e :: n', S'$ and $E \vdash e \to E' \vdash e'$, then $|E'| = n$ and $n \vdash e' :: n', S'$.*

[5]In the code fragment which is intended to resemble more concrete syntax we use `void` as it corresponds to the conventions of Java. In the formal analysis, as said, we use type `Unit` instead.

Proof. First observe that by the properties of $read$, $write$, and $extend$, $|E| = |E'|$. Proceed by case analysis over the operational rules of Table 8.3. The cases R-LOOKUP, R-UPD, R-CALL, and NEW are immediate. For R-COND$_1$, we need subsumption (R-COND$_2$ works analogously):

Case: R-COND$_1$

In this case, the expression e before the step is of the form `if` $v = v$ `then` e_1 `else` e_2 and the step is given as $E \vdash e \rightarrow E' \vdash e_1$. Note that $E' = E$, and hence $|E'| = n$. Concerning the typing, we are given $n \vdash$`if` $v = v$ `then` e_1 `else` e_2 $:: n', S'$, which implies by the premises of rule T-COND that $n \vdash e_1 :: n', S_1$ and $n \vdash e_2 :: n', S_2$ with $S' = S_1' \cup S_2'$. The result follows by subsumption (rule T-SUB). □

The global semantics accesses and changes the global environments Γ. These manipulations are captured in various functions, which are kept "abstract" in this semantics (as in [110]). To perform the subject reduction proof, however, we need to impose certain requirements on those functions:

Definition 7. *The properties of the abstract functions are specified as follows:*

1. *The function reflect satisfies the following condition: if reflect*$(p, E, \Gamma) = \Gamma'$ *and* $\Gamma = p_1{:}E_1, \ldots, p_k{:}E_k$*, then* $\Gamma' = p_1{:}E_1', \ldots, p_k{:}E_k'$ *with* $|E_i| = |E_i'|$ *(for all i).*
2. *The function* $spawn$ *satisfies the following condition: Assume* $\Gamma = p : E, \Gamma''$ *and* $p' \notin \Gamma$ *and* $spawn(p, p', \Gamma) = \Gamma'$*, then* $\Gamma' = \Gamma, p'{:}E'$ *s.t.* $|E| = |E'|$.
3. *The function start satisfies the following condition: if start*$(l, p_i, \Gamma) = \Gamma'$ *for a* $\Gamma = p_1{:}E_1, \ldots, p_i{:}E_i, \ldots, p_k{:}E_k$ *and for a fresh l, then* $\Gamma' = p_1{:}E_1, \ldots, p_i{:}E_i', \ldots, p_k{:}E_k$*, with* $|E_i'| = |E_i| + 1$.
4. *The function* $intranse$ *satisfies the following condition: Assume* $\Gamma = \Gamma'', p{:}E$ *s.t.* $E = E', l{:}\varrho$ *and* $intranse(l, \Gamma) = \vec{p}$*, then*
   (a) $p \in \vec{p}$ *and*
   (b) for all $p_i \in \vec{p}$ *we have* $\Gamma = \ldots, p_i : (E_i', l{:}\varrho_i), \ldots$.
   (c) for all threads p' *with* $p' \notin \vec{p}$ *and where* $\Gamma = \ldots, p'{:}(E', l'{:}\varrho'), \ldots$*, we have* $l' \neq l$.
5. *The function* $commit$ *satisfies the following condition: if* $commit(\vec{p}, \vec{E}, \Gamma) = \Gamma'$ *for a* $\Gamma = \Gamma'', p{:}(E, l{:}\varrho)$ *and for a* $\vec{p} = intranse(l, \Gamma)$ *then* $\Gamma' = \ldots, p_j{:}E_j', \ldots, p_i{:}E_i', \ldots$ *where* $p_i \in \vec{p}, p_j \notin \vec{p}$*,* $p_j{:}E_j \in \Gamma$*, with* $|E_j'| = |E_j|$ *and* $|E_i'| = |E_i| - 1$.

Lemma 8 (Subject reduction)**.** *If* $\Gamma \vdash P : ok$ *and* $\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$*, then* $\Gamma' \vdash P' : ok$*.*

Proof. Proceed by case analysis on the rules of the operational semantics from Table 8.4 (except rule G-COMMERROR for commit errors). For simplicity (and concentrating on the effect, not the values of expressions) we use ; for sequential composition in the proof, and not the more general `let`-construct.

Case: G-PLAIN

From the premises of the rule, we get for the form of the program that $P = P'' \parallel t\langle e\rangle$, furthermore for p's local environment $\Gamma \vdash p : E$ and $E \vdash e \rightarrow E' \vdash e'$ as a local step. Well-typedness $\Gamma \vdash P : ok$ implies $n \vdash e :: n', S'$ for some n' and S', where $n = |E|$. By subject reduction for the local steps (Lemma 6) $n \vdash e' :: n', S'$. By the properties of the $reflect$-operation, $|E'| = n$, so we derive for the thread p

$$\frac{n \vdash e' :: 0, \{0, \ldots\}}{\Gamma', p{:}E' \vdash p\langle e'\rangle : ok}$$

from which the result $\Gamma' \vdash P'' \parallel p\langle e'\rangle : \mathit{ok}$ follows (using T-PAR and the properties of *reflect* from Definition 7.1).

Case: G-SPAWN

In this case, $P = P'' \parallel p\langle \mathsf{spawn}\, e_1; e_2\rangle$ and $P' = P'' \parallel p\langle (); e_2\rangle \parallel p'\langle e_1\rangle$ (from the premises of G-SPAWN). The well-typedness assumption $\Gamma \vdash P : \mathit{ok}$ implies the following sub-derivation:

$$\dfrac{\dfrac{\dfrac{n \vdash e_1 : 0, S_1}{n \vdash \mathsf{spawn}\, e_1 : n, S_1 \cup \{0\}} \quad n \vdash e_2 : 0, S_2}{n \vdash \mathsf{spawn}\, e_1; e_2 : 0, \{0, \dots\}}}{p{:}E \vdash p\langle \mathsf{spawn}\, e_1; e_2\rangle : \mathit{ok}} \tag{8.7}$$

with $S_1 = \{0, \dots\}$ and $S_2 = \{0, \dots\}$. By the properties of *reflect*, the global environment Γ' after the reduction step is of the form $\Gamma, p'{:}E'$ where p' is fresh and $|E'| = |E|$ (see Definition 7.2). So we can derive

$$\dfrac{p{:}E \vdash p\langle (); e_2\rangle : \mathit{ok} \qquad \dfrac{n \vdash e_1 : 0, \{0, \dots\}}{p'{:}E' \vdash p'\langle e_1\rangle : \mathit{ok}}}{p{:}E, p'{:}E' \vdash p\langle (); e_2\rangle \parallel p'\langle e_1\rangle : \mathit{ok}}$$

The left sub-goal follows from T-THREAD,T-SEQ, T-UNIT, and the right sub-goal of the previous derivation (8.7). The right open sub-goal directly corresponds to the left sub-goal of derivation (8.7).

Case: G-TRANS

In this case, $P = P'' \parallel p\langle \mathsf{onacid}; e\rangle$ and $P' = P'' \parallel p\langle (); e\rangle$. The well-typedness assumption $\Gamma \vdash P : \mathit{ok}$ implies the following sub-derivation (assume that $|E| = n$):

$$\dfrac{\dfrac{n \vdash \mathsf{onacid} :: n+1, \emptyset \quad n+1 \vdash e :: 0, \{0, \dots\}}{n \vdash \mathsf{onacid}; e :: 0, \{0, \dots\}}}{p{:}E \vdash p\langle \mathsf{onacid}; e\rangle : \mathit{ok}} \tag{8.8}$$

For the global environment Γ' after the step, we are given $\Gamma' = \mathit{start}(l, p, \Gamma)$ from the premise of rule G-TRANS. By the properties of *start* from Definition 7.3, we have $\Gamma' = \Gamma'', p{:}E'$ with $|E'| = n + 1$. So with the help of right sub-goal of the previous derivation (8.8), we can derive for thread p after the step:

$$\dfrac{n+1 \vdash e :: 0, \{0, \dots\}}{p{:}E' \vdash p\langle e\rangle : \mathit{ok}}$$

Since furthermore the local environments of all other threads remain unchanged (cf. again Definition 7.3), the required $\Gamma' \vdash P' : \mathit{ok}$ can be derived, using T-PAR.

