UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

Hoare-style Reasoning with Higher-order Control: Continuations and Concurrency

PHD THESIS

Germán Andrés Delbianco

Departamentamento de Lenguajes y Sistemas Informáticos e Ingenieria de Software

Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

Hoare-style Reasoning with Higher-order Control: Continuations and Concurrency

Submitted in partial fulfillment of the requirements for the degree of:
Doctor of Philosophy in Computer Science

Author: **Germán Andrés Delbianco**
B.Sc. in Computer Science, Universidad Nacional de Rosario

Advisor:

Prof. Dr. Aleksandar Nanevski — IMDEA Software Institute

Thesis Committee:

Prof. Dr. Anindya Banerjee	IMDEA Software Institute
Prof. Dr. Ana Sokolova	Universität Salzburg
Prof. Dr. Cesar Sanchez Sanchez	IMDEA Software Institute
Prof. Dr. Bart Jacobs	KU Leuven
Prof. Dr. Manuel Carro Liñares	Universidad Politécnica de Madrid

2017

Abstract

The issue of software correctness is a long-standing problem in the computer science community, and it has always been relevant. Nowadays, even more so with the software industry becoming increasingly aware of the importance and benefits of formal verification. This comes as a consequence of realizing that having mathematical proof of the correctness of software systems is more efficient, even from an economical standpoint, than relying on *a posteriori* cycles of testing, debugging and re-implementing.

However, formal verification is *painstakingly hard*: it is a discipline closely connected to the semantic models on which programming languages are developed, usually involving complex mathematics. As technology paces forward, developing new logics to keep up becomes even harder. Since proving properties about complex programs is hard, it is preferable to have to do their proofs *at most* once. Unfortunately, this is not always possible.

The main goal of this thesis is the development and application of program logics aimed at the modular verification of stateful programs with higher-order control effects. In particular, it focuses of two different kinds of control effects: sequential control, in the shape of continuations and higher-order control operators like `call/cc`; and shared variable concurrency.

A continuation is a powerful abstraction which models the "future of a computation". The availability of higher-order control operators like `call/cc` make this execution context first-class citizens of a programming language, allowing client programs to have operational access (and *control*) over its execution context. The ability to manipulate "the future" makes these operators more powerful than plain `goto`-like instructions, but it also hinders the formal reasoning about programs. The contribution of this thesis in this regard is the development of a novel separation-like logic for the verification of higher-order stateful programs featuring `call/cc` and `abort` control operators.

As to shared-memory (or *shared-variable*) concurrency, we live in a world of

massively concurrent software systems running over increasing multi-processing power. In such a context, it is natural to expect programmers—and thus programming languages—to desire to exploit the available parallelism in order to produce more efficient software. Unfortunately, the intricacies of concurrency conspire against reasoning—*both formally and intuitively*—about the correctness of algorithms.

The contribution of this thesis in this regard is *Linking in Time*, a novel approach to specification of concurrent objects, in which the dynamic and non-local aspects inherent to concurrency reasoning can be represented in a procedure-local and thread-local manner. This technique has been formally mechanized in FCSL, a fine-grained concurrent separation logic, and it has been applied to prove the correctness of non-trivial concurrent objects with highly-speculative, non-obvious correctness. The approach is similar in its goals to *linearizability*, but is carried out exclusively using a separation-style logic to uniformly represent the state and time aspects of the data structure and its methods.

Resumen

La correctitud del software, es decir, el problema de decidir a ciencia cierta que un algoritmo o un programa es correcto antes de su ejecución, es una cuestión de larga data que siempre ha sido relevante en la comunidad de las Ciencias de la Computación. En los últimos tiempos cada vez más, con la industria comprendiendo la importancia y los beneficios de la verificación formal de software. Esto se produce como consecuencia de descubrir—o aceptar—que desarrollar una prueba formal de la correctitud de un sistema informático es una alternativa mucho más eficiente, incluso desde un punto de vista meramente económico, a depender de ciclos *a posteriori* de prueba, detección de errores y re-implementación.

Sin embargo, la verificación formal es una tarea *extremadamente compleja*: es una disciplina muy cercana a los modelos semánticos sobre los cuales se desarrollan los lenguajes de programación, involucrando usualmente modelos matemáticos complejos. A medida que la tecnología avanza, desarrollar nuevas lógicas, o nuevas herramientas orientadas a la verificación de estos nuevos avances se torna una empresa cada vez más difícil. Por otra parte, y dada la dificultad que conlleva comprobar formalmente propiedades de programas complejos, o de grandes sistemas, es preferible tener que hacer dichas demostraciones *como mucho* una única vez. Desafortunadamente, esto no es siempre factible.

El objetivo principal de esta tesis es el desarrollo y la aplicación de lógicas diseñadas específicamente para la verificación formal *modular* de programas con *efectos de control de alto orden*. En particular, esta tesis hace foco en dos clases diferentes de efectos de control: control secuencial, dado por construcciones como las *continuaciones* y operadores de alto orden como `call/cc`; y concurrencia de variables compartidas.

Una *continuación* es una abstracción que captura el contexto de ejecución de un cómputo o programa, *i. e.* captura el "futuro de un cómputo". Operadores de control de alto orden como `call/cc` transforman a dichos contextos de ejecución en objetos de primera clase de un lenguaje de programación, permitiendo a

los programas en ellos implementados tener acceso (y *control*) operacional a su contexto de ejecución. Esta capacidad de manipular "el futuro" convierte a este tipo de operadores más flexibles y expresivos que operaciones de primer orden que sólo permiten hacer saltos en la ejecución de un programa, *e. g.* `goto` y similares. Pero, también implica que razonar formalmente sobre éstos sea más complicado. La contribución de esta tesis en este sentido es el desarrollo de una nueva lógica, inspirada en la Lógica de Separación (o *Separation Logic)* diseñada para la verificación de programas de alto orden que utilizan operadores de control como `call/cc` y `abort`.

En cuanto a la concurrencia con variables compartidas, nos encontramos en presencia de sistemas masivamente concurrentes ejecutándose sobre capacidades de multiprocesamiento cada vez mayores. En este contexto, es comprensible que los programadores—y por lo tanto los lenguajes de programación—deseen explotar las capacidades de paralelización existentes para producir programas más eficientes. Desafortunadamente, la naturaleza misma de la concurrencia conspira en contra de razonar—*ya sea formal o intuitivamente*—sobre la correctitud de algoritmos concurrentes.

La contribución de esta tesis en este campo es *Linking in Time*, una nueva técnica de especificación de objetos concurrentes en la cual los aspectos dinámicos y no estructurales inherentes al razonamiento formal sobre concurrencia pueden ser representados de forma modular, tanto al nivel de funciones o procedimientos como al hilos de ejecución *(threads)*. Esta técnica ha sido mecanizada formalmente en FCSL, una lógica de separación para concurrencia no-bloqueante, y ha sido aplicada para demostrar formalmente la correctitud de objetos concurrentes complejos, cuyos argumentos de correctitud son intricados o altamente especulativos. Esta técnica es similar en sus objetivos a *linearizabilidad (linearizability)* pero, se desarrolla exclusivamente en una lógica de separación con el objetivo de capturar uniformemente los propiedades de estado (o memoria) de una estructura de datos concurrente, así como también expresar ciertos aspectos temporales.

Contents

List of Figures

1 Introduction

An ever present challenge in formal verification, specially as it comes to stateful reasoning, is modularity. But, not so much modularity of the programs themselves as proof-modularity, that is, the possibility to reuse the proofs of correctness of modular programs. Since proving properties of complex programs is hard, it is preferable to have to do their proofs *at most* once. Unfortunately, this is not always possible.

Although the state of the art has advanced significantly in this regard, most logics for modular stateful reasoning address *structured* programming idioms. One can even argue that a sequential mindset has driven the design of most logics for stateful reasoning, making most of the latter developments unsuitable for modular reasoning with programming constructs which implement higher-order control.

The main goal of this thesis is the development and application of dependent type theories aimed at the modular verification of stateful programs with higher-order control. As particular cases of programming idioms that have not, historically, got along well with logics for reasoning modularly about state, the thesis will focus on two: continuations and shared-memory concurrency. The premise is that by developing such higher-order control features in a dependent type theory, one can device the appropriate setting for the verification of program with complex control in a proof-modular manner.

1.1 Overview

The issue of software correctness is a long-standing problem in the computer science community, and it has always been relevant. Nowadays, even more so with the software industry becoming increasingly aware of the importance and benefits of formal verification. This comes as a consequence of realizing that having mathematical proof of the correctness of software systems (or at least particular, critical, smaller components) is more efficient, even from an economical standpoint, than relying on *a posteriori* cycles of testing, debugging and re-implementing. The

problem with formal verification is that it is closely connected to the semantic models on which programming languages are developed, usually involving complex mathematics. As technology paces forward, developing new logics to keep up becomes even harder.

The state of the art in formal verification has become so mature a discipline that its underlying theories are becoming not only a standard for reasoning with computer programs but also, they are considered rigorous enough to undertake the formalization of mathematics itself (Avigad & Harrison, 2014; Gonthier *et al.* , 2013). However, there are fundamental questions still unanswered and, in this thesis, I will address the issue of modularity in the context of stateful reasoning with unstructured programming constructs.

The problem with modular formal verification of stateful programs is not only the modularity of the programs themselves, but most importantly proof-modularity i.e. the ability to reuse the proofs of correctness of modular programs at different times. For instance, in the foundational development of Hoare Logic (Hoare, 1969) the proofs for two modularly developed stateful programs e.g. a client and a server for a complicated data structure could not be separated from each other. If at some point the internal implementation of the data structure changed, it meant that the proof for the whole system had to be redone including the one corresponding to parts that remained invariant. Since then, the state of the art has made significant advances with regards to modular verification, with Separation Logic (O'Hearn *et al.* , 2001; Reynolds, 2002) becoming the gold standard for reasoning with stateful, heap-manipulating computations, but with a steeper development in reasoning with well-structured programs. Understandably, because the latter are easier to reason with and have more accessible mathematical models.

As particular cases of intricate control flow constructs that have not, historically, got along well with logics for reasoning modularly about state, this thesis will focus on two: unstructured control flow and shared-memory concurrency.

Unstructured control flow is sometimes disregarded as a long-forsaken bad engineering practise mostly because of the ill-famous `goto` command—or, more precisely, due to Dijkstra's admonishment of its abuse (Dijkstra, 1968). There exists, however, further sensible programming constructs for unstructured control flow, continuations being one of them. *Continuations* are powerful abstractions that model the "future of a computation" (Reynolds, 1993). They have a ubiquitous presence in programming languages: they allow for a family of program transformation techniques in the style of many CPS transformations (Danvy & Filinski, 1992), they underlie the denotational semantics of programs with jumps (Felleisen *et al.* , 1988; Strachey & Wadsworth, 2000), they give computational content to classical proofs (Griffin, 1990), they have been used to structure computational effects (Filinski, 1994; Hyland *et al.* , 2007) and also to design compilation techniques (Appel, 1992). In spite of this, there has been little effort towards developing verification logics for language with first class continuations.

As to shared-memory (or *shared-variable*) concurrency, we live in a world of

massively concurrent software systems running over increasing multi-processing power. In such a context, it is natural to expect programmers—and thus programming languages—to desire to exploit the available parallelism in order to produce more efficient software. Reasoning about concurrent programs is difficult as it often entails considering all possible interactions between different threads. Unfortunately, this forces verification experts to abandon the safe waters of the sequential world where most of the advances for modular stateful reasoning have been made.

In a more general perspective, the premise of this thesis is that the challenges of software verification should be addressed in systems where the simultaneous development of programs and their proofs can be carried out in an integrated way, allowing that the same mechanisms for structuring and reuse be applied to both software development and their proofs of correctness. By judiciously structuring programs and proofs together, one can oversee that the proof burden introduced by verification does not blow up, while at the same time create new abstractions to reason with our programs.

Type Theories, in particular dependently-typed theories such as the one underlying Coq (Bertot & Castéran, 2004; The Coq Development Team, 2016), are strong, mature frameworks in which undertaking this task. In dependently-typed theories, programs and their proofs are indivisible, dual manifestations of the same phenomenon, namely the inhabitants of datatypes. Dependent Type theory is modular and supports the higher-order features that unstructured programming with higher-order control requires, but is usually *purely functional* i.e., there are no computational effects. In such type theories, one could develop programs and their correctness proofs, but the programs have to be purely functional and terminating.

This is an undesirable restriction, for two reasons. First, Hoare-style reasoning, as presented by Hoare Logic (Hoare, 1969) and Separation Logic (O'Hearn *et al.* , 2001; Reynolds, 2002) assert *partial correctness* properties, *i. e.* the postconditions hold if the program terminates and the—usually more complex—issue of termination is orthogonal. The second reason is that the high-order control effects that we consider in this thesis allow for programming idioms in which *non-termination* is natural, and we would not want to restrict ourselves to a *total* or *terminating* fragment. There are however, suitable precedents of integrating stateful reasoning and partial-correctness in the purely-functional, total setting of type theory, *Hoare Type Theory* (Nanevski *et al.* , 2006, 2008b,a, 2010) being one of them.

HTT addresses the difficulty of reconciling theorem proving with impure programming features by encapsulating sequential, stateful reasoning through computational types. This encapsulation, however, is quite different from the usual approach where one implements the abstract syntax of the programming language of choice as a datatype in the meta logic, and then reasons about abstract syntax trees. Instead, HTT uses Coq directly as a programming language, thus removing a level of indirection. HTT has carried out the initial steps to incorporate stateful

reasoning through effectful dependent type theories, but it has only considered one effect: dynamic state in a sequential setting. This thesis considers this experience to be a suitable stepping-stone to tame the verification of the high-order control features described above. We propose to adopt HTT's methodology to develop new program logics and verification techniques for modular stateful reasoning about continuations and shared-variable concurrency.

1.2 Thesis Statement and Contributions

The foremost objective of this thesis is the development and application of dependent type theories aimed at the modular verification of stateful programs with higher-order control effects. In particular, the thesis will focus on two separate kinds of control effects: continuations and shared-memory concurrency.

To this end, the aim of this thesis is to extend the type underlying dependently-typed theory of Coq with computational types which encapsulate the imperative higher-order control features under study. This has been done before, in HTT, but for a language with dynamic state in a sequential setting with no support for reasoning with *unstructured* effects. However, since both *continuations* and *concurrency* have been studied from a computational effects perspective (Hyland *et al.* , 2007; Thielecke, 2009; Abadi & Plotkin, 2009), it is natural to expect that those features can be embedded seamlessly into a dependent type theory, following this methodology, and thus enabling modular stateful verification of unstructured programming.

As we have discussed before, *continuations* are ubiquitous in the programming languages community. Although the state of the art concerning formal reasoning about them is vast, it has focused predominantly on the semantic modelling of higher-order control operators and CPS transformations (Danvy & Filinski, 1992), and verification of programs directly in the semantic model. In contrast, one of the goals of this thesis is to develop a Hoare-style logic in which one can systematically specify and verify full functional correctness of programs with higher-order jumps, in the presence of dynamic mutable state and first class continuations *i. e.*, where continuations can be returned as the result of computations, stored in the heap or passed as argument of high-order programs.

This thesis contribution in this regard is the development of HTTcc (Delbianco & Nanevski, 2013), a higher-order type theory for verification of programs with `callcc` control operators. HTTcc supports mutable state in the style of Separation logic, and, to the best of our knowledge, is the first Hoare logic or type theory to support the combination of higher-order functions, mutable state and control operators.

As for shared-memory concurrency there are two well established styles of program logics, customarily divided according to the supported kind of granularity of program interference. Logics for coarse-grained concurrency such as O'Hearn *et al.*

's Concurrent Separation Logic (CSL) (O'Hearn, 2007) restrict the interference to critical sections only, but generally lead to more modular specifications and simpler proofs of program correctness. Logics for *fine-grained* concurrency such as Jones' Rely-Guarantee (RG) (Jones, 1983) admit arbitrary interference, but their specifications have traditionally been more monolithic.

The goal of this thesis in this regard is to identify the essential ingredients required for compositional specification of concurrent programs, and combine them in a novel way which reconciles the two approaches. A first step towards that goal is the development of Fine-grained Concurrent Separation Logic (FCSL) (Nanevski *et al.* , 2014a), presented in Chapter 3. The latter logic features a novel concept of *fine-grained concurrent resources* which unify the best of the two worlds. These are concurrent computations with shared memory which allow for a simple, yet powerful, logical framework for uniform Hoare-style reasoning about partial correctness of coarse- and fine-grained concurrent programs in the *modularity-friendly, well-behaved manner* of CSL.

Another traditional school of thought in the field of concurrency verification, relies on the use of different correctness criteria or consistency models to specify the global behaviour of concurrent data structures, such as *sequential consistency* (Lamport, 1977) or *linearizability* (Herlihy & Wing, 1990). These criteria relate the concurrent history of the methods of a data structure with its sequential behaviour, allowing to reason with them using sequential program logics as if they were atomic—and therefore, there was no concurrency allowed. Library designers are thus required to prove that every possible behaviour of a method of a data structure satisfies the definition of the selected criterion.

Linearizability is the *de facto* correctness criterion for reasoning with modern concurrent fine-grained data-structures (Herlihy & Shavit, 2008; Raynal, 2013; Morrison, 2016). More precisely, for each concurrent history of an object, linearizability requires that there exists a mapping to a sequential history, such that the ordering of matching call/return pairs is preserved either if they are performed by the same thread, or if they do not overlap. To prove linearizability, one usually has to identify linearization points in programs or object methods, showing that this particular point is the single, atomic, point where the effect of the operation occurs. However, for certain classes complex, highly-scalable and efficient concurrent objects, *e. g.* (Jayanti, 2005; Dodds *et al.* , 2015; Morrison & Afek, 2013), proving them to be linearizable is not a straightforwards task: the linearization points of their methods are not fixed by the structure of the programs themselves, but rather depend on intricate interactions with the environment *(*i. e. *interference).* Traditionally, verifying such objects requires a dedicated metatheory (Turon *et al.* , 2013b; Liang & Feng, 2013; Henzinger *et al.* , 2013a), *e. g.* supporting prophecy variables (Abadi & Lamport, 1988), capable of reasoning about their highly speculative nature.

In contrast, this thesis proposes a different approach. In Chapter 4, we introduce *Linking in Time*, a lightweight technique for the verification, in a

concurrent separation logic, of complex concurrent objects with non-fixed, non-regional and future-dependent linearization points. By relying on the expressive power of FCSL, we are able to give them intuitive specification—and prove them correct!—that hide the speculative reasoning about the environment from the users of the logic. We build on the philosophy of FCSL, whereby relying upon simple, but powerful, abstractions like partial commutative monoids (FCSL), histories, and state transition systems it is possible prove tight specifications for concurrent fine-grained data-structures that resemble those of its sequential counterparts.

1.3 Position

Hoare-style stateful reasoning through dependent types

Hoare Type Theory (Nanevski *et al.* , 2006, 2008b,a, 2010) has been a recent development towards reconciling stateful reasoning with dependent type theories. HTT formalizes Separation Logic (O'Hearn *et al.* , 2001; Reynolds, 2002) in a dependently-typed setting, extending the purely-functional type theory underlying Coq (Bertot & Castéran, 2004; The Coq Development Team, 2016) to integrate Separation Logic into it. Its novelty is that its philosophy is quite different from the usual approach where one implements the abstract syntax of the programming language of choice as a datatype in the meta logic (a *deep embedding*), and then reasons about abstract syntax trees. Instead, HTT is implemented as a *shallow embedding*. This entails that HTT uses Coq directly as a programming language and thus removes a level of indirection.

The encapsulated effectful programs are classified by means of *Hoare types*, which also serve as specifications in the style of Separation Logic. A program has a Hoare type $\{P\}\,A\,\{Q\}$ if it is provably safe to run in a state satisfying the precondition P, and either diverges, or terminates with a value of type A, in a state satisfying the postcondition Q. Hoare types provide all the facilities usually required for working in Hoare logic *e.g.* specifying an invariant for a loop corresponds to providing a Hoare type for a recursive function. In such a framework, one can develop higher-order stateful programs, carry out proofs of their full functional correctness, and check the proofs mechanically. Programs and proofs can be organized into verified libraries, fulfilling the premise that modular programs should have modular, i.e. reusable, proofs.

All the logics and systems described above where designed to reason with *dynamic mutable state in a sequential setting* and they do not consider other computational effects beyond state. I propose to incorporate ideas for combining effects from the computational effects community (Plotkin & Power, 2002; Hyland *et al.* , 2007), and following HTT's philosophy develop new dependent type theories for stateful reasoning with continuations and concurrency.

Hoare-style logics for higher-order control

Crolard and Polonowski (Crolard & Polonowski, 2012) have recently developed a Hoare logic for control operators, in which specifications are carried out in types. While in this respect, the approach is similar to the one proposed in HTTcc from the high-level point of view, there is a number of differences. For example, Crolard and Polonowski only consider mutable stack variables with block scope, but no pointers or aliasing. Procedures are not allowed to contain free variables, and type dependencies contain first-order data only, such as natural numbers. Berger (Berger, 2009) presents a first-order Hoare logic for `callcc` in an otherwise purely functional language.

In contrast, I propose to follow HTT's methodology in order to develop a framework for Hoare-style reasoning with higher-order programs, control effects and mutable, dynamic state. The first step towards that goal is HTTcc, a dependent type-theory with first class continuations which uses computational types indexed by pre- and postconditions as our specifications in the style of Separation logic. Additionally, HTTcc features *algebraic* control operators, initially introduced by Jaskelioff (Jaskelioff, 2009).

A conclusion from this experience is that algebraic operators require less manual program annotations, than the non-algebraic variants. Consequently, HTTcc eases the verification of significant examples involving complicated data structures, solving further issues than just providing a sound rule for the control operators.

Program Logics for Shared-variable Concurrency

We have mentioned before that logics for shared variable concurrency can be classified in two families, grouped by the granularity of the interference allowed between concurrent threads.

Program logics for *coarse-grained* concurrency, such as Concurrent Separation Logic (CSL) (O'Hearn, 2007; Brookes, 2007), employ *shared resources* and associated *resource invariants* (Owicki & Gries, 1976), to abstract the interference between threads. A resource r is a chunk of shared state, and a resource invariant Inv is a predicate over states, which holds of r whenever all threads are outside the critical section. By mutual exclusion, when a thread enters a critical section for r, it acquires ownership and hence exclusive access to r's state. The thread may mutate the shared state and violate the invariant Inv, but it must restore Inv before releasing r and leaving the critical section.

Program logics for *fine-grained concurrency*, such as Jones' Rely-Guarantee (RG) (Jones, 1983) and its successors (Vafeiadis & Parkinson, 2007; Liang & Feng, 2013) admit arbitrary interference i.e., threads can read or write on shared variables at any time. The interaction between threads is directly specified by rely and guarantee transitions on states. A *rely* specifies the thread's expectations of state

transitions made by its environment. A *guarantee* specifies the state transitions made by the thread itself. RG is more expressive than CSL because transitions can encode arbitrary protocols on shared state, whereas CSL is specialized to a fixed mutual exclusion protocol on critical sections. But, CSL is more compositional in the manipulation of concurrent resources. Where a CSL resource invariant specifies the behaviour of an individual chunk of shared state, the atomic transitions in RG treat the whole state as monolithically shared.

This thesis aims to identify the essential ingredients required for compositional specification of concurrent programs, and combine them in a novel way to reconcile the two approaches. A first step towards that goal is the development of *fine-grained resources* (Nanevski *et al.* , 2014a; Ley-Wild & Nanevski, 2013). A fine-grained resource is specified by a resource invariant, as in CSL, but it also adds transitions in the form of relations between resource states. These transitions characterize the possible changes the threads can make to the shared state. In Chapter 3, we present FCSL a logic designed specifically towards this goal.

Correctness Criteria for Fine-grained Concurrent Objects

As we have mentioned before, linearizability is the *de facto* gold standard for reasoning with modern concurrent fine-grained data-structures. There has been a long line of research on establishing linearizability using forward-backwards simulations (Schellhorn *et al.* , 2012; Colvin *et al.* , 2006, 2005). These proofs usually require a complex simulation argument and are not modular, because they require reasoning about the entire data structure implementation, with all its methods, as a monolithic state-transition system. However, for certain complex concurrent objects, proving them to be linearizable is not a straightforwards task: the linearization points of their methods are not fixed by the structure of the programs themselves, but rather depend on intricate interactions with the environment. Traditionally, verifying such objects requires a dedicated metatheory (Henzinger *et al.* , 2013a; Chakraborty *et al.* , 2015), e.g. supporting prophecy variables, capable of reasoning about their highly speculative nature.

Instead, we propose the *Linking in Time* of atomic events. The technique we present in Chapter 4 allows us to specify and verify algorithms whose linearizability argument requires speculations, *i. e.*, depends on the *dynamic reordering* of events based on run-time information from the future. The twist is that this technique can be surprisingly implemented *off-the-shelf* in existing program logics by allowing certain *internal* (*i. e.*, not observable by clients) manipulations of the auxiliary state. The key realization comes from identifying linearization points as *pointers in time*, which can be manipulated as *fairly standard* auxiliary state in a concurrent separation-like logic. Modular reasoning is then achieved by means of separately proving properties of specific transitions of these state transition systems in FCSL, and then establishing specifications of programs, composed out of well-defined atomic commands, following the transitions, and respecting the STS invariants.

1.4 Structure of the thesis manuscript

The contributions of this thesis are twofold, and hence this document is structured to reflect the somewhat *parallel* nature of its contributions. Thus, Part I is concerned with the development of HTTcc, a Hoare-style logic for higher-order (sequential) control effects, and Part II is concerned with shared-memory concurrency, introducing *Linking in Time* and FCSL. Part III presents the conclusions of the thesis and discusses possible future work. Concretely:

Chapter 2 presents the first contribution of this thesis: a higher-order HTT for verification of programs with `call/cc` and `abort` control operators (Delbianco & Nanevski, 2013). The chapter explains the design and the implementation of the program logic—called HTTcc—and also it illustrates how to put the framework to work in order to verify a number of characteristic examples. To the best of our knowledge, this is the first Hoare-style logic for verification of higher-order programs with higher-order jumps.

Chapter 3 presents *fine-grained concurrent resources* (Nanevski *et al.* , 2014a), a novel model for scalable shared-variable concurrency verification and FCSL, a logic and verification framework which enables proof modular verification of complex concurrent data structures.

Chapter 4 presents *Linking in Time* (Delbianco *et al.* , 2017a), a technique implemented on top of FCSL, for reasoning about concurrent objects with non-fixed, future-dependent linearization points. We illustrate the method by verifying (mechanically in Coq) an intricate optimal snapshot algorithm due to Jayanti (Jayanti, 2005), together with some concurrent clients of the snapshot data structure.

Chapter 5 summarizes the contributions of this thesis and discusses several interesting open questions and some possible lines of future research directions.

1.4.1 Changes & Future Editions

In addition to general polishing, this version of the draft presents the following changes from the previous one:

- The *holes* in Chapter 3 have been completed to include a full account of FCSL semantics (Section 3.4), a sub-section on the *structural rules* of the logic (Section 3.2.2), and the Related Work (Section 3.5) has been revised.
- In Chapter 4, the Related Work 4.10 has been updated to discussed recently published developments in linearizability reasoning (Bouajjani *et al.* , 2017).

However, this thesis draft is still a work in progress. I intend to do the following changes to subsequent versions of this manuscript:

- The Introduction needs to be polished. Moreover, I plan to introduce examples to motivate the methodology of this thesis—*i. e.* the use of types as a vehicle for implementing program logics— and incorporate some general background from Hoare Type Theory (HTT), and on program logics in general.
- I envision the following improvements for Chapter 2:
    1. Expand the Overview section, Section 2.2, introducing simpler examples building up to **inc3**, the only example used in the opening of the original paper to introduce and motivate HTTcc.
    2. Expand the Discussion section, Section 2.7 with a more thorough description of the related work.
    3. Incorporate two more examples which have been verified in Coq and have not been included in the original manuscript.
    4. Add a detailed explanation for the proof outline for **ping–pong** in Section 2.6.
    5. Explain the alternative implementation of *roll-backing* control operators, together with an example.
- A comparisson with the new version of FCSL, currently in submission, should be included in the relevant Chapters.

Part I

Continuations

Preamble

Continuations are programming abstractions that allow for manipulating the *future* of a computation. Among their many applications, they enable implementing unstructured program flow through higher-order control operators such as `call/cc`. Hoare-style logics for the verification of control structures have focused, traditionally, only on first-order jumps i.e. plain `goto` and label commands (Crolard & Polonowski, 2012; Audebaud & Zucca, 1999; Arbib & Alagic, 1979; Tan & Appel, 2006), which can be implemented as a special—*i. e.* simplified—case of `call/cc`.

This part of this thesis presents, a new Hoare-style logic for the verification of programs with higher-order control, in the presence of dynamic, mutable state. This is done by designing a dependent type theory with first class callcc and abort operators, where pre- and postconditions of programs are tracked through types, building upon the lessons from Hoare Type Theory for sequential stateful reasoning. Moreover, the control operators are algebraic in the sense of Plotkin and Power (Plotkin & Power, 2004, 2003), and Jaskelioff (Jaskelioff, 2009), to reduce the annotation burden and enable verification by symbolic evaluation.

The resulting logic, which we call HTTcc, is implemented as a *shallow-embedding* in Coq, providing a unified framework in which to mechanize the meta-theory of the logic as well as its case studies. In order to further illustrate the latter point, we present the verification of a number of characteristic examples, featuring typical programming idioms using `call/cc`. To the best of our knowledge, HTTcc is the first Hoare-style logic for verification of higher-order programs with higher-order jumps.

The contents of this part of the thesis have been published in the following paper:

- Germán Andrés Delbianco and Aleksandar Nanevski. Hoare-style reasoning with (algebraic) continuations. In Morrisett, G. and Uustalu, T., editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27*, 2013, pages 363–376. ACM.

The contents of this part extend the original paper (Delbianco & Nanevski, 2013) by incorporating the appendices from the extended version of the paper, more examples and an updated and more thorough discussion of related work. The mechanization of HTTcc in Coq, together with the implementation of the case studies, is available online (Delbianco & Nanevski, 2017).

2

Hoare-style Reasoning with (Algebraic) Continuations

2.1 Introduction

Continuations are powerful abstractions that model the "future of a computation" (Reynolds, 1993). They have a ubiquitous presence in programming languages: they allow for a family of program transformation techniques in the style of many CPS transformations (Danvy & Filinski, 1992), they underlie the denotational semantics of programs with jumps (Felleisen *et al.* , 1988; Strachey & Wadsworth, 2000), they give computational content to classical proofs (Griffin, 1990), they have been used to structure computational effects (Filinski, 1994; Hyland *et al.* , 2007) and also to design compilation techniques. Moreover, certain programming languages provide first-class control operators which manipulate continuations, e.g. the variants of `call/cc` in Scheme, ML and Haskell, or the related $\mathcal{C}$ and $\mathcal{F}$ control operators (Felleisen *et al.* , 1986, 1987).

The ability to manipulate "the future" makes these operators more powerful than plain `goto`-like instructions, but it also hinders the formal reasoning about programs. Although the state of the art concerning formal reasoning about continuations is vast, it has focused predominantly (with notable exceptions discussed below) on the semantic modelling of higher-order control operators and CPS transformations (Felleisen *et al.* , 1986; Griffin, 1990; Danvy & Filinski, 1992; Thielecke, 1997), and verification of programs directly in the semantic model (Støvring & Lassen, 2007; Dreyer *et al.* , 2010).

In contrast, we are interested in developing a Hoare-style logic in which one can systematically specify and verify full functional correctness of programs with higher-order jumps, in the presence of dynamic mutable state and the ability to capture continuations and return them as results of computations, potentially encapsulated into closures.

The ability to capture and return continuations makes our task more difficult when compared to the previous work on Hoare logics for first-order jumps

(i.e. `gotos`) in high-level languages (Clint & Hoare, 1972; Kowaltowski, 1977; Audebaud & Zucca, 1999; Arbib & Alagic, 1979) and low-level machine code (Saabas & Uustalu, 2007; Tan & Appel, 2006; Jensen *et al.* , 2013). In particular, the higher-order nature of `call/cc` entails the need for a Hoare logic capable of reasoning about (potentially higher-order) functions. It also makes it somewhat more difficult to design the specification methodology, i.e. decide on just what kind of information should the proof developer provide in the form of annotations when specifying a program involving `call/cc`, and how should that information relate the context in which the continuation is captured to the context in which it is invoked.

The presence of dynamic state significantly complicates matters, and differentiates our work from recent Hoare-style logics for higher-order jumps (Berger, 2009; Crolard & Polonowski, 2012). From the semantic point of view, supporting dynamic state requires building a model in which executing a continuation does not roll back the mutable state to the point at which the continuation is captured. From the specification point of view, it requires reconciling `call/cc` with Separation logic (O'Hearn *et al.* , 2001; Reynolds, 2002). In this work, we accomplish the task in a novel manner, combining Separation *assertion* logic with large footprint semantics and large footprint inference rules for verification in the style of symbolic evaluation.