Case: G-COMM

In this case, $P = P'' \parallel \vec{p}\langle \mathsf{commit}; \vec{e}\rangle$ and $P' = P'' \parallel \vec{p}\langle \vec{e}\rangle$. The well-typedness assumption $\Gamma \vdash P : \mathit{ok}$ implies the following sub-derivation for thread p:

$$\dfrac{\dfrac{n \vdash \mathsf{commit} :: n-1, \emptyset \quad n-1 \vdash e_i : 0, \{0, \dots\}}{n \vdash \mathsf{commit}; e_i : 0, \{0, \dots\}}}{p_i{:}E_i \vdash p_i\langle \mathsf{commit}; e_i\rangle : \mathit{ok}} \tag{8.9}$$

For the global environment Γ' after the step, we are given $\Gamma' = \mathit{commit}(\vec{p}, \vec{E}, \Gamma)$ from the premise of rule G-TRANS, where $\vec{p} = \mathit{intranse}(l, \Gamma)$ and $\vec{E}$ are the corresponding local environments. By the properties of *commit* from Definition 7.5, we have for the local environments

$\vec{E}'$ of threads $\vec{p}$ after the step that $|E'_i| = n - 1$. So we obtain by T-THREAD, using the right sub-goal of derivation (8.9):

$$\frac{n - 1 \vdash e_i :: 0, \{0, \ldots\}}{p_i{:}E'_i \vdash p_i\langle e_i \rangle : ok}$$

For the threads $p_j\langle e_j \rangle$ different from $\vec{p}$, according to the Definition 7.5, we have $|E'_j| = |E_j|$ so $p_j{:}E'_j \vdash p_j\langle e'_j \rangle : ok$ straightforwardly. As a result, we have $\Gamma' \vdash P' : ok$. □

Lemma 9. *If $\Gamma \vdash P : ok$ then it is not the case that $\Gamma \vdash P \Longrightarrow error$.*

Proof. Let $\Gamma \vdash P : ok$ and assume for a contradiction that $\Gamma \vdash P \to error$. From the rules of the operational semantics it follows that $P = p\langle \mathsf{commit}; e \rangle \parallel P'$ for some thread t, where the step $\Gamma \vdash P \to error$ is done by p (executing the commit-command). Furthermore, the local environment E for the thread t is empty:

$$\frac{E = \emptyset}{\Gamma', p{:}E \vdash p\langle \mathsf{commit}; e \rangle \parallel P' \to error}\ \text{G-COMM}$$

To be well-typed, i.e., for the judgment $\Gamma \vdash p\langle \mathsf{commit}; e \rangle \parallel P' : ok$ to be derivable, it is easy to see that the derivation must contain $\Gamma', p{:}\emptyset \vdash p\langle \mathsf{commit}; e \rangle : n, S$ as sub-derivation (for some n and S). By inverting rule T-THREAD, we get that $0 \vdash \texttt{let commit in } e : 0, \{0, 0, \ldots\}$ is derivable (since $|E| = 0$). This is a contradiction, as the balance after commit would be negative (inverting rules T-LET and T-COMMIT). □

Corollary 10 (Well-typed programs are commit-error free)**.** *If $\Gamma \vdash P : ok$ then it is not the case that $\Gamma \vdash P \Longrightarrow^* error$,*

Proof. A direct consequence of the subject reduction Lemma 8 and Lemma 9. □

8.5 Conclusion

This work took the TFJ language design from [110] as starting point. That paper is not concerned with static analysis, but develops and investigates two different operational semantics for TFJ that assure transactional guarantees. As mentioned, however, the flexibility of the language may lead to run-time errors when executing a commit outside any transaction; we called such situations *commit-errors*. To statically prevent commit-errors, we presented a static type and effect system, which keeps track of the commands for starting and finishing transactions. We proved soundness of the type system.

In the following we discuss related work, and especially compare transactions in TFJ with the flexible ways of lock manipulation supported in Java 5, to consider our approach in that setting.

A comparison with explicit locks of Java The built-in support for concurrency control in Java is lock-based; each object comes equipped with a (re-entrant) lock, which can be used to specify synchronized blocks and, as a special case, synchronized methods. The lock can achieve mutual exclusion between threads that compete for the lock before doing something critical. Thus, the built-in, lock-based (i.e., "pessimistic") concurrency control in Java offers *lexically scoped* protection based on mutual exclusion. While offering basic concurrency control, the scheme has been criticized as too rigid, and consequently, Java 5 now additionally supports explicit locks with non-lexical scope. The `ReentrantLock` class and the `Lock` interface allow more freedom, offering explicit `lock` and `unlock` operations. Locking and unlocking can be compared, to some extent, to starting and committing a transaction, even if there are differences especially wrt. failure and progress properties. See e.g., [33] for a discussion of such differences. Besides the more behavioral differences, such as different progress guarantees or deadlocking behavior, the lock handling in Java 5 and the transactional model of TFJ differ in the following aspects, as relevant for the type analysis (cf. Table 8.6).

One basic difference is that we proposed a *static* scheme to catch commit errors, whereas in Java, improper use of locking and unlocking is checked at *run-time*. Both schemes, as mentioned, have all the flexibility of non-lexical scoping. The rest of Table 8.6 deals with the structure of protected areas (the transaction or the execution protected by a lock) and the connection to the threading model. One difference is that locks have an identity available at the program level, whereas transactions have not. Furthermore, locks and monitors in Java are *re-entrant*, i.e., one particular thread holding a lock can recursively re-enter a critical section or monitor. Re-entrance is not an issue in TFJ: a thread leaves a transaction by committing it (which terminates the transaction), hence re-entrance into the same transaction makes no sense. Transactions in TFJ can be nested. Of course, in Java, a thread can hold more than one lock at a time; however, the critical sections protected by locks do not follow a first-in-last-out discipline, and the sections are not nested as they are independent. For nested transactions in contrast, a commit to a child transaction is propagated to the surrounding parent transaction, but not immediately further, until that parent commits its changes in turn. Finally, TFJ allows multi-threaded transactions, whereas monitors and locks in Java are meant to ensure mutual exclusion. In particular, if an activity inside a monitor spawns a new thread, the new thread starts executing *outside* any monitor, in other words, a new thread holds *no* locks. In [133], we discuss the differences and similarities in more depth by comparing the analysis developed here with a corresponding one that deals with the safe use of a statically allocated number of locks.

Related work There have been a number of further proposals for integrating transactional features into programming languages. , e.g. AtomCaml [168], X10 [41], Fortress [19], Chapel [53]. The paper [2] presents the AME calculus, a calculus for *automatic mutual conclusion*, a concept proposed in [107]. The sequential core is a λ-calculus with references and imperative update, extended by the possibility to create asynchronous threads and means for atomic execution. Unlike other approaches, where the user is required to mark parts of the code intended for atomic execution, in AME, atomic execution is the default. For code parts where transactional behavior is not intended or possible (e.g., legacy code from libraries) can be marked as unprotected. A calculus and a proof method (implemented in the tool QED) for atomic actions is presented in [69]. For transactional languages, lexical scope for transactions, so called atomic blocks, have been proposed, using e.g., an `atomic`-construct or similar. Examples are Atomos [39], the AME calculus [2], and many proposals for software transactional memory [85, 192][53], but none of them deals with assuring statically proper use of the corresponding constructs. Besides, many early language designs, especially for data base programming, supported non-lexical scoping of the transactional constructs, cf. e.g. CICS [93], R* [129], Camelot [70], Argus [130], Quicksilver [87], Arjuna [174], Avalon [60]. A recent proposal to integrate software transactional memory into a full-fledged general purpose language is Clojure [96], an extension of Lisp. Static analysis is a well-established method to assure desired properties ranging from resource consumption (e.g., concerning memory, time ...), absence of deadlocks and race conditions. When dealing with concurrency, most static analyses (e.g., [35, 4]

. . .) [150]. . .) focus on avoiding data races and deadlocks, especially for multi-threaded Java programs. Static type systems have also been used to impose restrictions assuring transactional semantics, e.g. in [86, 2, 107]. A type system for *atomicity* is presented in [73, 72]. [32] [127]develops a type system for statically assuring proper lock handling for the JVM, i.e., on the level of byte code. Their system assures what is known as *structured locking*, i.e., (in our terminology), each method body is balanced as far as the locks are concerned, and at no point, the balance reaches below 0. Since the work does not consider non-lexical locking as in Java 5, the conditions apply *per method* only. Also the Rcc/Java type system tries to keep track of which locks are held (in an approximate manner), noting which field is guarded by which lock, and which locks must be held when calling a method. [148] presents a type and effect system for a transactional calculus to assure *strong* isolation, which assures non-interference of an atomic block even with code that is not marked as atomic. Slicing [176, 186] is one particular general static analysis and program optimization technique that has been applied (among many other uses) to analyze multi-threaded Java programs [88]. The analysis has been implemented in the context of the Bandera tool [50]. Especially *safe lock* analysis, supported e.g. by the Indus tool [163][106] as part of Bandera, is a static analysis that checks whether a lock is held indefinitely (in the context of multi-threaded Java). Software model checking is a prominent, alternative way to assure quality of software. The underlying techniques often are based on automatic (static) abstractions of the programs, albeit mostly not formulated as type systems. By using some form of abstraction (typically ignoring data parts and working on an abstract, automata-based representation), model checking can be used as a form of static analysis of concrete programs, as well (as opposed of full verification of a given "model"). The Blast analyzer [94] ("Berkeley Lazy Abstraction Software verification Tool")allows automatic verification for checking temporal safety properties of C programs (using counter-example guided abstraction refinement), and has been extended to deal with concurrent programs, as well [56]. Similarly, Java PathFinder is an automatic, model-checking tool (based on Spin) to analyze Java programs [89][90]. The *sat-based* static analyzer Saturn [193][171][16] has been used, amongst other thing to analyze C code (a Linux kernel) to check for *lock errors*.

Future work The work presented here can be extended to deal with more complex language features, e.g. when dealing with higher-order functions. In that setting, the effect part and its connection to the type system become more challenging. Furthermore, we plan to adopt the results for a different language design, more precisely for the language Creol [119], which is based on asynchronously communicating, active objects, in contrast to Java, whose concurrency is based on multi-threading. As discussed, there are similarities between lock-handling in Java 5 and the transactions as treated here. We plan to use similar techniques as explored here to give static guarantees for lock-based concurrency, as well. It is worthwhile to try to generalize the techniques also beyond the simple lock-based (or transaction based) synchronization discipline to more recently proposed ones such as chords [30], state classes [55], or non-uniform object behavior by temporal constraints as proposed in Jeeg [143]. Of practical relevance is to extend the system from type *checking* to type *inference*, potentially along the lines of [104].

Acknowledgements

We thank the anonymous reviewers for their helpful suggestions.

$$\frac{}{n \vdash x :: n, \emptyset}\ \textsc{T-Var} \qquad \frac{}{n \vdash () :: n, \emptyset}\ \textsc{T-Unit} \qquad \frac{}{n \vdash v.f :: n, \emptyset}\ \textsc{T-Lookup}$$

$$\frac{}{n \vdash \mathsf{new}\, C :: n, \emptyset}\ \textsc{T-New} \qquad \frac{n \vdash v_1 :: n, \emptyset \qquad n \vdash v_2 :: n, \emptyset}{n \vdash v_1.f_i := v_2 :: n, \emptyset}\ \textsc{T-Upd}$$

$$\frac{}{n \vdash \mathsf{onacid} :: n+1, \emptyset}\ \textsc{T-Onacid} \qquad \frac{n \geq 1}{n \vdash \mathsf{commit} :: n-1, \emptyset}\ \textsc{T-Commit}$$

$$\frac{K = C(\vec{f} : \vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\} \qquad mtype(C, \vec{M}) :: \vec{n}_1 \rightarrow \vec{n}_2, \vec{S} \qquad \vdash \vec{M} :: \vec{n}_1 \rightarrow \vec{n}_2, \vec{S}}{\vdash \mathsf{class}\ C\{\vec{f} : \vec{T}; K; \vec{M}\} :: ok}\ \textsc{T-Class}$$

$$\frac{n_1 \vdash e :: n_2, S \qquad S = \{0, \ldots\} \text{ or } S = \emptyset}{\vdash m(\vec{x} : \vec{T})\{e\} :: n_1 \rightarrow n_2, S}\ \textsc{T-Meth} \qquad \frac{n \vdash e :: n', S_1 \qquad S_1 \leq S_2}{n \vdash e :: n', S_2}\ \textsc{T-Sub}$$

$$\frac{n_0 \vdash e_1 :: n_1, S_1 \qquad n_1 \vdash e_2 :: n_2, S_2}{n_0 \vdash \mathtt{let}\ x : T = e_1\ \mathtt{in}\ e_2 :: n_2, S_1 \cup S_2}\ \textsc{T-Let} \qquad \frac{n \vdash e :: n', S}{n \vdash \mathsf{spawn}\, e :: n, S \cup \{n'\}}\ \textsc{T-Spawn}$$

$$\frac{n \vdash v :: n, \emptyset \qquad n \vdash e_1 :: n', S_1 \qquad n \vdash e_2 :: n', S_2}{n \vdash \mathtt{if}\ v\ \mathtt{then}\, e_1\ \mathtt{else}\ e_2 :: n', S_1 \cup S_2}\ \textsc{T-Cond}$$

$$\frac{n \vdash v :: n, \emptyset \qquad n \vdash v_i :: n, \emptyset \qquad mtype(C, m) :: n_1' \rightarrow n_2', S \qquad S \neq \emptyset \qquad n = n_1'}{n \vdash v.m(\vec{v}) :: n_2' + (n - n_1'), S + (n - n_1')}\ \textsc{T-Call}_1$$

$$\frac{n \vdash v :: n, \emptyset \qquad n \vdash v_i :: n, \emptyset \qquad mtype(C, m) :: n_1' \rightarrow n_2', \emptyset \qquad n > n_1'}{n \vdash v.m(\vec{v}) :: n_2' + (n - n_1'), \emptyset}\ \textsc{T-Call}_2$$

$$\frac{|E| \vdash e :: 0, \{0, 0, \ldots\}}{t{:}E \vdash p\langle e \rangle : ok}\ \textsc{T-Thread} \qquad \frac{\Gamma_1 \vdash P_1 : ok \qquad \Gamma_2 \vdash P_2 : ok}{\Gamma_1, \Gamma_2 \vdash P_1 \parallel P_2 : ok}\ \textsc{T-Par}$$

Table 8.5: Effect system

	Java 5.0	TFJ
when?	run-time	compile time
non-lexical scope	yes	yes
program level identity	yes	no
re-entrance	yes	no
nested transactions/critical sections	no	yes
internal multi-threading	no	yes

Table 8.6: Transactional Featherweight Java and explicit locks of Java

CHAPTER 12

Design issues in concurrent object-oriented languages and observability

12.1 Introduction

Compositionality is important in large and distributed systems as it allows programmers to build a larger system from smaller components. In general, a large system cannot be built from scratch by a single programmer or by a company. It often requires a cooperation from different organizations and from different places. In that setting, we need a good mechanism to compose components or replace a component with another one. In general, to replace a component with another, one needs to observe their behaviors to see whether two components are observably equal, if no context can see a difference. One of the most widely adopted solutions is to consider components as black-boxes, and only observe the interactions via their interfaces. So the question is that what can be observed or seen from the outside, from the "client code".

In an object-oriented setting, an open program interacts with its environment via method calls or message exchange. Besides message passing, of course, different communication and synchronization mechanisms exists (shared variable concurrency, multi-cast, black-board communication, publish and subscribe and many more). We concentrate here, however, on basic message passing using method calls. In that setting, the interface behavior of an open program C can be characterized by the set of all those message sequences (traces) t, for which there *exists* an environment E such that C and E exchange the messages recorded in t. Thereby we abstract away from any concrete environment, but consider only environments that are compliant to the language restrictions (syntax, type system, etc.). Consequently, interactions are not arbitrary traces $C \overset{t}{\Longrightarrow}$; instead we consider behaviors

$$C \parallel E \underset{\bar{t}}{\overset{t}{\Longrightarrow}} \acute{C} \parallel \acute{E} \tag{12.1}$$

where E is a *realizable* environment and trace $\bar{t}$ is complementary to t, i.e., each input is replaced by a matching output and vice versa. The notation $C \parallel E$ indicates that the component C runs in parallel with its environment or observer E. To account for the abstract environment ("there exists an E s.t. ...”), the open semantics is given in an *assumption-commitment* way:

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta} , \tag{12.2}$$

where Δ (as an abstract version of E) contains the *assumptions* about the environment, and dually Θ the *commitments* of the component. Abstracting away also from C gives a language characterization by the set of all possible traces between any component and any environment.

Such a behavioral interface description is relevant and useful for the following reasons. 1) The set of possible traces given this way is more restricted (and realistic) than the one obtained when ignoring the environments. When reasoning about the trace-based behavior of a component, e.g., in compositional verification, with a more precise characterization one can carry out stronger arguments. 2) When using the trace description for *black-box testing*, one can describe test cases in terms of the interface traces and then synthesize appropriate test drivers from it. Clearly, it makes no sense to specify impossible interface behavior, as in this case one cannot generate a corresponding tester. 3) A representation-independent behavior of open programs paves the way for a compositional semantics, a two-level semantics for the nested composition of program components. It allows furthermore optimization of components: only if two components show the same external, observable behavior, one can replace one for the other without changing the interaction with any environment. 4) The formulation gives insight into the semantic nature of the language, here, to different design choices concerning concurrency and object orientation. Some technical material underlying this paper can be found in [180]; here we concentrate on discussing the influence of various design choices from a more practical and global view.

In Section 12.2, we discuss closer observability and the problem of characterizing the interface behavior. Afterwards in Section 12.3 resp. 12.4, we discuss the influence of classes and inheritance, resp. of the concurrency model. In Section 12.5 we conclude by summarizing lessons learned from the theoretical approach in more practical terms.

12.2 Observable behavior

In this section, to give an intuitive understanding of the formal framework, we first sketch in Section 12.2.1 the object-oriented calculus which concentrates on the object-oriented features we are interested in; later we extend or modify it by inheritance, by two different concurrency models, and considering synchronization. Afterwards, Section 12.2.2 describes the steps of an open semantics, based on the ideas mentioned in the introduction.

12.2.1 An object-oriented, concurrent core calculus

The abstract syntax is given in Table 12.1 (where *run-time* syntax is underlined). The calculus is rather standard and a class-based variant similar to the object calculi of Abadi and Cardelli [3][81]. The syntax supports objects as instances of classes, local variables, and standard control constructs like conditionals; later we will add concurrency. Objects carry a name or reference, likewise classes and later threads, and via destructive field update, the model supports mutable heap and aliasing.