More concretely, our contribution in this part of the thesis is the development of HTTcc, a framework for Hoare-style reasoning with higher-order programs, control effects and mutable, dynamic state. To the best of our knowledge, this is the first formal system for reasoning about such combination of features. We define a dependent type-theory with first class continuations which uses monadic types indexed by pre- and postconditions as our specifications in the style of Separation logic. This has been done before for a language with state, e.g. in *Hoare Type Theory* (HTT) (Nanevski *et al.* , 2008a, 2010), but now we extend the approach to a language with continuations. In particular, we rely on *dependent records* (i.e. iterated Σ-types) as an essential tool for specification of programs which "capture the continuation" in a closure that is later invoked.

In order to make specification and verification in HTTcc more palatable, we focus on a specific choice of control operators which are *algebraic* in the sense of Plotkin and Power (Plotkin & Power, 2004, 2003) and Jaskelioff (Jaskelioff, 2009); that is, the control operators commute with sequential composition. We argue that the algebraic control operators are less burdensome for verification than the non-algebraic alternatives, because in practice they require less annotations to be manually provided by the user. In particular, the algebraic `call/cc` typically requires manual description of only the jumping behaviour of the code, whereas, in our experience, the non-algebraic variants require manual description of both the jumping and the normal termination of the `call/cc` block.

To test our design in practice, we have implemented HTTcc as a shallow embedding in the Calculus of Inductive Constructions (CIC), as realized in

1. $\{x \mapsto v\}$
2. $c \leftarrow \mathbf{callcc}\, f.\ \mathbf{ret}\ (\mathbf{abort}\ f\ (x\ \mathbf{:=}\ !x+1\,;\ \mathbf{ret}\ (\mathbf{ret}\ ())));$
3. $\{x \mapsto v \curlywedge \{\, x \mapsto v+1 \,\}\, \mathbf{c}\, \{\, \bot \,\} \curlyvee$
   $x \mapsto v+2 \curlywedge \{\, x \mapsto v+3 \,\}\, \mathbf{c}\, \{\, x \mapsto v+3 \}\}$
4. $x\ \mathbf{:=}\ !x+1\,;$
5. $\{x \mapsto v+1 \curlywedge \{\, x \mapsto v+1 \,\}\, \mathbf{c}\, \{\, \bot \,\} \curlyvee$
   $x \mapsto v+3 \curlywedge \{\, x \mapsto v+3 \,\}\, \mathbf{c}\, \{\, x \mapsto v+3 \}\}$
6. c
7. $\{x \mapsto v+3\}$

Figure 2.1: An idealized proof outline for **inc3**, implementing a *backwards* or *re-entrant* jump. Notice the use of *nested* Hoare triples to cope with the high-order nature of this function: c, the return value of the **callcc** call in line 2 is a computation, executed later at line 6.

Coq (The Coq Development Team, 2016; Bertot & Castéran, 2004) and Ssreflect (Gonthier *et al.* , 2008). We have mechanized its denotational semantics and soundness proofs and used the framework to verify a survey of standard examples which cover the usual continuation idioms. A subset of the examples is presented in Section 2.6. All of our Coq code is available on-line (Delbianco & Nanevski, 2017).

2.2 Overview

The language of HTTcc uses monads, as in Haskell, to separate the purely functional and the imperative fragment of the language. In addition to capturing continuations via **callcc** and jumping to them via **abort**, the imperative fragment supports recursion (which we consider a side effect), and heap-mutating commands such as allocation, deallocation, reading from and writing into heap locations. We use $\mathbf{ret}\, v$ for a monadic command returning a value v, and $x \leftarrow e_1; e_2$ for monadic bind (i.e. a sequential composition), which first executes e_1, then binds the return result to x before executing e_2. We abbreviate with $e_1; e_2$ when $x \notin \mathsf{FV}\,(e_2)$. As customary with monadic languages, we assume that a command is evaluated only upon being bound in a sequential composition. Until then, the command's execution is suspended.

2.2.1 Algebraicity

As our **callcc** and **abort** are non-standard, we briefly illustrate them by example. Figure 2.1 shows an *informal* proof outline—in a style of partial correctness Hoare logic or Separation logic—for a function **inc3**, which uses a backward jump to increment the value of pointer x by 3. Our actual syntax for **inc3**, and the manual assertions needed for its specification, will be introduced in Section 2.2.3. We first discuss the behaviour of the function, and then return to the proof outline.

In the first command (line 2), **inc3** captures the continuation, which, at that point, is $\lambda\, c.\ x \mathrel{:=} !x + 1;\ c$, corresponding to the code in lines $4 - 6$, with the variable c abstracted. The continuation is encapsulated inside a *continuation object* f, which may be viewed as a label for jumps. Next, the body of **callcc** is executed, and the suspended command $\mathbf{abort}\ f\,(x \mathrel{:=} !\,x + 1;\ \mathbf{ret}\,(\mathbf{ret}\ ()))$ is bound to c.[1] The program continues by incrementing x (line 4), after which c follows (line 6). Execution of c causes a jump to the continuation encapsulated inside f. However, before the control is passed to the continuation, the second argument of **abort**—$x \mathrel{:=} !\,x + 1;\ \mathbf{ret}\,(\mathbf{ret}\ ())$— is executed. Thus, x is incremented again, and $\mathbf{ret}\,()$ is passed as the argument c to the continuation encapsulated inside f. Passing the control to the continuation corresponds to a backward jump to line 4. Thus, x is incremented once again, followed by execution of c. Since the latter variable is now bound to $\mathbf{ret}\,()$, its execution falls through and **inc3** returns $()$, after having incremented x three times.

The non-standard aspect of our control operators is that **abort** allows executing arbitrary side-effectful command—in the above case $x \mathrel{:=} !\,x + 1$—as part of performing the jump. This is different from the customary **callcc** and **throw** (Moggi, 1989; Thielecke, 2009; Wadler, 1994), as the latter only passes a *value* upon a jump. We refer to this side-effectful command as *finalization* code, as it is executed immediately before the jump, thus ending the normal control flow. Obviously, **throw** can be mimicked by **abort** by using a trivial finalization code which immediately returns. Dually, $\mathbf{abort}\ f\ e$ can be implemented by sequential composition which executes e, followed by throwing the obtained value to f.

However, choosing **abort** as a primitive, awards special status to the finalization code, which makes the control operators *algebraic* (Jaskelioff, 2009; Plotkin & Power, 2004, 2003). More concretely, **callcc** commutes with sequential composition, a property we use in Section 2.4 to formulate a methodology for Hoare-style specification and verification by symbolic evaluation. We illustrate how the commutation arises by the following equations, which can be derived

[1] As usual with monads, the binding strips the outer **ret**.

from Jaskelioff (Jaskelioff, 2009).

$$x \leftarrow (\mathbf{callcc}\, f.\, e_1); e_2 = \mathbf{callcc}\, g.x \leftarrow [(g \triangleright x.\, e_2)/f]e_1;\ e_2 \tag{2.1}$$

$$\mathbf{abort}\ (g \triangleright x.\, e_2)\ \ e_1 = \mathbf{abort}\ g\ \ (x \leftarrow e_1;\ \ e_2) \tag{2.2}$$

$$\textit{where} \quad g \triangleright x.\, e_2 \triangleq \lambda t.\ g\ (x \leftarrow t; e_2)$$

Consider equation (2.1). The continuation captured inside f on the left side of the equation includes e_2. On the right side, **callcc** has been commuted out of the scope of x, and thus e_2 cannot be part of the continuation captured inside g. To induce that the expressions on the two sides of equality behave the same, jumps to g on the right side must be *preceded* by an execution of e_2. Because our primitives provide for finalization code, we could enforce such a discipline by using e_2 as finalization code for every **abort** to g. This is achieved by uniformly substituting f in e_1 with a new continuation object $g \triangleright x.e_2$. The new object is engineered so that aborting to it object behaves like aborting to g with the finalization code extended by e_2, as captured in equation (2.2). Thus, execution of e_2 precedes the jumps to g, just as required.

2.2.2 Proof outline

As customary in Separation Logic, heaps are finite maps from the type **ptr** of pointers (isomorphic to $\mathbb{N}$) to values. The predicate $x \mapsto v$ holds only of a singleton heap with a pointer x, storing the value v. We use $\curlywedge$, $\curlyvee$, for point-wise conjunction and disjunction of heap *predicates*, and $\top$ and $\bot$ for the always true and always false predicate, respectively. In subsequent examples, we will also use separating conjunction $P * Q$, which holds of a heap h if h can be split into disjoint parts satisfying predicates P and Q, respectively. We will also use the predicate $\mathsf{this}\ h$, which holds only of heaps equal to h. We write $P\ h$ or alternatively, $h \in P$, when the predicate P holds of the heap h. We retain $\wedge$, $\vee$, True and False for the customary propositional (i.e., non-separation) connectives.

Referring to Figure 2.1, the line 1 states that the program starts with the initial heap containing a pointer x storing the integer v (though we omit the type annotations). HTTcc uses *large footprint* annotations which describe the full heap in which the program runs, rather than just a subheap that the program needs (in contrast to the *small footprint* annotations from Separation logic). Thereby, the proof outline in Figure 2.1 describes the behaviour of **inc3** when the heap contains *exactly* the pointer x storing v, but no other pointers. For now, we restrict ourselves to this simple case in order to focus on algebraicity, but we explain in Section 2.3 how to generalize the annotations of **inc3** to cover larger heaps.

Going back to Figure 2.1, at line 3, after the continuation is captured in line 2, the current heap is unchanged, but the program variable c is bound to the command $\mathbf{abort}\ f\,(\mathbf{ret}\,(\mathbf{ret}\ ()))$, which itself has to be specified. The Hoare

triple $\{x \mapsto v+1\}\ c\ \{\bot\}$ indicates that c should be executed only after x is incremented (precondition $x \mapsto v+1$), and that the execution of c causes a jump (postcondition $\bot$). This behaviour precisely corresponds to the intended use for c in line 6. However, as c in line 6 executes a backward jump, the assertion in line 3 has to describe the state right after the jump, but before the program proceeds with executing line 4 for the second time. This is the role of the second disjunct in line 2. It shows that the program point is reached for the second time with x incremented twice. As described in Section 2.2.1, at that point c is bound to **ret ()**, and can be specified by $\{x \mapsto v+3\}\ c\ \{x \mapsto v+3\}$ to indicate that this second instance of c will be executed after x is incremented once more (precondition $x \mapsto v+3$). The second execution of c does not jump, but falls through with the heap unchanged (postcondition $x \mapsto v+3$).

After x is incremented in line 4, the assertion in line 5 accounts for the change in the heap: compared to line 3, the value of x is incremented in both disjuncts, while the specifications for c remain unchanged. Now, command c is safe to execute in line 6, because the heap in both disjuncts satisfies the respective preconditions for c. The program terminates satisfying $x \mapsto v+3$ (line 7), which is a disjunction of the postcondition of c from line 5.

2.2.3 Proof annotations as dependent types

The crucial point in the proof outline for **inc3** is deciding on the specifications for c in line 3, which indicate the intended use of c in the rest of the program (i.e., c is executed when x stores $v+1$ and $v+3$, jumping in the first case, and falling through in the second). Such specifications depend on the structure of the rest of the program, and cannot be gleaned solely from the body of **callcc** in line 2. In this paper, we adopt the approach that such information is provided by the programmer in the form of annotations. This is similar to the way loop invariants often have to be provided when verifying structured programs. In HTTcc, we use *type annotations* for this purpose. However, because the annotations clearly depend on run-time values (e.g., the contents of the pointer x in Figure 2.1), we have to use *dependent types*.

In particular, HTTcc features two type constructors which we use to provide annotations for side-effectful programs and for continuation objects. Intuitively, the type $\mathsf{SK}^*\ \{P\}\,A\,\{Q\}$ classifies programs with precondition P, postcondition Q and return value of type A. The type $\mathsf{Kont}^*\ \{R\}\{P\}\,A\,\{Q\}$ classifies continuation objects. R describes what holds when the continuation is captured by **callcc**, and is referred to as the *initial condition*. Precondition P describes what must hold of the heap when aborting to the continuation object; thus immediately before the finalization code is executed. postcondition Q describes the heap and the value obtained after the execution of the finalization code, but before the actual jump. In both types, the assertions R, P and Q may depend on program values. Additionally, Q may depend on the dedicated variable $r{:}A$, naming the return

$$
\begin{array}{l}
\textbf{inc3}\,(x{:}\mathsf{ptr}) : [v].\ \mathsf{SK}^*\ \{x \mapsto v\}\ ()\ \{x \mapsto v+3\} \triangleq \\
\quad \textbf{do}\ c \leftarrow \textbf{callcc}_j \\
\qquad f : [v].\ \mathsf{Kont}^*\ \{j \in x \mapsto v\}\ \{x \mapsto v+1\}\ \Sigma{\bullet}\mathsf{SK}\,() \\
\qquad\qquad\qquad \{r.\ x \mapsto v+2 \curlywedge \mathsf{spec}\, r \sqsubseteq (x \mapsto v+3, x \mapsto v+3)^*\}. \\
\qquad \textbf{do}\ (\textbf{ret}\ [\textbf{abort}\ f\ (x := !x+1\,;\ \textbf{ret}\ [\textbf{ret}\ ()])]) \\
\qquad : [v].\ \mathsf{SK}^*\ \{x \mapsto v \curlywedge j \in x \mapsto v\}\ \Sigma{\bullet}\mathsf{SK}\,() \\
\qquad\qquad \{r.\ x \mapsto v \curlywedge \mathsf{spec}\, r \sqsubseteq (x \mapsto v+1, \bot)^*\}; \\
\quad x := !x+1\,; \\
\quad \textbf{cmd}\ c.
\end{array}
$$

Figure 2.2: Specification of **inc3** via type annotations.

value.

We employ the notation $[v_1{:}A_1, \ldots, v_n{:}A_n].\ \mathsf{SK}^*\ \{P\}\,A\,\{Q\}$, often omitting the types A_i, to specify that $v_1, \ldots, v_n$ are variables that may scope through P and Q (and similarly for R, P and Q in Kont^* types). In Hoare logic terminology, such variables are known as *logical*; they may appear in assertions, but not in the code. In first-order Hoare logics, logical variables have global scope and are used to relate the initial and ending states of a computation. In our setting, the scope of logical variables is *local* to the type in which they are bound. This is required in any Hoare logic for a language with procedures and recursion (Kleymann, 1999) such as HTTcc, where a logical variable used in the specification of a recursive procedure, may have to be instantiated differently to satisfy the preconditions of different recursive calls.

Additionally, we employ a *dependent record type* $\Sigma{\bullet}\mathsf{SK}\,A$, which packages a precondition and a postcondition together with a computation, and thus abstracts existentially over them. In other words, a value of type $\Sigma{\bullet}\mathsf{SK}\,A$ is a *structure* of the form $[P, Q, e]$, where $e : \mathsf{SK}^*\ \{P\}\,A\,\{Q\}$. As we show subsequently, values of this type will be used whenever we "nest" the monadic types, i.e. build computations that return other computations, which may potentially capture continuations. Given $c : \Sigma{\bullet}\mathsf{SK}\,A$, we will use $\mathsf{spec}\,c$ and $\textbf{cmd}\ c$ to project the components when necessary:

$$
\begin{array}{l}
\mathsf{spec}\ : [P, Q, e] \mapsto (P, Q) \\
\textbf{cmd}\ : [P, Q, e] \mapsto e
\end{array}
$$

We will abuse the notation and write $[e]$ instead of $[(P, Q), e]$, as the Hoare triple type of e—and hence its P and Q components—can usually be inferred, as we describe in Section 2.4. We will also use the symbol $\sqsubseteq$ to denote the usual Hoare ordering (i.e., pre-strengthening/post-weakening) on pairs (P, Q).

We now present in Figure 2.2 the fully annotated version of **inc3**, as it is written in HTTcc. Apart from the obvious typing annotations and the explicit use of the aforementioned constructors and projections for $\Sigma\bullet\mathsf{SK}\,(\,)$, there are additional syntactic elements that did not appear in Figure 2.1. We use the expression $\mathbf{do}\ e : \mathsf{SK}^*\ \{P\}\,A\,\{Q\}$ (potentially with logical variables) whenever we want to explicitly ascribe the specification (P, Q) to e, rather than use the tightest specification that the system infers for e. Such ascription will entail a proof obligation that (P, Q) is valid for e, as we explain in Section 2.4. When the type ascription is explicitly bound to a variable x, we write $x : \mathsf{SK}^*\ \{P\}\,A\,\{Q\} \triangleq \mathbf{do}\ e$.

We also make explicit that **callcc** captures the *current heap*, in addition to the current continuation, through the *heap* variable j that is bound by **callcc** and which we declare as an index (e.g., $\mathbf{callcc}_j$). The variable j scopes over the whole **callcc** body, including the type of the continuation object f. However, it is introduced strictly for purposes of specification, and we shall use it only in the annotations, but not in the executable code over which it scopes. The role of j is to relate the values of the various logical variables in its scope. In Figure 2.2, e.g., the assertion $j \in x \mapsto v$ appears in the type of the continuation object f and in the type of the body of **callcc**, thus implying that the (distinct) logical variables named v in the two types, in fact denote the same value – that stored in x at the entry to **callcc**. In Figure 2.1 we used a single *global* logical variable v for this purpose, but, as explained, global logical variables do not scale to Hoare-style reasoning about procedural languages.

Figure 2.2 uses the constructor $[\,\_\,]$ twice. Once around $c_1 = \mathbf{abort}\ f\,(x := !x + 1\,;\ \mathbf{ret}\,[\mathbf{ret}\,(\,)])$ and again around $c_2 = \mathbf{ret}\,(\,)$, which is embedded inside c_1. As explained in Section 2.2.1 at different points of execution, both c_1 and c_2 are assigned to the variable c, and thus, all three must have the same type. Because the Hoare types of c_1 and c_2 actually differ in the pre- and postconditions, we employ $[\,\_\,]$ to abstract over the whole specifications, coercing c_1 and c_2 to $\Sigma\bullet\mathsf{SK}(\,)$. The individual specifications of c_1 and c_2 are then re-established using the $\mathsf{spec}\ r$ projection in other program annotations.

For example, $[c_1]$ is the return value of **callcc**. The explicitly ascribed SK^* type states that $\mathsf{spec}\,r \sqsubseteq (x \mapsto v + 1, \bot)^*$, exposing that c_1 performs a jump and should be executed only when x is incremented once.[2] On the other hand, $[c_2]$ is the value of the finalization code of the **abort** to f in c_1. Hence, the variable r in the postcondition of f's type stands for $[c_2]$, and the formula $\mathsf{spec}\,r \sqsubseteq (x \mapsto v + 3, x \mapsto v + 3)^*$ in f's postcondition exposes that c_2 does not change the state, but should be executed only when x is incremented by 3. The types of f and the **callcc** body in Figure 2.2 explicitly provide the required information about the intended uses of the code bound to c at various execution points. Indeed, the postconditions of these two types are essentially the two disjuncts from line 3,

[2]The reader can ignore the operation $(-)^*$ for now; it will be defined in Section 2.3.

Figure 2.1, with the formulas involving $\mathsf{spec}\, r$ replacing the Hoare triples over c. In this sense, the spec projection out of the record type $\Sigma\bullet\mathsf{SK}$ represents Hoare triples when they are *nested*, i.e. used within the assertions of other Hoare triples.

It is important, however, that the two disjuncts in line 3, Figure 2.1 are specified in two different places in **inc3** from Figure 2.2. In particular, the type of f provides the disjunct describing what happens when f is jumped to, whereas the inner type ascription provides the disjunct describing the normal return value of the **callcc** block. This pattern whereby f specifies only the jumping behaviour is characteristic of the algebraic **callcc** operator. On the other hand, as we shall see in Section 2.6, the annotations describing the non-jumping behaviour in the body of **callcc** can often be inferred (though in the case of **inc3** we had to explicitly ascribe them because the return value nests a jump). In the non-algebraic case, in our experience, f has to *always* be manually annotated with the full disjunction of the jumping and non-jumping cases, resulting in larger and more cumbersome annotations and proofs.

2.3 Notation, logical variables, large footprint

In the previous sections we have given an informal presentation of HTTcc. In this section we will present the typing rules for the HTTcc language together with the properties and lemmas that constitute our verification framework. Furthermore, we will present the interpretation of algebraicity in our setting, and we will show how to use algebraicity for deriving verification lemmas that will allow us to structure the proofs by symbolic evaluation.

While logical variables are very useful in specifications, they are somewhat inconvenient to work with in the meta-theory. The main hindrance is that premises of inference rules may contain Hoare triples with differing contexts of logical variables, which have to coalesce in some way into a logical context of the Hoare triple in the conclusion. Typically, simple conjoining of contexts is not what is wanted, as some sharing of variables is desired. But then, it becomes problematic how to specify just exactly which variables should be shared in the conclusion, and which should not. To circumvent the issue, in this section we introduce SK and Kont types which do not use logical variables at all, and show how SK^* and Kont^* types with logical variables from Section 2.2, become merely notational abbreviations.

As a first step, we introduce the type $\mathsf{SK}\ A\ (P, Q)$ of effectful computations where P is a precondition over heaps, as discussed in the previous section, but Q is a *binary* postcondition, ranging over the ending result of the program (of type A) and the initial and ending heaps, much as in VDM-style specifications (Bjørner & Jones, 1978; Nanevski *et al.* , 2010). We use CIC-style notation to classify logical propositions by the type Prop, and represent predicates as functions into Prop.

Definition 2.1 (A-specs)**.** Given $P : \mathsf{preT}$ and $Q : \mathsf{postT}\ A$ an A-specification, or A-*spec*, is a pair $(P, Q) : \mathsf{specT}\ A$ with:

$$\begin{aligned} \mathsf{preT} &\triangleq \mathsf{heap} \to \mathsf{Prop} \\ \mathsf{postT}\ A &\triangleq A \to \mathsf{heap} \to \mathsf{heap} \to \mathsf{Prop} \\ \mathsf{specT}\ A &\triangleq \mathsf{preT} \times \mathsf{postT}\ A \end{aligned}$$

The notation $[\Delta].\ \mathsf{SK}^*\ \{P\}\ A\ \{Q\}$ from Section 2.2, with unary $\{P\}$ and $\{Q\}$ is an abbreviation, as we illustrate next.

Example 2.2. The type $[v].\ \mathsf{SK}^*\ \{x \mapsto v\}\ ()\ \{x \mapsto v + 3\}$ of **inc3** from Figure 2.2, is an abbreviation of the specification:

$$\mathsf{SK}\ ()\ (\lambda i.\, \exists v.\ i \in x \mapsto v,\ \ \lambda r\, i\, m.\, \forall v.\, i \in x \mapsto v \to m \in x \mapsto v + 3)$$

The SK type introduces an explicit quantification over v in the precondition to guarantee that **inc3** is safe to execute in the heap containing (only) the pointer x. The universal quantification in the postcondition expresses that upon termination, the value pointed to by x is incremented by 3. To express this property, the precondition $x \mapsto v$ has to be repeated as part of an implication in the binary postcondition, which is obviously cumbersome, thus motivating the following notation, generalizing to a context Δ.[3]

Definition 2.3 (SK^*)**.** Given a type A, $P : \mathsf{preT}$, $Q : \mathsf{postT}\ A$ the type notation $[\Delta].\ \mathsf{SK}^*\ \{P\}\ A\ \{Q\}$ is defined as:

$$[\Delta].\ \mathsf{SK}^*\ \{P\}\ A\ \{Q\} \ \triangleq\ \mathsf{SK}\ A\ (P, Q)^*_\Delta$$

where $(P, Q)^*_\Delta$ is the A-specification defined as follows:

$$(P, Q)^*_\Delta \triangleq (\lambda i.\ \exists \Delta.\, i \in P,\ \ \lambda r\, i\, m.\ \forall \Delta.\, i \in P \to m \in Q)$$

The second step is to introduce the type $\mathsf{Kont}\ A\ R\ Q$ of continuation objects. R:preT is the initial condition, describing the heap at the point of a jump. Q:postT A is a binary postcondition relating the initial heap with the ending heap and value of the finalization code. The Kont type will always be in scope of a variable j denoting the heap at the point of continuation capture (the index of **callcc** in Figure 2.2); thus R and Q may depend on j as well.

We next show how the the type Kont^* from Section 2.2, is a notation over Kont, but first we need to generalize somewhat. As the following example illustrates, the Kont^* types actually require *two* different kinds of logical variables.

[3] The notation also explains the operator $(-)^*$ (with the empty context Δ), that was used in Figure 2.2.

Example 2.4. Consider a continuation object which is captured when the heap $j = x \mapsto v$, can be jumped to only when x is incremented by *some* p, and whose finalization code increments p further by 1. The scenario uses the program variable x and two logical variables, v and p. However, v and p clearly have different nature; v is implicitly universally quantified, while quantification over p is existential when describing the precondition for the jump, and universal in the description of the finalization code. Thus, we put v and p into two different contexts, and describe the continuation object by the type

$$[v], \langle p \rangle . \, \mathsf{Kont}^* \, \{j \in x \mapsto v\}\{x \mapsto v + p\} \, A \, \{x \mapsto v + p + 1\}$$

We refer to the two kinds of variables and contexts as *box* and *diamond* variables and contexts, respectively. In Figure 2.2 we only used the box contexts, and in general, when the diamond context is empty, we simply omit it. However, diamond context is not always empty, as we will illustrate in the **ping–pong** program in Section 2.6.2.

Definition 2.5 (Kont^*)**.** Let Δ and Γ be contexts of logical variables, and j be a distinguished heap variable, possibly occurring freely in $R : \mathsf{preT}$, $P : \mathsf{preT}$ and $Q : \mathsf{postT}\, A$. Then the notation $[\Delta], \langle \Gamma \rangle . \, \mathsf{Kont}^* \, \{R\}\{P\} \, A \, \{Q\}$, is an abbreviation for the following Kont type without logical variable contexts.

$$[\Delta], \langle \Gamma \rangle . \, \mathsf{Kont}^* \, \{R\}\{P\} \, A \, \{Q\} \triangleq$$
$$\mathsf{Kont} \; A \; (\lambda \, i. \; \forall \Delta . \, R \to \exists \, \Gamma . \, i \in P) \; (\lambda \, r \, m \, i. \; \forall \Delta . \, R \to \forall \Gamma . \, i \in P \to m \in Q)$$

All the variables in Δ are universally quantified in the notation, whereas the variables in Γ are existentially quantified in the initial condition, and universally in the postcondition.

We close the section by revisiting the issue of large footprint annotations. In Section 2.2, **inc3** could execute only in a heap with *exactly* the pointer x, and no others. We now explain how to specify **inc3** to admit execution in the presence of additional pointers. We will use a (box) logical variable of type heap, to describe the sub-heaps that *do not* contain x. For example, the more general annotation for **inc3** is given below.

$$
\begin{array}{l}
\textbf{inc3}\,(x : \mathsf{ptr}) : \ [v,h].\ \mathsf{SK}^*\ \{x \mapsto v * \mathsf{this}\,h\}\ ()\ \{x \mapsto v+3 * \mathsf{this}\,h\} \triangleq \\
\quad \textbf{do}\ c \leftarrow \textbf{callcc}_j \\
\qquad\qquad f : [v,h].\ \mathsf{Kont}^*\ \{j \in x \mapsto v * \mathsf{this}\,h\} \\
\qquad\qquad\qquad\qquad \{x \mapsto v+1 * \mathsf{this}\,h\}\ \Sigma\bullet\mathsf{SK}\,() \\
\qquad\qquad\qquad\qquad \{r.x \mapsto v+2 * \mathsf{this}\,h \\
\qquad\qquad\qquad\qquad\ \curlywedge\ \mathsf{spec}\,r \sqsubseteq (x \mapsto v+3 * \mathsf{this}\,h,\ x \mapsto v+3 * \mathsf{this}\,h)^*\}. \\
\qquad\quad \textbf{do}\ (\textbf{ret}\ [\textbf{abort}\ f\ (x := !x+1\,;\ \textbf{ret}\ [\textbf{ret}\ ()])]) \\
\qquad\quad : [v,h].\ \mathsf{SK}^*\{x \mapsto v * \mathsf{this}\,h\ \curlywedge\ j \in x \mapsto v * \mathsf{this}\,h\}\ \Sigma\bullet\mathsf{SK}\,() \\
\qquad\qquad\qquad\quad \{r.\ x \mapsto v * \mathsf{this}\,h\ \curlywedge\ \mathsf{spec} \sqsubseteq (x \mapsto v+1 * \mathsf{this}\,h, \bot)^*\}; \\
\qquad x := !x+1\,; \\
\qquad \textbf{cmd}\ c.
\end{array}
$$

The annotations of **inc3** introduce a logical variable h, the assertions $\mathsf{this}\,h$, and separating conjunction $*$, to name the part of the heap disjoint from the pointer x. The occurrence of the same $\mathsf{this}\,h$ in both the pre- and postcondition of **inc3**, specifies that **inc3** keeps this part of the heap invariant. Moreover, h is local to the Hoare triples in which it appears (unlike in first-order Hoare or Separation logic, where logical variables are global). Thus, a specification of **inc3** can be extended to a larger heap merely by instantiating h, rather than by means of a dedicated *frame rule* as in Separation logic. Because of the repeated occurrences of $\mathsf{this}\,h$ in the assertions, this style of annotation is a bit more verbose than in Separation logic, where the invariance of residual heaps is implicit. However, in practice, the extra logical variable did not affect our proofs. We discuss in Section 2.7 the reasons for needing large footprints in HTTcc, and some alternative designs.

2.4 Inference rules

We develop the semantics of HTTcc using the Calculus of Inductive Constructions (CIC) as the meta logic. The choice provides us directly with a method to prototype HTTcc as a shallow embedding in Coq, thus inheriting a number of useful constructs, such as dependent Π and Σ types, which we have already used in Section 2.2. For the sake of brevity, we omit the treatment of such standard constructs (it can be found in (The Coq Development Team, 2016; Bertot & Castéran, 2004)), and freely assume the standard typing rules and the various syntactic categories of CIC, such as, e.g., variable contexts. We only present our *impure* monadic extensions: the typing rules for the SK and Kont types, and related terms. In Section 2.5, we develop the semantic model for HTTcc, which we mechanized in Coq, to show the soundness of the extension.

The specific rules of HTTcc are of two distinct kinds. The first kind consists of *typing rules*, which serve to infer the default program specifications (weakest precondition for memory safety, and strongest postcondition wrt. that precondition). The inference is important in practice because it minimizes the amount of annotations that the user has to provide manually. The second kind consists of *structural lemmas* that formalize the reasoning about Hoare-ordering of specifications. As illustrated in Figure 2.2, such reasoning is needed in several situations: (1) We may explicitly need to ascribe a custom specification to a program, and this requires a proof that the default specification can be pre-weakened and post-strengthened into a desired one, and (2) We may use the relation $\sqsubseteq$, to explicitly declare that a $[-]$-abstracted command satisfies a predetermined specification. The two kinds of rules are discussed in Sections 2.4.1 and 2.4.2, respectively.

2.4.1 Typing rules

$$\begin{array}{ll}
\textbf{alloc}: & \Pi v{:}A.\,[h].\,\mathsf{SK}^*\ \{\mathsf{this}\,h\}\ \mathsf{ptr}\ \{r.\ r \mapsto v * \mathsf{this}\,h\} \\
\textbf{dealloc}: & \Pi x : \mathsf{ptr}.\,[B, v{:}B, h].\,\mathsf{SK}^*\ \{x \mapsto v * \mathsf{this}\,h\}\ ()\ \{\mathsf{this}\,h\} \\
\textbf{:=}: & \Pi x : \mathsf{ptr}.\,\Pi v : A.\,[B, w{:}B, h].\,\mathsf{SK}^*\{x \mapsto w * \mathsf{this}\,h\}\ ()\{x \mapsto v * \mathsf{this}\,h\} \\
\textbf{!}: & \Pi x : \mathsf{ptr}.\,[v{:}A, h].\,\mathsf{SK}^*\ \{x \mapsto v * \mathsf{this}\,h\}\ A\ \{r.\ x \mapsto v * \mathsf{this}\,h \curlywedge r = v\}
\end{array}$$

Figure 2.3: Typing assignment rules for the *heap-manipulating* fragment of HTTcc commands. Notice the *large-footprint* nature of these rules, as witnessed by the use of the logical variable h to explicitly quantify over the rest of the heap.

From here on, we use s and variants to range over specifications (P, Q), with $\mathsf{pre}\ s$ and $\mathsf{post}\ s$ projecting out the components.

Bind rule in Figure 2.4 perhaps best exemplifies the inference nature of our typing rules. Given programs e_1 and e_2 with specification s_1 and s_2, respectively, the rule infers the tightest specification for the sequential composition $x \leftarrow e_1; e_2$ as follows. Because the execution of the compositions starts with e_1, the inferred precondition must require that $\mathsf{pre}\,s_1$ holds of the initial heap i. After e_1 terminates with an intermediate value x and heap h satisfying $\mathsf{post}\,s_1\,x\,i\,h$, it must be that $\mathsf{pre}\,(s_2\,x)\,h$ so that e_2 is safe to run. The inferred postcondition declares the existence of an intermediate value x, and and a heap obtained after e_1 but before e_2, as per relational composition $\mathsf{post}\,s_1\,x \circ \mathsf{post}\,(s_2\,x)\,r$. BIND, as well as the other rules in Figure 2.4, use SK and Kont types *without* logical variable contexts, which is why we introduced such types in Section 2.3 in the first place. Omitting logical variables facilitates specification inference, as it circumvents the issue of

$$\frac{\Gamma \vdash e_1 : \mathsf{SK}\ A\ s_1 \qquad \Gamma, x : A \vdash e_2 : \mathsf{SK}\ B\ (s_2\, x)}{\Gamma \vdash x \leftarrow e_1;\, e_2 : \mathsf{SK}\ B\ (\lambda i.\, \mathsf{pre}\ s_1\, i \wedge \forall x\, h.\, \mathsf{post}\ s_1\, x\, i\, h \rightarrow \mathsf{pre}\ (s_2\ x)\, h,\ \lambda r\, i\, m.\ \exists x.(\mathsf{post}\ s_1\, x \circ \mathsf{post}\ (s_2\ x)\, r)\ i\, m)}\ \textsc{Bind}$$

$$\frac{\Gamma,\, j : \mathsf{heap},\, f : \mathsf{Kont}\, A\, (R\, j)\, (Q\, j) \vdash\ e : \mathsf{SK}\ A\ (s\, j)}{\Gamma \vdash \mathbf{callcc}_j\, f\, .e :\ \mathsf{SK}\ A\ (\lambda i.\ \mathsf{pre}\, (s\, i)\, i,\ \lambda r\, i\, m.\ \mathsf{post}\, (s\, i)\, r\, i\, m \vee (R \circ Q\, i\, r)\ i\, m)}\ \textsc{CallCC}$$

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{return}\, v :\ \mathsf{SK}\ A\ (\lambda i.\ \mathsf{True},\, \lambda r\, i\, m.\ m = i \wedge r = v)}\ \textsc{Ret}$$

$$\frac{\Gamma \vdash f : \mathsf{Kont}\, A\, R\, Q \qquad \Gamma \vdash e : \mathsf{SK}\ A\ s}{\Gamma \vdash\ \mathbf{abort}_B\ f\ e : \mathsf{SK}\ B\ (\lambda i.\ R\, i \wedge\ s \sqsubseteq (R \curlywedge \mathsf{this}\, i, Q),\, \lambda r\, i\, m.\ \mathsf{False})}\ \textsc{Abort}$$

$$\frac{\Gamma \vdash\ e_1 : \mathsf{SK}\ A\ s_1 \quad s_1 \sqsubseteq s_2}{\Gamma \vdash \mathbf{do}\ e_1 : \mathsf{SK}\ A\ s_2}\ \textsc{Do}$$

$$\frac{\Gamma \vdash\ e : \mathsf{bool} \qquad \Gamma \vdash e_1 : \mathsf{SK}\ A\ s_1 \qquad \Gamma \vdash e_2 : \mathsf{SK}\ A\ s_2}{\Gamma \vdash \mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 : \mathsf{SK}\ A\ (\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2)}\ \textsc{If}$$

$$\frac{\Gamma \vdash f\ : (\Pi\, x{:}A.\, \mathsf{SK}\ (B\, x)\ (s\, x)) \rightarrow \Pi\, x{:}A.\, \mathsf{SK}\ (B\, x)\ (s\, x)}{\Gamma \vdash \mathbf{fix}\, f : \Pi\, x{:}A.\, \mathsf{SK}\ (B\, x)\ (s\, x)}\ \textsc{Fix}$$

Figure 2.4: HTTcc typing rules for specification inference.

reconciling potentially different contexts of logical variables that may appear in the type for e_1 and the type for e_2.

Abort rule infers a precondition that has a dual role. The first conjunct $R\,i$ ensures that the continuation object f is **abort**ed to only in heaps i for which the initial condition R is satisfied. The second conjunct $s \sqsubseteq (R \curlywedge \mathsf{this}\, i, Q)$ ensures that e is an appropriate finalization code for f; that is, the specification s of e can be weakened into a precondition R and postcondition Q, as required by the type of f. Additionally, $\mathsf{this}\, i$ allows the proof of the weakening to exploit the knowledge that the heap in which the finalization code executes is exactly i. The exact definition of $\sqsubseteq$ will be given in Section 2.4.2. Because **abort** does not return any values, its return type B is arbitrary and can be supplied by the user. We omit annotating this type in examples as it can be usually inferred from the context.

CallCC rule in Figure 2.4, infers the specification for **callcc** $f.\, e$. The premise of the rule introduces the heap variable j, which, as illustrated in Section 2.2, provides a common point for f and e to "synchronize" on, thereby fixing the values of various logical variables in relation to j. In the conclusion of the rule, j will be instantiated with the initial heap of **callcc**; that is, with the heap at the point of continuation capture.

The specification of f allows aborting to f only in heaps satisfying $R\ j$. After f's finalization code terminates, the resulting heap and value satisfy $Q\ j$. If e has a specification s, then the tightest specification of **callcc** $f.\, e$ can be inferred as follows. Because the execution of the whole command starts with e, the inferred precondition has to be derived out of e's precondition $\mathsf{pre}\,(s\, j)$. The unknown j is instantiated with the actual initial heap, to obtain the tightest precondition $\lambda\, i.\, \mathsf{pre}\,(s\, i)\, i$. The postcondition is a disjunction expressing that e may produce two different outcomes: it either aborts to f, or it does not. The left disjunct $\mathsf{post}\,(s\, i)\, r\, i\, m$ describes the non-aborting case; it simply equals the postcondition of e with j instantiated by i. In those cases when e actually aborts, the disjunct will be False, as it embeds the postcondition of the Abort rule. The right disjunct is a relational composition $(R \circ Q\, i\, r)\, i\, m$ which describes the aborting case, as follows. In the aborting case, there exists a heap—call it h—that is current at the point of **abort**. Due to the specification of f, R must relate i and h. Additionally, m and r are obtained after the finalization code of f is executed at h, and thus $Q\, i\, r$ must relate h and m as well.

The annotations specifying the continuation object (R and Q) appear in a negative position in the premise of CallCC, and cannot be inferred automatically. In contrast, the specification s for e is generated by e's typing derivation. The property that *only* the aborting case has to be specified manually, differentiates the algebraic **callcc** from the standard, non-algebraic alternatives. A detailed

comparison is presented in Section 2.7.

Other rules The rule Do implements a type ascription, requiring a proof that the specification s_1 can be weakened into s_2, as in the usual Hoare logic rule of consequence. The If rule uses the type-level conditional, available in CIC, to compute the specification of a program-level conditional out of the types of the components. The Fix rule implements the usual typing for recursive procedures, requiring that an SK type be established for the procedure body, under a hypothesis that the recursive calls satisfy the same type. The primitive stateful commands have standard Separation logic specifications (Figure 2.3), except that they are extended to the large footprint idiom by naming the unused heap with the logical variable h.

Additional notation In the rest of this chapter, we will use explicit names for the derived specifications in Figure 2.4. For example, we write $\mathsf{callcc\_s}\ R\ Q\ s$ for $(\lambda i.\ \mathsf{pre}\ (s\,i)\,i, \lambda r\,i\,m.\ \mathsf{post}\ (s\,i)\,r\,i\,m \vee (R \circ Q\,i\,r)\ i\,m)$, $\mathsf{abort\_s}\ R\ Q\ s$ for $(\lambda i.\ R\ i \wedge s \sqsubseteq (R \curlywedge \mathsf{this}\,i, Q), \lambda r\,i\,m.\,\mathsf{False})$, and similarly for $\mathsf{bind\_s}$, $\mathsf{read\_s}$, etc.

2.4.2 Structural lemmas and symbolic evaluation

The Hoare ordering on two specifications s_1 and s_2 is defined as:

$$s_1 \sqsubseteq s_2 \triangleq \forall i.\ \mathsf{pre}\ s_2\,i \to \mathsf{verify}\,i\,s_1\,(\lambda r{:}A\,m.\ \mathsf{post}\ s_2\,y\,i\,m)$$

where for any $\kappa : A \to \mathsf{heap} \to \mathsf{prop}$:

$$\mathsf{verify}\,i\,s\,\kappa \triangleq \mathsf{pre}\ s\,i \wedge \forall r\,m.\,\mathsf{post}\ s\,r\,i\,m \to \kappa\ r\ m$$

The definition of $s_1 \sqsubseteq s_2$ states that $\mathsf{pre}\ s_1$ weakens into $\mathsf{pre}\ s_2$ and $\mathsf{post}\ s_2$ strengthens into $\mathsf{post}\ s_1$, as in the Hoare logic rule of consequence. It is split into two stages in order to exploit the hypothetical reasoning of CIC and Coq. In practice, the hypothesis $\mathsf{pre}\ s_2\ i$ will always move into the hypothesis context of Coq, leaving verify to describe the remaining proof goal.

The HTTcc proof obligations arise directly from the side condition about Hoare ordering in the rule Do in Figure 2.4, when the user wants to ascribe a desired specification to a program. In the practical work with HTTcc, the proof obligations are discharged by applying a number of carefully crafted lemmas about verify that implement verification by *symbolic evaluation*. We illustrate the process here, and discuss the relationship to algebraicity of control operators in Section 2.4.3.

For instance, let e_1 and e_2 be computations with specifications s_1 and $s_2\ x$, respectively. The inferred specification for $x \leftarrow e_1; e_2$ is $\mathsf{bind\_s}\ s_1\ s_2$. An HTTcc proof obligation establishing that in some heap i, the execution of the sequential composition produces a heap and a value satisfying κ, will have the form:

$$\mathsf{verify}\ i\ (\mathsf{bind\_s}\ s_1\ s_2)\,\kappa$$

The idea of symbolic evaluation is to discharge such a goal as follows. It suffices to show that verifying a composite B-spec ($\mathsf{bind\_s}\ s_1\ s_2$) can be reduced to verifying s_1 against a κ', where κ' itself involves verifying s_2 against κ. The process is iterated as long as s_2 contains sequential compositions, and can be seen as a sequence of applications of the following lemmas in HTTcc:

$$\textsc{Step} \quad : \mathsf{verify}\ i\ s_1(\lambda\, y\, m.\ \mathsf{verify}\ m\ (s_2\ y)\ \kappa) \to \mathsf{verify}\ i\ (\mathsf{bind\_s}\ s_1\, s_2)\, \kappa$$
$$\textsc{ValDo} : \mathsf{pre}\ s\ i \to (\forall\, x\, i\, m.\ \mathsf{post}\ s\, x\, i\, m \to \kappa\, x\, m) \to \mathsf{verify}\ i\ s\, \kappa$$

The Step lemma implements the iterative step, and ValDo lemma applies in the end, when there are no outstanding sequential compositions to be stepped through.

The ValDo lemma may be specialized to streamline the symbolic evaluation for specific commands. For example, let $f : \mathsf{Kont}\, A\, R\, Q$ and $e : \mathsf{SK}\, A\, s$. The inferred spec for **abort** $f\ e$ is $\mathsf{abort\_s}\ R\ Q\ s = (\lambda\, i.\ R\, i \wedge s \sqsubseteq (R, Q), \lambda\, r\, i\, m.\, \mathsf{False})$. Taking this spec for s in ValDo, after some simplification, we obtain:

$$\textsc{ValAbort} : \ R\ i \to \mathsf{verify}\ i\ s\ (\lambda r.\ \lambda m.\ Q\, r\, i\, m) \to \mathsf{verify}\ i\ (\mathsf{abort\_s}\ R\ Q\ s)\, \kappa$$

In other words, to verify **abort** to f, we need to show that R holds of the current heap, and that the supplied finalization code satisfies Q after running in i.

In case $e_1 = \mathbf{callcc}_j\, f.\, e$, the inferred spec is

$$\mathsf{callcc\_s}\ R\ Q\ s = (\lambda\, i.\ \mathsf{pre}\, (s\, i)\, i, \lambda\, r\, i\, m.\ \mathsf{post}\, (s\, i)\, r\, i\, m \vee (R \circ Q\, i\, r)\ i\, m)$$

for some j and $f : \mathsf{Kont}\, A\, (R\ j)\, (Q\ j)$ and $e : \mathsf{SK}\ A\ (s\ j)$. Taking this spec for s, and after some rearrangement of the disjunction in the postcondition of callcc_s, ValDo can be specialized into the following lemma.

$$\begin{aligned}\textsc{ValCC} : \ & \mathsf{verify}\, i\, (s\ i)\, \kappa \to (\forall\, r{:}A\, m.\ (R \circ (Q\, i\, r))\, i\, m \to \kappa\, r\, m) \to \\ & \mathsf{verify}\, i\, (\mathsf{callcc\_s}\ R\, Q\, s)\, \kappa\end{aligned}$$

The first hypothesis corresponds to the case when e does not abort. In that case, the goal reduces to verifying $(s\ i)$ against κ. The second hypothesis corresponds to the aborting case. In that case e produces an ending heap m and value x satisfying $(R \circ (Q\ i))\, x\, i\, m$; that is e first reaches an aborting heap out of i (predicate R), and then executes the finalization code $(Q\ i)$. Then we just need to prove that κ holds after the execution of the finalization code.

Specialised symbolic execution lemmas can be proved for all other primitive commands. Furthermore, because verify is an ordinary logical definition, users can establish such lemmas for their own programs as well, directly in the logic.

Example 2.6. We can implement the usual **throw** command, as an **abort** with an immediately-returning finalization code. Given $f : \mathsf{Kont}\, A\, R\, Q$ and a value $v{:}A$, we define:

$$\mathbf{throw}\ g\ v \triangleq \mathbf{abort}\ f\ (\mathbf{ret}\, v)$$

The inferred specification $\mathsf{throw\_s}\ R\ Q\ v = \mathsf{abort\_s}\ R\ Q\ (\mathsf{ret\_s}\ v)$ can be proved to satisfy a streamlined version of ValAbort, which exploits the trivial nature of the finalization code to simplify one of the hypotheses.

$$\textsc{ValThrow}: \quad R\ i \to Q\ v\ i\ i \to \mathsf{verify}\ i\ (\mathsf{throw\_s}\ R\ Q\ v)\ \kappa$$

2.4.3 Algebraicity at the level of specifications

In this section, we show that the algebraicity property of **callcc** can be expressed as a lemma over specifications, similar to the symbolic evaluation lemmas from the previous section. This lemma, which we call AlgCC, expresses that we can commute the specification forms $\mathsf{bind\_s}$ and $\mathsf{callcc\_s}$, thus lifting to the level of specifications the algebraicity equation (1) from Section 2.2. However, stating the AlgCC lemma requires generalizing somewhat the definition of $\mathsf{callcc\_s}\ R\ Q\ s_1$. We first introduce the generalized definition, which we rename $\mathsf{algcc\_s}\ R\ s_0\ s_1$, and then describe the intuition behind it.

Consider a program of the form

$$x \leftarrow (\mathbf{callcc}_j\ f.\ e_1); e_2$$

where $f : \mathsf{Kont}\ A\ (R\ j)\ (Q\ j)$, $e_1 : \mathsf{SK}\ A\ (s_1\ j)$, and $x{:}A \vdash e_2 : \mathsf{SK}\ B\ (s_2 x)$. Then, side-by-side, the two definitions look like:

$$\begin{aligned}
\mathsf{callcc_s}\ R\ Q\ s_1 &= (\lambda\, i.\ \mathsf{pre}\ (s_1\ i)\ i, \\
&\qquad \lambda\, r\ i\ m.\ \mathsf{post}\ (s_1\ i)\ r\ i\ m \vee (R \circ Q\ i\ r)\ i\ m) \\
\mathsf{algcc_s}\ R\ s_0\ s_1 &= (\lambda\, i.\ \mathsf{pre}\ (s_1\ i)\ i \wedge \forall\, h. R\ i\ h \to \mathsf{pre}\ (s_0\ i)\ h, \\
&\qquad \lambda\, r\ i\ m.\ \mathsf{post}\ (s_1\ i)\ r\ i\ m \vee (R \circ \mathsf{post}\ (s_0\ i)\ r)\ i\ m)
\end{aligned}$$

The main difference is that where $\mathsf{callcc\_s}$ uses only the postcondition Q (abstracting over j) to specify the finalization code, $\mathsf{algcc\_s}$ takes a full specification s_0 (also abstracting over j), which includes a precondition as well. In $\mathsf{callcc\_s}$, the precondition for the finalization code is assumed to be the trivially true heap predicate, and we can prove the equation

$$\mathsf{callcc\_s}\ R\ Q\ s_1 = \mathsf{algcc\_s}\ R\ (\lambda\, j.\ (\top, Q\ j))\ s_1.$$

Intuitively, $\mathsf{callcc\_s}$ could use the trivial precondition for the finalization code, because R already describes what holds of the heap at the point of abort, and this is the same heap in which the finalization code executes. However, the main property of algebraic commutation is that it changes the finalization code by extending it with e_2, though it does not change the point where the continuation is aborted. The new definition $\mathsf{algcc\_s}$ thus divorces R and $\mathsf{pre}\ s_0$, so that the algebraicity lemma can express that R remains fixed, while $\mathsf{pre}\ s_0$ changes. The changes to $\mathsf{pre}\ s_0$ cannot be arbitrary, however, as the finalization code always

executes in a heap in which **abort** is called. Thus, the precondition of algcc_s includes a conjunct that R implies pre s_0 for every initial heap i, and aborting heap h.

We can now state the algebraicity lemma for **callcc**, with the omitted proof included in our Coq files (Delbianco & Nanevski, 2017).

$$\begin{aligned}\textsc{AlgCC}: \quad & \mathsf{verify}\, i\ (\mathsf{bind\_s}\ (\mathsf{algcc\_s}\ R\, s_0\, s_1)\, s_2)\ \kappa \quad \longleftrightarrow \\ & \mathsf{verify}\, i\ (\mathsf{algcc\_s}\ R\ (\lambda\, j.\, \mathsf{bind\_s}\ (s_0\ j)\, s_2)\ (\lambda\, j.\, \mathsf{bind\_s}\ (s_1\, j)\, s_2))\ \kappa\end{aligned}$$

2.5 Denotational semantics

In this section we present the semantics of HTTcc as a shallow embedding into CIC. That is, we provide a semantic interpretation function $[\![-]\!]$ that maps HTTcc types SK and Kont into types defined in CIC, but acts as identity on all the other types inherited from CIC, such as nat or heap. Similarly, the interpretation of terms maps the monadic constructs such as **callcc**, **abort**, etc., into CIC-terms, while acting as an identity on all the other terms inherited from CIC. The interpretation extends homomorphically to variable contexts as well.

2.5.1 Semantics of types

As customary in the case of control operators, our denotational semantics is parameterized by the type X of return results for the continuations. We further require that X is a *complete lattice*, thus providing us with means to model the fixed point combinator in CIC, using the Knaster-Tarski theorem.

Definition 2.7. Given a type A, and predicates P : preT and Q : postT A, the types SK A (P, Q) and Kont $A\, P\, Q$ are interpreted as follows.

$$[\![\mathsf{SK}\ A\ (P, Q)\,]\!] \triangleq \Pi i{:}\mathsf{heap}.\, [\![P]\!]\, i \to (\Pi r{:}[\![A]\!].\, \Pi m{:}\mathsf{heap}.\, [\![Q]\!]\, r\, i\, m \to X) \to X$$

$$[\![\mathsf{Kont}\, A\, P\, Q]\!] \triangleq [\![\mathsf{SK}\ A\ (P, Q)]\!] \to \Pi j{:}\mathsf{heap}.\, [\![P]\!]\, j \to X$$

A computation of SK type takes a heap i satisfying the precondition P, and a *continuation* requiring the postcondition Q as a precondition. Intuitively, the continuation applies to the ending value and heap of the computation, to produce a result in X. As the latter type is a complete lattice, the possible results include divergence, denoted by the bottom element of the lattice. We don't model the faulting behaviour such as type or memory errors (e.g., de-referencing a dangling pointer), but instead rely on the proofs of $[\![P]\!]\, i$ and $[\![Q]\!]\, r\, i\, m$ to statically ensure that a computation executes only in heaps and continuations for which such errors do not occur. In this sense, well-typed (i.e., well-specified) computations in HTTcc do not fault, as usual for type systems and for *fault-avoiding* Hoare logics such as Separation logic.

A further useful intuition about our model may be gained if one erases the dependencies on P and Q in Definition 2.7. This results in the following informal equations:

$$\begin{aligned}[\![\mathsf{SK}\,A]\!] &= \mathsf{heap} \to ([\![A]\!] \to \mathsf{heap} \to X) \to X\\ [\![\mathsf{Kont}\,A]\!] &= [\![\mathsf{SK}\,A]\!] \to \mathsf{heap} \to X\end{aligned}$$

The first equation shows that the $\mathsf{SK}\,A$ type is essentially the standard *state-passing continuation monad*. The second equation shows that a continuation object semantically requires two arguments: an explicit finalization code (the $\mathsf{SK}\,A$ type), and a heap. Of course, the finalization code is explicitly provided by the program as an argument of abort. Importantly, the heap argument is *implicitly* supplied by the denotation of **abort** as the heap current at the point of aborting to f.

Of course, the parameterization with finalization code is what makes our control operators algebraic, and is directly inspired by Jaskelioff (Jaskelioff, 2009). On the other hand, the further heap argument makes the $\mathsf{Kont}\,A$ type implement *non-rollbacking* continuations. In contrast, the algebraic **callcc** presented by Jaskelioff for a state and continuations monad does roll-back the state to the one captured together with the current continuation.

2.5.2 Semantics of computations

For the sake of brevity, we present only the denotational semantics of the control operators **callcc** and **abort**. Moreover, we illustrate only the simplified setting where the dependencies on P and Q are erased from the types (as in the above informal equations), and the dependencies on the proofs of $[\![P]\!]\,i$ and $[\![Q]\!]\,r\,i\,m$ are erased from the terms. The full dependency-respecting denotations for all the monadic commands from Figures 2.4 and 2.3 are implemented in the companion Coq files.

Definition 2.8 (**callcc** $f.\,e$)**.** Given $f : [\![Kont\,A]\!]$ and $e : [\![\mathsf{SK}\,A]\!]$, the denotation of **callcc** $f.\,e$ of type $[\![\mathsf{SK}\,A]\!]$ is defined as: [4]

$$\begin{aligned}[\![\mathbf{callcc}]\!] &: ([\![\mathsf{Kont}A]\!] \to [\![\mathsf{SK}\,A]\!]) \to [\![\mathsf{SK}\,A]\!]\\ [\![\mathbf{callcc}\,f.\,e]\!] &\triangleq \lambda i : \mathsf{heap}.\,\lambda k : [\![A]\!] \to \mathsf{heap} \to X.\\ &\qquad [\lambda c : [\![\mathsf{SK}\,A]\!].\,\lambda h : \mathsf{heap}.\,c\ h\ k/f][\![e]\!]\ i\ k\end{aligned}$$

Intuitively, executing **callcc** $f.\,e$ corresponds to applying the denotation to the initial (i.e., captured) heap i and continuation k. The body e is executed using the same heap and continuation. However, first the variable $f : [\![\mathsf{Kont}\,A]\!]$ is bound to a continuation object that, when supplied the finalization code c and a heap h that is current at the point of aborting to f, executes c in h passing the control (i.e., jumping) to k.

[4] Omitting the index j to **callcc**, which may only appear in the erased dependencies.

Definition 2.9 (**abort**$_B$ f e)**.** Given $f : [\![\mathsf{Kont}\, A]\!]$ and $e : [\![\mathsf{SK}\, A]\!]$, the denotation of **abort**$_B$ f e of type $[\![\mathsf{SK}\, B]\!]$ is given by:

$$\begin{aligned}&[\![\mathbf{abort}_B]\!] \quad : [\![\mathsf{Kont}\, A]\!] \to [\![\mathsf{SK}\, A]\!] \to [\![\mathsf{SK}\, B]\!]\\ &[\![\mathbf{abort}_B\ f\ e]\!] \triangleq \lambda i : \mathsf{heap}.\, \lambda k : [\![B]\!] \to \mathsf{heap} \to X.\ [\![f]\!]\ [\![e]\!]\ i\end{aligned}$$

Theorem 2.10 (Soundness)**.** *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash_{CIC} [\![e]\!] : [\![A]\!]$.

Proof. The proof is by induction on the structure of e. The interesting cases are when e is one of the monadic commands (correspondingly, when A is an SK type), as in all other cases the semantic function is trivial. When e is a monadic command, the soundness proof for the command is intertwined with the definition of the denotation of e. For example, in the case of $e = \mathbf{callcc}_j\ f.\, e$ the denotation will involve parameterization on the proofs of $[\![P]\!]\, i$ and $[\![Q]\!]\, r\, i\, m$, for the appropriate P and Q, that we have simplified away in our discussion. Such proof parameters are used in the denotations to build larger proofs on-the-fly, as necessary to make the various sub-terms of the denotation type-check, until ultimately the whole denotation term type-checks wrt. the specification given in Figure 2.4. Similar considerations apply for other monadic terms as well. One exception is the fixed point construct **fix** f, whose soundness is proved by an appeal to Knaster-Tarski theorem over the monotone completion of f. The Knaster-Tarski theorem applies because $[\![ST\, A\, (P, Q)]\!]$ is a complete lattice, being defined as a function space into a complete lattice X. We have mechanized all the steps of the proof in our Coq mechanization. □

2.6 A short verification survey

We present the verification of two examples exhibiting non-trivial patterns for programming with continuations. The first example, **rember-up-to-last**, illustrates *downwards or exit* continuations, whereby a jump is used to escape early from a recursive call, whilst discarding suspended computations. The second example, **ping–pong**, illustrates *upwards* continuations or *unstructured* loops, where the executions of a captured continuation are interleaved with user code, thereby mimicking the cooperating behaviour of (a simplified version of) coroutines. In the companion files (Delbianco & Nanevski, 2017), we verify other examples as well, such as escaping from infinite loops and using continuations to implement error handlers.

2.6.1 rember-up-to-last

Let A be a type supporting decidable equality; i.e., there exists a function $== : A \to A \to \mathsf{bool}$. Given $x : A$ and a list $xs : \mathsf{list}\, A$, **rember-up-to-last** $x\, xs$, returns the ending segment of xs after the last occurrence of x; if x does not occur

1. **rember-up-to-last** $(x : A)\,(xs :\ \mathsf{list}\,A) :$
$\quad [h].\,\mathsf{SK}^*\ \{\mathsf{this}\,h\}\ (\mathsf{list}\,A)\ \{r.\ \mathsf{this}\,h\ \curlywedge\ r = \mathsf{rember}\ x\,xs\,[\,]\} \triangleq$
2. $\quad$ **do** $(\mathbf{callcc}_j$
$\quad\quad exit\ :\ [h].\,\mathsf{Kont}^*\ \{j \in \mathsf{this}\,h\}\ \{\mathsf{if}\ x \in xs\ \mathsf{then}\ \mathsf{this}\,h\ \mathsf{else}\ \bot\}$
$\quad\quad\quad\quad (\mathsf{list}\,A)\ \{r.\ \mathsf{this}\,h\ \curlywedge\ r = \mathsf{rember}\ x\,xs\,[\,]\}.$
3. $\quad\quad$ **fix** $(\lambda\, f :\ \mathsf{remberT}.\ \lambda\, ys : \mathsf{list}\,A.$
4. $\quad\quad\quad$ if ys is $y :: ys'$ then
5. $\quad\quad\quad\quad zs \leftarrow f\ ys';$
6. $\quad\quad\quad\quad$ if $x == y$ then **throw** $exit\ zs$
7. $\quad\quad\quad\quad$ else **ret** $(y{::}zs)$
8. $\quad\quad\quad$ else **ret** $[\,])\ xs)$

where

$$\mathsf{remberT} \triangleq \Pi ys : \mathsf{list}\,A.\ \mathsf{SK}^*\ \{\mathsf{this}\,j\ \curlywedge\ \exists ps.\, xs = ps{+\!\!+}ys\}\ (\mathsf{list}\,A)\ \{r.\ \mathsf{this}\,j\ \curlywedge\ r \in \mathsf{remberP}\ x\,ys\,[\,]\}$$

$$\mathsf{remberP}\ x\ xs\ acc \triangleq \begin{array}{l}\text{if } xs \text{ is } y :: ys' \\ \text{then if } x == y \text{ then } \emptyset \text{ else } \mathsf{remberP}\ x\ ys'\ (acc{+\!\!+}[y]) \\ \text{else } \{ps \mid ps = acc\}\end{array}$$

Figure 2.5: **rember-up-to-last** in HTTcc.

```
1. rember (x : A) (xs acc : list A) : list A ≜
2.         if xs is y::ys' then
3.             if x == y then rember x ys' [ ]
4.             else rember x ys' (acc++[y])
5.         else acc.
```

Figure 2.6: Purely functional rember.

in xs, the whole xs is returned (Friedman & Felleisen, 1996, pp. 55). For example, given $xs = [23, 16, 42, 4, 42, 8, 15, 16]$, we have:

$$\begin{aligned}
&\textbf{rember-up-to-last}\ \ 4\ xs = [42, 8, 15, 16]\\
&\textbf{rember-up-to-last}\ 16\ xs = [\,]\\
&\textbf{rember-up-to-last}\ 42\ xs = [8, 15, 16]\\
&\textbf{rember-up-to-last}\ \ 7\ xs = xs
\end{aligned}$$

Figure 2.5 implements **rember-up-to-last** in HTTcc, following the ML implementation given in Hayo Thielecke's phD thesis (Thielecke, 1997). For simplicity, we use purely-functional lists instead of imperative, heap allocated lists. The implementation works as follows. In line 2, it captures the continuation with **callcc**. Then, it recurses over the input list xs, searching for x (lines 3–5), rebuilding the input list on the way back (line 7). If x is found (line 6), it jumps out of the loop, by throwing to the captured continuation (Example 2.6). The returned value zs is the list rebuilt so far, while the outstanding iterations of the loop intended to further rebuild the list, are discarded.

We also develop a purely-functional version rember (Figure 2.6) by induction on xs. rember does not use **callcc**, but instead relies on *tail recursion* and the accumulator *acc* to keep track of the currently rebuilt list. Of course, the implementation with jumps is preferable from these efficiency standpoint, but we require the pure rember in order to *specify* **rember-up-to-last**.

We now analyse the type annotations presented in Figure 2.5. The specification given to **rember-up-to-last** (line 1) is quite intuitive. The function can run in an arbitrary heap h (this h in the precondition). It leaves the heap unchanged (this h in the postcondition), and the returned result r is the same as running the tail-recursive rember with the empty initial accumulator. The continuation object *exit* (line 2) is specified as follows. The initial condition exposes that the captured heap j equals h. A jump to exit may occur only when the precondition is satisfied; in this case, only when $x \in xs$. The postcondition states that the heap is unchanged, and the return value equals running rember, as expected.

The most interesting annotation is the *loop invariant* remberT provided as the

type of the recursive function f. The precondition states that f is only ever applied to the tail ys of xs; hence xs can be partitioned as $ps{+\!\!+}ys$. The partitioning is made explicit in the precondition, as it will be required when proving that we are throwing the correct ending segment in line 6. In the postcondition, remberT has to state that f produces the correct result. Importantly, however, it cannot use the helper function rember. The latter function requires an accumulator argument, but as f itself *is not* tail recursive, it is not clear which value to supply for the accumulator. Unlike the specifications of **rember-up-to-last** and exit, it is incorrect to use $[\,]$, as that does not reflect the looping behaviour. It is also incorrect to existentially abstract over the accumulator, since that produces too weak a property which then does not imply the postcondition of **rember-up-to-last**, where the accumulator is instantiated to $[\,]$.

The workaround is to define a helper function remberP, which is similar to rember, but records when the jumps in f appear, as follows. remberP returns either an empty *set* of values, to signalize a jump, or a singleton set, with the correct value, in the non-jumping case. More precisely, we have the following lemma:

$$\mathsf{rmb\_rmbP} : r \in \mathsf{remberP}\ x\ xs\ acc\ \rightarrow r = \mathsf{rember}\ x\ xs\ acc.$$

Then, the loop invariant remberT can assert that the return value r is always in the set defined by remberP. In the case of a jump, this property is evidently false, since the set is empty. But, in such a case, the program behaviour is described by the annotation on exit anyway, and the loop invariant need not bother describing it again.