A *component* C is a collection of classes $c[\![M, F]\!]$, objects $o[c, M, F]$, and (for now) one single thread $\natural\langle e\rangle$, with empty component $\mathbf{0}$. The ν-binder is used for hiding and dynamic scoping, as known from the π-calculus [145]. An object o references the class c it instantiates, contains *embedded* the methods it supports plus the fields. The thread $\natural\langle e\rangle$ contains the running

$$\begin{array}{rcll}
C & ::= & \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid c[\![O]\!] \mid \underline{o[c,O]} \mid \underline{\natural\langle e\rangle} & \text{component} \\
O & ::= & M, F & \text{object} \\
M & ::= & m \;=\; \varsigma(n{:}T).\lambda(\vec{x}{:}\vec{T}).e, \ldots & \text{method suite} \\
F & ::= & f \;=\; fd, \ldots & \text{fields} \\
fd & ::= & v \mid \bot_c & \text{field} \\
e & ::= & v \mid \texttt{stop} \mid \texttt{let}\ x{:}T = e\ \texttt{in}\ e & \text{expressions} \\
 & \mid & \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ e & \\
 & \mid & v.m(\vec{v}) \mid v.f \mid v.f := v \mid \texttt{new}\ c & \\
v & ::= & x \mid n \mid () & \text{values} \\
n & ::= & o \mid c & \text{names}
\end{array}$$

Table 12.1: Syntax of an oo core calculus

code, basically as incarnation of method bodies "in execution". The expression e is basically a sequence of expressions, where the let-construct is used for sequencing and for local declarations. Sequential composition $e_1; e_2$ abbreviates `let` $x{:}T = e_1$ `in` e_2, where x does not occur free in e_2. The (closed) semantics can be given operationally in a standard way, describing steps of the form $C \xrightarrow{\tau} C'$, modulo standard algebraic laws for parallel composition (such as associativity and commutativity). Being closed, the steps of the semantics are labelled with an internal τ-label, only.

12.2.2 Characterizing the observable behavior

Whereas the closed semantics uses internal steps, the open semantics interacts with the environment via communication labels a, and the behavior can be characterized by sequences or traces t of such interaction labels. In an object-oriented setting, the message labels are categorized in method calls, returning the results, and ν-labels which communicate fresh names and which correspond to instantiate a new object from a class. In a concurrent setting later, also fresh thread names are created and communicated via ν-labels. The communication labels are either *incoming* (from the environment to the component) or dually *outgoing* (marked ? resp. !).

As said, without concrete environment, the *open* semantics uses *assumptions* as existential abstraction of all potential environments, and the interaction steps are of the following form:

$$\Delta \vdash C : \Theta \xrightarrow{a} \acute{\Delta} \vdash \acute{C} : \acute{\Theta}\ , \tag{12.3}$$

(cf. also the traces as sketched in equation (12.2)). In the step, Δ is the mentioned assumption context, and Θ a commitment context, describing dually relevant interface information about the component C. In software engineering, the terms *provided* and *required* interface are also used instead of commitments and assumptions. In the steps of the open semantics, the two contexts are used as follows: For *incoming* communication, originating from the environment, (mainly) the assumption context Δ is used to *check* whether there *exists* an environment that can send the incoming step (written $\Delta, \Theta \vdash a$ below). The information exchanged over the interface via the communication label a *updates* the (assumption and commitment) contexts in the step (written $\acute{\Delta}, \acute{\Theta} = \Delta, \Theta + a$ below).

The steps of equation (12.3) existentially abstract away from the environment but keep the component part C of the configuration concrete. Abstracting away in the same way from C as well gives a representation-independent characterization of interaction traces possible for

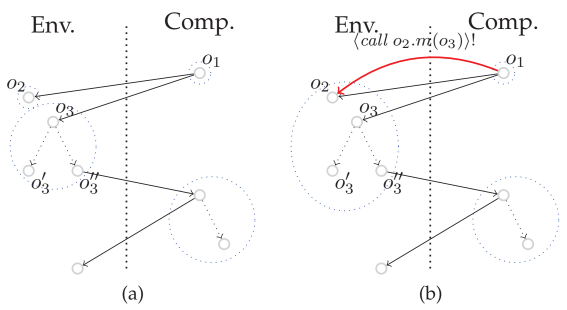


Figure 12.1: Connectivity

well-formed and well-typed open programs. The corresponding judgment

$$\Delta, \Theta \vdash r \rhd t : trace$$

captures the statement: "under assumptions Δ and given commitments Θ, and with history r, the further trace t is possible". The rule of equation (12.4) inductively formalizes the basic step of that judgment:

$$\frac{\Delta, \Theta \vdash a \qquad \acute{\Delta}, \acute{\Theta} = \Delta, \Theta + a \\ \acute{\Delta}, \acute{\Theta} \vdash r\ a \rhd t : trace \qquad \text{other conditions}}{\Delta, \Theta \vdash r \rhd a\ t : trace} \tag{12.4}$$

The details of the check $\Delta, \Theta \vdash a$ and the update $\Delta, \Theta + a$ as well as the "other conditions" depend on the design decisions concerning the language constructs. We describe the influence of classes, inheritance, two forms of concurrency and related synchronization mechanisms in the following.

12.3 Classes and inheritance

Since Simula [54], classes are a central concept in most object-oriented languages. Actually, classes combine different roles in programming languages: 1) they structure the code and offer an abstraction mechanism (as they are also used as type or at least implement an interface). 2) They offer a mechanism of code reuse, typically via inheritance. 3) Finally, they are generators of objects. We describe the influence on the interface behavior of all three roles in turn.

12.3.1 Classes as units of composition

In class-based, object-oriented languages, classes describe data together with operations or *methods* to operate on the data and thus provide a programming abstraction, similar to abstract data types [48]. At run-time, the global heap contains the state of all instances, where objects are identified via their references or addresses, and objects may refer to each other via references stored in their instance variables.

When considering a global class-based program as an open system and taking classes as the unit of composition, some classes belong to the component, and others are external, i.e., belong to the environment. Consequently, also the the method implementations are only partially part of the component. This distinction between the program and environment, both given by sets of classes and their methods, is a *static* distinction, to start with. Considering the open *behavior* of the set of program classes in an environment, however, requires to think which part of the *run-time* configuration belongs to the component, and which part conceptually to the environment. In other words, the distinction between "component" vs. "environment" becomes relevant at run-time as well, when investigating the open behavior starting from a static program where the overall set of classes is split into environment and program classes. In particular the heap can be considered only partially part of the component: conceptually, instances of component classes reside in the component part of the heap, and instances of external environment classes are part of the "environment heap", i.e., abstracted away when modelling the behavior of the component.

If in that situation a component object creates an environment object by an instantiation across the border between component and environment, then, without further interface communication, the new environment object cannot be connected to any other object, in the sense that the new object itself cannot contain references to any other object and also that it itself cannot be pointed to by other environment objects. The reason is that all communications which would put the new object in connection in this way would be visible at the interface. For instance, in Figure 12.1, after the component object o_1 has created the environment objects o_2 and o_3, indicated by the two arrows, it is guaranteed that both o_2 and o_3 are unconnected with each other and no other objects from the environment can point to them. However, o_3 could create environment instances in turn, to which it would be connected, but that would *not* be visible at the interface. The fact that o_3 has created 2 objects *internal* to the environment is indicated by the two corresponding dotted arrows. The (potential) connectivity of objects among each other is important for describing the interface behavior, since certain communications are impossible. For instance, in the situation described so far, no incoming call label a of the form $\langle call\ o.m(o_2, o_3)\rangle?$ (for some callee o) is possible, since no caller object in the environment can point to both o_2 and o_3, as they are necessarily unconnected.

To prevent such impossible interface interaction, the assumption context must contain an over-approximation of such connectivities as an existential abstraction of the heap structure. In particular, the check and update mentioned in the rule of equation (12.4) must check the connectivity, resp. update that information appropriately.

The potential connectivity of, for instance, environment objects among each other is a reflexive, transitive, and symmetric relation (written $\leftrightharpoons$). We call the equivalence class of potentially connected objects a *clique.* Note that the reflexive, transitive, and symmetric closure over the father-son arrows does *not* apply to the arrows crossing the border. For instance, in the described situation, the check whether o_2 and o_3 are potentially connected, i.e., members of the same environment clique, would fail, i.e.,

$$\Delta, \Theta \nvdash o_2 \leftrightharpoons o_3 \quad \text{but} \quad \Delta, \Theta \vdash o_3 \leftrightharpoons o_3' \leftrightharpoons o_3'' \,. \tag{12.5}$$

In Figure 12.1, the arrows show the tree of object creation (with cross-border instantiation as full arrows), and the resulting cliques of objects as dotted bubbles.

The picture so far is static and the cliques are "induced" by the tree of creation. Communication over the interface updates the connectivity information, as formalized by the update $\acute{\Delta}, \acute{\Theta} = \Delta, \Theta + a$. For instance, if the component sends an outgoing call $\langle call\ o_2.m(o_3)\rangle!$ with o_1 as caller, o_2 as callee and o_3 as argument, then all four objects o_2, o_3, o_3', and o_3'' are assumed to be connected after the step (see Figure 12.1b).