Figure 2.7 presents a proof outline for **rember-up-to-last** $x\ xs$. The trivially true assertion $\exists h.\,\mathsf{this}\,h$ in line 1 corresponds to unfolding the notation for the type SK^* with a precondition $\mathsf{this}\,h$, and a logical variable h. In line 3, the current heap h is captured into the variable j, corresponding to substituting h for j in the rule BndCC, Section 2.4. In the proof outline, we cannot represent the substitution because j appears in the rest of the code, so instead we equate j with h in the assertion. Line 6 starts the verification of the loop; it shows that the precondition of the loop invariant remberT holds; the current heap is unchanged with regard to the captured one, and there exists a partition $xs = ps{+\!\!+}ys$. One critical point in the proof is Line 13, where we need to establish $zs = \mathsf{rember}\ x\ xs\ [\,]$, which is the precondition for **throw**. This property can be proved out of the partitioning $xs = ps{+\!\!+}(x :: ys')$ and $zs \in \mathsf{remberP}\ x\ ys'\ [\,]$ available in Line 12, by using rmb_rmbP and additional two helper lemmas, whose derivation we elide:

$$\begin{array}{ll} \mathsf{rmb\_cat}: & \mathsf{rember}\ x\ (ps +\!\!+ ys)\ acc = \mathsf{rember}\ ys\ (\mathsf{rember}\ x\ acc\ ps) \\ \mathsf{rmb\_in}: & \mathsf{rember}\ x\ xs\ acc = \mathsf{if}\ x \in xs\ \mathsf{then}\ \mathsf{rember}\ x\ xs\ [\,]\ \mathsf{else}\ (acc +\!\!+ xs) \end{array}$$

Another critical point is line 17, where we need to prove $y{::}zs \in \mathsf{remberP}\ x\ (y{::}ys')\ [\,]$, required to establish the postcondition of the loop. The property is proved out

1. $\{\exists\, h.\ \mathsf{this}\, h\}$
2. $\mathbf{do}\ (\ \mathbf{callcc}_j\ exit\ .$
3. $\{\exists\, h.\ \mathsf{this}\, h \curlywedge j = h\}$
4. $\{\mathsf{this}\, j \curlywedge xs = [\,]{+\!\!+}xs\}$
5. $\quad \mathbf{fix}\ (\lambda\ f\ ys.$
6. $\{\exists\, ps.\ \mathsf{this}\, j \curlywedge xs = ps{+\!\!+}ys\}$
7. $\qquad\qquad\qquad$ if ys is $y :: ys'$ then
8. $\{\exists\, ps.\ \mathsf{this}\, j \curlywedge xs = ps{+\!\!+}(y :: ys')\}$
9. $\qquad\qquad\qquad\quad zs \leftarrow f\ ys';$
10. $\{\exists\, ps.\ \mathsf{this}\, j \curlywedge xs = ps{+\!\!+}(y :: ys') \curlywedge zs \in \mathsf{remberP}\, x\, ys'\, [\,]\}$
11. $\qquad\qquad\qquad\quad \mathbf{if}\ x == y\ \mathbf{then}$
12. $\{\exists\, ps.\ \mathsf{this}\, j \curlywedge xs = ps{+\!\!+}(x :: ys') \curlywedge zs \in \mathsf{remberP}\, x\, ys'\, [\,]\}$
13. $\{\mathsf{this}\, j \curlywedge x \in xs \curlywedge zs = \mathsf{rember}\ x\ xs\ [\,]\}$
14. $\qquad\qquad\qquad\qquad\quad \mathbf{throw}\ exit\ zs$
15. $\{\bot\}$
16. $\qquad\qquad\qquad\quad \mathbf{else\ ret}\ \ (y :: zs)$
17. $\exists\, ps.\ \mathsf{this}\, j \curlywedge xs = ps{+\!\!+}(y :: ys') \curlywedge y :: zs \in \mathsf{remberP}\, x\, (y :: ys')\, [\,]\}$
18. $\qquad\qquad\qquad$ else $\mathbf{ret}\ \ [\,]$
19. $\{\exists\, ps.\ \mathsf{this}\, j \curlywedge xs = ps \curlywedge [\,] \in \mathsf{remberP}\, x\, [\,]\, [\,]\}$
20. $\qquad\qquad\qquad\qquad\quad)\ xs)$
21. $\{\mathsf{this}\, j \curlywedge r \in \mathsf{remberP}\, x\, xs\, [\,]\}$
22. $\{\exists\, h.\ \mathsf{this}\, h \curlywedge (r \in \mathsf{remberP}\, x\, xs\, [\,] \curlyvee (x \in xs \curlywedge r = \mathsf{rember}\, x\, xs\, [\,]))\}$
23. $\{\exists\, h.\ \mathsf{this}\, h \curlywedge r = \mathsf{rember}\, x\, xs\, [\,]\}$

Figure 2.7: Proof outline for **rember-up-to-last** $x\ xs$.

of $zs \in \mathsf{remberP}\ x\ ys'\ [\,]$ available in line 10, after unfolding the definition of $\mathsf{remberP}$ once, and using the following lemma about $\mathsf{remberP}$:

$$zs \in \mathsf{remberP}\, x\, ys\, acc \to y{::}zs \in \mathsf{remberP}\ x\, ys\, (y{::}acc)$$

Line 19 describes what holds in the else branch of the main conditional in the loop, and line 21 establishes that the loop invariant holds at the end of the loop. Line 21 is a common weakening of lines 15, 17 and 19. Line 22 includes a disjunction, showing that the line can be reached by a normal termination of the loop (line 21), or by a jump to exit. Line 23 is obtained out of line 22 by applying $\mathsf{rmb\_rmbP}$, and establishes the specified postcondition of **rember-up-to-last**.

2.6.2 Ping-Pong cooperation

In Section 2.2 we presented **inc3**, which used a closure to capture a continuation and execute it twice. Here, we generalize the idea to n calls to **abort**, thus iterating n times the captured continuation. The result is an interleaving between the captured continuation and the finalization code in the closure, which creates a cooperation pattern resembling that of coroutines (Thielecke, 1997)—albeit one without the full power of coroutine *fork* or *yield* operations, which further require storing continuations into the heap (Springer & Friedman, 1989; Reppy, 1999). Unlike **inc3**, where both the captured continuation and the finalization code executed the same code, we will have different computations for the captured continuation, ping, and for the finalization code in the closure, pong, and use the pre- and postconditions to show that these are interleaved in the evaluation of **ping**–**pong**. Since our motivation is to verify the cooperation pattern, rather than ping and pong *per se*, we give a trivial implementation for the latter functions: $\mathsf{ping} \triangleq \mathsf{incr}\ x$ and $\mathsf{pong} \triangleq \mathsf{incr}\ y$, where:

$$\begin{aligned}&\mathsf{incr}\ z : [v\ h].\ \mathsf{SK}^*\ \{z \mapsto v * \mathsf{this}\, h\}\ (\,)\ \{z \mapsto v+1 * \mathsf{this}\, h\}\\ &\quad \triangleq \mathbf{do}(v \leftarrow\ !x;\ x\ :=\ v+1\,)\end{aligned}$$

Figure 2.8 presents **ping**–**pong**, whose structure is close to that of **inc3**. In line 2, **callcc** captures the continuation corresponding to the rest of the program $\lambda c.\ \mathsf{ping}; \mathbf{cmd}\ c$ and binds it to the continuation object k. The body of **callcc** returns a function f defined recursively on n. If n is non-zero (line 5), f returns a closure which aborts to the captured continuation with the finalization code consisting of pong followed by the recursive call to f. In the zero case, f returns the computation pong. The result of this recursive function is bound to c in the sequel. When $n > 0$, c will be bound to a closure with n calls to **abort** nested in the finalization code:

$$[\mathbf{abort}\ \underbrace{k\ (\mathsf{pong}; \mathbf{ret}\,[\mathbf{abort}\ k\ (\cdots\ \mathbf{abort}\ k}_{n \text{ calls to } \mathbf{abort}\, k}\,(\mathbf{ret}\,[\mathsf{pong}]))])]$$

1. **ping–pong** $(n : \mathsf{nat})$:
   $[v\ w\ h].\ \mathsf{SK}^*\ \{x \mapsto v * y \mapsto w * \mathsf{this}\, h\}()$
   $\{x \mapsto v + n + 1 * y \mapsto w + n + 1 * \mathsf{this}\, h\} \triangleq$
2. **do** $(c \leftarrow$ **callcc**$_j$
   $k\ :\ [v\ w\ h]\langle p\rangle.$
   $\mathsf{Kont}^*\ \{j \in x \mapsto v * y \mapsto w * \mathsf{this}\, h\}$
   $\{\mathsf{if}\ \ n \neq 0\ \ \mathsf{then}\ \ x \mapsto v + p + 1 * y \mapsto w + p * \mathsf{this}\, h\ \ \mathsf{else}\ \ \bot\}$
   $\Sigma\bullet\mathsf{SK}\ ()$
   $\{r.\ x \mapsto v + p + 1 * y \mapsto w + p + 1 * \mathsf{this}\, h$
   $\curlywedge\ \mathsf{spec}\ r \sqsubseteq (x \mapsto v + p + 2 * y \mapsto w + p + 1 * \mathsf{this}\, h,$
   $\mathsf{if}\ \ p + 1 = n\ \ \mathsf{then}$
   $x \mapsto v + n + 1 * y \mapsto w + n + 1 * \mathsf{this}\, h$
   $\mathsf{else}\ \ \bot)^*\}.$
3. **fix** $(\lambda\ f\ :\ \mathsf{pingpongT}.\ \lambda\ z : \mathsf{nat}.$
4. if z is $\mathsf{Suc}\, z'$ then
5. **ret** $[\mathbf{abort}\ k\, (\mathsf{pong}; (f\ z'))]$
6. else **ret** $[\mathsf{pong}])\ n;$
7. ping;
8. **cmd** $c).$

where

$\mathsf{pingpongT} \triangleq \Pi z : \mathsf{nat}.$
$[v\ w\ p\ h].\ \mathsf{SK}^*\ \{x \mapsto v + p\ * y \mapsto w + p * \mathsf{this}\, h$
$\curlywedge\ j \in x \mapsto v * y \mapsto w * \mathsf{this}\, h\ \curlywedge\ z \leq n\ \curlywedge\ p + z = n\}$
$\Sigma\bullet\mathsf{SK}\ ()$
$\{r.\ x \mapsto v + p\ * y \mapsto v + p * \mathsf{this}\, h$
$\curlywedge\ \mathsf{spec}\ r \sqsubseteq (x \mapsto v + p + 1 * y \mapsto v + p * \mathsf{this}\, h,$
$\mathsf{if}\ \ p = n\ \ \mathsf{then}$
$x \mapsto v + n + 1 * y \mapsto w + n + 1 * \mathsf{this}\, h$
$\mathsf{else}\ \ \bot)^*\}$

Figure 2.8: **ping–pong** cooperation.

After **callcc**, ping is evaluated to increment x, and then c is evaluated. If $n = 0$, and thus c is bound to [pong], the value at y is incremented an the function terminates. If $n > 0$, the *outermost* **abort** in the closure above is executed, thus running pong as the finalization code and passing the rest of the nested computations in the closure, bound to c, to the captured continuation. This results in a backward jump to line 7. The loop continues until the closure is consumed in full and the function terminates. As a result ping and pong are interleaved $n + 1$ times.

This behaviour is reflected in the type of **ping–pong**: when the function is executed in a heap containing at least the pointers x and y storing some natural number, the result after $n+1$ executions of ping and pong is a heap with the same shape, where the values of x and y are both incremented $n + 1$ times.

The type invariant pingpongT describes the specification of the loop that defines the closure f described above: on each iteration of the recursive call, we insert calls to pong deeper into the closure, which will execute after the corresponding ping. Then, we make explicit that on each recursive call, the values stored in the heap have been incremented appropriately. We introduce a (box) logical variable p to account for this fact: when the recursive call to f occurs, p calls to ping and p calls to pong have occurred. The recursive call produces a closure whose spec we define using $\sqsubseteq$. The closure should be run in a heap after the $(p+1)$-execution of ping. As for the postcondition, If $p \neq n$, then the closure's head is a call to **abort**, line 5, and the postcondition is $\bot$. If $p = n$, i.e. this is the last iteration of f, then the closure corresponds to the one in line 6, entailing that this is the last execution of the loop, if $n > 0$ or that there was no loop at all otherwise. Hence, the final heap of the closure results in the assertion:

$$x \mapsto v + n + 1 * y \mapsto w + n + 1 * \mathsf{this}\, h \tag{2.3}$$

Unlike the case of **inc3**, the continuation object k here is aborted to more than once. As a result, the precondition in Kont^* has to accommodate for the changes in the state in each of the different jumping points. This is solved by using a (diamond) logical variable p, which allows us to discriminate the changes of each particular jumping point with regard to the captured heap j. The precondition in k states that the calls to **abort** occur when $n > 0$. Then, the heap at each jumping point should reflect the effect of $p+1$ executions of ping and p executions of **pong**. The postcondition states that the finalization code performs $p + 1$ execution of pong, and that it returns a closure which, again, is meant to run after the next ping. The postcondition in the closure is similar to the one in pingpongT, albeit the current iteration being $p + 1$ rather than p, as one ping has already executed.

We present a proof outline for **ping–pong** in Figure 2.9, the full proof can be consulted in our HTTcc source files (Delbianco & Nanevski, 2017). For clarity sake, we introduce the following notation to abbreviate the asserted shape of the

1. $\{\exists\, h.\ x \mapsto v * y \mapsto w + n * \mathsf{this}\, h\}$
2. **do** $(\mathbf{callcc}_j\, k\,.$
3. $\quad \{\exists\, h.\ \mathsf{shapeP}[v\ w\ h]\, 0\, 0\ \curlywedge\ j \in \mathsf{shapeP}[v\ w\ h]\, 0\, 0\}$
4. $\quad$ **fix** $(\lambda\ (f : \mathsf{pingpongT})\ (z : \mathsf{nat}).$
5. $\quad \{\exists\, h\, p.\ \mathsf{shapeP}[v\ w\ h]\, p\, p\ \curlywedge\ j \in \mathsf{shapeP}[v\ w\ h]\, 0\, 0\ \curlywedge\ z \le n\ \curlywedge\ p + z = n\}$
6. $\quad\quad$ if z is $\mathsf{Suc}\, z'$ then
7. $\quad\quad\quad \{\exists\, h\, p.\ \mathsf{shapeP}[v\ w\ h]\, p\, p\ \curlywedge\ j \in \mathsf{shapeP}[v\ w\ h]\, 0\, 0$
$\quad\quad\quad\quad\quad \curlywedge\ z' + 1 \le n\ \curlywedge\ p + z' + 1 = n\}$
8. $\quad\quad\quad$ **ret** $[\mathbf{abort}\ k\, (\mathsf{pong}; (f\ z'))]$
9. $\quad\quad\quad \{\exists\, h\, p.\ \mathsf{shapeP}[v\ w\ h]\, p\, p\ \curlywedge\ j \in \mathsf{shapeP}[v\ w\ h]\, 0\, 0\ \curlywedge\ z' + 1 \le n$
$\quad\quad\quad\ \curlywedge\ p + z' + 1 = n\ \curlywedge\ \mathsf{spec}\ r \sqsubseteq (\mathsf{shapeP}[v\ w\ h]\, (p+1)\, p,\ \bot)^*\}$
10. $\quad\quad$ else **ret** $[\mathsf{pong}])\ n);$
11. $\quad\quad \{\exists\, h\, p.\, \mathsf{shapeP}[v\ w\ h]\, p\, p\ \curlywedge\ j \in \mathsf{shapeP}[v\ w\ h]\, 0\, 0\ \curlywedge\ p = n$
$\quad\quad\ \curlywedge\ \mathsf{spec}\ r \sqsubseteq (\mathsf{shapeP}[v\ w\ h]\, (n+1)\, n, \mathsf{shapeP}[v\ w\ h]\, (n+1)\, (n+1))^*\}$
12. $\quad \{\exists\, h.\, \mathsf{shapeP}[v\ w\ h]\, 0\, 0\ \curlywedge\ j \in \mathsf{shapeP}[v\ w\ h]\, 0\, 0$
$\quad\ \curlywedge\ \mathsf{spec}\ r \sqsubseteq (\mathsf{shapeP}[v\ w\ h]\, 1\, 0,\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{shapeP}[v\ w\ h]\, 1\, 1\ \mathsf{else}\ \bot)^*\}$
13. $\{\exists\, h\, p.\, \mathsf{shapeP}[v\ w\ h]\, 0\, 0$
$\curlywedge\ \mathsf{spec}\ r \sqsubseteq\ (\mathsf{shapeP}[v\ w\ h]\, 1\, 0,\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{shapeP}[v\ w\ h]\, 1\, 1\ \mathsf{else}\ \bot)^*$
$\curlyvee\ \mathsf{if}\ n \neq 0\ \mathsf{then}$
$\quad \mathsf{shapeP}[v\ w\ h]\, (p+1)\, (p+1)$
$\quad\ \curlywedge\ \mathsf{spec}\ c \sqsubseteq (\mathsf{shapeP}[v\ w\ h]\, (p+2)\, (p+1),$
$\quad\quad\quad\quad \mathsf{if}\ p + 1 = n\ \mathsf{then}\ \mathsf{shapeP}[v\ w\ h]\, (n+1)\, (n+1)\ \mathsf{else}\ \bot)^*$
$\quad \mathsf{else}\ \bot\}$
14. $\quad$ ping;
15. $\quad \{\exists\, h\, p.\, \mathsf{shapeP}[v\ w\ h]\, 1\, 0$
$\quad \curlywedge\ \mathsf{spec}\ r \sqsubseteq\ (\mathsf{shapeP}[v\ w\ h]\, 1\, 0,\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{shapeP}[v\ w\ h]\, 1\, 1\ \mathsf{else}\ \bot)^*$
$\quad \curlyvee\ \mathsf{if}\ n \neq 0\ \mathsf{then}$
$\quad\quad \mathsf{shapeP}[v\ w\ h]\, (p+1)\, (p+1)$
$\quad\quad\ \curlywedge\ \mathsf{spec}\ c \sqsubseteq (\mathsf{shapeP}[v\ w\ h]\, (p+2)\, (p+1),$
$\quad\quad\quad\quad\quad \mathsf{if}\ p + 1 = n\ \mathsf{then}\ \mathsf{shapeP}[v\ w\ h]\, (n+1)\, (n+1)\ \mathsf{else}\ \bot)^*$
$\quad\quad \mathsf{else}\ \bot\}$
16. $\quad$ **cmd** $c).$
17. $\{\exists\, h.\ x \mapsto v + n + 1 * y \mapsto w + n + 1 * \mathsf{this}\, h\}$

Figure 2.9: A proof outline for **ping**–**pong**.

current heap:

$$\mathsf{shapeP}\,[v\ w\ h]\,n\,m \triangleq x \mapsto v + n * y \mapsto w + m * \mathsf{this}\,h$$

Notice we explicitly mark the logical variables as such using the $[\_]$ notation for logical variables contexts, although this has no *semantic* meaning: the intended effect is to remember that those variables will always be instantiated to the (same) logical values throughout the proof outline. Using this notation, the heap assertion in the precondition of **ping**–**pong** in Figure 2.8 would be $\mathsf{shapeP}[v\ w\ h]\,0\,0$, whereas the one in Equation 2.3 above is rendered as $\mathsf{shapeP}\,[v\ w\ h]\,(n+1)\,(n+1)$).

2.7 Discussion and related work

Reasoning with non-algebraic callcc To understand the difference in specification and reasoning between algebraic and non-algebraic control operators, we have repeated our formal development for a non-algebraic set of operators, which we also make available online (Delbianco & Nanevski, 2017). As a basis, we gave control operators a more familiar type for *non-parameterized* continuation monads (Wadler, 1994):

$$\begin{aligned}&\mathsf{callcc} : ((A \to \mathsf{SK}\,B) \to \mathsf{SK}\,A) \to \mathsf{SK}\,A\\ &\mathsf{throw} : (A \to \mathsf{SK}\,B) \to A \to \mathsf{SK}\,B\end{aligned}$$

Continuation objects are ordinary side-effectful functions of type $A \to \mathsf{SK}\,B$, which do not make provisions for finalization code, and are thus not algebraic (Jaskelioff, 2009). The same remark applies to the type given to the $\mathcal{C}$-operator in (Thielecke, 2009), which is a different, but closely related control operator (Sabry & Felleisen, 1993; Felleisen *et al.* , 1987).

We managed to soundly parameterize the monad with Hoare-style assertions using the following typing rules.

$$\frac{\begin{array}{c}j,\ f : \Pi\, x{:}A.\ \mathsf{SK}\ B\ (S\,x\,j,\ \lambda r\,i\,m.\ \mathsf{False}) \vdash e : \mathsf{SK}\ A\ (P\,j, Q\,j)\\ \forall j.\,(P\,j, Q\,j) \sqsubseteq (\lambda i.j = i \wedge R\,i, S)\end{array}}{\vdash\ \mathbf{callcc}_j\ f.\ e : \mathsf{SK}\ A\ (R,\ S)}$$

$$\frac{\vdash f : \Pi\, x{:}A.\ \mathsf{SK}\ B\ (R,\ \lambda r i\,m.\ \mathsf{False}) \qquad \vdash e : A}{\vdash\ \mathbf{throw}\ f\,e : \mathsf{SK}\ B\ (R,\ \lambda r\,i\,m.\,\mathsf{False})}$$

The intuition for the **callcc** rule is that the user must provide the ending specification pair (R, S). The (binary) postcondition S for the whole command is used as a precondition for the continuation object f, after S is first instantiated with the value x that is passed to f, and the captured heap j. The rule has a side condition requiring that the specification $(P\,j, Q\,j)$ inferred for the body e, can

be weakened into the desired (R, S), under the knowledge that the captured heap j equals the initial heap i.

The requirement that the specification (R, S) has to be provided by hand, practically differentiates the algebraic and non-algebraic operators. In the non-algebraic **callcc**, the specification is monolithic, and the postcondition S is usually a disjunction whose cases specify both the jumping and the non-jumping behaviour of the code. The algebraic **callcc** separates the two cases; the jumping is manually specified in the type of f, but the non-jumping specification can often be inferred by the typing rules from the structure of e, say, if e is straight line code, or by using the invariants provided with the loops in e, otherwise. Our examples **rember-up-to-last** and **ping–pong** illustrate the point, whereby e's specification directly corresponds to the supplied loop invariants. In the non-algebraic case, specifying these two examples incurs an overhead that the same annotation has to be provided twice; once as the loop invariant, and again as part of the specification of **callcc**.

Small vs. large footprints The need for large footprints and explicit naming of residual heaps arises in HTTcc due to the control operators. The HTTcc typing rule for ABORT, requires first discharging a precondition that the heap i at the point of the jump is related by the initial condition R to the heap j at the point of continuation capture. R is an ordinary predicate on heaps, rather than a Hoare triple. Thus, the usual idea of Separation logic, whereby a Hoare triple leaves the unused parts of a program implicitly unchanged, does not directly apply, and we have to name the unused parts in i and j in order to explicitly state their equality.

We have considered an alternative whereby the denotational semantics of control operators may automatically determine the unused part of the heap, by subtracting out of the current heap the portion described by the assertions. However, for this to work, the assertions would have to be *precise*, i.e., uniquely determine the portion to be subtracted. But then, the precision of each used assertion has to be formally proved. Thus, we opted for slightly increasing the specification burden by introducing explicit $\mathsf{this}\, h$ predicates, as a trade-off for not having to prove precision of assertions.

Despite the slight overhead of large footprints, we have not found them too problematic in practice. The naming of the residual heaps is intuitive and can be done systematically. Moreover, the logical variables used in the naming are *local* to the Hoare triple. This makes a big difference from ordinary first-order Hoare or Separation logic, where the global nature of logical variables makes such a naming scheme—and correspondingly, the use of large footprints—a complete non-starter. But mostly, it is the presence of separating conjunction $*$, which makes HTTcc capable of reasoning about heap disjointness with the same ease inherent in Separation logic, irrespective of whether the annotations describe full or partial heaps.

Specification-only variables and implicit constructions Our **callcc** primitive is indexed by a specification-only variable j, which binds the heap at the point of continuation capture. j should be used only in the assertions and proofs, but not in the executable parts of the **callcc** block. Unfortunately, Coq (and consequently HTTcc) does not currently provide any means for enforcing this syntactic distinction. Declaring variables such as j as specification-only *is* natively supported by Coq*, an extension of Coq based on the *Implicit Calculus of Constructions* (Barras & Bernardo, 2008). In the future, we plan to explore embedding HTTcc into Coq*, to make use of this feature.

Higher-order heaps and semantic models for callcc Dreyer et al. (Dreyer *et al.* , 2010) and Støvring and Lassen (Støvring & Lassen, 2007) develop semantics models and methods for equational reasoning in such models, for programs with continuations and mutable store. A specific focus in both works is on higher-order heaps (Krishnaswami, 2011; Yoshida *et al.* , 2008; Schwinghammer *et al.* , 2011); that is, the ability to store computations (and continuations as a special case) into the heap. HTTcc's model is much simpler in this particular respect. While it allows programs that return continuations, by encapsulating them through a closure, it does not allow programs that store side-effectful computations into the heap. The reason is that we defined SK and Kont types *in terms of* heap, rather than *mutually recursively with* heap, as required for stored computations. In the future, we plan to develop a model for HTTcc with stored computations. We plan to build on the model for HTT by Svendsen et al. (Svendsen *et al.* , 2011), which includes higher-order heaps, but no control operators. Our ultimate aim is to implement proper coroutines, following Reppy (Reppy, 1999, cp. 10), and use them to verify concurrency primitives. Moreover, our inference of weakest pre- and strongest postconditions by the rules presented in Figure 2.4 is also in the spirit of *characteristic formulae* (Park, 1981; Aceto & Ingólfsdóttir, 2007; Charguéraud, 2010).

Hoare logics for higher-order control Crolard and Polonowski (Crolard & Polonowski, 2012) have recently developed a Hoare logic for control operators, in which specifications are carried out in types. While in this respect, the approach is similar to HTTcc from the high-level point of view, there is a number of differences. For example, Crolard and Polonowski only consider mutable stack variables with block scope, but no pointers or aliasing. Procedures are not allowed to contain free variables, and type dependencies contain first-order data only, such as natural numbers. In contrast, in HTTcc, we allow the full expressiveness of CIC, including Σ-types over specifications, which, as we illustrated, is required for specifying closures that return captured continuations. Berger (Berger, 2009) presents a first-order Hoare logic for **callcc** in an otherwise purely functional language. One of the main features of the logic is the polarity distinction between the types of

programs that perform jumps ('jumping-to') and the types of labels for jumps ('being-jumped-to'). From the point of view of reasoning, the logic allows nesting Hoare triples inside the assertions. This is necessary for specifying closures with captured continuations, and achieves the same effect as Σ-types over specifications in our dependently-typed setting.

Hoare reasoning through dependent types Related systems that employ Hoare-style specification via types are HTT (Nanevski *et al.* , 2010) and F^* (Swamy *et al.* , 2013). HTT is a direct precursor of HTTcc, but does not include the control operators. It uses an embedding of (small footprint) Separation logic via monads into Coq to formulate annotations and discharge verification conditions. A similar idea of Hoare monads in Coq, without control operators, has also been considered by Swierstra (Swierstra, 2009). F^* specifies computations using a somewhat different monad from the above work. Instead of postconditions ranging over the input and output heaps, F^* considers predicate transformers ranging over *sets* of input and output heaps. F^* does not include a separate form of preconditions to specify safety; thus, its Hoare logic is not fault-avoiding as is HTTcc, or other systems based on Separation logic. F^* relies on Z3 for automatic discharge of verification conditions. In order to facilitate automation, its assertion logic is a first-order fragment supported by Z3. F^* does not consider **callcc**, or other abstractions required by it, such as Σ-types over specifications.

CPS translation of HTTcc CPS translation in the case of dependent types has been studied by Barthe and Uustalu (Barthe & Uustalu, 2002) who show the impossibility of CPS-translating dependent inductive and Σ-types. As HTTcc essentially relies on Σ-types to encode nested Hoare triples – as inhabitants of the Σ•SK type – it seems impossible to present HTTcc as a CPS translation into a **callcc**-free fragment of HTTcc.

Roll-backing vs. persistent state Unlike the case of the *algebraic* **callcc** in (Jaskelioff, 2009), where the state is rolled back when the continuation is restored by abort, we chose to make the changes in the state persistent, as we believe it is a more reasonable approach to an imperative-style dynamic state. A direct consequence of this decision is the need for providing the R_f annotation, although we believe it is a fair trade-off.

2.8 Summary

This chapter presented a higher-order type theory for verification of programs with `call/cc` and `abort` control operators. The implemented theory, that we have named HTTcc, supports mutable state in the style of Separation logic, and, to the best of our knowledge, is the first Hoare logic or type theory to support

the combination of higher-order functions, mutable state and control operators. The support for mutable state comes with a twist, however. While our assertion logic embeds separating conjunction $*$, we used large footprint specification style, which we found necessary to relate heaps captured with the continuation to heaps at the point of a jump. We use *algebraic* control operators, initially introduced by Jaskelioff (Jaskelioff, 2009), which we here adapt to *non-rollbackable* state. We argue that in practice, the algebraic operators require less manual program annotations, than the non-algebraic variants. We have implemented HTTcc as a shallow embedding in Coq, and verified a number of characteristic example programs that use **callcc**. The implementation of the logic and the case studies is available online (Delbianco & Nanevski, 2017).

Part II

Concurrency

Preamble

Arguments about correctness of a concurrent data structure are typically carried out by using the notion of *linearizability* (Herlihy & Wing, 1990) and specifying the linearization points of the data structure's procedures. Such arguments are often cumbersome as the linearization points' position in time can be *dynamic* (depend on the interference, run-time values and events from the past, or even future), *non-local* (appear in procedures other than the one considered), and whose position in the execution trace may only be determined after the considered procedure has already terminated.

In Chapter 4, we propose a new method, based on an existing separation-style logic—FCSL, described in Chapter 3—, for reasoning about concurrent objects with such linearization points. We embrace the dynamic nature of linearization points, and encode it as part of the data structure's *auxiliary state*, so that it can be dynamically modified in place by auxiliary code, as needed when some appropriate run-time event occurs. We name the idea *Linking-in-time*, because it reduces temporal reasoning to spatial reasoning. For example, modifying a temporal position of a linearization point can be modeled similarly to a pointer update in separation logic. Furthermore, the auxiliary state provides a convenient way to concisely express the properties essential for reasoning about clients of such concurrent objects. We illustrate the method by verifying (mechanically in Coq) an intricate optimal snapshot algorithm due to Jayanti (Jayanti, 2005), together with some concurrent clients of the snapshot data structure.

The contents of this part of the thesis are based in the following papers, coauthored by the author:

- Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski and Anindya Banerjee. Concurrent Data Structures Linked in Time. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain*. LIPIcs 74, pages 8:1–8:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

- Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey and Germán Andrés Delbianco: Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014, Proceedings*, volume 8410 of LNCS, pages 290—310. Springer, 2014.

Chapter 4 extends the original paper (Delbianco *et al.* , 2017a) by incorporating the appendices from the extended version of the paper together with an updated and more thorough discussion section. The Coq developments presented in this chapter, together with full FCSL sources were unanismously validated as an

artifact by ECOOP's AEC, which is available online on the DARTS artifact series (Delbianco *et al.* , 2017b).

Chapter 3 summarizes the development of FCSL presented in the original paper (Nanevski *et al.* , 2014a) and incorporates the formal details about the logic from an online extended version (Nanevski *et al.* , 2014b).

3

FCSL: A Fine-Grained Concurrent Separation Logic

This chapter presents a novel model of concurrent computations with shared memory and provide a simple, yet powerful, logical framework for uniform Hoare-style reasoning about partial correctness of both coarse- and fine-grained concurrent programs. We call this framework FCSL, which stands for *Fine-grained Concurrent Separation Logic*. The key idea behind FCSL is to specify arbitrary resource protocols as communicating *state transition systems* (STS) that describe valid states of a resource and the transitions the resource is allowed to make, including—*but not limited to!*—transfer of heap ownership. By reasoning in terms of communicating STS, the logic makes it easy to crystallize behavioural invariants of a resource.

Thus, FCSL takes the classical rules from Concurrent Separation Logic (CSL) (O'Hearn, 2007; Brookes, 2007; Brookes & O'Hearn, 2016), such as scoped resource allocation, and generalizes them to cope with fine-grained resource management. As a result, the user of the logic can give rich specifications to concurrent objects, with complex and expressive environment interference interactions in the style of Rely-Guarantee (Jones, 1983; Vafeiadis & Parkinson, 2007), albeit in a concise, modular way. The ending result is a flexible framework which regains the compositionality of CSL-style resource management.

Following the methodology of HTT and HTTcc, FCSL has been formalized as a shallow embedding in Coq (FCSL, 2017) and has been used to prove the correctness of different concurrent data structures, with varying complexity and correctness criteria, in the same unified setting: FCSL has been used to verify both coarse-grained lock-based structures (Nanevski *et al.* , 2014a) and fine-grained concurrent objects such as snapshots (Delbianco *et al.* , 2017a; Sergey *et al.* , 2015b), stacks (Sergey *et al.* , 2015b) and other advanced objects (Sergey *et al.* , 2016), and to prove the correctness of concurrent graph algorithm implementations (Sergey *et al.* , 2015a).

$$
\begin{array}{lll}
\texttt{incr}\ n & : & \{\sigma_{\mathsf{s}} = \emptyset \wedge \mu_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha_{\mathsf{s}} = 0\}\ ()\\
 & & \{\sigma'_{\mathsf{s}} = \emptyset \wedge \mu'_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha'_{\mathsf{s}} = n\}@\ \mathcal{L}_{\mathrm{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}\\
 & \widehat{=} & \textbf{do}\ \ (\ \texttt{lock};\\
 & & \quad r \leftarrow \textbf{inject}\ (\mathsf{read}\ x);\\
 & & \quad \textbf{inject}\ (\mathsf{write}\ x\ (r+n));\\
 & & \quad \texttt{unlock})
\end{array}
$$

Figure 3.1: Coarse-grained concurrent incrementer in FCSL, `incr`

3.1 Overview

Specifications FCSL specifications take the form of Hoare-type ascription $e : \{P\}\, A\, \{Q\}@\ \mathcal{C}$ expressing that the A-value returning computation e has a precondition P and a postcondition Q over a state space and under transitions defined by the concurrent resource $\mathcal{C}$, which in FCSL takes the role of a resource context from CSL. In a similar way we have done for HTTcc type ascription in Chapter 2, we might need logical variables to specify our triples. In those cases, we make the context of logical variables explicit in the type:

$$e : [\Gamma].\ \{P\}\, A\, \{Q\}@\ \mathcal{C}$$

Note that, unlike standard Hoare-style logics where logical variables are global to a specification or a correctness proof, FCSL logical variables scopes are delimited to a given triple. The reason behind this requirement is to enable the specification of recursive procedures, where a logical variable may be instantiated differently in each recursive call (Kleymann, 1999). Moreover, logical variables can **only** appear in P and Q, *i. e.* they do not appear in program e, nor the type A nor the concurrent resource $\mathcal{C}$.

The state s of a concurrent resource $\mathcal{C}$ in FCSL always consists of three distinct auxiliary variables that we name s_{s}, s_{o} and s_{j}, such that $s = (s_{\mathsf{s}}, s_{\mathsf{j}}, s_{\mathsf{s}})$. These stand for the abstract *self*, *other*, and *joint* projections of the state. However, the user can pick the types of these variables based on the application. It is essential that a_{s} and a_{o} have a common type, which moreover, exhibits the algebraic structure of a *partial commutative monoid* (PCM). The subjective assertions (or *getters*) constrain the value of one state component, assuming others to be existentially quantified over.

Coarse-Grained Incrementer We illustrate the specification language by visiting the implementation of a coarse-grained concurrent incrementer (Owicki & Gries, 1976; Ley-Wild & Nanevski, 2013) resource $\mathcal{L}_{\mathrm{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}$, whose main method `incr` is presented in Figure 3.1. The resource $\mathcal{L}_{\mathrm{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}$ consists of by a lock lk which protects a shared heaplet h. A CSL-style resource invariant (O'Hearn,

2007; Brookes, 2007; Brookes & O'Hearn, 2016) $\mathcal{I}$ is associated to the protected heap h, $\mathcal{I}\ n\ h \mathrel{\widehat{=}} h = x \mapsto a$. The latter is used to define the *state invariant* of $\mathcal{L}_{\text{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}$, $\mathsf{Coh}_{\mathcal{L}_{\text{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}}$, to be the states that satisfies the following assertion:

$$\rho_{\mathsf{j}} = lk \mapsto b \cup h \wedge \ \begin{array}{l} \mathit{if}\ b\ \mathit{then}\ h = \emptyset\ \wedge \mu_{\mathsf{s}} \bullet \mu_{\mathsf{o}} = \mathsf{Own} \\ \mathit{else}\ \mathcal{I}\ (\alpha_{\mathsf{s}} + \alpha_{\mathsf{o}})\ h \wedge \mu_{\mathsf{s}} \bullet \mu_{\mathsf{o}} = \cancel{\mathsf{Own}} \end{array}$$

We first describe the thread-relative assertions. The getter ρ_{j} exposes the fact that the joint heap owned by the resource contains a Boolean pointer lk encoding a lock that protects the heap h. The getters μ_{s} and μ_{o} assert the *self* (and respectively, *other*) ownership of the lock lk, taking values from the mutex $(\{\mathsf{Own}, \cancel{\mathsf{Own}}\}, \bullet, \cancel{\mathsf{Own}})$, where $\cancel{\mathsf{Own}} \bullet m = m = m \bullet \cancel{\mathsf{Own}}$ and $\mathsf{Own} \bullet \mathsf{Own}$ is undefined. Finally, the α_{s} and α_{o} getters project, respectively the *self* and *other* cumulative contributions to the incrementer.

Then, the $\mathsf{Coh}_{\mathcal{L}_{\text{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}}$ asserts that if the lock lk is taken ($b = \mathsf{true}$) then the heap $x \mapsto \_$ is given away by the resource, and otherwise it satisfies the resource invariant $\mathcal{I}$. In either case, the thread-relative getters' values μ_{s}, μ_{o}, α_{s}, α_{o} are consistent with the resource's joint view of lk and h through ρ_{j}. Indeed, notice how μ_{s}, μ_{o} and $\alpha_{\mathsf{s}}, \alpha_{\mathsf{o}}$ are first $\bullet$-joined and then related to b and h; the former implicitly by the conditional, the latter explicitly, by the resource invariant $\mathcal{I}$, which is now parametrized by $\alpha_{\mathsf{o}} + \alpha_{\mathsf{s}}$, asserting then that $x \mapsto \alpha_{\mathsf{s}} + \alpha_{\mathsf{o}}$.

We consider now the specification of `incr` in Figure 3.1: the function acquires the lock to ensure exclusive access to x, increments x by n, and releases the lock. The specification states that `incr` runs in an empty private heap—$\sigma_{\mathsf{s}} = \emptyset$ —the lock is not owned—$\mu_{\mathsf{s}} = \cancel{\mathsf{Own}}$—by the calling thread initially, and will not be owned in the end—$\mu'_{\mathsf{s}} = \cancel{\mathsf{Own}}$. The addition of calling thread to x increases from 0 to n, as witness by the *getters* $\alpha_{\mathsf{s}} = 0$ in the precondition and $\alpha'_{\mathsf{s}} = n$ in the postcondition. We use a VDM-like notation (Bjørner & Jones, 1978), using primed getters in the postcondition, in order to distinguish pre-values and post-values in the post-conditions, as both getters are allowed to appear in post-conditions[1].

Programming Language FCSL uses monads to separate the purely functional fragment of Coq to the imperative fragment of the logic, denoted by Hoare-types $\{P\}\ B\ \{Q\}@\ \mathcal{C}$. In the example in Figure 3.1, we use $x \leftarrow e_1; e_2$ for monadic bind (i.e. a sequential composition), which first executes e_1, then binds the return result to x before executing e_2. We abbreviate with $e_1; e_2$ when $x \notin \mathsf{FV}\,(e_2)$. The expression $\mathbf{do}\, e : \{P\}\ B\ \{Q\}@\ \mathcal{C}$ (potentially with logical variables) whenever we want to explicitly ascribe the specification (P, Q) to e, rather than use the tightest specification that the system infers for e. Such ascription will entail a proof obligation that (P, Q) is valid for e, as we explain in Section 3.2. The expression **inject** e allows us to *lift* operations from a smaller resource to a bigger

[1]This is a notational convention which is implemented in our Coq Files using a dedicated logical variable in order to connect pre and post values.

resource. `lock` and `unlock` are two operations from the $\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}$*resource* which acquire and release the lock lk, specified as follows:

$$\begin{aligned}
\texttt{lock} \;:\; & \{\sigma_\mathsf{s} = \emptyset \wedge \mu_\mathsf{s} = \cancel{\mathsf{Own}} \wedge \alpha_\mathsf{s} = 0\}() \\
& \{\exists\, \mathsf{a_o}.\ \sigma'_\mathsf{s} = x \mapsto \mathsf{a_o} \wedge \mu'_\mathsf{s} = \mathsf{Own} \wedge \alpha'_\mathsf{s} = 0 \wedge \alpha'_\mathsf{o} = \mathsf{a_o}\}@\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}} \\
\texttt{unlock} \;:\; & [\mathsf{a_s}, \mathsf{a_o}].\ \{\sigma_\mathsf{s} = x \mapsto \mathsf{a_s} + \mathsf{a_o} \wedge \mu_\mathsf{s} = \mathsf{Own} \wedge \alpha_\mathsf{s} = \mathsf{a_s} \wedge \alpha_\mathsf{o} = \mathsf{a_o}\}() \\
& \{\sigma'_\mathsf{s} = \emptyset \wedge \mu'_\mathsf{s} = \cancel{\mathsf{Own}} \wedge \alpha'_\mathsf{s} = \mathsf{a_s}\}@\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}
\end{aligned}$$

`lock` assumes that lock is not taken, and that the *self thread* so far has added 0 to x. Thus, Note that acquire does not have to be atomic: as implemented, it just spins on lk, and after acquisition, the heaplet $x \mapsto \mathsf{a_o}$ is transferred into the private heap of *self*. $\mathsf{a_o}$ must be existentially quantified, because other threads may add to x while `lock` is spinning.

`unlock` assumes that lock is taken by *self*, and that prior to taking lock, *self* and *other* have added 0 and $\mathsf{a_o}$ to x, respectively. After acquiring x, *self* has mutated it, so that its contents is $\mathsf{a_s} + \mathsf{a_o}$. After releasing, x is moved back to the resource's control. The post-condition does not mention x nor the getter ρ_j for the *joint* heap resource, as the contents asserted by the latter are not stable. Indeed, other thread may acquire the lock and change x after `unlock` terminates. However, that thread cannot change the *self* view of x, α_s, which is now set to $\mathsf{a_s}$.

Parallel Composition The ability to choose user-defined PCM structures (inherited from SCSL (Ley-Wild & Nanevski, 2013)), makes it possible to define the inference rule for parallel composition in a generic way:

$$\frac{\Gamma \vdash e_1 : \{P_1\}\, A\, \{Q_1\}@\,\mathcal{C} \qquad \Gamma \vdash e_2 : \{P_2\}\, B\, \{Q_2\}@\,\mathcal{C}}{\Gamma \vdash e_1 \parallel e_2 : \{P_1 \circledast P_2\}\, (A \times B)\, \{r.\ [r.1/r]Q_1 \circledast [r.2/r]Q_2\}@\,\mathcal{C}}\ \textsc{Par}$$

Here, $\circledast$ is defined over state predicates P_1 and P_2 as follows:

Definition 3.1 (Subjective Star)**.** Given assertions P_1, and P_2, and a *subjective* state s, we define FCSL subjective separating conjunction $\circledast$ as follows:

$$\begin{aligned}
(s_\mathsf{s}, s_\mathsf{j}, s_\mathsf{o}) \models P_1 \circledast P_2 \iff\ & \exists\, x_1\ x_2.\ \ s_\mathsf{s} = x_1 \bullet x_2 \wedge\ (x_1, s_\mathsf{j}, x_2 \bullet s_\mathsf{o}) \models P_1 \\
& \wedge\ (x_2, s_\mathsf{j}, x_1 \bullet s_\mathsf{o}) \models P_2
\end{aligned}$$

The inference rule, and the definition of $\circledast$, formalize the intuition that when a parent thread forks e_1 and e_2, then e_1 is part of the environment for e_2 and vice-versa. This is so because the *self* component a_s of the parent thread is split into x_1 and x_2; x_1 and x_2 become the *self* parts of e_1, and e_2 respectively, but x_2 is also added to the *other* component a_o of e_1, and dually, x_1 is added to the *other* component of e_2. For example, in the case of an assertion using the heap

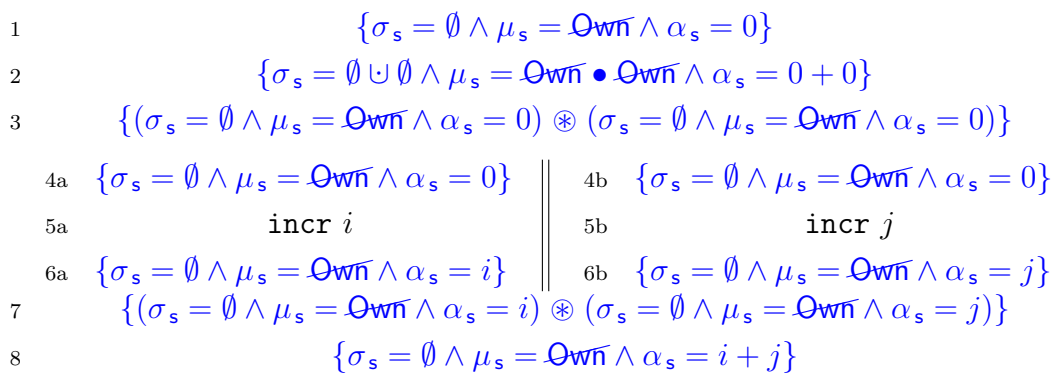

Figure 3.2: FCSL proof outline for `two` $\widehat{=}$ **do** (`incr` $i \parallel$ `incr` j).

getters σ from the specification of `incr` above, we have:

$$\begin{array}{l} \sigma_s = x \mapsto a \cup y \mapsto b \land \sigma_o = z \mapsto c \\ \Longrightarrow \\ (\sigma_s = x \mapsto a \land \sigma_o = z \mapsto c \cup y \mapsto b) \circledast (\sigma_s = y \mapsto b \land \sigma_o = z \mapsto c \cup x \mapsto a) \end{array}$$

The implication encodes the idea of a *forking shuffle* from Rely-Guarantee (Jones, 1983; Vafeiadis & Parkinson, 2007), but via states, rather than transitions as in RG. It allows us to use the *same* concurrent resource $\mathcal{C}$ to specify the transitions of both e_1 and e_2 in Par, much like the parallel composition rule of CSL uses the same resource context. Essentially, we rely on the recombination of thread views to select the transitions of $\mathcal{C}$ available to each of e_1 and e_2, instead of providing distinct transitions for e_1 and e_2 as in ParRG. We commonly encounter cases where the *other* views are existentially abstracted. In those cases, where we only have *self* assertions we can derived a simplified bi-implication. For instance, using the getters σ_s, μ_s, and α_s from the specification of `incr` above, we derive:

$$\begin{array}{l} \sigma_s = x \mapsto a \cup y \mapsto b \land \mu_s = m_1 \bullet m_2 \land \land \alpha_s = n + m \\ \Longleftrightarrow \\ (\sigma_s = x \mapsto a \land \mu_s = m_1 \land \alpha_s = n) \circledast (\sigma_s = y \mapsto b \land \mu_s = m_2 \land \alpha_s = m) \end{array} \tag{3.1}$$

Many logics that employ auxiliary state, going back to Owicki and Gries (Owicki & Gries, 1976), require that their procedures are annotated for a specific number and forking pattern of interfering threads. When this pattern changes, the programs have to be re-annotated, and proofs redone. But this is not the case in FCSL. In order to illustrate this claim, we now show how the Par rule is used to prove that the parallel combination of incr $i \parallel$ incr j increments x by $i + j$ in Figure 3.2. Notice we omit the resource annotation $@\mathcal{L}_{\text{CSL}\ \{\mathsf{lock}, lk, \mathcal{I}\}}$ from the proof

outline, as it is does not change throughout the proof: this will be a standard practice throughout all the FCSL proof outlines presented in this thesis.

The proof proceeds at follows: on line 2, we use the unit of the PCMs to express the contents of the getters as unions. Then, we use the bi-implication (3.1) on line 3 to move between $\circledast$-separated assertions and $\bullet$-joined views. The proof is compositional in the sense that the same verification of incr is used as a black box in both parallel threads (through lines 4a–6a and, respectively, 4b–6b), with each sub-proofs merely instantiating the parameter n with i and j respectively. On line 7, we recover the $\circledast$-separated assertion as a post-condition of the Par rule. Finally, we use the bi-implication (3.1) in the opposite direction to gather the contributions on line 8.

Framing Being a separation-like logic, FCSL provides a frame rule. The frame rule can be seen then as a special case of the parallel composition rule, obtained when e_2 is taken to be the idle thread:

$$\frac{\Gamma \vdash e : \{P\}\, A\, \{Q\} @\, \mathcal{C} \qquad R \text{ stable } \mathcal{C}}{\Gamma \vdash e : \{P \circledast R\}\, A\, \{Q \circledast R\} @\, \mathcal{C}}\ \textsc{Frame}$$

Since R describes both the pre-state and post-state, it has to be *stable* under $\mathcal{C}$ that is, it determines a subset of $\mathcal{C}$'s states that remains fixed under the interference of *other* threads (see Definition 3.27). We illustrate the use of the Frame rule a small client which composes two incrementer calls sequentially:

$$\begin{aligned} \texttt{twice}\ n \quad : \quad & \{\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = k\}\ () \\ & \{\sigma'_s = \emptyset \wedge \mu'_s = \cancel{\mathsf{Own}} \wedge \alpha'_s = 2 * n\} @\ \mathcal{L}_{\text{CSL}\ \{\mathsf{lock}, lk, \mathcal{I}\}} \\ \widehat{=} \quad & \textbf{do}\ (\texttt{incr}\ n; \texttt{incr}\ n;) \end{aligned}$$

The proof is presented in Figure 3.3. On line 4, we exploit the bi-implication (3.1), to frame the contribution of the first call with $\circledast$. The assertion:

$$(\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = i)$$

1 $\{\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = 0\}$
2 incr n;
3 $\{\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = n\}$
4 $\{(\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = 0) \circledast (\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = n)\}$
5 incr n;
6 $\{(\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = n) \circledast (\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = n)\}$
7 $\{\sigma_s = \emptyset \wedge \mu_s = \cancel{\mathsf{Own}} \wedge \alpha_s = 2 * n\}$

Figure 3.3: FCSL proof outline for twice $n \widehat{=}$ **do** (incr n; incr n).

will be trivially stable as it only asserts *self* values. Then, we apply the FRAME rule to run `incr` n again, in order to get as a result the $\circledast$-separated assertion on line 6. Finally, we apply the bi-implication 3.1 to gather the contributions together, asserting that $\alpha'_{\mathsf{s}} = 2 * n$ in the end. It should be noticed that we do not use VDM-style primed notation in the proof outline to avoid an explosion of *primes* on getters and instead we resort to the standard Hoare-Logic practice of introducing existentially quantified naming new logical variables when we want to *remember* a particular value.

Entanglement and Injection In FCSL fine-grained resources can be combined, allowing to combine "smaller" resources into "bigger" ones, and re-using their methods and specifications modularly. For example, if $\mathcal{P}$ is the FCSL resource for private heaps and $\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$ is the FCSL resource for a lock (with a lock, lock pointer lk and protected heap described by the coarse-grained resource invariant I), we can construct the $\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}$ resource as an entanglement:

$$\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}} = \mathcal{P} \bowtie \mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$$

The entanglement can be iterated, to obtain a STS for *two coarse-grained resources*, and so on. In this way, FCSL fine-grained resources generalize the notion of resource context from the RESOURCE rule in CSL (O'Hearn, 2007), with entanglement modelling the addition of new resources to the context. We will dwell further into the formal details about entanglement and resource combination in Section 3.3.5.

We now explain the **inject** e command which allows us to *lift* commands from a smaller resource $\mathcal{U}$ to an extended resource $\mathcal{U} \bowtie \mathcal{V}$. The command is governed by the INJECT rules, described as follows:

$$\frac{e : \{p\}\, A\, \{q\}@\, U \qquad R \text{ stable } \mathcal{V}}{\textbf{inject } c : \{P * R\}\, A\, \{Q * R\}@\, \mathcal{U} \bowtie \mathcal{V}}\ \textsc{Inject}$$

Reading the rule bottom-up, it says we can ignore $\mathcal{V}$, as $\mathcal{V}$'s transitions and e operate on disjoint resources. $\mathcal{V}$ may change $\mathcal{U}$'s state by communication, but the change is bounded by $\mathcal{U}$'s external transitions. Thus, we are justified in verifying e against $\mathcal{U}$ alone. In this sense, INJECT may be seen as generalizing the rule for resource context weakening of CSL.

Whereas in CSL the separating conjunction * splits the heap, in FCSL the connective $*$ splits the *state* according to the components of $\mathcal{U}$ and $\mathcal{V}$, P and Q describe the part controlled by $\mathcal{U}$, and R describes the part controlled by $\mathcal{V}$. Formally:

Definition 3.2 (Separating Conjunction)**.** Given assertions P_1, and P_2, and a state s, we define the $*$ separating conjunction in FCSL as:

$$s \models P_1 * P_2 \iff \exists\, s_1\ s_2.\ s = s_1 \cup s_2 \wedge\ s_1 \models P_1 \wedge\ s_2 \models P_2$$

where the separated union of FCSL states is given on Definition 3.11

1 $\{\sigma_{\mathsf{s}} = \emptyset \wedge \mu_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha_{\mathsf{s}} = 0\}$
2 `lock`;
3 $\{\exists \mathsf{a}_{\mathsf{o}}.\ \sigma_{\mathsf{s}} = x \mapsto \mathsf{a}_{\mathsf{o}} \wedge \mu_{\mathsf{s}} = \mathsf{Own} \wedge \alpha_{\mathsf{s}} = 0 \wedge \alpha_{\mathsf{o}} = \mathsf{a}_{\mathsf{o}}\}$
4 $r \leftarrow$ **inject** (read x);
5 $\{\exists \mathsf{a}_{\mathsf{o}}.\ \sigma_{\mathsf{s}} = x \mapsto \mathsf{a}_{\mathsf{o}} \wedge r = \mathsf{a}_{\mathsf{o}} \wedge \mu_{\mathsf{s}} = \mathsf{Own} \wedge \alpha_{\mathsf{s}} = 0 \wedge \alpha_{\mathsf{o}} = \mathsf{a}_{\mathsf{o}}\}$
6 **inject** (write x $(r + n)$);
7 $\{\exists \mathsf{a}_{\mathsf{o}}.\ \sigma_{\mathsf{s}} = x \mapsto (n + \mathsf{a}_{\mathsf{o}}) \wedge r = \mathsf{a}_{\mathsf{o}} \wedge \mu_{\mathsf{s}} = \mathsf{Own} \wedge \alpha_{\mathsf{s}} = 0 \wedge \alpha_{\mathsf{o}} = \mathsf{a}_{\mathsf{o}}\}$
8 `unlock`
9 $\{\sigma_{\mathsf{s}} = \emptyset \wedge \mu_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha_{\mathsf{s}} = n\}$

Figure 3.4: A proof outline for `inr` n in FCSL

We will illustrate the use of the INJECT rule, by verifying `incr`. First, we need to specify the atomic commands `write` and `read` from the private state resource $\mathcal{P}$, which will be *lifted* to $\mathcal{L}_{\text{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}$ via **inject**

$$\begin{array}{lll} \texttt{read}\ x & : & [v].\ \{\sigma_{\mathsf{s}} = x \mapsto v\}\ A\ \{\sigma'_{\mathsf{s}} = x \mapsto v \wedge r = v\}@\ \mathcal{P} \\ \texttt{write}\ x & : & \{\sigma_{\mathsf{s}} = x \mapsto \_\}\ A\ \{\sigma'_{\mathsf{s}} = x \mapsto v \wedge r = v\}@\ \mathcal{P} \end{array}$$

Figure 3.4 presents the proof outline for `incr` n in FCSL. INJECT is used twice, to lift `read` and `write` from $\mathcal{P}$ to $CSL_{\mathsf{lock},lk,I}$. These commands manipulate the contents of priv, but retain the framing predicate $\mu_{\mathsf{s}} = \mathsf{Own} \wedge \alpha_{\mathsf{s}} = 0 \wedge \alpha_{\mathsf{o}} = \mathsf{a}_{\mathsf{o}}$. This predicate is stable with regard to $\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$. Intuitively, because *self* owns the lock, other threads cannot acquire x and add to it. Thus, no matter what steps the environment performs, the assertion α_{o} and the framed predicate remain invariant.

3.2 FCSL Rules and Verification Framework

In the previous section we have given an informal presentation of FCSL. In this section we will present the typing rules for the FCSL language together with the properties and lemmas that constitute our verification framework.

Again, it should be noticed that the latter logic is implemented as a shallow embedding in CiC in a similar way to the implementation of HTTcc in Chapter 2. Therefore, we will concentrate just in the effectful rules FCSL which implement effectful commands $e : \{P\}\ A\ \{Q\}@\ \mathcal{C}$, ignoring the pure inherited from the embedding into CiC as it is standard. Moreover, we will distinguish between *computational* inference rules which correspond to commands in FCSL syntax and are thus implemented through language definitions, and *structural* or *logical* rules

which are presented as derived lemmas[2].

3.2.1 FCSL Inference Rules

FCSL's judgements are *hypothetical* under a context Γ that maps CiC *variables* x, v to their *pure* types A, B and *commands* e to their specifications of type $\{P\}\, A\, \{Q\}@\, \mathcal{U}$. We allow each specification to depend on the variables declared to the left. Γ does not bind logical variables, however. As we have mentioned before, we limit the scope of logical variables in FCSL to the Hoare-types in which they appear. We note that the assertions and the annotations in the rules (*e. g.*, Φ in the Hide rule) may freely use the variables in Γ. To reduce clutter, we silently assume the checks that all such specification level-entities are well-typed in Γ. We have already discussed rules Par and Hide in Section 3.1. We describe the rest of the rules in Figure 3.5 in the sequel.

Monadic Combinators FCSL is implemented as a monadic extension of Coq. The monadic *bind* (*i. e.* sequential composition) $x \leftarrow c_1; c_2$ runs c_1 then substitutes its result v_1 for x to run c_2 (we write $c_1; c_2$ when $x \notin \mathsf{FV}(c_2)$). The associated Seq is, then, a standard Hoare logic rule for sequential composition. The Ret rule says that the value returning command **ret** is safe to run in any state satisfying a stable precondition. After **ret** returns, the state is left unchanged, modulo environment interference. The Fix rule implements the usual typing for recursive procedures in Hoare Type Theories (*cf.*. HTTcc's typing rule for Fix in Figure 2.4), requiring that a Hoare-type specification can be established for the procedure body, under a hypothesis that the recursive calls satisfy the same type. This is type invariant is akin to the role of *loop invariants* in standard Hoare/separation logics.

Conseq The rule Conseq uses the judgement $\Gamma \vdash (P_1, Q_1) \sqsubseteq (P_2, Q_2)$, which generalizes the customary side conditions $P_2 \implies P_1$ for strengthening the precondition and $Q_1 \implies Q_2$ for weakening the postcondition. We use this rule whenever we want to explicitly ascribe the specification (P, Q) to e, rather than use the tightest specification that the system infers for e.

In first order Hoare logics, the logical variables have global scope, so the standard pre-strengthening, post-weakening implications over P_1, P_2 and Q_1, Q_2

[2]This presentation deviates from the one in the original paper (Nanevski *et al.* , 2014a). The presentation in the latter was written aimed at an audience familiar with program logics (for concurrency), but not necessarily keen on reading about Hoare-Types or *shallow embeddings in Coq*. In this thesis manuscript, however, we favour the presentation in this section as it has several advantages: (1) it is consistent with the presentation of HTTcc in Chapter 2 and thus emphasizes the common approach to the design and implementation of program logics, (2) it simplifies the presentation of the semantics of the programming language, and (3), there is no need to present trivial *structural* rules, *e. g.* Hyp or App (*cf.* (Nanevski *et al.* , 2014a, Figure 2)), as either they are immediately derived from the underlying metatheory of Coq, or could be implemented easily as Lemmas 3.2.2.

$$\frac{\Gamma \vdash e_1 : \{P\}\, B\, \{R\}@\, \mathcal{C} \quad \Gamma,\, x{:}B \vdash e_2 : \{[x/r]R\}\, A\, \{Q\}@\, \mathcal{C} \quad x \notin \mathsf{FV}(Q)}{\Gamma \vdash x \leftarrow c_1; c_2 : \{P\}\, A\, \{Q\}@\, \mathcal{C}}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash e_1 : \{P_1\}\, A\, \{Q_1\}@\, \mathcal{C} \qquad \Gamma \vdash e_2 : \{P_2\}\, B\, \{Q_2\}@\, \mathcal{C}}{\Gamma \vdash e_1 \parallel e_2 : \{P_1 \circledast P_2\}\, (A \times B)\, \{r.\ [r.1/r]Q_1 \circledast [r.2/r]Q_2\}@\, \mathcal{C}}\ \textsc{Par}$$

$$\frac{\Gamma \vdash v : A \qquad P\ \mathsf{stable}\ \mathcal{C}}{\Gamma \vdash \mathbf{ret}\ v : \{P\}\, A\, \{r.\ P \wedge r = v\}@\, \mathcal{C}}\ \textsc{Ret}$$

$$\frac{\Gamma \vdash e : \{P_1\}\, B\, \{Q_1\}@\, \mathcal{C} \qquad (P_1,\, Q_1) \sqsubseteq (P_2,\, Q_2)}{\Gamma \vdash \mathbf{do}\ e : \{P_2\}\, A\, \{Q_2\}@\, \mathcal{C}}\ \textsc{Conseq}$$

$$\frac{\Gamma \vdash f : (\Pi\, x{:}A.\ \{P\, x\}\, (B\, x)\, \{Q\, x\}@\, \mathcal{C}) \to (\Pi\, x{:}A.\ \{P\, x\}\, (B\, x)\, \{Q\, x\}@\, \mathcal{C})}{\Gamma \vdash \mathbf{fix}\, f : \Pi\, x{:}A.\ \{P\, x\}\, (B\, x)\, \{Q\, x\}@\, \mathcal{C}}\ \textsc{Fix}$$

$$\frac{\Gamma \vdash e : \{P\}\, A\, \{Q\}@\, \mathcal{U} \qquad R\ \mathsf{stable}\ \mathcal{V}}{\Gamma \vdash \mathbf{inject}\ e : \{P * R\}\, A\, \{Q * R\}@\, \mathcal{U} \rtimes \mathcal{V}}\ \textsc{Inject}$$

$$\frac{\Gamma \vdash e : \{\sigma_{\mathsf{s}} = h * P\}\, A\, \{\sigma'_{\mathsf{s}} = h' * Q\}@\, (\mathcal{P} \rtimes \mathcal{U}) \rtimes \mathcal{V}}{\begin{array}{l}\Gamma \vdash \mathbf{hide}_{\Phi,g}\ e\ :\ \ \{\Psi\ g\ h * (\Phi\,(g) \mathrel{-\!\!*} P)\}\, A \\ \qquad\qquad\qquad\quad \{\exists\, g'.\ \Psi\ g'\ h' * (\Phi\,(g') \mathrel{-\!\!*} Q)\}@\, \mathcal{P} \rtimes \mathcal{U}\end{array}}\ \textsc{Hide}$$

$$\text{where } \Psi\ g\ h\ = \exists\, k{:}\mathsf{heap}.\, \sigma_{\mathsf{s}} = h \cup k \wedge \Phi\,(g) \downarrow k$$

$$\frac{\begin{array}{c}a = (\mathcal{U}, A, \mathsf{Pre}_a, \mathsf{Rel}_a) \text{ is an } \textit{action} \\ \Gamma \vdash (\mathsf{Pre}_a \wedge \mathsf{this}\ w, \lambda w'.\, (w, w', r) \in \mathsf{Rel}_a) \sqsubseteq (P, Q) \qquad P, Q\ \mathsf{stable}\ \mathcal{U}\end{array}}{\Gamma \vdash \mathbf{act}\ a : \{P\}\, A\, \{Q\}@\, \mathcal{U}}\ \textsc{Action}$$

Figure 3.5: Inference rules for FCSL commands. We assume that predicates P, R, Q, *etc.* in an FCSL ascription $e : \{P\}\, A\, \{Q\}@\, \mathcal{C}$ assert properties over states whose labels coincide with those of $\mathcal{C}$. In the same spirit, we consider that resource operations $\mathcal{U} \rtimes \mathcal{V}$ are valid, and therefore their sets of label are disjoint (*cf.* Definition 3.38). The $P\ \mathsf{stable}\ \mathcal{U}$ assertion recovers the familiar notion of *stability* under environment interference, and it is formally defined in Definition 3.27. FCSL *actions* are defined in Definiton 3.28. The $\sqsubseteq$ ordering between specifications used in rules Conseq and Action is defined in Definition **??**

suffice. In FCSL, however, logical variables scope locally over Hoare triples, and this scope has to be reflected in the semantic definition of $\sqsubseteq$ by introducing quantifiers. The definition is similar to the one by Kleymann (Kleymann, 1999).

Definition 3.3 (Conseq)**.** The Hoare ordering on two specifications (P_1, Q_1) and (P_2, Q_2) is defined as:

$$\begin{aligned}(P_1, Q_1) \sqsubseteq (P_2, Q_2) \iff \\ \forall w\ w'.\ (w \models \exists \bar{v}_2.\, P_2 \implies w \models \exists \bar{v}_1.\, P_1) \wedge \\ (\forall \bar{v}_1\ r.\, w \models P_1 \implies w' \models Q_1) \implies (\forall \bar{v}_2\ r.\, w \models P_2 \implies w' \models Q_2)\end{aligned}$$

where $\bar{v}_i = \mathsf{FLV}(P_i, Q_i)$ denotes the free logical variables from each pre- and postcondition pair.

Action Actions perform atomic steps from state to state, such as, *e. g.*, realigning the boundaries between, or changing the contents of *self*, *joint* and *other* state components. The actions thus serve to *synchronize* the changes to the underlying machine state (*i. e.*, heaps), with the changes to the logical information required for verification (*i. e. auxiliary* or *ghost* state: $\mathsf{a_s}$, $\mathsf{a_o}$, *etc.*). An action is a tuple $a = (\mathcal{C}, A, \mathsf{Pre}_a, \mathsf{Rel}_a)$ as it will be specified later on in Definition 3.28. We explain briefly the contents of the 4-tuple, and delay the formal treatment of actions to Section 3.3.4. The type A denotes the type of the return value of the command. The predicate Pre_a denotes the set of states in which is safe to execute the action a. The relation Rel_a realizes the effect of the atomic action a, connecting the pre- and post-states of the action and the returned value. It should be noticed that Pre_a and Rel_a are given in a large-footprint style, giving fully the heaps and the auxiliaries they accept.