12.3.2 Classes as units of code reuse

Besides describing the implementation of their instances, one common role of classes is that they are units of code reuse via *inheritance*. The most established form of inheritance is single inheritance, on which we concentrate, even if the results apply to multiple inheritance, as well, only the details get more involved. To represent (single) inheritance, the syntax requires a small addition, only: each class mentions its immediate super-class, i.e., for $c_1[\![c_2, M, F]\!]$, c_2 is the super-class of c_1.

In the open semantics in Section 12.3.1, the heap is split into a component heap and to an (abstracted) environment heap, containing instances of component resp. of environment classes. Introducing inheritance means that instances may contain members (fields or methods) whose code is provided by the component as well as ones whose code is provided by the environment. Concerning the *state* of the open system, this existence of component and environment fields in one instance has the following consequence. Not only is the heap separated into component instances on one side and environment instances on the other, now each instance state *itself* is split into two halves, one containing the content of the instance's component fields and the other that of the environment fields.

Figure 12.2a schematically sketches that split when instantiating an instance of a component class c_2 which inherits from an environment super-class c_1. The new object o_2 contains fields from c_2 and from c_1. It is thus depicted as consisting of two halves, where the absent environment half is drawn shaded. Similar to the situation in Section 12.3.1, and without further communication, the *environment* fields of the new object o_2 do not point to any other object and furthermore, o_2 is not pointed at by environment fields of any object. In that sense, the *environment half* of o_2 forms a separate clique, indicated by the dotted circle and unconnected until interface interaction puts it into connection. We assume the standard good practice that fields are "private", so a method added in a sub-class cannot access (via `this`) fields inherited from a super-class. As a consequence of that privacy restriction, component fields can be accessed only by component methods, and analogously for fields and methods of the environment.

A second consequence is that, due to late binding and overriding, seemingly internal implementation details are actually externally observable. One symptom of that is known in software engineering as the fragile base class problem [141]. A base class in an inheritance hierarchy is a (common) super-class, and fragile means that replacing one base class by another, seemingly satisfying the same interface description, may break the code of the client of the base class, i.e., change the behavior of the "environment" of the base class. Consider the following code fragment.

Listing 12.1: Fragile base class

```
class A  {                          class B extends A {
  void add () {...}                   void add () {
  void add2 () {...}                    size = size + 1;
  ...                                   super.add(); }
}                                     void add2 () {
                                        size = size + 2;
                                        super.add2();}
                                      }
```

The two methods add and add_2 are intended to add one respectively two elements to some container data structure. Even if informally, this completely describes the intended behavior of A's methods. Class B in addition keeps information about the size of the container. Due to late-binding, this implementation of B is wrong if the add_2-method of the super-class A is

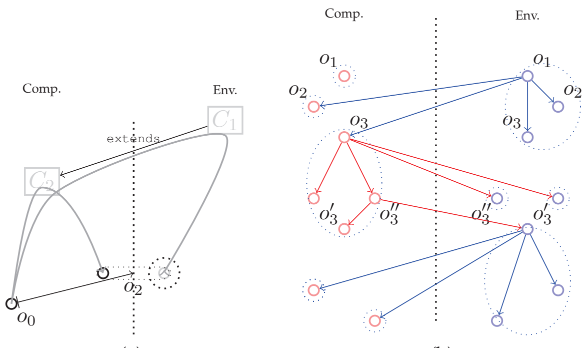


Figure 12.2: Connectivity and inheritance

implemented via *self*-calls using two times the add-method. The problem is that *nothing* in the interface, e.g., in the form of pre- and post-conditions of the methods, helps to avoid the problem. The interface specification is too weak to allow to consider the base class as a black box which can be safely substituted based on its interface specification only. In other words, due to late binding, the version of A where add_2 implements its functionality via seemingly internal self-calls add and the version where it directly implements it are observationally different. In the open semantics of the form of equation (12.3), this is reflected by the fact that the mentioned self-call constitutes a call across the interface; assuming e.g., that A is a component class and the sub-class B an environment class, a self call from method add_2 of A is an outgoing call from the component to the environment if add_2 is executed on an instance of B, as in that case the `this` refers to the environment method add of B. Note that the observability of the self-call does not depend on the use of the `super`-keyword (which is used here only to make the example more plausible).

When formulating the open semantics, the assumption and commitment contexts must represent two equivalence relations, one as abstraction of the environment fields and their connectivity and one for the component fields. These abstractions are used to check the interface steps and are correspondingly updated similar to before. Taking the tree of object creation from Figure 12.1a, in the setting with inheritance, the clique structure now looks schematically as in Figure 12.2b. Since each object is now split into two halves, all objects from the original figure have now a "mirrored" counterpart. For instance, if the component half of o_1 instantiates o_2 and o_3 (as before), the *environment half* of o_1 is connected to both o_2 and o_3 (indicated here by 2 pairs of arrows). Note that the clique structures on the component side and on the environment side differ. For instance, in the figure, the component half of o_3 is connected to o_3' and o_3'', however, for the environment part o_3, o_3', and o_3'' are all members of different cliques. Also in case of communication (as shown in Figure 12.1b in the setting without inheritance), the two clique structures are updated differently. For instance, an outgoing communication merges only cliques at the receiving side of the component, i.e., only environment cliques.

12.3.3 Classes as generators of objects

One last aspect of classes we shortly discuss is that classes are *generators* of objects. In particular, two instances of the same class are, until the first differentiating incoming input, identical up to their name or address. That means, that their behavior, when confronted with the same sequence of inputs must be identical (up to the object names). In the description of the open behavior and the possible traces from equation (12.4), the "other conditions" must therefore require that the trace is "deterministic" in that sense, i.e., given the history r, the next step a must not contradict behavior seen earlier on an "equivalent" instance of the same class. This restriction of course only applies when the language is deterministic, i.e., in particular in absence of concurrency. Note further that the problem of assuring deterministic reactions from instances rests also on the fact that objects may be *unconnected* from other objects (by being in different cliques). In the presence of *global* variables, for instance, class variables, no newly instantiated object would have "a fresh start" and would therefore not be required to show the same behavior as an earlier instantiation of the same class. Of course the complication does not occur when objects are not created from classes, or each class is instantiated only once. The formal study of full abstraction in a class-less object-oriented calculus [112] basically allows to instantiate objects only once (and besides that disallows cross-border instantiation/inheritance, which therefore avoids these problems.

12.4 Concurrency model

How to marry concurrency and object-orientation has been a long-standing issue; see e.g., [21] for an early discussion of different design choices. Actually, the basic distinction already discussed in [21] is between considering threads and objects as separate concepts on the one hand or considering objects conceptually as a unit of concurrency (and state) on the other. The first model is the multi-threading model, prominently represented by object-oriented languages like Java [82] and C$^\sharp$, [68]. The alternative is known as *active objects* or actor model. When comparing both models, we assume that part of the model are *locks* as mechanism for concurrency control, i.e., objects are now of the form $o[F, M, L]$ were L is the lock.

12.4.1 Multi-threading

The concurrency model, as known from languages like Java, separates the unit of state (the objects) from the units of concurrency (the threads).[1] The locks to protect shared data are *re-entrant* locks. To formalize that model means to extend the calculus of Table 12.1 by a `spawn` e-expression plus operations for synchronization such as `wait` and `notify`. In order to characterize the interface behavior, the formalization of legal traces from equation (12.4) needs to be adapted capturing the following two aspects,

- re-entrant calls, and
- the lock-status.

We discuss the two issues in turn.

[1]The fact that in Java concretely a thread is created as an instance of a specific thread class does not change the conceptual distinction between threads and objects.

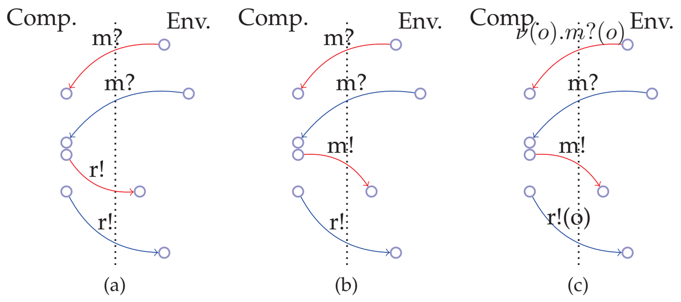


Figure 12.3: Locks

Re-entrant calls

In the multi-threading model, call-backs and re-entrant calls between component and environment are possible, e.g., an outgoing call by one thread from the component to the environment is not directly answered by the corresponding return giving back the result, but followed by an incoming call. Each thread for a legal trace from equation (12.4) must therefore be a) strictly alternating wrt. incoming and outgoing communications and b) each return must be preceded by a matching call. This amounts to a *context-free* condition on the interactions of one thread p, where on equation (12.4), one needs one check of a form $\vdash r \rhd a : \mathit{wbalanced}^+_p$ where p is the name of the thread executing a and "weak balance" formalize the condition just mentioned, requiring that the next interaction a must be incoming for instance ($\vdash r \rhd a : \mathit{wbalanced}^-_p$ is dual).