The rule Action takes an action a and checks that a satisfies that Pre_a can be strengthened into P and Rel_a can be weakened into Q. As Rel_a is not a postcondition, but rather a relation taking input states, we first introduce a fresh logical variable w to name the input state using a predicate this. Then the predicate expressing post states for the action is computed out of Rel_a and w, and it is this predicate which is weakened into Q. Again, we require that P and Q are *stable* assertions under $\mathcal{U}$'s interference, in order to to account for the possibility that another thread modifies the shared state of the resource just before—or just after, or both—the atomic action a is executed. We implicitly require also that the action a is a *proper* action (Proposition 3.29) and moreover, that it is *operational* (Definition 3.35).

The rule Action does not prescribe however specific guidance on how an action manipulates the logical state, as long as it satisfies the $\sqsubseteq$ entailment. In this sense, the rule is generic, as it delegates the operational aspect of changing the state atomically to a user-selected action for the particular resource $\mathcal{C}$. As an example we consider the action $\mathsf{release}$, which implements the command `unlock` used in

Figure 3.1 to release a lock and transfer the pointer x from a private heap of a thread to the ownership of the lock resource:

Example 3.4 (release action)**.** The action release for the $\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}$ resource is a tuple $\mathsf{release} = (\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}, (\,), \mathsf{Pre}_{rls}, \mathsf{Rel}_{rls})$ such that:

$$
\begin{array}{lrcl}
w \in \mathsf{Pre}_{rls} & \iff & w = & \mathsf{priv} \mapsto [x \mapsto (\mathsf{a_s} + \mathsf{a'_s} + \mathsf{a_o}) \cup h_\mathsf{s} \mid \emptyset \mid h_\mathsf{o}] \cup \\
& & & \mathsf{lock} \mapsto [(\mathsf{Own}, \mathsf{a'_s}) \mid lk \mapsto \mathsf{true} \mid (\cancel{\mathsf{Own}}, \mathsf{a_o})] \\
(w, w', r) \in \mathsf{Rel}_{rls} & \iff & w = & \mathsf{priv} \mapsto [x \mapsto (\mathsf{a_s} + \mathsf{a'_s} + \mathsf{a_o}) \cup h_\mathsf{s} \mid \emptyset \mid h_\mathsf{o}] \ \cup \\
& & & \mathsf{lock} \mapsto [(\mathsf{Own}, \mathsf{a'_s}) \mid lk \mapsto \mathsf{true} \mid (\cancel{\mathsf{Own}}, \mathsf{a_o})] \ \wedge \\
& & w' = & \mathsf{priv} \mapsto [h_\mathsf{s} \mid \emptyset \mid h_\mathsf{o}] \cup \\
& & & \mathsf{lock} \mapsto [\ (\cancel{\mathsf{Own}}, \mathsf{a_s} + \mathsf{a'_s}) \mid \\
& & & \qquad lk \mapsto \mathsf{false} \cup x \mapsto (\mathsf{a_s} + \mathsf{a'_s} + \mathsf{a_o}) \mid \\
& & & \qquad (\cancel{\mathsf{Own}}, \mathsf{a_o})] \ \wedge \\
& & r = & (\,)
\end{array}
$$

This action is defined over the entangled resource $\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}} = \mathcal{P} \bowtie \mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$ as it transfers the ownership of $(x \mapsto -)$. It can be executed in states in which the lock is taken by the *self* thread, and the pointer x is in the private heap. The contents of x is $\mathsf{a_s} + \mathsf{a'_s} + \mathsf{a_o}$, for some (existentially quantified) $\mathsf{a_s}$,$\mathsf{a'_s}$ and $\mathsf{a_o}$, so that once x is transferred to the ownership of the lock resource, it satisfies the resource invariant. Once the states are flattened into heaps (*cf.*. Definition 3.12), the Pre_{rls} and Rel_{rls} components of release reduce to predicates and relations on heaps which describe the behaviour of a memory mutation on the pointer lk. Thus, operationally, release can be implemented as a single mutation to the lk pointer, and satisfies the *operational* requirement on actions[3].

We apply now the release action to define the `unlock` command introduced in Section 3.1:

$$
\begin{array}{lcl}
\texttt{unlock} & : & [j,\, k].\ \{\sigma_\mathsf{s} = x \mapsto j + k \wedge \mu_\mathsf{s} = \mathsf{Own} \wedge \alpha_\mathsf{s} = j \wedge \alpha_\mathsf{o} = k\}(\,) \\
& & \qquad \{\sigma'_\mathsf{s} = \emptyset \wedge \mu'_\mathsf{s} = \cancel{\mathsf{Own}} \wedge \alpha'_\mathsf{s} = j\} @ \mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}} \\
& \widehat{=} & \textbf{do}\ (\textbf{act}\ \mathsf{release})
\end{array}
$$

We will explain now how its specification arises from the rule Action. First, we have already established that release is a *proper* and *operational* action. Then, we need to assert the *stability* of the pre- and postconditions under $\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}$ environment interference.

[3]We deliberately leave aside here the proofs that release satisfies the formal requirements in order to be a *proper* FCSL action from Proposition 3.29, as they are straightforwards. In our Coq implementation, however, we do not have this luxury: actions are implemented through a type class (technically, a *canonical structure*) which requires such proofs in order to accept release as a proper instance.

As for the precondition, we know that if the *self* thread owns the lock lk, as witnessed by $\mu_{\mathsf{s}} = \mathsf{Own}$, *other* threads cannot increase their contributions, so the assertion $\alpha_{\mathsf{o}} = \mathsf{a}_{\mathsf{o}}$ is valid for all possible steps the environment can perform and also the value of *self getters* is fixed under *other* steps (*cf.*. the formal definition of $\mathsf{R}_{\mathcal{U}}$ in Definition 3.26). The postcondition is trivially stable after the latter fact, given that it only specifies assertions using *self* getters.

The interesting step here is to show show that

$$(\mathsf{Pre}_{rls} \wedge \mathsf{this}\ w, \lambda w'.\,(w, w', r) \in \mathsf{Rel}_{rls}) \sqsubseteq (P', Q')$$

where P' and Q' are the specifications above. For the *pre-strengthening* part of $\sqsubseteq$, it boils down to proving that $w \models P' \implies w \in \mathsf{Pre}_{rls}$, *i. e.* that a state which satisfies the precondition of `unlock` is *safe* to execute release. This is can be done immediately given the specification and the definition rls in Example 3.4 above. It suffices to instantiate $\mathsf{a}_{\mathsf{s}} = 0$, $\mathsf{a}'_{\mathsf{s}} = j$ and $\mathsf{a}_{\mathsf{o}} = k$. As for the post-weakening part, we need to show that if w is a state satisfying the pre-condition, then $(w, w', ()) \in \mathsf{Rel}_{rls} \implies w' \models Q'$, *i. e.* we need to show that the end-state after executing the atomic action release satisfies the post-condition Q'. Again, this is done straightforwards, using the same instances as before.

Hide In FCSL, *Hiding* refers to the ability to construct a fine-grained resource $\mathcal{V}$ from the underlying thread-private heap (governed by the resource $\mathcal{P}$), in the local scope of a thread e. Thus, the children forked by e can share and interfere with $\mathcal{V}$'s state, respecting $\mathcal{V}$'s transitions, but $\mathcal{V}$ is hidden from the environment of the parent thread c. In this environment, the changes in the state done following $\mathcal{V}$'s rules will be reflected as changes to (a part of) the private heap of c. In this sense, hiding generalizes *resource* rules of Concurrent Separation Logic (O'Hearn, 2007) to FCSL fine-grained resources.

Since creating the resource $\mathcal{V}$ requires to consume a *chunk* of private heap, the rule Hide requires the overall resource to support private heaps, *i. e.*, to be an entanglement $\mathcal{P} \rtimes \mathcal{U}$, with an arbitrary resource $\mathcal{U}$. Notice that we introduce $\mathcal{U}$ so as to be able to nest calls to the Hide rule, as well as to provide resources other than $\mathcal{P}$. We can always instantiate $\mathcal{U}$ with the empty resource $\mathcal{E}$ (Definition 3.43), for which $\mathcal{P} \rtimes \mathcal{E} = \mathcal{P}$ (Lemma 3.44).

As we have mentioned before for Hide, the rule Hide also assumes a formation lemma on the entanglement operations which requires that they are well defined, and therefore $\mathcal{U}$, $\mathcal{P}$, and $\mathcal{U}$ should have disjoint label sets (*cf.* Definition 3.38).

The formal properties of hiding and the annotation Φ are presented in Section 3.3.5. For now, we give an intuition of its role creating the new resource $\mathcal{V}$ out of $\mathcal{P}$. $\Phi(g)$ corresponds to a set of *concrete states* of a resource $\mathcal{V}$ to be created. Its parameter g is a meaningful abstraction of such a set (*e. g.*, $(\mathsf{m}_{\mathsf{s}}, \mathsf{a}_{\mathsf{s}})$ for $\mathcal{L}_{\{\mathsf{lock}, lk, \mathcal{I}\}}$) and can be thought of as an "abstract state". The annotation g (which we colloquially refer to as the *seed* for Φ) is the initial abstract state: upon creation, the state of $\mathcal{V}$ satisfies $\Phi\,(g)$.

1 $\{\sigma_{\mathsf{s}} = lk \mapsto \mathsf{false} \cup x \mapsto 0\}@\mathcal{P}$
2 $\{\Psi\ (\cancel{\mathsf{Own}}, 0)\ \emptyset\}@\mathcal{P}$
3 $\{\Psi\ (\cancel{\mathsf{Own}}, 0)\ \emptyset * (\Phi\ (\cancel{\mathsf{Own}}, 0) \mathrel{-\!\!*} (\mu_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha_{\mathsf{s}} = 0))\}@\mathcal{P} \rtimes \mathcal{E}$
4 $\mathtt{hide}_{\Phi,(\cancel{\mathsf{Own}},0)}$ $\{\ \sigma_{\mathsf{s}} = \emptyset \wedge \mu_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha_{\mathsf{s}} = 0\}@\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}$
5 $\mathtt{incr}\ n$
6 $\{\sigma_{\mathsf{s}} = \emptyset \wedge \mu_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha_{\mathsf{s}} = n\}@\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}}$
7 $\{\exists\, g_2.\ \Psi\ g_2\ \emptyset * (\Phi\ g_2 \mathrel{-\!\!*} (\mu_{\mathsf{s}} = \cancel{\mathsf{Own}} \wedge \alpha_{\mathsf{s}} = n))\}@\mathcal{P} \rtimes \mathcal{E}$
8 $\{\Psi\ (\cancel{\mathsf{Own}}, n)\ \emptyset\}@\mathcal{P}$
9 $\{\sigma_{\mathsf{s}} = lk \mapsto \mathsf{false} \cup x \mapsto n\}@\mathcal{P}$

Figure 3.6: A proof outline for **hide** ($\mathtt{incr}\ n$). Notice that on line 4, $\mathcal{L}_{\text{CSL }\{\mathsf{lock},lk,\mathcal{I}\}} = \mathcal{P} \rtimes \mathcal{E} \rtimes \mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$, which allows us to apply the HIDE rule.

In the premise of the HIDE rule, the assertion $\sigma_{\mathsf{s}} = h$ and $\sigma'_{\mathsf{s}} = h'$ describe the behaviour of e on the private heaps of $\mathcal{P}$, described as transition from the heap h to the heap h'. Moreover, P and Q specify the behaviour of e with regard to components of $\mathcal{U}$ and $\mathcal{V}$ **only!**— as per the definition of the separating conjunction on resources described in Section 3.1. In the conclusion, $\Psi\ g\ h$ and $\Psi\ g'\ h'$ map the abstract states g and g' into private heaps h and h'. This follows from the definition of Ψ, in which $\Phi\,(g) \downarrow k$ indicates that states satisfying $\Phi\,(g)$ *flattens* to the private heap k (see Definition 3.13). Thus, changes that e imposes on the abstract states on $\mathcal{V}$, will be seen as changes to the private heaps in the environment of $\mathsf{hide}_{\Phi,g}\ e$. In the conclusion, the assertion $\Phi\,(g) \mathrel{-\!\!*} p$ states that attaching any state satisfying $\Phi\,(g)$ to the chunk of the initial state identified by the labels from $\mathcal{U}$ produces a state in which P holds, "compensating" for the component k in Ψ. That is, P corresponds to an abstract state g and c can be safely executed in such a state. The rule guarantees that if e terminates with a postcondition Q, then Q corresponds to some abstract state g'.

We illustrate the use of HIDE with a proof outline for the following program $\mathbf{hide}_{\Phi,g}\ (\mathtt{incr}\ n)$ in Figure 3.6. The program implements the functionality of creating a *spin-lock*: outside the resource scope, we will start with the resource $\mathcal{P}$, and the private heap containing pointers lk and x. Then, we will use **hide** to install the $\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$ resource, which makes x a shared pointer, protected by the lock lk. Then it runs $\mathsf{incr}(n)$ in the new context, after which the local resource is disposed, and lk and x return to the private heap. Finally, we prove that if initially $x \mapsto 0$, then in the end $x \mapsto n$.

The abstract states are pairs $(\mathsf{m_s}, \mathsf{a_s})$, encoding of the *self* views of the concrete state of lock. Φ maps a *self* view into a predicate on the full state of lock, specifying

joint and *other* views as well.

$$\begin{aligned}\Phi\ (\mathsf{m_s}, \mathsf{a_s}) \ \widehat{=}\ & \mu_\mathsf{s} = \mathsf{m_s} \wedge \alpha_\mathsf{s}\mathsf{a_s}\ \wedge \mu_\mathsf{o} = \cancel{\mathsf{Own}} \wedge \alpha_\mathsf{s} = 0 \wedge \\ & \mathit{if}\ \mathsf{m_s} = \cancel{\mathsf{Own}}\ \mathsf{then}\ \rho_\mathsf{j} = lk \mapsto \mathsf{false} \cup x \mapsto \mathsf{a_s} \\ & \mathit{else}\ \rho_\mathsf{j} = lk \mapsto \mathsf{true}\end{aligned}$$

We choose the initial state $g = (\mathsf{m_s}, \mathsf{a_s}) = (\cancel{\mathsf{Own}}, 0)$ on line 4, entailing that the lock is installed with lk unlocked, and x set to 0. Then, we observe that:

$$\Phi\ (\cancel{\mathsf{Own}}, n) \downarrow lk \mapsto \mathsf{false} \cup x \mapsto n$$

ad thus we have $\Psi\ (\cancel{\mathsf{Own}}, 0)\ \emptyset = \sigma_\mathsf{s} = lk \mapsto \mathsf{false} \cup x \mapsto 0$ on line 2 and $\Psi\ (\cancel{\mathsf{Own}}, n)\ \emptyset = \sigma_\mathsf{s} = lk \mapsto \mathsf{false} \cup x \mapsto 9$ on line 9. Finally, we use the fact that $\Phi\ (\mathsf{m_s}, \mathsf{a_s}) \multimap (\mu_\mathsf{s} = \mathsf{m'_s} \wedge \alpha_\mathsf{s} = \mathsf{a'_s})$ is equivalent to $(\mathsf{m_s}, \mathsf{a_s}) = (\mathsf{m'_s}, \mathsf{a'_s})$ in the label-free state.

The soundness of Hide depends on a number of properties and definitions that will be presented later on Section 3.3.5. In particular, Φ satisfies the properties described in Proposition 3.46. The complete proofs are given in our mechanization (FCSL, 2017).

3.2.2 Structural rules of FCSL

In this chapter's Overview (§ 3.1) we introduced FCSL's Frame rule. However, we have not listed that rule, nor any other *structural* rule in Figure 3.5. This is because, as we did for HTTcc in Chapter 2, we follow the philosophy of later HTT implementations (Nanevski *et al.* , 2010) which distinguish between a core of commands whose inference rules are *instrumented*, *i. e.* implemented in Coq via a definition which implements their semantics, and *structural* rules for the logic, which are implemented as derived lemmas.

In order to present the latter, we will present first an alternative type for computations $e : \mathsf{Prog}_\mathcal{C}\ A$ which strips FCSL commands of their specifications. In Section 3.4.1, we will define this type to give a denotational model of the logic's commands. Full specifications are recovered by combining this bare type of computations with a predicate transformer, verify, which realizes FCSL's verification framework: a FCSL assertion will be presented as in Coq as a proof goal

$$\mathsf{verify}_\mathcal{U}\ e\ Q\ w$$

where w is the current state, $e : \mathsf{Prog}_\mathcal{C}\ A$ and Q is a post-condition of type $Q : A \to \mathsf{state} \to \mathsf{Prop}$. In Section 3.4, the implementation of verify (Definition 3.62) will show that the partial assertion $\mathsf{verify}_\mathcal{U}\ e\ Q$ entails the computation of the *weakest precondition* for the command e.

However, we do not need to unfold the definition of verify to reason with clients. A user of the logic discharges proof obligations by the application of

several lemmas that allows us to reason in terms of verify assertions. Thus, we keep the predicate transformer verify abstract, and rely on these *structural* rules implement derived lemmas that enable logical manipulation of the goal and also reflect internal semantic properties to the proof outline.

$$
\begin{array}{lcl}
\textsc{VrfCoh} & : & \mathsf{verify}_{\mathcal{U}}\ e\ Q\ w \implies w \in \mathsf{Coh}_{\mathcal{U}} \\
\textsc{VrfStable} & : & \mathsf{verify}_{\mathcal{U}}\ e\ Q\ w \implies Q\ \mathsf{stable}\ \mathcal{U} \\
\textsc{VrfPost} & : & (\forall\, r\, s.\ s \in \mathsf{Coh}_{\mathcal{U}} \implies Q_1\ r\ s \implies Q_2\ r\ s) \implies \\
 & & \quad \mathsf{verify}_{\mathcal{U}}\ e\ Q_1\ w \implies \mathsf{verify}_{\mathcal{U}}\ e\ Q_2\ w \\
\textsc{Frame} & : & R\ \mathsf{stable}\ \mathcal{U} \implies R\ [p \mid w_{\mathsf{j}} \mid w_{\mathsf{o}} \circ w_{\mathsf{j}}] \implies \\
 & & \quad \mathsf{verify}_{\mathcal{U}}\ e\ Q\ (w \triangleright p) \implies \\
 & & \quad\quad \mathsf{verify}_{\mathcal{U}}\ e\ (\lambda\, r.\, Q\ r \circledast R)\ (w \triangleleft p) \\
\textsc{Conjunction} & : & \mathsf{verify}_{\mathcal{U}}\ e\ P\ w \implies \mathsf{verify}_{\mathcal{U}}\ e\ Q\ w \implies \mathsf{verify}_{\mathcal{U}}\ e\ (P \wedge Q)\ w \\
\textsc{Implication} & : & (\forall\, s.\, s \in \mathsf{Coh}_{\mathcal{U}} \implies P\ s \implies \mathsf{verify}_{\mathcal{U}}\ e\ Q\ s) \implies \\
 & & \quad \mathsf{verify}_{\mathcal{U}}\ e\ (\lambda\, r\, m.\, P\ m \implies Q\ r\, m)\ w
\end{array}
$$

Figure 3.7: Selected FCSL structural rules.

Figure 3.7 presents a subset of these lemmas, all of which have been proved correct in our Coq implementation. The VrfCoh and VrfStable lemmas are example of lemmas that reflects internal properties of the commands or the resources to the verification task. The former says that if a state is in $\mathsf{verify}_{\mathcal{U}}\ e\ Q$, then it must also be in the set of valid states of the resource $\mathcal{U}$, $\mathsf{Coh}_{\mathcal{U}}$. In other words, our proof outlines are only concerned with states that are valid for the underlying resource. The latter, VrfStable reflects the fact that FCSL operational semantics interleaves *self* with environment *steps* and thus we need to provide stable specifications for verify. This fact will become more evident when we present the definition of the verify predicate together with the semantic of FCSL Hoare-triples later in Section 3.4.

The other *structural* lemmas allow us to manipulate the current assertion of a proof outline. The vrfPost lemma, for instance, allows us to weaken the postcondition Q_1 into Q_2 if the first implies the second for every return value r and state s. When proving Q_2 out of Q_1, it is sound to further assume $w \in \mathsf{Coh}_{\mathcal{U}}$, because verify is only concerned with states that are valid for the resource $\mathcal{U}$. The Conjunction and Implication rules are straightforward implementations of the corresponding Hoare/separation logic structural rules.

The most relevant *structural* rule is the Frame rule, which implements the intuition we have presented as a rule in the Overview (§ 3.1). Consider a predicate R which is stable under the environment interference of the resource $\mathcal{U}$ ($R\mathsf{stable}\mathcal{U}$), and a PCM-map p such that R holds in a state whose *self* component is determined

by p, and the *other* component takes *self* and *other* components from a state w, *i. e.* $R\ [p \mid w_\mathsf{j} \mid w_\mathsf{o} \circ w_\mathsf{j}]$. Then, if we can verify that Q holds in a *small* state w, where p is part of the environment—$\mathsf{verify}_\mathcal{U}\ e\ Q\ \ (w \triangleright p)$—, we can verify $Q \circledast R$ in a *larger* state $w \triangleleft p$ where we have *zigged* p together with w's *self* component.

Notice that, unlike other separation logics where frames can materialize *out of thin air*, the full-world assumption of FCSL state forces us to distinguish a piece of the state where such framing occurs. Still, we can take p to be the full *self* component of a state and thus recover the usual trick of pushing all previous contributions to the frame, as we did after the first call to `incr` (line 4) in the proof of `twice` in Figure 3.3.

3.3 Formal Structures

3.3.1 PCMs & Subjective State

The state w of a concurrent resource $\mathcal{C}$ in FCSL always consists of three distinct auxiliary variables that we name a_s, a_o and a_j, such that $w = [a_\mathsf{s} \mid a_\mathsf{j} \mid a_\mathsf{s}]$. These stand for the abstract *self*, *other*, and *joint* projections of the state. However, the user can pick the types of these variables based on the application. It is essential that a_s and a_o have a common type, which moreover, exhibits the algebraic structure of a *partial commutative monoid* (PCM)[4]:

Definition 3.5 (PCM)**.** A *partial commutative monoid* is a tuple $(\mathbb{U}, \bullet, \epsilon)$ where $\mathbb{U}$ is a carrier set and $\bullet$ is a *partial*, commutative, and associative binary operator with $\epsilon \in \mathbb{U}$ as a unit.

Definition 3.6 (Defined)**.** Given a PCM $(\mathbb{U}, \bullet, \epsilon)$, we associate a predicate $\mathsf{defined}$ which asserts whether the operation $\bullet$ is defined over a particular choice of elements.

O'Hearn *et al.* first identified the partial commutative structure of heaps in separation logic. In SCSL (Ley-Wild & Nanevski, 2013), it was proposed to use user defined PCM structures to abstract the space of ghost states, as a basis for defining subjective state in a concurrent setting. We next list some characteristic examples of PCMs which feature regularly in the design of the auxiliary state of concurrent resources:

Example 3.7 (PCM examples)**.** We list some examples of PCMs:

1. $(\{\, p \mapsto v \mid p \in \mathbb{N}^+ \}, \cup, \mathsf{empty})$ *i. e.* Separation logic heaps, with disjoint heap union $\cup$ and the empty heap empty as a unit; such that

$$\mathsf{defined}\ (p \mapsto \_ \cup q \mapsto \_) \iff p \neq q$$

[4]Oddly enough, the acronym PCM is also used to abbreviate the slightly different algebraic structure of a *partially-commutative* or *trace* monoid. In the latter structure, in contrast, the binary operator is always *defined*, but not all products commute.

2. $(\mathbb{N}, +, 0)$ *i. e.* the commutative monoid defined by natural numbers under addition. Note that this PCM is in fact total. Any *total* commutative monoid is trivially a PCM.

3. $(\{\,\mathsf{Own}, \cancel{\mathsf{Own}}\}, \bullet, \cancel{\mathsf{Own}})$ the PCM of mutex ownership, with:

$$\mathsf{defined}\ (m_1 \bullet m_2) \iff \neq (m_1 = m_2 = \mathsf{Own})$$

4. $(\{\, q \mid q \in \mathbb{Q} \wedge 0 \leq q \leq 1\}, \bullet, 0)$ *i. e.* the PCM of *fractional permissions* where $\bullet$ is defined by addition such that:

$$\mathsf{defined}\ (q_1 \bullet q_2) \iff q_1 + q_2 \leq 1$$

5. Given a type A, the PCM of *finite sets*, $(\{t \mid t \in A\}, \cup, \emptyset)$, with disjoint union $\cup$ and the empty set $\emptyset$, such that

$$\mathsf{defined}\ (A \cup B) \iff A \cap B = \emptyset$$

6. Given a type A, the PCM of *finite maps*, $(\mathbb{N} \rightharpoonup_{\text{fin}} A, \cup, \emptyset)$, with disjoint union $\cup$ and the empty map $\emptyset$, such that

$$\mathsf{defined}\ (f \cup g) \iff \mathsf{dom}(f) \cap \mathsf{dom}(g) = \emptyset$$

The PCMs of mutexes, natural numbers and heaps constitute the different components of the auxiliary state of the $\mathcal{L}_{\text{CSL}\ \{\mathsf{lock}, lk, \mathcal{I}\}}$ fine-grained resource introduced in Section 3.1. The combination of the PCMs of *heaps* and *fractional permissions* is the basis for the $x \overset{q}{\mapsto} v$ assertions which allowed limited sharing of heap variables as shared resources in the work by Bornat *et al.* (Bornat *et al.* , 2005)[5]. Moreover, permissions are a regular feature of the design of several program logics for shared variable concurrency reasoning inspired after CSL, *e. g.* (Dinsdale-Young *et al.* , 2010; Svendsen & Birkedal, 2014; Dodds *et al.* , 2009; Jacobs & Piessens, 2011), where permissions realize *ownership* of a concurrent resource, or the ability to execute atomic changes in some notion of shared space. The PCM of *finite sets* is used in of the implementation of concurrent coarse-grained sets (Ley-Wild & Nanevski, 2013; Nanevski, 2016), as well in other FCSL case studies (Nanevski *et al.* , 2014b; Sergey *et al.* , 2016). We use *finite maps* to define history-based resources aimed at the specifications of fine-grained data structures in Chapter 4. History-based fine-grained resources have been a regular feature of recent verification efforts in FCSL (Sergey *et al.* , 2015b,a, 2016).

[5]The original formulation of fractional permissions in (Bornat *et al.* , 2005) actually varies slightly from the one given in Example 3.7 above: the carrier set is defined to be $\{q \mid q \in \mathbb{Q} \wedge O < q \leq q\}$. This forbids 0 from being the unit and it also entails that the structure, as is, it is not an actual PCM. This situation is however changed in subsequent works where 0 permissions actually make sense.

A remark on the implementation of PCMs in Coq It should be noticed that in order to make the implementation of FCSL cleaner, and also, in order to make the verification experience smoother, we reckon it is better to deal with *partiality* by making the binary operation $\bullet$ total. A more "dependently-typed" approach could have been to encode *defined by construction* $\bullet$ operations *e. g.* the one for *fractional permissions* as:

$$\begin{array}{lcl} \textsc{safeJoin} & : & \forall\,(p\,q{:}\mathbb{Q}).\ p+q \leq 1 \to \mathbb{Q} \\ \textsc{SafeJoin}\ p\,q\,pf & \mathrel{\widehat{=}} & p+q \end{array}$$

This definition might look elegant *a priori* but, we think it is a paradigmatic case of "trying too hard to be smart" with Coq, which would eventually had created the need to carry around proof terms like *pf* in derived definitions. Instead, we use the Maybe monad—or in more appropriate Coq terms, the option data type—to lift of the carrier set $\mathbb{U}$ to $\mathbb{U}^\dagger \mathrel{\widehat{=}} \mathbb{U} \cup \{\,\mathsf{undefined}\}$. Not only this choice prevents the excessive proliferation of proof arguments but also it has the advantage of providing an immediate definition of defined as:

$$\mathsf{defined}\ e \mathrel{\widehat{=}} e \neq \mathsf{undefined}$$

PCM constructions Partial commutative monoids are closed under *products*. This fact allow us to combine PCMs as the ones presented in Example 3.7 to define the state space of complex resources. The product PCM is defined as follows:

Definition 3.8 (Product PCM)**.** Given two PCMs $(A, \bullet, \epsilon)$ and $(B, \circ, \varepsilon)$ we define the *product* PCM by point-wise lifting each operation to the product of the sets. That is the *product* is a PCM $(A \times B, \star, (\epsilon, \varepsilon))$ where:

$$(a_1, b_1) \star (a_2, b_2) \mathrel{\widehat{=}} (a_1 \bullet a_2, b_1 \circ b_2)$$

is such that,

$$\mathsf{defined}\,((a_1, b_1) \star (a_2, b_2)) \iff \mathsf{defined}\,(a_1 \bullet a_2) \wedge \mathsf{defined}\,(b_1 \circ b_2)$$

In order to define FCSL states, we need an auxiliary construction on PCMs, *PCM-maps*. These are a particular class of finite maps with a *non-disjoint* union which distributes over $\bullet$.

Definition 3.9 (PCM-map and *zip*)**.** A PCM-map is a finite map from labels (isomorphic to nat) to $\Sigma_{U:\mathrm{pcm}}U$ which associates each label with a pair of a PCM U and a value $v \in U$. PCM-maps support a partial *zipping* operation which $\bullet$-joins the components of two PCM-maps:

$$\begin{array}{ll} \emptyset \circ \emptyset & \mathrel{\widehat{=}}\ \emptyset \\ (\mathsf{l} \mapsto_U v_1) \uplus m_1) \circ ((\mathsf{l} \mapsto_U v_2) \uplus m_2) & \mathrel{\widehat{=}}\ (\mathsf{l} \mapsto_U v_1 \bullet_U v_2) \uplus (m_1 \circ m_2) \\ \_ \mapsto \_ \circ \_ \mapsto \_ & \quad \mathsf{undefined}\ \text{otherwise} \end{array}$$

Definition 3.10 (States)**.** Given a type A, a FCSL state w of state-type $\mathsf{state}\,A$ is a triple $[s \mid j \mid o]$, where s, o are PCM-maps, and j is an A-map.

Definition 3.11 (State Union)**.** The disjoint union of FCSL states is defined by lifting the disjoint union of the components component-wise:

$$s \cup w \;\widehat{=}\; \begin{cases} [s_{\mathsf{s}} \cup w_{\mathsf{s}} \mid s_{\mathsf{j}} \cup w_{\mathsf{j}} \mid s_{\mathsf{o}} \cup w_{\mathsf{o}}] & \textit{iff } \mathsf{defined}(s_i \cup w_i)\ i \in \{\mathsf{s},\mathsf{j},\mathsf{o}\} \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Notation We abbreviate $[\mathsf{l} \mapsto v_s \mid \mathsf{l} \mapsto v_j \mid \mathsf{l} \mapsto v_o]$ with $\mathsf{l} \mapsto [v_s \mid v_j \mid v_o]$. Notice also that we overload the $\cup$ symbol for *disjoint* unions of finite maps $\cup$ to other structures that "look like a finite map" *e. g.* states, heaps, PCM-maps, etc.

Definition 3.12 (State Flattening)**.** State flattening $\lfloor w \rfloor$ is the disjoint union of the underlying heap footprint of a state w:

$$\begin{array}{ll} \lfloor \emptyset \rfloor & \widehat{=}\ \emptyset \\ \lfloor l \mapsto_U [s \mid j \mid o] \cup m \rfloor & \widehat{=}\ \mathsf{flatts}\,U\,s \cup \mathsf{flatts}\,U\,s \cup \mathsf{flatts}\,U\,o \cup \lfloor m \rfloor \\ & \quad \text{where } \mathsf{flatts}\,U\,h \;\widehat{=}\; \textit{if } U = \mathsf{heap}\ \textit{then } h \textit{ else } \emptyset \end{array}$$

Given the definition of flattening above, we can now define formally the assertion $w \downarrow h$ from the Hide rule in Figure 3.5 as follows:

Definition 3.13 (Erasure)**.** Given a state w and a heap k we define $w \downarrow k$ as:

$$w \downarrow k \iff \lfloor w \rfloor = k$$

Definition 3.14 (Valid States)**.** A state w is valid if and only if the following properties hold:

- w_{s}, w_{j}, w_{o} have the same label domains *i. e.*: $\mathsf{dom}(w_{\mathsf{s}}) = \mathsf{dom}(w_{\mathsf{j}}) = \mathsf{dom}(w_{\mathsf{o}})$.
- Zipping w_{s} and w_{o} is defined *i. e.* $\mathsf{defined}\ (w_{\mathsf{s}} \circ w_{\mathsf{o}})$.
- The underlying heap flattening of w is defined, *i. e.* $\mathsf{defined}\,(\lfloor w \rfloor)$. This entails that the heap components in w_{s}, w_{o} and w_{j} are disjoint.

Definition 3.15 (State transposition)**.** Given a state $w = [a_s \mid a_j \mid a_o]$, the *transposition* of w is the state $w^{\top} = [a_o \mid a_j \mid a_s]$.

Definition 3.16 (State framing)**.** Let w be a state of type $\mathsf{state}\,T$ and p a PCM-map. Then *self-framing of w with p* is the state $w \triangleright p = (a_s \bullet p, a_j, a_o)$. Dually, *other-framing of s with w* is $s \triangleleft p = (a_s, a_j, p \bullet a_o)$.

3.3.2 Transitions

We formalize now the concept of an FCSL transitions. As we have hinted informally before in Section 3.1, fine-grained resources in FCSL can be characterized as state transition systems over *subjective* states, whose transitions denote possible atomic manipulations of the—both real and auxiliary—state.