Lock status

In Java, each object is equipped with a (re-entrant) lock which can be used to protect the internal state of the object from interference or also for programming synchronized blocks. The actual state of a lock is not observable.[2] However, from the interface trace, under certain circumstances, one can draw the conclusion that a lock of an object is definitely taken. That is the case when a call is answered by a call-back. With the information of a lock being definitely taken, certain interface interactions are known then to be impossible and must therefore be excluded form the legal traces. Those conditions complicate the description of the interface behavior considerable [7]. The complications are caused, basically, by the important fact that interface interactions have no *instantaneous* effect on the state. This *decoupling* in the formulation of the open behavior is crucial, because enabledness of a step of, say, the component must *not* depend on the (internal) state of the environment and vice versa. This would contradict a compositional description of the open system. In our setting, e.g., sending a call across the interface is *independent* on the state of the callee's lock, as the lock itself is unobservable.

The issue is illustrated in Figure 12.3. The scenario of Figure 12.3a shows a trace with interactions of 2 threads (red and blue), where after two incoming calls of the two threads, each one responds with a corresponding return. Since neither the interface interaction in the form of a call takes the lock instantaneously nor a return indicates the exact point of lock release,

[2]In our theoretical calculus we do not include a method as `isLocked()` from Java's `ReentrantLock`-class which allows direct inspection of the lock status.

the behavior of 12.3a is actually possible: since the object lock guarantees mutual exclusion, either p_1 executes its the method body completely before the method body executed by p_2, or conversely. Both *serialized* executions are consistent with the shown interface behavior of 12.3a.

The situation of Figure 12.3b is similar, only now the red thread p_1 responds with a call-back instead of a return. Also this scenario possible: The call-back of p_1 makes it observable that p_1 at that point actually holds the lock. The outgoing return of p_2 makes it observable, that at some point in the past, p_2 must have held the lock (but does no longer). A possible serial execution consistent with the shown scenario therefore is that p_2 takes the lock first, releases it again, and afterwards p_1 takes it and holds it till the end of the scenario. Unlike the situation of 12.3a, this time the observed behavior imposes an *order* in which the method bodies are entered.

As discussed so far, the constraints on the order derived from the observable interaction depended only on observations concerning *synchronization* via the locks. Object creation and thus the exchange of dynamically created identifiers, e.g., object references, impose another ordering constraint: a value cannot be communicated before it has not been created. Consider Figure 12.3c, which shows the same communication pattern as 12.3b except that the first incoming call of thread p_1 sends a freshly created object identifier o as argument, indicated by the binder $\nu(o)$. The shown scenario, where p_2 returns the o in the last interaction is *impossible.* As discussed for 12.3b, the serialization constraint concerning the locks enforces that p_2 is executed before p_1. The data dependence concerning o requires that p_1 is executed before p_2.

In summary, to capture the legal behavior in the presence of lock synchronization and re-entrant multi-threading concurrency, the conditions of equation (12.4) need to keep track of the mentioned order constraints and the rule need to check that the order constraints remain *acyclic.*

12.4.2 Active objects

An alternative to the multi-threaded model of concurrency is one based on *active objects*, where the object is not only the unit of (encapsulated) state but also a unit of concurrency. One way to move from multi-threading to active objects is to replace standard method calls $v.m(\vec{v})$ by asynchronous method calls, written say $v@m(\vec{v})$. In an asynchronous call, the caller can proceed concurrently with the called method and get the result back from the call only if and when it needs it. In this way, each asynchronous method calls spawns an new thread; hence there is no stack of method calls. For the interface behavior that means the balance condition of Section 12.4.1 is not needed, resp. it degenerates to check that there is no return without prior call and the condition degenerates from a *context-free* restriction to a *regular* one (per thread). Also, in the rule of equation (12.4), when checking $\Delta, \Theta \vdash r \rhd t : \mathit{trace}$, the history r needs no longer be remembered, when formalizing the check for possible traces.

As in the multi-threaded setting the encapsulated state of an active object needs to be protected against unwanted interference. Without re-entrance, that can be achieved by simple *binary* locks, as opposed to re-entrant locks. Likewise as before, the actual lock status is not directly observable and in particular an interface interaction (still) does not indicate the instantaneous acquisition (in case of a call) or release (in case of returning the value) of a lock. Without re-entrance and call-backs, the scenario shown in Figures 12.3b and 12.3c is not possible, the one of 12.3a of course is. Remember that in Figure 12.3b it is the call-back which makes the fact observable that the lock is actually taken. In 12.3a, no information about the *current* lock status can ever be observed, which means the lock status needs not be represented when formalizing the legal traces of an open system which simplifies the description considerably. This corresponds to the situation with active objects.

Both facts —regular behavior is simpler than context-free and non-observability of the internal lock status for active objects— are a clear formal indication that the active object models with its simpler interface behavior are better suited for open systems and when considering compositionality.

12.5 Conclusion

In this paper we discussed issues for object-oriented, class-based languages from the perspective of compositionality and observable behavior. We gave an overview of how design decisions in an object-oriented language influence the description of the black box behavior when considering classes as units of composition. The question of observable black-box behavior can be and has been studied theoretically. Apart from the theoretical problems, there are also practical lessons to be learned: a clean and simple description of the component behavior is an indication that the chosen constructs are suitable for modular design and compositional reasoning. Or conversely, if the open semantics makes clear that the behavior directly or indirectly exposes internal details at the interface, this is an indication of an inherently non compositional design, for instance not providing enough encapsulation, resp. that the interfaces are too abstract to reflect the reality of what actually is observable.

Based on the sketched theoretical results, we opine the following three main points.

Object-orientation and modularity

Using classes as units of composition and as generators of objects exposes an abstract representation of the heap (in the form of connectivity of objects) as part of the interface behavior, which is a considerable complication. Taking into account also inheritance across component boundaries the description becomes even more involved. Basically, classes and sets of objects are no good units of composition, especially if it is allowed to instantiate instances of classes from another component or if inheritance between component borders is possible.

Concurrency

The comparison between the two main competing models of concurrency for object-oriented programs in Section 12.4 clearly showed that the multi-threading model is unsuitable as interaction model between components. Components are better considered as communicating asynchronously. Furthermore, the discussion in Section 12.3.3 indicates that components should be considered as inherently concurrent from the start since assuming a sequential model actually *complicates* the description of the interface behavior. In other words, a concurrent model of interaction surprisingly simplifies composition.

Synchronization

The presence of concurrency requires concurrency control. Especially in connection with multi-threading, the (in principle unobservable) status of the lock may sometimes be inferred by interacting with an object. This fact further complicated the multi-threaded setting by introducing some order constraints. One underlying reason for that seems to be that lock based synchronization is a rather low-level means of guarding against interference. The purpose of using locks is to protect critical regions against unwanted interference, but the way it's achieved is by low-level lock acquisition and release on shared locks. A more compositional, declarative, and high-level way on the user level to achieve protection would be based on *transactional* constructs. Known long from databases, such constructs have recently been proposed as user-level constructs for programming languages, for instance for Java [110] as well as for other languages. We leave the study of observational semantics for such designs as future work.

List of Figures

List of Tables

Listings

Bibliography

[1] M. Abadi. Automatic mutual exclusion and atomicity checks. In *Concurrency, Graphs, and Models*, volume 5065 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In POPL'08 [14].

[3] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[4] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[5] E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In M. Bonsangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors, *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 296–316 (21 pages). Springer-Verlag, 2005.

[6] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(7):491–518 (28 pages), 2009. Special issue with selected contributions of NWPT'07. The paper is a reworked version of an earlier UiO Technical Report TR-364, Oct. 2007.

[7] E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. In R. Gorrieri and H. Wehrheim, editors, *FMOODS '06*, volume 4037 of *Lecture Notes in Computer Science*, pages 218–232 (15 pages). Springer-Verlag, 2006.

[8] E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. *Theory of Computing Systems*, 43(3-4):322–361 (40 pages), Dec. 2008.

[9] E. Ábrahám, T. Mai Thuong Tran, and M. Steffen. Observable interface behavior and inheritance. Technical Report 409, University of Oslo, Dept. of Informatics, Apr. 2011. `www.ifi.uio.no/~msteffen/publications.html#techreports`.

[10] E. Ábrahám, T. Mai Thuong Tran, and M. Steffen. Observable interface behavior and inheritance (extended abstract). In NWPT'11 [156].

[11] E. Ábrahám, T. Mai Thuong Tran, and M. Steffen. Observable interface behavior and inheritance. Nov. 2012. Accepted for publication in the Special Issue on Behavioral Types in the Journal of Mathematical Structures in Computer Science (MSCS).

[12] S. Abramsky, S. J. Gay, and R. Nagarajan. A type-theoretic approach to deadlock freedom of asynchronous systems. In M. Abadi and T. Ito, editors, *Proceedings of TACS '97*, volume 1281 of *Lecture Notes in Computer Science*, pages 295–320. Springer-Verlag, 1997.

[13] ACM. *ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[14] ACM. *37th Annual Symposium on Principles of Programming Languages (POPL)*, Jan. 2008.

[15] G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1987.