Definition 3.17 (Transitions)**.** A transition g of a resource $\mathcal{U}$ is a relation on states such that the following properties hold:

$$\begin{aligned}
&\textsc{Guarantee} \quad : \quad (w, w') \in \mathsf{g} \implies w_\circ = w'_\circ \\
&\textsc{Locality} \quad : \quad \forall t{:}\mathsf{PCMmap}.\ w_\circ = w'_\circ \implies \\
&\qquad\qquad\qquad\qquad (w \triangleright t, w' \triangleright t) \in \mathsf{g} \implies (w \triangleleft t, w' \triangleleft t) \in \mathsf{g}
\end{aligned}$$

Additionally, we will only consider transitions that relates states which belong to the *state space* $\mathsf{Coh}_\mathcal{U}$ (see Definition 3.22), *i. e.*

$$(w_1, w_2) \in \mathsf{g} \implies w_1 \in \mathsf{Coh}_U \wedge w_2 \in \mathsf{Coh}_U$$

The Guarantee property restricts g to only modify the *self* and *joint* components. Therefore, g describes the behaviour of a viewing thread in the subjective setting, but not of the thread's environment. In the terminology of Rely-Guarantee logics (Jones, 1983; Feng *et al.* , 2007; Feng, 2009; Vafeiadis & Parkinson, 2007), g is a *guarantee* relation. The Locality property ensures that if g relates states with a certain *self* components, then g also relates states in which the *self* components have been simultaneously *framed* by a PCM-map t, *i. e.*, enlarged according to t. This definition generalizes the notion of *locality* from separation logic (Reynolds, 2002). However, unlike the case of separation logic frames which can materialize out of nowhere, FCSL is designed on a "closed world" model and thus t has to be appropriated from the *other* component; that is, taken out—or transferred—from the ownership of the environment.

Moreover, even if FCSL transitions are perceived as *Guarantee* relations, they encode in fact, both *Guarantee* and Rely relations. This is because the behaviour of the environment is not free but rather a dual construction on the possible self-moves of the STS. Then, in order to describe the behaviour of the thread's environment, i.e. obtain a *rely* relation, we merely *transpose* the *self* and other components of g as follows:

Definition 3.18 (Transition Transposition)**.** We lift the definition of *state transposition* from Definition 3.15above to act point-wise over transitions:

$$\mathsf{g}^\top = \{(w_1^\top, w_2^\top) \mid (w_1, w_2) \in \mathsf{g}\}$$

Furthermore, FCSL transitions can be classified in *internal* and *external* transitions. Moreover, external transitions comes in pairs of symmetric *acquire* and *release* transitions.

Definition 3.19 (Internal Transitions)**.** An *internal* transition Int_U is a transition which is *reflexive* and preserves heap footprints, *i. e.*

$$\begin{array}{lll} \textsc{Reflexivity} & : & \forall\, w.\ (w,w) \in \mathsf{Int}_U \\ \textsc{Footprint Preservation} & : & \forall\, w_1\, w_2.\ (w_1,w_2) \in \mathsf{Int}_U \implies \\ & & \qquad \mathsf{dom}(\lfloor w_1 \rfloor) = \mathsf{dom}(\lfloor w_2 \rfloor) \end{array}$$

Internal transitions are reflexive so that programs specified by such transitions may be *idle* or *stuttering*—*i. e.*, they transition from a state to itself—. The Footprint Preservation requires internal transitions to preserve the domains of heaps obtained by state flattening. Internal transitions may exchange the ownership of sub-heaps between the *self* and *joint* components, or change the contents of individual heap pointers, or change the values of auxiliary state, which flattening erases. However, they cannot add new pointers to a state or remove old ones, which is the task of external transitions.

An external transition, then, is a function, mapping a heap h into a relation on $\mathsf{Coh}_\mathcal{V}$. As we mentioned above, they manifest in pairs, consisting of an *acquire* and a *release* transition.

Definition 3.20 (Acquire/Release Pair)**.** An *acquire* transition $\mathsf{acq}_\mathcal{U}$, and a *release* transition $\mathsf{rls}_\mathcal{U}$ are functions mapping heaps to transitions which extend and reduce heap footprints, respectively, as formalized below.

$$\begin{array}{lll} \textsc{Footprint Extension} & : & \forall\, h{:}\mathsf{heap}.\ (w,w') \in (\alpha\ h) \implies \\ & & \qquad \mathsf{dom}\ (\lfloor w \rfloor \cup h) = \mathsf{dom}\ \lfloor w' \rfloor \\ \textsc{Footprint Reduction} & : & \forall\, h{:}\mathsf{heap}.\ (w,w') \in (\rho\ h) \implies \\ & & \qquad \mathsf{dom}\ (\lfloor w' \rfloor \cup h) = \mathsf{dom}\ \lfloor w \rfloor \end{array}$$

These allow for the composition of different resources by *entangling* them (Section 3.3.5), *i. e.* interconnecting some (or all of) their dually polarized external transition pairs of two FCSL resources, to obtain a larger one.

Example 3.21 ($\mathsf{id}_\mathcal{U}$)**.** The *identity* or *diagonal* transition over $\mathsf{Coh}_\mathcal{U}$, is the least (reflexive) internal transition over $\mathsf{Coh}_\mathcal{U}$:

$$(a,b) \in \mathsf{id}_\mathcal{U} \iff a = b \wedge a \in \mathsf{Coh}_\mathcal{U}$$

3.3.3 Fine-Grained Resources

Definition 3.22 (Fine-grained Resource)**.** A FCSL *Fine-grained resource* or *concurroid*[6] $\mathcal{V}$ is a 4-tuple $\mathcal{V} = (\mathsf{Labs}_\mathcal{V}, \mathsf{Coh}_\mathcal{V}, \mathsf{Int}_\mathcal{V}, \mathsf{Ext}_\mathcal{V})$ where: (1) $\mathcal{L}$ is a set of labels (again, isomorphic to $\mathbb{N}$); (2) $\mathsf{Coh}_\mathcal{V}$ is a set of states which denotes the *Invariant* or *State space* of the resource; (3) $\mathsf{Int}_\mathcal{V}$ is the set of *internal transitions*

of the resource; (4) $\mathsf{Ext}_\mathcal{V}$ is a set of pairs $(\mathsf{acq}_\mathcal{V}, \mathsf{rel})$, where $\mathsf{acq}_\mathcal{V}$ and rel are *external transitions* of $\mathcal{V}$. Moreover, the states in $\mathsf{Coh}_\mathcal{V}$ satisfy the following properties:

$$\begin{aligned}
\textsc{State Validity} &: \forall w \in \mathsf{Coh}_\mathcal{V}.\ \mathsf{valid}\, w \\
\textsc{Label Set} &: \mathsf{Labs}_\mathcal{V} = \mathsf{dom}(w_\mathsf{s}) = \mathsf{dom}(w_\mathsf{j}) = \mathsf{dom}(w_\mathsf{o}) \\
\textsc{Fork-join Closure} &: \forall t{:}\mathsf{PCM} - \mathsf{map}.\ w \triangleleft t \in \mathsf{Coh}_\mathcal{V} \iff w \triangleright t \in \mathsf{Coh}_\mathcal{V}
\end{aligned}$$

The property State Validity asserts that every state $w \in \mathsf{Coh}_\mathcal{V}$ is *valid* as per Definition 3.14. Label Set commands that the labels of the resource matches matches with that of the states in the invariant space state $\mathsf{Coh}_\mathcal{V}$. As a consequence, states w' which have extra labels are not part of the *invariant space*, even if they have those labels required to satisfy the assertions of the fine-grained resource Invariant. Finally, the Fork-join Closure property requires that $\mathsf{Coh}_\mathcal{V}$ is closed under the realignment of self and other components, when they exchange a PCM-map t between them. Such realignment is part of the definition of $\circledast$, and thus appears in proofs whenever the rule Par is used, *i. e.* whenever threads fork or join. Fork-join closure ensures that if a parent thread forks in a state from $\mathsf{Coh}_\mathcal{V}$, then the child threads are supplied with states which also are in $\mathsf{Coh}_\mathcal{V}$, and dually for joining.

As an example, we give now the complete definitions for the fine-grained resources of *private state*, $\mathcal{P}$ and a single-cell *spin lock*, $\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$, which were presented in Section 3.1, as components of $\mathcal{L}_{\mathrm{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}$.

Example 3.23 ($\mathcal{P}$)**.** The fine-grained resource for *private heaps* is given by the tuple $\mathcal{P} = (\{\mathsf{priv}\}, \mathsf{Coh}_\mathcal{P}, \mathsf{Int}_\mathcal{P}, \{(\mathsf{acq}_\mathcal{P}, \mathsf{rls}_\mathcal{P})\})$, with:

$$\begin{aligned}
\mathsf{Coh}_\mathcal{U} &\mathrel{\widehat{=}} \{\ \mathsf{priv} \mapsto [h_\mathsf{s} \mid \emptyset \mid h_\mathsf{o}] \mid \mathsf{dom}(h_\mathsf{s}) \cap \mathsf{dom}(h_\mathsf{o}) = \emptyset\} \\
(w, w') \in \mathsf{Int}_\mathcal{P} &\iff w_\mathsf{s} = \mathsf{priv} \mapsto h_\mathsf{s} \wedge w'_\mathsf{s} = \mathsf{priv} \mapsto h'_\mathsf{s} \wedge \\
&\quad\ \mathsf{dom}(h_\mathsf{s}) = \mathsf{dom}(h'_\mathsf{s}) \wedge w_\mathsf{o} = w'_\mathsf{o} \\
(w, w') \in \mathsf{acq}_\mathcal{P}\ h &\iff w_\mathsf{s} = \mathsf{priv} \mapsto h_\mathsf{s} \wedge w'_\mathsf{s} = \mathsf{priv} \mapsto h_\mathsf{s} \cup h \wedge w_\mathsf{o} = w'_\mathsf{o} \\
(w, w') \in \mathsf{rls}_\mathcal{P}\ h &\iff w_\mathsf{s} = \mathsf{priv} \mapsto h_\mathsf{s} \cup h \wedge w'_\mathsf{s} = \mathsf{priv} \mapsto h_\mathsf{s} \wedge w_\mathsf{o} = w'_\mathsf{o}
\end{aligned}$$

The internal transition $\mathsf{Int}_\mathcal{P}$ admits arbitrary footprint-preserving changes to the private heap h_s, *e. g.* reading from and writing to the threads *owned* private heap. The acquire and release transitions $\mathsf{acq}_\mathcal{P}$ and $\mathsf{rls}_\mathcal{P}$ simply add and remove the the heap h from h_s. Notice that how the acquire and release transitions of $\mathcal{P}$ realize ownership transfer of the *real*—non-auxiliary—memory uniformly, in the same way as with any other auxiliary state. Notice that the explicit $w_\mathsf{o} = w'_\mathsf{o}$ statement implies the Guarantee property of transitions. This entails that the heap h being interchanged with acquire-release pairs cannot materialize from the

[6] In the original FCSL paper (Nanevski *et al.* , 2014a), we introduced the name *concurroid* as a short name for fine-grained concurrent resources but it has since fallen out of favour—at least in FCSL papers: we do still use the term colloquially sometimes, as it is shorter.

environment's contribution, but rather has to be provided by a dual transition (from a different resource). We will discuss this feature in more detail when when we present *entanglement* in Section 3.3.5.

Example 3.24 ($\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$)**.** The $\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}$ concurrent resource models a *spin* lock implemented on a single pointer cell lk, which protects the heap h enforcing the resource invariant $\mathcal{I}$ (O'Hearn, 2007). Given, the following definition for the *invariant space* $\mathsf{Coh}_{\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}}}$

$$
\begin{array}{rcl}
\mathsf{Coh}_{\mathcal{L}} & \widehat{=} & \{w \mid\ w \models \mathsf{Spin}_{\mathcal{I}}\} \\
\mathsf{Spin}_{\mathcal{I}} & \widehat{=} & \mathsf{lock} \overset{s}{\mapsto} (\mathsf{m_s}, \mathsf{a_s}) \land \mathsf{lock} \overset{o}{\mapsto} (\mathsf{m_o}, \mathsf{a_o}) \land \mathsf{lock} \overset{j}{\mapsto} ((lk \mapsto b) \cup h) \land \\
 & & \mathit{if}\ b\ \mathit{then}\ h = \emptyset \land \mathsf{m_s} \bullet \mathsf{m_o} = \mathsf{Own} \\
 & & \mathit{else}\ \mathcal{I}\ (\mathsf{a_s} \bullet \mathsf{a_o})\ h \land \mathsf{m_s} \bullet \mathsf{m_o} = \cancel{\mathsf{Own}}
\end{array}
$$

we define the resource $\mathcal{L}_{\{\mathsf{lock},lk,\mathcal{I}\}} = (\{\mathsf{lock}\}, \mathsf{Coh}_{\mathcal{L}}, \mathsf{Int}_{\mathcal{L}}, \{(\mathsf{acq}_{\mathcal{L}}, \mathsf{rls}_{\mathcal{L}})\})$, with the following transitions:

$$
\begin{array}{lcl}
(w, w') \in \mathsf{Int}_{\mathcal{L}} & \iff & w = w' \\
(w, w') \in \mathsf{acq}_{\mathcal{L}}\ h & \iff & w_{\mathsf{s}} = \mathsf{lock} \mapsto (\mathsf{Own}, \mathsf{a_s}) \land w_{\mathsf{j}} = \mathsf{lock} \mapsto (lk \mapsto \mathsf{true}) \land \\
 & & w'_{\mathsf{s}} = \mathsf{lock} \mapsto (\cancel{\mathsf{Own}}, \mathsf{a'_s}) \land \\
 & & w'_{\mathsf{j}} = \mathsf{lock} \mapsto ((lk \mapsto \mathsf{false}) \cup h) \land \\
 & & w'_{\mathsf{o}} = w_{\mathsf{o}} \\
(w, w') \in \mathsf{rls}_{\mathcal{L}}\ h & \iff & w_{\mathsf{s}} = \mathsf{lock} \mapsto (\cancel{\mathsf{Own}}, \mathsf{a_s}) \land \\
 & & w_{\mathsf{j}} = \mathsf{lock} \mapsto ((lk \mapsto \mathsf{false}) \cup h) \land \\
 & & w'_{\mathsf{s}} = \mathsf{lock} \mapsto (\mathsf{Own}, \mathsf{a_s}) \land w'_{\mathsf{j}} = \mathsf{lock} \mapsto (lk \mapsto \mathsf{true}) \\
 & & w'_{\mathsf{o}} = w_{\mathsf{o}}
\end{array}
$$

The internal transition $\mathsf{Int}_{\mathcal{L}}$ does not admit changes to the state w, and only allows for *stuttering* transitions. The $\mathsf{acq}_{\mathcal{L}}$ transition corresponds to unlocking, and hence to the (re-)acquisition of the heap h. It changes the ownership ghost from Own to $\cancel{\mathsf{Own}}$ and the contents of the lk pointer (the *ownership bit*) from true to false, and adds the heap h to the resource state. The $\mathsf{rls}_{\mathcal{L}}$ transition corresponds to *locking*, and is dual to $\mathsf{acq}_{\mathcal{L}}$. When locking, the $\mathsf{rls}_{\mathcal{L}}$ transition keeps the auxiliary view $\mathsf{a_s}$ unchanged. Thus, the resource *remembers* the auxiliary view at the point of the last call to lock. Upon unlocking, the $\mathsf{acq}_{\mathcal{L}}$ transition changes this view into $\mathsf{a'_s}$, where $\mathsf{a'_s}$ is some value that is coherent with the acquired heap h, *i. e.*, which makes the resource invariant $\mathcal{I}(\mathsf{a_s} \bullet \mathsf{a_o})\ h$ hold, and thus, the state satisfies the state invariant $\mathsf{Coh}_{\mathcal{L}}$.

We conclude this section with the formalization of the concepts of Rely and Guarantee of a fine-grained resource. This will be useful later on to define the denotational meaning of *FCSL* triples.

Definition 3.25 (Guarantee)**.** Given a resource $\mathcal{U} = (\mathsf{Labs}_\mathcal{U}, \mathsf{Coh}_\mathcal{U}, \mathsf{Int}_\mathcal{U}, \mathsf{Ext}_\mathcal{U})$, the *guarantee* or *self-stepping* relation $\mathsf{G}_\mathcal{U}$ of $\mathcal{U}$ unions its internal and all external transitions of $\mathcal{U}$, and then takes a transitive closure:

$$\mathsf{G}_\mathcal{U} \mathrel{\widehat{=}} \left(\mathsf{Int}_\mathcal{U} \cup \bigcup_{h,(\mathsf{a},\mathsf{r}) \in \mathsf{Ext}_\mathcal{U}} (\mathsf{a}\ h) \cup (\mathsf{r}\ h) \right)^*$$

Definition 3.26 (Rely)**.** Given a resource $\mathcal{U} = (\mathsf{Labs}_\mathcal{U}, \mathsf{Coh}_\mathcal{U}, \mathsf{Int}_\mathcal{U}, \mathsf{Ext}_\mathcal{U})$, the *rely*, *other-* or *environment-stepping* relation $\mathsf{R}_\mathcal{U}$ of $\mathcal{U}$ is defined as the transitive closure of the union of the *transposed* (*cf.* Definition 3.18) internal transition with the iterated union of all transposed external transitions from $\mathsf{Ext}_\mathcal{U}$:

$$\mathsf{R}_\mathcal{U} = \left(\mathsf{Int}_\mathcal{U}^\top \cup \bigcup_{h,(\mathsf{a},\mathsf{r}) \in \mathsf{Ext}_\mathcal{U}} (\mathsf{a}\ h)^\top \cup (\mathsf{r}\ h)^\top \right)^*$$

With the later definition, we can now revisit the notion of *stability*, giving a complete and formal—and obvious!—definition based on the *Rely* of the resource:

Definition 3.27 (Stability)**.** An FCSL assertion P is stable under the environment interference of the fine-grained resource $\mathcal{U}$ *if and only if* for all w, w' such that $(w, w') \in \mathsf{R}_\mathcal{U}$, $w \models P \implies w' \models P$. We denote stable assertions P for a resource $\mathcal{U}$ as $P\ \mathsf{stable}\ P\mathcal{U}$.

3.3.4 Actions of a concurrent resource

Definition 3.28 (Actions)**.** Given a fine-grained resource $\mathcal{U}$ and a type A, we define *actions* by a 4-tuple $a = (\mathcal{U}, A, \mathsf{Pre}_a, \mathsf{Rel}_a)$ where:

1. $\mathcal{U}$ is the fine-grained resource whose internal transitions a respects.
2. A is the type of the action's return value.
3. Pre_a is a predicate on states describing the subset of states of $\mathsf{Coh}_\mathcal{U}$ in which the action could be executed, *i. e.* the *safety* predicate of the action.
4. $\mathsf{Rel}_a : \mathsf{state} \times \mathsf{state} \times A \to \mathsf{Prop}$ is the *stepping* relation, which relates the initial state, the ending state, and the ending result of the action.

It should be noticed that the type A is expected to be *pure*—*i. e.* not a FCSL triple[7]. Moreover, Pre_a and Rel_a are given in a large-footprint style, giving the full heaps and the auxiliary states they accept. The *stepping relation* Rel_a implements the effect of the atomic action.

[7]This is because of the fact that FCSL has first-order states, as it was the case with HTTcc in Chapter 2. We dwell further into this matter in Section 3.6 and in the Conclusions (Chapter 5).

Proposition 3.29 (Proper actions). *Given a resource* $\mathcal{U} = (\mathsf{Labs}_{\mathcal{U}}, \mathsf{Coh}_{\mathcal{U}}, \mathsf{Int}_{\mathcal{U}}, \mathsf{Ext}_{\mathcal{U}})$, *an action* $a = (\mathcal{U}, \mathsf{Pre}_a, \sigma, \mathsf{Rel}_a)$ *A is required to satisfy the following properties:*

$$
\begin{aligned}
\textsc{Coherence} &: w \in \mathsf{Pre}_a \implies w \in \mathcal{W} \\
\textsc{Safety Monotonicity} &: w \triangleright t \in \mathsf{Pre}_a \implies w \triangleleft t \in \mathsf{Pre}_a \\
\textsc{Step Safety} &: (w, w', r) \in \mathsf{Rel}_a \implies w \in \mathsf{Pre}_a \\
\textsc{Internal Stepping} &: (w, w', r) \in \mathsf{Rel}_a \implies (w, w') \in \mathsf{Int}_{\mathcal{U}} \\
\textsc{Framing} &: w \triangleright t \in \mathsf{Pre}_a \implies (w \triangleleft t, w', r) \in \mathsf{Rel}_a \implies \\
&\quad \exists w''.\, w' = w'' \triangleleft t \wedge (w \triangleright t, w'' \triangleright t, v) \in \mathsf{Rel}_a \\
\textsc{Erasure} &: \mathsf{defined}(\lfloor w \rfloor \cup h) \implies \lfloor w \rfloor \cup h = \lfloor w' \rfloor \cup h' \implies \\
&\quad (w, w_1, r) \in \mathsf{Rel}_a \implies (w', w_1', r') \in \mathsf{Rel}_a \implies \\
&\quad r = r' \wedge \lfloor w_1 \rfloor \cup h = \lfloor w_1' \rfloor \cup h' \\
\textsc{Totality} &: \forall w.\, w \in \mathsf{Pre}_a \implies \exists w'\ v.\, (w, w', v) \in \mathsf{Rel}_a
\end{aligned}
$$

The Coherence, Step Safety and Internal Stepping properties are straightforward. The Safety-Monotonicity property states that if the action is safe in a state with a smaller *self* component (because the other component is enlarged by t), the action is also safe if we increase the *self* component by t.

The Framing property says that if a steps in a state with a large *self* component $w \triangleleft t$, but is already safe to step in a state with a smaller *self* component $w \triangleright t$, then the result state and value obtained by stepping in $w \triangleleft t$ can be obtained by stepping in $w \triangleright t$, and moving t afterwards.

The Erasure property shows that the behaviour of the action on the concrete input state obtained after erasing the auxiliary fields and the logical partition, doesn't depend on the erased auxiliary fields and the logical partition. In other words, if the input state have *compatible* erasures (that is, erasures which are sub-heaps of a common heap), then executing the action in the two states results in equal values, and final states that also have compatible erasures. This is a standard property proved in concurrency logics that deal with auxiliary state and code (Owicki & Gries, 1976; Brookes, 2007).

The Totality property shows that an action whose safety predicate is satisfied always produces a result state and value. It doesn't loop forever, and more importantly, it doesn't crash. We will use this property of actions in the semantics of programs to establish that if the program's precondition is satisfied, then all of the approximations in the program's denotation are either done stepping, or can actually make a step (i.e., they make progress).

So far, the actions are defined in a so-called *large footprint* style, *e. g.*, the definition of the action $\mathsf{release}$ in Example 3.4 mentions existentially-quantified heap h explicitly. In order to enable writing various actions in a *small footprint* style, we also enforce the following Locality property:

$$
\textsc{Locality} : w_{\mathsf{o}} = w'_{\mathsf{o}} \implies (w \triangleright t, w' \triangleright t, v) \in \mathsf{Rel}_a \implies (w \triangleleft t, w' \triangleleft t, v) \in \mathsf{Rel}_a
$$

Atomic Actions The FCSL actions presented above act on the whole, decorated, FCSL state. In order to prove that this actions correspond to atomic operations on real state, we need to introduce the notion of *atomic actions*.

Definition 3.30 (Atomic Action)**.** We say that an FCSL action is an *atomic action*, and denote it by the triple $a = (A, \mathsf{Pre}_a, \mathsf{Rel}_a)$, if it is a special kind of action defined *only* over heaps, rather than over *decorated* states.

That is, unlike FCSL states which are decorated with additional information such as auxiliary state and partitioning between *self*, *joint* and *other*. As it was before, A denotes the return type, Pre_a is the safety predicate and Rel_a is the stepping relation, all ranging over heaps. Atomic actions can then, be lifted to any fine-grained resource $\mathcal{U}$ with a non-empty heap foot-print, *i. e.* a resource whose state flattening (*cf.* Definition 3.12) results in a non-empty heap. The basic building block of FCSL atomic operations is given by a generic family of *Read-Modify-Write* (or RMW) operations, which act on a single heap cell (Herlihy & Shavit, 2008, §5.6).

Definition 3.31 (RMW)**.** We define the family of $\mathsf{RMW}^{A,B}\,(x,\ f,\ g)$ atomic *Read-Modify-Write* operations through the following (indexed) atomic actions:

$$\mathsf{RMW}^{A,B}\,(x,\ f,\ g) \mathrel{\widehat{=}} (B, (x \mapsto_A -) \uplus h, (x \mapsto v) \uplus h \rightsquigarrow (x \mapsto f\,v) \uplus h \wedge r = g\,v)$$

These atomic actions atomically replace the current value v of pointer x with $f\,v$ for some pure function f, and return the result according to the function g. A denotes the type of the pre-value of the register x, r denotes the returned value and h here is a logical variable denoting the rest of the heap. Using the RMW action, we define the basic, *primitive*, atomic actions for FCSL:

Definition 3.32 (Primitive atomic actions)**.** We define the basic Read, and Write atomic actions by instantiating RMW as follows:

$$\begin{aligned}\mathsf{Read}^A\ x \quad &\mathrel{\widehat{=}} \mathsf{RMW}^{A,A}\,(x,\ \lambda\,v.\,v,\ \lambda\,v.\,v)\\ \mathsf{Write}\ (x, w) &\mathrel{\widehat{=}} \mathsf{RMW}^{A,()}\,(x,\ \lambda\,\_.\,w,\ \lambda\,\_.\,())\end{aligned}$$

Moreover, we define the *idle* atomic action Skip as:

$$\mathsf{Skip} \mathrel{\widehat{=}} (\mathsf{unit}, h, h \rightsquigarrow h)$$

Notice that we define Skip in this way in order to allow actions with an *empty* heap footprint, *i. e.* actions which implement pure *auxiliary* or *ghost code*, and are introduced for verification purposes[8]. We can use RMW to implement several other classical concurrency primitives:

[8]Otherwise, had we intended to use RMW to define Skip, and therefore making RMW the unique building block of atomic actions, we could have defined it as $\mathsf{RMW}^{(),()}\,(p,\ \lambda\,v.\,v,\ \lambda\,\_.\,())$. However, we would have needed to assert the existence of a *non-null* dedicated pointer p just to comply with *RMW*'s signature—Yikes!—. Of course, we rather not.

Example 3.33 (Atomic Instructions)**.** We instantiate RMW in order to provide FCSL implementations of the following atomic instructions:

$$
\begin{aligned}
\mathsf{CAS}\ (x, a, b) &\mathrel{\widehat{=}} \mathsf{RMW}^{A,B}\ (x,\ \lambda v.\ \mathit{if}\ v = a\ \mathit{then}\ b\ \mathit{else}\ v,\ \lambda v.\ v)\\
\mathsf{Compare\&Set}\ (x, a, b) &\mathrel{\widehat{=}} \mathsf{RMW}^{A,\mathbb{B}}\ (x,\ \lambda v.\ \mathit{if}\ v = a\ \mathit{then}\ b, \mathit{else}\ v,\ \lambda v.\ v = a)\\
\mathsf{Get\&Set}\ (x, w) &\mathrel{\widehat{=}} \mathsf{RMW}^{A,A}\ (x,\ \lambda \_.\ w,\ \lambda v.\ v)\\
\mathsf{Test\&Set}\ x &\mathrel{\widehat{=}} \mathsf{RMW}^{\mathbb{B},\mathbb{B}}\ (x,\ \lambda \_.\ \mathsf{True},\ \lambda v.\ v)\\
\mathsf{F\&I}\ x &\mathrel{\widehat{=}} \mathsf{RMW}^{\mathsf{Int},\mathsf{Int}}\ (x,\ \lambda v.\ v + 1,\ \lambda v.\ v)\\
\mathsf{F\&A}\ (x, a) &\mathrel{\widehat{=}} \mathsf{RMW}^{\mathsf{Int},\mathsf{Int}}\ (x,\ \lambda v.\ v + a,\ \lambda v.\ v)
\end{aligned}
$$

Compare-and-Swap (CAS) atomically reads the pointer x and if the value matches a swaps it to b. In any case, it returns the original value of the pointer x. The variants which returns True if succeeded and False otherwise is usually called *Compare-and-Set*. *Get-and-Set*, *a. k. a.* as *atomic swap* fetches the old value and replaces it with a new one. In the case that the new value is True (or 1), it is usually referred as *Test and Set* (TAS). *Fetch-and-Increment* (F&I) atomically increments the counter x, while returning the pre-value. *Fetch-and-Add* generalizes the latter to perform atomic incrementation by an integer value a passed as a parameter.

It should be noticed that, since FCSL's model of heaps is *size-agnostic*, all of our implementations are indeed abstract and do not implement *real-world* atomic instructions over registers whose size is limited to 64 bits. This could be done, for example, by building a specialized resource $\mathcal{P}_{64}$ whose invariant space $\mathsf{Coh}_{\mathcal{P}_{64}}$ enforces such restrictions on the contents of the heaps.

We connect some of the atomics in Example 3.33 to the corresponding fine-grained resources developed for different FCSL case studies. Test&Set is used to implement the CSL-style resource $\mathcal{L}_{\mathrm{CSL}\ \{\mathsf{lock},lk,\mathcal{I}\}}$'s *lock* atomic action. We use CAS in the Coq implementation of the concurrent resource $\mathcal{J}$ that will be presented in Chapter 4 to implement a snapshot algorithm's "single-writer/single-scanner" *thread-exclusion* requirement. Finally, F&I is used in (Nanevski *et al.* , 2014b) to implement a Ticketed Lock.

Erasure and *Operational actions* An important property of FCSL actions is the fact that they erase to heap-only atomic actions, acting on a unique pointer. We devote the rest of this sub-section to formalize this claim.

Definition 3.34 (Action erasures)**.** Given an action a, the erasures $\lfloor \mathsf{Pre}_a \rfloor$ and $\lfloor \mathsf{Rel}_a \rfloor$ of a's safety predicate and stepping relation are relations on heaps defined as follows:

$$
\begin{aligned}
\lfloor w \rfloor \in \lfloor \mathsf{Pre}_a \rfloor &\iff w \in \mathsf{Pre}_a\\
(\lfloor w \rfloor, \lfloor w' \rfloor, r) \in \lfloor \mathsf{Rel}_a \rfloor &\iff (w, w', r) \in \mathsf{Rel}_a
\end{aligned}
$$

Definition 3.35 (Operational action)**.** We call an action *a operational* if its erasure corresponds to one of the atomic actions, *i. e.*, if there exists an atomic command $b \in \{\mathsf{Skip}, \mathsf{RMW}^{A,B}\ (x, f, g)\}$ such that

$$\lfloor \mathsf{Pre}_a \rfloor \subseteq \mathsf{Pre}_b \wedge \forall h \in \lfloor \mathsf{Pre}_a \rfloor\ h'\ r.\, (h, h', r) \in \lfloor \mathsf{Rel}_a \rfloor \implies (h, h', r) \in \mathsf{Rel}_b$$

In the examples throughout this thesis—and elsewhere—we have only considered operational actions, though the inference rules and the implementation of FCSL in Coq does not currently enforce this requirement, and therefore the fact that they are, indeed, operational actions has been proved by hand. Moreover, when we define an action, or as in the rule ACTION (Figure 3.5) which states that a is an action for the resource $\mathcal{U}$, we will always assume that the action is proven to be *operational* by the definition above and, moreover, it is a *proper* action satisfying the properties described in Proposition 3.29.

3.3.5 Entanglement, Injection and Hiding

Entanglement We give now the formal definitions involved in the cosntruction of *entangled* fine-grained resources.

The state space predicate $\mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}}$ combines the individual states of $\mathcal{U}$ and $\mathcal{V}$ by taking the union of their label sets, while ensuring that the labels contain only non-overlapping heaps.

Definition 3.36 ($\mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}}$)**.** We define the *state space* for the entangled resource $\mathcal{U} \rtimes \mathcal{V}$ as follows:

$$\mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}} \ \widehat{=}\ \{w_{\mathcal{U}} \uplus w_{\mathcal{V}} \mid w_{\mathcal{U}} \in \mathsf{Coh}_{\mathcal{U}} \wedge w_{\mathcal{V}} \in \mathsf{Coh}_{\mathcal{V}} \wedge \mathsf{dom}(\lfloor \mathcal{U} \rfloor) \cap \mathsf{dom}(\lfloor \mathcal{V} \rfloor) = \emptyset\}$$

Thus, the space state $\mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}}$ identifies the point-wise component of each state and, provided that the heap footprint is disjoint, asserts that each component contributed from the "smaller" resources $\mathcal{U}$ and $\mathcal{V}$, satisfy their respective state invariants. In order to define the transition components of $U \rtimes V$, we first need the auxiliary concept of transition interconnection.