[16] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, P. Hawkins, and B. Hackett. An overview of the Saturn project. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, June 2007.

[17] E. Albert, S. Genaim, and M. G.-Z. Gil. Live heap space analysis for languages with garbage collection. In *International Symposium on Memory Management*, 2009.

[18] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM '07*, New York, NY, USA, 2007. ACM.

[19] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress language specification. Sun Microsystems, 2005.

[20] P. America. POOL-T: A parallel object-oriented language. In M. Tokoro and A. Yonezawa, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[21] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[22] T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[23] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. *SIGPLAN Notices*, 36:16–30, Oct. 2001.

[24] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Hemel Hempstead, Hertfordshire, 2nd edition, 1996.

[25] D. Aspinall, R. Atkey, K. MacKenzie, and D. Sannella. Symbolic and analytic techniques for resource analysis of java bytecode. In M. Wirsing, M. Hofmann, and A. Rauschmayer, editors, *TGC'10*, number 6084 in Lecture Notes in Computer Science. Springer-Verlag, 2010.

[26] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, May 2008.

[27] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005.

[28] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *SEFM '05*, Washington, DC, USA, 2005. IEEE.

[29] N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 15–26, New York, NY, USA, 2007. ACM.

[30] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769 – 804, Sept. 2004.

[31] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[32] G. Bigliardi and C. Laneve. A type system for JVM threads. In *In Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, page 2003, 2000.

[33] C. Blundell, E. C. Lewis, and M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[34] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '02 (Seattle, USA)*. ACM, Nov. 2002. In *SIGPLAN Notices*.

[35] C. Boyapati, R. Lee, and M. Rinard. A type system for preventing data races and deadlocks in Java programs. In *In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[36] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '01*. ACM, 2001.

[37] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5), 2006.

[38] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, New York, NY, USA, 1995.

[39] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Oluktun. The ATOMOΣ transactional programming language. In *ACM Conference on Programming Language Design and Implementation (PLDI) (Ottawa, Ontario, Canada)* [13].

[40] A. T. Chamillard. *An empirical comparison of static concurrency analysis techniques*. PhD thesis, University of Massachusetts Amherst, 1996. AAI9709581.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '05*, pages 519–538. ACM, 2005. In *SIGPLAN Notices*.

[42] C. G. Chas Emerick, Brian Carper. *Clojure Programming*. O'Reilly Media, 2011.

[43] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory usage verification for OO programs. In *Proceedings of SAS '05*, volume 3672 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[44] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10), 1999.

[45] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[46] P. Clauss, F. J. Fernandez, D. Garbervetsky, and S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Transactions on Very Large Scale Integration Systems*, 17, 2009.

[47] E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

[48] W. Cook. Object-Oriented Programming Versus Abtract Data Types. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages (REX Workshop)*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.

[49] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Seventeenth Annual Symposium on Principles of Programming Languages (POPL) (San Fancisco, CA)*, pages 125–135. ACM, January 1990. Also in the collection [84].

[50] J. C. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[51] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL, Los Angeles, CA*. ACM, January 1977.

[52] K. Crary and S. Weirich. Resource bound certification. In *POPL '00*. ACM, 2000.

[53] Cray. Chapel specification, Feb. 2005.

[54] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67, common base language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

[55] F. Damiani, E. Giachino, P. Giannini, and S. Drossopoulou. A type safe state abstraction for coordination in Java. *Acta Informatica*, 45(7-8):479–536, 2008.

[56] A. Davare. Concurrent BLAST, 2003. Internal Report, EECS Berkely. Mentors: Mentors Rupak Majumdar and Ranjit Jhala.

[57] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Vienna, Austria.*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[58] W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Nov. 2001.

[59] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, June 2001.

[60] D. L. Detlevs, M. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, pages 57–69, 1988.

[61] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [77].

[62] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, Sept. 1965.

[63] E. W. Dijkstra. Cooperating sequential processes. In P. B. Hansen, editor, *The origin of concurrent programming*, pages 65–138, New York, NY, USA, 2002. Springer-Verlag.

[64] J. Donahue. On the semantics of data type. *SIAM J. Computing*, 8:546–560, 1979.

[65] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science State-of-the-Art-Survey*, pages 41–82. Springer-Verlag, 1999.

[66] S. Drossopoulou and T. Valkevych. Java exceptions throw no surprises. Technical report, Dept. of Computing, Imperial College of Science, London, 2000.

[67] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing, Imperial College of Science, London, Mar. 2000.

[68] ECMA International Standardizing Information and Communication Systems. *C# Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.

[69] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proceedings of POPL '09*. ACM, Jan. 2009.

[70] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[71] Findbugs - find bugs in Java programs. http://findbugs.sourceforge.net/, May 2012. Last updated 05/11/2012.

[72] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL '04*, pages 256–267. ACM, Jan. 2004.

[73] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM Conference on Programming Language Design and Implementation (PLDI) (San Diego, California)*. ACM, June 2003.

[74] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 1993. In *SIGPLAN Notices* 28(6).

[75] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[76] N. H. Gehani and W. D. Roome. Concurrent C++: Concurrent programming with class(es). *Software — Practice and Experience*, 18:1157–1177, 1988.

[77] F. Genyus. *Programming Languages*. Academic Press, 1968.

[78] P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Programming Language Approaches to Concurrency and Communication-eCentric Software EPTCS 17*, pages 79–93, 2010.

[79] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[80] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implemementation*. Addison-Wesley, Reading, MA, 1983.

[81] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[82] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.

[83] J. Gray and A. Reuter. *Transaction Processing. Concepts and Techniques*. Morgan Kaufmann, 1993.

[84] C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics, and Language Design*. Foundations of Computing Series. MIT Press, 1994.

[85] T. Harris and K. Fraser. Language support for lightweight transactions. In *Eighteenth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '03*. ACM, 2003. In *SIGPLAN Notices*.

[86] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In J. Ferrante, D. A. Padua, and R. L. Wexelblat, editors, *PPoPP'05*, pages 48–60. ACM, 2005.

[87] R. Haskin, Y. Malachai, W. Sawdon, and G. Chan. Recovery management in Quicksilver. *Database Engineering*, 8(1):82–108, 1985.

[88] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In A. Cortesi and G. Filé, editors, *Proceedings of SAS '99*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1999.

[89] K. Havelund. *Java PathFinder User Guide*. NASA, 1999.

[90] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000.

[91] K. Havelund and T. Pressburger. Model checking Jave programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, Mar. 2000.

[92] C. T. Haynes. A theory of data type representation independence. In G. Kahn, D. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 157–176. Springer-Verlag, 1984.

[93] P. Helland. Transaction monitoring facility. *Database Engineering*, 8(1):9–18, June 1988.

[94] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. K. Rajamani, editors, *Proc. SPIN2003*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[95] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93, 20th Intl. Symp. in Computer Structures*, pages 289–301. ACM, May 1993.

[96] R. Hickey. The Clojure language home page. available at `http://clojure.org/`, 2010.

[97] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[98] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[99] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[100] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL '03*. ACM, Jan. 2003.

[101] M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an object-oriented language). In P. Sestoft, editor, *Proceedings of ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[102] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. *SIGPLAN Notices*, 34(9), 1999.

[103] IEEE. *Seventeenth Annual Symposium on Logic in Computer Science (LICS) (Copenhagen, Denmark)*. Computer Society Press, July 2002.

[104] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.

[105] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, volume 34(10), pages 132–146. ACM, 1999. In *SIGPLAN Notices*.

[106] `http://indus.projects.cis.ksu.edu/`, 2010.

[107] M. Isard and A. Birell. Automatic mutual exclusion. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, 2007.

[108] F. Iwama, A. Igarashi, and N. Kobayashi. Resource usage analysis for a functional language with exceptions. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipumlation (PEPM)*, 2006.

[109] F. Iwama and N. Kobayashi. A new type system for JVM lock primitives. In *ASIA-PEPM '02: Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 71–82, New York, NY, USA, 2002. ACM.

[110] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, Aug. 2005.

[111] jChord – a static and dynamic program analysis framework for Java. Webpage: http://jlint.sourceforge.net/, June 2012.

[112] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In LICS'02 [103], pages 101–112.

[113] A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In M. Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

[114] Jlint - find bugs in Java programs. http://jlint.sourceforge.net/, June 2012. Last accessed 18th June 2012.

[115] E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java (extended abstract). In *Proceedings of the 22nd Nordic Workshop on Programming Theory (NWPT'10)*, volume 57 of *TUCS General Publication*, pages 5–7. Turku Centre for Comuter Science, Nov. 2010.

[116] E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java. Technical Report (revised version) 402, University of Oslo, Dept. of Informatics, Jan. 2011. `www.ifi.uio.no/~msteffen/publications.html#techreports`.

[117] E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java. In *Proceedings of the International Workshop on Foundations of Software Engineering (Theory and Practice) (FSEN'11)*, volume 7141 of *Lecture Notes in Computer Science*. Springer-Verlag, 2012. 16 pages.

[118] E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java with exceptions. *Journal of Logic and Algebraic Programming, special issue of selected contributions to NWPT'10*, Mar. 2012. available online 3. March 2012.

[119] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[120] C. B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[121] D. Kafura and K. Lee. Inheritance in actor based concurrent object oriented languages. Technical Report TR-88-53, Virginia Polytechnic Institute and State University, 1988.

[122] D. G. Kafura and R. G. Lavender. Concurrent object-oriented languages and the inheritance anomaly. In *In ISIPCALA'93*, pages 183–213. IEEE Press, 1993.

[123] D. Kester, M. Mwebesa, and J. S. Bradbury. How good is static analysis at finding concurrency bugs? *Proc of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2010*, pages 115–124, 2010.

[124] V. Koutavas and M. Wand. Reasoning about class behavior. In *Informal Workshop Record of FOOL 2007*, Jan. 2007.

[125] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[126] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, Mar. 1966.

[127] C. Laneve. A type system for JVM threads. *Theoretical Computer Science*, 290(1):741 – 778, 2003.

[128] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[129] B. Lindsay, C. Mohan, P-Wilms, and R. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems*, 2(1):24–38, Feb. 1984.

[130] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. The implementation of Argus. In *Proceedings of SOSP'87: Symposium on Operating Systems Principles*, pages 111–122, 1987.

[131] T. Mai Thuong Tran. Safe commits for Transactional Featherweight Java (extended abstract). In *Informal proceedings of the Young Researchers Forum at MFCS-CLS, Brno, August 2010*, Aug. 2010.

[132] T. Mai Thuong Tran. Inheritance can not be observed easily (extended abstract). In *Proceedings of the Young Researchers Workshop on Concurrency Theory (CONCUR satellite workshop)*, Sept. 2011.

[133] T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe typing for transactional vs. lock-based concurrency in multi-threaded Java. In S. B. Pham, T.-H. Hoang, B. McKay, and K. Hirota, editors, *Proceedings of the Second International Conference on Knowledge and Systems Engineering, KSE 2010*, pages 188 – 193. IEEE Computer Society, Oct. 2010.

[134] T. Mai Thuong Tran and M. Steffen. Safe commits for Transactional Featherweight Java. In D. Méry and S. Merz, editors, *Proceedings of the 8th International Conference on Integrated Formal Methods (iFM 2010)*, volume 6396 of *Lecture Notes in Computer Science*, pages 290–304. Springer-Verlag, Oct. 2010. An earlier and longer version has appeared as UiO, Dept. of Informatics Technical Report 392, Oct. 2009.

[135] T. Mai Thuong Tran and M. Steffen. Design issues in concurrent object-oriented languages and observability. In *Proceedings of the Third International Conference on Knowledge and Systems Engineering (KSE 2011), Hanoi 14th-17th Oct, 2011*, pages 135–142. IEEE Computer Society CPS, June 2011.

[136] T. Mai Thuong Tran, M. Steffen, and H. Truong. Estimating resource bounds for software transactions. Technical report 414, University of Oslo, Dept. of Informatics, Dec. 2011.

[137] T. Mai Thuong Tran, M. Steffen, and H. Truong. Estimating resource bounds for software transactions (extended abstract). In NWPT'11 [156], pages 77–79.

[138] T. Mai Thuong Tran, M. Steffen, and H. Truong. Compositional static analysis for implicit join synchronization in a transactional setting. Apr. 2013. Submitted for conference publication.

[139] S. Matsuoka and A. Yonezawa. Analysis of the inheritance anomaly in object-oriented concurrent programming. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[140] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Report in a Conference of the NATO Science Committee*, pages 138–150, 1969.

[141] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–354. Springer-Verlag, 1998.

[142] G. Milicia and V. Sassone. The inheritance anomaly: Ten years after. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274, 2004.

[143] G. Milicia and V. Sassone. Jeeg: Temporal constraints for the synchronization of concurrent objects. *Practice and Experience*, 17(5-6):539–572, 2005.

[144] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[145] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.

[146] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[147] J. C. Mitchell. Representation independence and data abstraction. In *Thirteenth Annual Symposium on Principles of Programming Languages (POPL) (St. Peterburg Beach, FL)*, pages 263–276. ACM, January 1986.

[148] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In POPL'08 [14].

[149] G. J. Myers. *The Art of Software-Testing*. John Wiley & Sons, New York, 1979.

[150] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI) (Ottawa, Ontario, Canada)* [13], pages 308–319.

[151] E. Najm, A. Nimour, and J.-B. Stefani. Liveness properties through behavioral typing of objects. In *Proceedings of the Joint International Conference on Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV'99)*. Kluwer, 1999.

[152] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1), Mar. 1992.

[153] F. Nielson. Annotated type and effect systems. *ACM Computing Surveys*, 28(2), June 1996.

[154] F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[155] H. R. Nielson and F. Nielson. *Semantics with Applications. An Appetizer*. Springer-Verlag, 2007.

[156] *Proceedings of the 23nd Nordic Workshop on Programming Theory (NWPT'11)*, volume 254/2011. Märdalen Real-time Research Centre, Märdalen University, Oct. 2011.

[157] S. Oaks and H. Wong. *Java Threads*. O'Reilly, Third edition, Sept. 2004.

[158] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Series. Artima Press, 2011.

[159] P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about program that alter data structures. In *Proceedings of CSL 2001*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.

[160] T.-H. Pham, A.-H. Truong, N.-T. Truong, and W.-N. Chin. A fast algorithm to compute heap memory bounds of Java Card applets. In *SEFM'08*, 2008.

[161] G. D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[162] A. Poetzsch-Heffter and J. Schäfer. A representation-independent behavioral semantics for object-oriented components. In M. M. Bonsangue and E. B. Johnsen, editors, *FMOODS '07*, volume 4468 of *Lecture Notes in Computer Science*, pages 157–173. Springer-Verlag, June 2007.

[163] V. P. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal of Software Tools and Technology Transfer*, 9(5):489–504, 2007.

[164] The weak memory home page: Relaxed-memory concurrency. http://www.cl.cam.ac.uk/ pes20/weakmemory/index.html, 2012.

[165] J. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la programmation (Paris, France)*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[166] J. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. IFIP, North-Holland, 1983.

[167] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In LICS'02 [103].

[168] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *ACM International Conference on Functional Programming*, pages 92–104. ACM, 2005. In *SIGPLAN Notices*.

[169] C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '00*, pages 208–228. ACM, 2000. In *SIGPLAN Notices*.

[170] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[171] The Saturn home page: Precise and scalable software analysis. `http://saturn.stanford.edu`, 2010.

[172] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[173] D. A. Schmidt. Programming language semantics. In *In CRC Handbook of Computer Science*, pages 28–1. CRC Press, 1995.

[174] S. K. Shrivastava, G. N. Dixon, F. Hedayati, G. Parrington, and S. Wheater. A technical overview of Arjuna: A system for reliable distributed computing. Technical report, Computing Laboratory, University of Newcastle upon Tyne, 1985.

[175] C. Skalka, S. Smith, and D. van Horn. A type and effect system for flexible abstract interpretation of Java (extended abstract). *Electronic Notes in Theoretical Computer Science*, 131:111–124, 2005.

[176] Wisconsin program-slicing project. Available at `http://www.cs.wisc.edu/wpis/html`, 2010.

[177] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '86 (Portland, Oregon)*, pages 38–45. ACM, 1986. In *SIGPLAN Notices* 21(11).

[178] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[179] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '95*, pages 200–214. ACM, 1995. In *SIGPLAN Notices*) 30(10).

[180] M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, July 2006. 281 pages.

[181] M. Steffen and T. M. T. Tran. Safe commits for Transactional Featherweight Java. Technical Report 392, University of Oslo, Dept. of Informatics, Oct. 2009. 23 pages.

[182] M. Steffen and T. M. T. Tran. Safe commits for Transactional Featherweight Java (extended abstract). In *Proceedings of the Nordic Workshop on Programming Theory, NWPT'09*, Oct. 2009.

[183] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[184] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *APLAS 2008*, volume 5356 of *Lecture Notes in Computer Science*, pages 155–170. Springer-Verlag, 2008.

[185] T. Terauchi. Checking race freedom via linear programming. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–10. ACM, 2008.

[186] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[187] H. Truong and M. Bezem. Finding resource bounds in the presence of explicit deallocation. In *ICTAC'05*, volume 3722 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, 2005.

[188] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In L. D. Zuck, P. D. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proceedings of the 4th International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2003*, volume 2575 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[189] A. Van Wijngaarden, B. J. Mailloux, J. Peck, and C. H. A. Koster. Draft report on the algorithmic language algol 68. *ALGOL Bull.*, (Sup 26):1–84, Mar. 1968.

[190] R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.

[191] G. Weikum and G. Vossen. *Fundamentals of Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[192] A. Welc, S. Jagannathan, and A. Hosking. Transactional monitors for concurrent objects. In M. Odersky, editor, *18th European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.

[193] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 2005.

[194] T. V. Xuan, H. T. Anh, T. Mai Thuong Tran, and M. Steffen. A type system for finding upper resource bounds of multi-threaded programs with nested transactions. In *ACM Proceedings of the 3rd ACM International Symposium on Information and Communication Technology SoICT*, pages 20–31. ACM, Aug. 2012.