Definition 3.37 (Transition Interconnnection)**.** Given two resources $\mathcal{U}$ and $\mathcal{V}$, and transitions $\mathsf{f}_{\mathcal{U}}$ and $\mathsf{g}_{\mathcal{V}}$ over $\mathsf{Coh}_{\mathcal{U}}$ and, respectively, $\mathsf{Coh}_{\mathcal{V}}$, the interconnection $\mathsf{f}_{\mathcal{U}} \bowtie \mathsf{g}_{\mathcal{V}}$ is a transition on $\mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}}$ which behaves as $\mathsf{f}_{\mathcal{U}}$ (resp. $\mathsf{g}_{\mathcal{V}}$) on the part of the states labelled by $\mathcal{U}$, and respectively $\mathcal{V}$:

$$\mathsf{f}_{\mathcal{U}} \bowtie \mathsf{g}_{\mathcal{V}} \ \widehat{=}\ \left\{ (w_1 \uplus w_2, w_1' \uplus w_2') \;\middle|\; \begin{array}{l} (w_1, w_1') \in \mathsf{f}_{\mathcal{U}}\ \wedge (w_2, w_2') \in \mathsf{g}_{\mathcal{V}}\ \wedge \\ w_1 \uplus w_2 \in \mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}} w_1' \uplus w_2' \in \mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}} \end{array} \right\}$$

Definition 3.38 (Entanglement-$\rtimes$)**.** Given two fine-grained resources $\mathcal{U}$ and $\mathcal{V}$. The entanglement $U \rtimes V$ is another fine-grained resource defined as follows:

- The label set of the entangled resource is derived by the point-wise disjoint union of each label set, *i. e.* $\mathcal{L}_{U \rtimes V} = \mathcal{L}_U \cup \mathcal{L}_V$.

- The state invariant $\mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}}$ (Definition 3.36).

- The internal transition of $U \rtimes V$ is defined as follows:

$$\begin{aligned} \mathsf{Int}_{U \rtimes V} \;\widehat{=}\; & (\mathsf{Int}_U \bowtie \mathsf{id}_V) \;\cup (\mathsf{id}_U \bowtie \mathsf{Int}_V) \cup \\ & \bigcup_{h, (a_U, r_U), (a_V, r_V)} (a_U\ h \bowtie r_V\ h) \cup (a_V\ h \bowtie r_U\ h) \end{aligned}$$

  where $(a_U, r_U) \in \mathsf{Ext}_{\mathcal{U}}$, $(a_V, r_V) \in \mathsf{Ext}_{\mathcal{V}}$, and $\mathsf{id}_{\mathcal{U}}$ is the diagonal of $\mathsf{Coh}_{\mathcal{U}}$ (Definition 3.21)

- The external transitions of $\mathcal{U} \rtimes \mathcal{V}$, $\mathsf{Ext}_{\mathcal{U} \rtimes \mathcal{V}}$, are those of $\mathcal{U}$, framed with regard to the labels of $\mathcal{V}$:

$$\mathsf{Ext}_{\mathcal{U} \rtimes \mathcal{V}} = \{(\lambda h.\, (\mathsf{acq}_{\mathcal{U}}\ h) \bowtie \mathsf{id}_{\mathcal{V}}, \lambda h.\, (\mathsf{rls}_{\mathcal{U}}\ h) \bowtie \mathsf{id}_V) \mid (\mathsf{acq}_{\mathcal{U}}, \mathsf{rls}_{\mathcal{U}}) \in \mathsf{Ext}_{\mathcal{U}}\}$$

Then the internal transition $\mathsf{Int}_{\mathcal{U} \rtimes \mathcal{V}}$ prescribes that the entangled resource steps internally whenever $\mathcal{U}$ steps and $\mathcal{V}$ stays *idle*, or when $\mathcal{V}$ steps and $\mathcal{U}$ stays *idle*, or when there exists a heap h which $\mathcal{U}$ and $\mathcal{V}$ exchange ownership over by synchronizing their external transitions. We note that $\mathsf{Ext}_{\mathcal{U} \rtimes \mathcal{V}}$ makes an arbitrary choise to frame on the transitions of $\mathcal{U}$ rather than those of $\mathcal{V}$. Hence, this definition interconnects the external transitions of $\mathcal{U}$ and $\mathcal{V}$, but it keeps those of $\mathcal{U}$ "open" in the entanglement, while it "shuts down" those of $\mathcal{V}$. The notation $\mathcal{U} \rtimes \mathcal{V}$ is meant to symbolize this asymmetry. The asymmetry is important for our example of encoding CSL resources, as it enables us to iterate the (non-associative) addition of new resources as $((\mathcal{P} \rtimes \mathcal{L}_{\mathsf{lock}_1, lk_1, \mathcal{I}_1}) \mathcal{L}_{\mathsf{lock}_2, lk_2, \mathcal{I}_2}) \rtimes \cdots$ while keeping the external transitions of $\mathcal{P}$ open in order to possibly exchange heaps with new resources.

Clearly, this choice entails that there are several ways to interconnect transitions of two resources and select which transitions to keep open. In our implementation (FCSL, 2017), we have identified several operators implementing common interconnection choices, and proved a number of equations and properties about them (*e. g.*, all of them validate an instance of the Inject rule).

Injection We define the predicate $\mathsf{injects}\ \mathcal{V}\ \mathcal{U}$, which intuitively means that the "larger" fine-grained resource $\mathcal{U}$ can be considered as an entanglement of the "smaller" fine-grained resource $\mathcal{V}$ with some additional fine-grained resource $\mathcal{W}$.

Definition 3.39 (Injection Predicate)**.** We say that $\mathcal{V}$ *injects* into $\mathcal{U}$, $\mathsf{injects}\ \mathcal{V}\ \mathcal{U}$, *if and only if* there exists $\mathcal{W}$, such that the following three statements hold:

$$
\begin{aligned}
&(1)\quad w \in \mathsf{Coh}_{\mathcal{U}} \iff w = w_1 \uplus w_2 \wedge w_1 \in \mathsf{Coh}_{\mathcal{V}} \wedge w_2 \in \mathsf{Coh}_{\mathcal{W}}\\
&(2)\quad \forall\, w_1\ w_2\ w.\ \ w_1 \uplus w \in \mathsf{Coh}_{\mathcal{U}} \wedge w_2 \uplus w \in \mathsf{Coh}_{\mathcal{U}} \wedge (w_1, w_2) \in \mathsf{Int}_V \implies\\
&\qquad\qquad (w_1 \uplus w, w_2 \uplus w) \in \mathsf{Int}_U\\
&(3)\quad \forall\, w_1\ w_1'\ w_2\ w_2'.\ \ s_1 \in \mathsf{Coh}_{\mathcal{V}} \wedge s_2 \in \mathsf{Coh}_{\mathcal{V}} \wedge (s_1 \uplus s_1', s_2 \uplus s_2') \in \mathsf{G}_{\mathcal{U}} \implies\\
&\qquad\qquad (s_1, s_2) \in \mathsf{G}_{\mathcal{V}} \wedge (s_1', s_2') \in \mathsf{G}_{\mathcal{W}}
\end{aligned}
$$

where $\mathsf{G}_V, \mathsf{G}_W$ are, respectively, the *guarantee* relations the fine-grained resources $\mathcal{V}$, and, respectively, W (Definition 3.25).

The first requirement ensures that the states of $\mathcal{U}$ can be constructed as Cartesian products of states of $\mathcal{V}$ and $\mathcal{W}$. The second requirement ensures that internal transitions of the smaller fine-grained resource $\mathcal{V}$ do not break the coherence of it enclosing fine-grained resource $\mathcal{U}$. The third requirement ensures that the transitions within $\mathcal{U}$ can be always uniquely split to the transitions done by $\mathcal{V}$ and by the "addition" of $\mathcal{W}$. Notably, the first requirement ensures that if $\mathsf{injects}\ V\ U$ holds, there always exists a "smaller" state w_V, such that $w_V \in \mathsf{Coh}_{\mathcal{V}}$ for any $w_U \in \mathsf{Coh}_{\mathcal{U}}$.

We will also introduce another auxiliary definition in order to have a handle to the existential fine-grained resource $\mathcal{W}$ in Definition 3.39.

Definition 3.40. We will write $U = V \bowtie W$, when $\mathsf{injects}\ V\ U$ and W is an additional fine-grained resource from Definition 3.39.

The notation $\bowtie$ hints that the entanglement operators $\rtimes$, $\ltimes$, *etc.*can be seen as particular cases of $\bowtie$. We close this paragraph with some usufeul properites of the entanglement operator, which we have proven in our Coq files.

Lemma 3.41 (World Expansion)**.** $\mathsf{injects}\ V\ (V \rtimes W)$ *for every* $\mathcal{V}$ *and* $\mathcal{W}$ *with disjoint label sets.*

Lemma 3.42 (Exchange law for $\rtimes$)**.** $(\mathcal{U} \rtimes \mathcal{V}) \rtimes \mathcal{W} = (\mathcal{U} \rtimes \mathcal{W}) \rtimes \mathcal{V}$

Example 3.43 (Empty Resource)**.** We define of the *empty* resource $\mathcal{E}$ as

$$\mathcal{E} = (\emptyset, \mathsf{Coh}_{\mathcal{E}}, \mathsf{id}, \emptyset)$$

where, $w \in \mathsf{Coh}_{\mathcal{E}} \iff w = \emptyset$, *i. e.* w is the empty state without any labels.

Lemma 3.44 ($\rtimes$–Right Unit)**.** *The* empty *resource* $\mathcal{E}$ *is the right unit of the* $\rtimes$ *operator.*

Refinement We next define the refinement relation on fine-grained resources. The fine-grained resource $\mathcal{U}$ is refined into $\mathcal{V}$, if we can, intuitively, elaborate the states of $\mathcal{U}$ into those of $\mathcal{V}$. In other words, if $\mathcal{V}$'s states can be seen to contain the states of $\mathcal{U}$, plus some other additional state. Abstractly, we capture the dependence by positing an elaboration predicate $\widehat{\Phi}$ between the state spaces of $\mathcal{U}$ and $\mathcal{V}$, with a number of properties shown below. Additionally, $\widehat{\Phi}$ takes a value $g : G$ of user-specified type, which is an abstraction of the mentioned additional state (which we call a *seed*). Thus, we read $\widehat{\Phi}(g)(w, w')$ to say that g elaborates w into w' (via $\widehat{\Phi}$).

Definition 3.45 (Elaboration predicate)**.** Given a type G of abstract states and a function $\widehat{\Phi} : G \to (\mathsf{state}\,\mathcal{U}, \mathsf{state}\,\mathcal{V}) \to \mathsf{Prop}$, $\mathsf{refines}\ \mathcal{U}\ \mathcal{V}\ \widehat{\Phi}$ *if and only if* the following statements hold:

1. $\widehat{\Phi}(g)(w, w_1) \land \widehat{\Phi}(g)(w, w_2) \implies w_1 = w_2$
2. $\widehat{\Phi}(g_1)(w_1, w) \land \widehat{\Phi}(g_2)(w_2, w) \implies g_1 = g_2 \land w_1 = w_2$
3. $\widehat{\Phi}(g)(w, w') \implies w \in \mathsf{Coh}_{\mathcal{U}} \land w' \in \mathsf{Coh}_{\mathcal{V}}$
4. $\widehat{\Phi}(g)(w_1, w_1') \land (w_1, w_2) \in \mathsf{R}_{\mathcal{U}} \implies \exists w_2', \widehat{\Phi}(g)(w_2, w_2') \land (w_1', w_2') \in \mathsf{R}_{\mathcal{V}}$
5. $\widehat{\Phi}(g_1)(w_1, w_1') \land (w_1', w_2') \in \mathsf{Int}_{\mathcal{V}} \implies \exists g_2\ w_2.\, \widehat{\Phi}(g_2)(w_2, w_2') \land (w_1, w_2) \in \mathsf{Int}_{\mathcal{U}}$
6. $\widehat{\Phi}(g_1)(w_1, w_1') \land \widehat{\Phi}(g_2)(w_2, w_2') \implies (w_{1\,\circ} = w_{1\,\circ} \iff w_{1\,\circ}' = w_{2\,\circ}')$
7. $\forall p. \exists q.$

$$\begin{array}{rl} \forall g\ w\ w'. & \widehat{\Phi}(g)(w \triangleright p, w') \implies \left(\exists w_1', w' = w_1' \triangleright q \land \widehat{\Phi}(g)(w \triangleleft p, w_1' \triangleleft q)\right) \\ \land \forall g\ w\ w'. & \widehat{\Phi}(g)(w, w' \triangleleft q) \implies \left(\exists w_1, w = w_1 \triangleleft p \land \widehat{\Phi}(g)(w_1 \triangleright p, w' \triangleright q)\right) \end{array}$$

8. $\widehat{\Phi}(g)(w, w') \implies \lfloor w \rfloor = \lfloor w' \rfloor$

If the properties in Definition 3.45 are satisfied, we call $\widehat{\Phi}$ and elaboration predicate for $\mathcal{U}$ and $\mathcal{V}$. In Definition 3.45, (1) states that $\widehat{\Phi}$ uniquely determines refined states (however, some states in $\mathcal{U}$ need not refine into anything in $\mathcal{V}$, *i. e.*, $\widehat{\Phi}$ is a partial function); (2) states that the $\widehat{\Phi}$ is injective: abstracting back from a refinement is unique. Thus, (1) and (2) state that $\widehat{\Phi}$ is a partial bijection. (3) ensures that the elaboration maps well-formed states to well-formed states; (4) specifies that environment steps in the "coarse" world do not change introduced abstractions in the "fine" world, as environment steps in the coarse world can't see the refinement; (5) states that "abstracting back" preserves internal transitions. The properties (4) and (5) can be undesrtood also as *simulation properties*[9]. In (4), a rely transitions of $\mathcal{U}$ can always be matched by a rely transition of $\mathcal{V}$. In

(5), an internal transition of $\mathcal{V}$ may always be matched by an internal transition of $\mathcal{U}$. (6) postulates that refinement realigns *self* and *joint* parts, but it does not modify the contents of *other* by adding or removing (*i. e. zigging* or *zagging*) to its component; (7) is a version of framing, namely, every every extension to *other* of a coarse state is is uniquely refined into an addition to fine state; finally, (8) ensures that refinement only deals with abstract parts of the state and does not change the heap.

Hiding The abstraction function Φ is a user-specified annotation on the **hide** command. It maps values $g : G$ (where G is also user specified) to assertions, that is, predicates over states (equivalently, sets of states) of a concurroid $\mathcal{V}$. For the soundness of the hiding rule, Φ is required to satisfy the following properties.

Proposition 3.46 (Φ functions for Hide)**.** *The Φ function in the* Hide *rule satisfies the following properties, where s, w denote states of $\mathcal{W}$ and $f, g : G$, where G is a user specified type.*

$$
\begin{array}{lcl}
\textsc{Coherence} & : & w \in \Phi(g) \implies w \in \mathsf{Coh}_{\mathcal{W}} \\
\textsc{Injectivity} & : & w \in \Phi(f) \implies w \in \Phi(g) \implies f = g \\
\textsc{Surjectivity} & : & s \in \Phi(f) \implies w \in \mathsf{Coh}_{\mathcal{W}} \implies s_{\mathsf{o}} = w_{\mathsf{o}} \implies \exists g.\, w_2 \in \Phi(g) \\
\textsc{Guarantee} & : & s \in \Phi(f) \implies w \in \Phi(g) \implies s_{\mathsf{o}} = w_{\mathsf{o}} \\
\textsc{Precision} & : & s \in \Phi(g) \implies w \in \Phi(g) \implies \lfloor s \rfloor \cup h = \lfloor w \rfloor \cup h' \implies s = w
\end{array}
$$

The Coherence and Injectivity assertions are immediate from their definitions. The property Surjectivity states that for every state w of the concurroid $\mathcal{W}$ one can find an image g, under the condition that the *other* component of w is well-formed according to Φ (In most cases, this typically ammounts to the fact that *other* component is equal to the unit of the PCM-map monoid for $\mathcal{W}$). Guarantee formalizes that environment of **hide** can't interference on $\mathcal{V}$, as $\mathcal{V}$ is installed locally. Thus, whatever the environment does, it can not influence the *other* component of the states w described by Φ. Finally, Precision is a technical property common to separation-style logics, though here it has a somewhat different flavor. Thus, precision ensures that for every value g, $\Phi(g)$ precisely describes the underlying heaps of its circumscribed states; that is, each state $\Phi(g)$ is uniquely determined by its heap erasure.

Lemma 3.47 (Hiding and refinement)**.** *Given two resources $\mathcal{U}$ and $\mathcal{V}$, and let Φ be an abstraction function from a* Hide *command,* i. e. *Φ satisfies the properties*

[9]Informally, at least. The point is that they reek of commutativity or naturality and they beckon us to engage in a deeper study into the mathematics of FCSL's concurrent resources.

about the resource $\mathcal{V}$ *from Proposition 3.46. Let* $\widehat{\Phi}$ *be constructed as follows.*

$$\begin{aligned}\widehat{\Phi}(g)(w,w') \iff \exists\ h\ w_1\ w_2, \quad & w = (\mathit{priv} \mapsto (h \uplus \lfloor w_1 \rfloor)) \uplus w_2 \\ & \wedge\ w' = (\mathit{priv} \mapsto h) \uplus w_1 \uplus w_2 \\ & \wedge\ w_1 \in \Phi(g) \\ & \wedge\ w \in \mathsf{Coh}_{(P \rtimes \mathcal{U})} \wedge w' \in \mathsf{Coh}_{(P \rtimes \mathcal{U} \rtimes \mathcal{V})}\end{aligned}$$

Then $\widehat{\Phi}$ *is a* valid elaboration predicate*, that is* $\mathsf{refines}\ (P \rtimes \mathcal{U})\ (P \rtimes \mathcal{U} \rtimes \mathcal{V})\ \widehat{\Phi}$.

Finally we pack this Lemma with the properties required for Φ as a property, which will constitute the proof obligation required in order to assert a valid resource introduction with the HIDE rule:

Definition 3.48 (hides predicate)**.** We say that an abstraction function Φ, which satisifies the properties from Proposition3.46, hides $\mathcal{V}$ from $\mathcal{U}$, *i. e.* $\mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi$, *if and only if Phi* defines a valid elaboration predicate following Lemma 3.47. In short,

$$\mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi \ \widehat{=}\ \mathsf{refines}\ (P \rtimes \mathcal{U})\ (P \rtimes \mathcal{U} \rtimes \mathcal{V})\ \widehat{\Phi}$$

3.4 Semantics of FCSL

Akin to the development of HTTcc presented in Chapter 2, we implement FCSL as a *shallow embedding* in Coq (Bertot & Castéran, 2004; The Coq Development Team, 2016). This has several important benefits, as it allows us to program and prove directly *with the semantic objects*, thus immediately lifting FCSL to a full-blown programming language and verification framework with higher-order functions, abstract types, abstract predicates, and a module system. We also gain a powerful dependently-typed λ-calculus, which we use to formalize all the semantic definitions and the meta-theory, including the assertion language and we gain support for higher-order procedures, inductive types, modules, and other features of the Coq type-system for free. Last but not least, we can directly re-use Coq and Ssreflect libraries—*e. g.* facts about lists, different number theories, permutations, finite-functions *etc.*— in the verification of FCSL client programs without having to re-implement or re-prove them.

The structure of this section follows the bottom up process of the shallow embedding of FCSL in Coq. First, we define action trees and their semantics[10]with regard to instrumented FCSL states. We will use action trees to define the type of program models $\mathsf{Prog}_{\mathcal{U}}\ A$. Then, we will relate such program models to the high-level transitions of a concurrent resource by an always state-transformer predicate that ensures a tree is resilient to any amount of interference.

As a first step into defining the semantic of Hoare triples $\{P\}\,A\,\{Q\}@\,\mathcal{U}$, we translate those types to *Hoare-types* $\mathsf{ST}_{\mathcal{U}}\ s$ akin to those in Chapter 2. The denotational semantics of FCSL Hoare-types will be given by a *complete lattice*

of action trees that are always-safe to run from any initial configuration that satisfies precondition p and if they terminate produce a final configuration that satisfies postcondition q (under possible interference from programs respecting the transitions of $\mathcal{U}$). The complete lattice structure makes the semantic domain suitable for modelling recursion.

Finally, we establish the *soundness* of FCSL by showing that the denotations of FCSL commands satisfy an appropriate instance of the always predicate (*i. e.*, adhere to the *progress-and-preservation* property).

3.4.1 Action trees and their operational semantics

Action Trees The semantic model for FCSL programs largely relies on *action trees* (Ley-Wild & Nanevski, 2013), which implement finite, partial approximations of the behaviour of FCSL commands. Action trees generalize Brookes' *action traces* (Brookes, 2007). Action trees have richer structure than action traces: while the latter are sequences of actions and their results, action trees contains an action and a *continuation* which itself is a tree parametrized with regard to the output of the action. Concretely, action trees are implemented by the following inductive data-type definition.

Definition 3.49 (Action trees)**.** The type $\mathsf{tree}\ \mathcal{U}\ A$ of A-returning action trees is defined by an inductive definition, as follows:

$$
\begin{aligned}
&\mathsf{tree}\ \mathcal{U}\ A\ \ \widehat{=}\\
&\quad \mid \mathsf{Unfinished}\\
&\quad \mid \mathsf{Ret}\ (v : A)\\
&\quad \mid \mathsf{Act}\ (a : \mathsf{action}\ \mathcal{U}\ A)\\
&\quad \mid \mathsf{Seq}\ (t : \mathsf{tree}\ \mathcal{U}\ B)\ (k : B \to \mathsf{tree}\ \mathcal{U}\ A\\
&\quad \mid \mathsf{Par}\ (t_1 : \mathsf{tree}\ \mathcal{U}\ B_1)\ (t_2 : \mathsf{tree}\ \mathcal{U}\ B_2)(k : B_1 \times B_2 \to \mathsf{tree}\ \mathcal{U}\ A)\\
&\quad \mid \mathsf{Inject}\ (t : \mathsf{tree}\ \mathcal{V}\ A)\\
&\quad \mid \mathsf{Hide}_{\Phi,g}\ (t : \mathsf{tree}\ \mathcal{V}\ A)
\end{aligned}
$$

We briefly explain the constructors in Definition 3.49. The Unfinished tree indicates an incomplete approximation to divergent computation. Ret v is a terminal computation that returns value $v : A$. The constructor Act takes as a parameter an A-returning action $a = (\mathcal{U}, A, \mathsf{Pre}_a, \mathsf{Rel}_a)$, defined for the resource $\mathcal{U}$. Seq $t\ k$ sequentially composes a B-returning tree t with a continuation k that takes t's return value and generates the rest of the approximation. Par $t_1\ t_2\ k$

[10] FCSL being a *shallow embedding*, this is not an operational semantics *per se*, as the latter is given by the host language. It would be perhaps more appropriate then to speak of an operational semantics *on the model*, as it will be made more clear by the denotation of FCSL triples in Definition **??**.

is the parallel composition of trees t_1 and t_2, and a continuation k that takes the pair of their results when they join. The underlying type theory of Coq's support for iterated inductive definition permits the recursive occurrences of tree to be *nonuniform* (*e. g.*, tree B_i in Par) and *nested* (*e. g.*, the *positive* occurrence of tree A in the continuation). Moreover, since the function space $\rightarrow$ includes case-analysis, the continuation may branch upon the argument, and thus capture the pure computation of conditionals. This closely corresponds to the operational intuition and leads to a straightforward denotational semantics: there is no need to implement conditionals and we get, for instance, case analysis on *pure* inductive types for free.

The Inject constructor embeds a tree t : tree $\mathcal{V}$ A (of a *different*, *i. e.*, smaller, resource $\mathcal{V}$) for the underlying computation and generates a tree in the resource $\mathcal{U}$. Needless to say, $\mathcal{V}$ and $\mathcal{U}$ cannot not be arbitrary fine-grained resources, as there will be a requirement that $\mathcal{V}$ *injects* into $\mathcal{U}$.

In a similar way, the Hide constructor embeds a *generalized refinement function* Φ, an abstract state $g : \mathcal{V}$ and a tree t : tree $\mathcal{V}$ A for the underlying computation and generates a tree in the resource $\mathcal{U}$. Again, $\mathcal{U}$ and $\mathcal{V}$ are not arbitrary but it has to be possible to *refine* $\mathcal{V}$ into $\mathcal{U}$.

Finally, it should be noticed that in the implementation of FCSL in Coq, the proofs about the correctness of the injection from $\mathcal{V}$ injects into $\mathcal{U}$ has to be an annotation provided to the Inject constructor. The same applies to Hide and the proof of the corresponding elaboration. We elide the annotations here for the sake of a cleaner presentation.

Operational semantics of action trees The judgement for small-step operational semantics of action trees has the form $w; t \xrightarrow{\pi} w', t'$. It operates on program states w and paths π to step the tree t from initial state w to a reduced tree t' in ending state w'. Intuitively, the path π identifies the position in the tree to be reduced as follows::

Definition 3.50 (Paths)**.** The inductive type path defines a *path*, *i. e.* the reduction on a tree that can be mandated by a *scheduler* (a list of *paths*):

$$
\begin{array}{rcl}
\mathsf{path} & \widehat{=} & \mathsf{ChoiceAct} \mid \mathsf{SeqRet} \mid \mathsf{SeqStep}\ (\pi{:}\mathsf{path}) \\
& & \mid \mathsf{ParRet} \mid \mathsf{ParL}\ (\pi{:}\mathsf{path}) \mid \mathsf{ParR}\ (\pi{:}\mathsf{path}) \\
& & \mid \mathsf{HideStep}\ (\pi{:}\mathsf{path}) \mid \mathsf{HideRet} \mid \mathsf{InjStep}\ (\pi{:}\mathsf{path}) \\
& & \mid \mathsf{InjRet}
\end{array}
$$

Figure 3.8 presents the rules for the operational steps a pair $w; t$ can perform. Stepping is undefined for the Unfinished and Ret trees. Given a tree denoting a step by an atomic action Act a, the rule ActStep steps to a Ret v tree which feeds the return value of the action. The side-condition of the rule checks that the initial state w satisfies the internal precondition of the action, $w \in \mathsf{Pre}_a$.

$$\frac{(w, w', v) \in \mathsf{Rel}_a}{w; \mathsf{Act}\ a \xrightarrow{\mathsf{ChoiceAct}} w'; \mathsf{Ret}\ v}\ \textsc{ActStep}$$

$$\frac{}{w; \mathsf{Par}\ (\mathsf{Ret}\ v_1)\ (\mathsf{Ret}\ v_2)\ k \xrightarrow{\mathsf{ParRet}} w; k\ (v_1, v_2)}\ \textsc{ParKont}$$

$$\frac{w; t_1 \xrightarrow{\pi} w'; t_1'}{w; \mathsf{Par}\ t_1\ t_2\ k \xrightarrow{\mathsf{ParL}\ \pi} w'; \mathsf{Par}\ t_1'\ t_2\ k}\ \textsc{ParLeft}$$

$$\frac{w; t_2 \xrightarrow{\pi} w'; t_2'}{w; \mathsf{Par}\ t_1\ t_2\ k \xrightarrow{\mathsf{ParR}\ \pi} w'; \mathsf{Par}\ t_1\ t_2'\ k}\ \textsc{ParRight}$$

$$\frac{}{w; \mathsf{Seq}\ (\mathsf{Ret}\ v)\ k \xrightarrow{\mathsf{SeqRet}} w; (k\ v)}\ \textsc{SeqKont}$$

$$\frac{w; t_1 \xrightarrow{\pi} w'; t_2}{w; \mathsf{Seq}\ t_1\ k \xrightarrow{\mathsf{SeqStep}\ \pi} w'; \mathsf{Seq}\ t_2\ k}\ \textsc{SeqStep}$$

$$\frac{}{w; \mathsf{Inject}\ (\mathsf{Ret}\ v) \xrightarrow{\mathsf{InjRet}} w; \mathsf{Ret}\ v}\ \textsc{InjRet}$$

$$\frac{\mathsf{injects}\ \mathcal{V}\ \mathcal{U} \quad w_U = w_V \uplus w_0 \quad w_V; t_1 \xrightarrow{\pi} w_V'; t_2 \quad w_U' = w_V' \uplus w_0}{w_U; \mathsf{Inject}\ t_1 \xrightarrow{\mathsf{InjStep}\ \pi} w_U'; \mathsf{Inject}\ t_2}\ \textsc{InjStep}$$

$$\frac{}{w; \mathsf{Hide}\ \Phi\ g\ (\mathsf{Ret}\ v) \xrightarrow{\mathsf{HideRet}} w; \mathsf{Ret}\ v}\ \textsc{HideRet}$$

$$\frac{\mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi \quad \widehat{\Phi}(g)(w_U, w_V) \quad w_V; t_1 \xrightarrow{\pi} w_V'; t_2 \quad \widehat{\Phi}(g')(w_U', w_V')}{w_U; \mathsf{Hide}\ \Phi\ g\ t_1 \xrightarrow{\mathsf{HideStep}\ \pi} w_U'; \mathsf{Hide}\ \Phi\ g'\ t_2}\ \textsc{HideStep}$$

Figure 3.8: Small-step operational semantics on *action trees*: The judgement $w; t \xrightarrow{\pi} w'; t'$ describes how a configuration $w; t$ evolves into $w'; t'$ when the path π is scheduled. The predicate $\mathsf{injects}\ \mathcal{V}\ \mathcal{U}$ (Definition 3.39) asserts that is possible to coerce $\mathcal{V}$ to $\mathcal{U}$. The predicate $\mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi$ (Definition 3.48) asserts that Φ elaborates $\mathcal{V}$, hiding it from $\mathcal{U}$. $\widehat{\Phi}$ is a *valid elaboration predicate* constructed from Φ(*cf.* Lemma 3.47).

The rules for parallel composition ParLeft, ParRight and ParKont; and those for sequential composition SeqStep and SeqKont, treat the respective constructors Par and Seq as execution contexts. For a tree which is a parallel composition Par t_1 t_2 k, the path π identifies on which child to perform the appropriate reduction, by the appropriate choice between ParLeft and ParRight rules. This entails that the operational semantic on action trees is deterministic. The rule ParKont captures the operational intuition that the parent's execution is suspended until both children have terminated.

The rules for injection and hiding treat the constructors as *resource contexts*, delimiting the framing on the underlying resources. For an Inject t tree in the InjStep rule, the reduction of the tree t_1 is performed for a smaller state w_V instead of the large state w_U, and the resulted state w'_V is "plugged" back using the addition w_0 in order to obtain the resulting large state w'_U. Moreover, we are required to provide evidence that $\mathcal{V}$ can be lifted into a larger resource $\mathcal{U}$, by the assertion injects $\mathcal{V}$ $\mathcal{U}$ (Definition 3.39). When the Inject-guarded executions reach a Ret v tree, the rules InjRet rule discards the resource-framed context.

The case for Hide trees is similar: the coarse-grained state w_U and the fine-grained state w_V should be related by the elaboration functions Φ. The same applies for the end states. Note that the modifications of the changes to the abstract values g are recorded into g'. We are also required to proof that Φ is a proper elaboration function for these resources, as denoted by hides $\mathcal{U}$ $\mathcal{V}$ Φ (Definition 3.48). Finally, the rule HideRet discards the current layer of resource-scoping context.

The rules in 3.8 may fail to make a configuration $w; t$ step for two different reasons. The first reason is that the scheduled path π does not actually determine an action or a redex in the tree t. For example, we may have $t =$ Unfinished and $\pi =$ ParR. We consider such paths that do not determine an action or a redex in a tree ill-formed. The second reason arises when π is indeed well-formed. Here, the constructors of the path uniquely determine a rule of the operational semantics that should be applied to step the tree. However, it cannot be applied because some side-condition is not satisfied by the current state. We distinguish the two cases by the following definitions:

Definition 3.51 (Good paths)**.** Let t : tree $\mathcal{U}$ A and π : path. Then the predicate

$\mathsf{good}\ t\ \pi$ is defined as follows:

good	(Act a)	ChoiceAct	$\widehat{=}$	true
good	(Seq (Ret v) _)	SeqRet	$\widehat{=}$	true
good	(Seq t _)	SeqRet π	$\widehat{=}$	good $t\ \pi$
good	(Par (Ret _) (Ret _) _)	ParRet	$\widehat{=}$	true
good	(Par t_1 t_2 _)	ParL π	$\widehat{=}$	good $t_1\ \pi$
good	(Par t_1 t_2 _)	ParR π	$\widehat{=}$	good $t_2\ \pi$
good	(Inject (Ret _))	InjRet	$\widehat{=}$	true
good	(Inject t)	InjStep π	$\widehat{=}$	good $t\ \pi$
good	(Hide _ _ (Ret _))	HideRet	$\widehat{=}$	true
good	(Hide _ _ t)	HideStep π	$\widehat{=}$	good $t\ \pi$
good	t	π	$\widehat{=}$	false otherwise

Definition 3.52 (Safe path for an action tree)**.** Let t : tree $\mathcal{U}\ A$, π : path, and state $w \in \mathsf{Coh}_{\mathcal{U}}$. We say it is *safe* to step a configuration $w; t$ following the path π, *i. e.* $\mathsf{safe}\ t\ \pi\ w$ if and only if:

safe	(Act a)	ChoiceAct	w	$\widehat{=}$	$w \in \mathsf{Pre}_a$
safe	(Seq (Ret v) _)	SeqRet	w	$\widehat{=}$	true
safe	(Seq t _)	SeqStep π	w	$\widehat{=}$	safe $t\ \pi\ w$
safe	(Par (Ret _) (Ret _) _)	ParRet	w	$\widehat{=}$	true
safe	(Par t_1 t_2 _)	ParL π	w	$\widehat{=}$	safe $t_1\ \pi\ w$
safe	(Par t_1 t_2 _)	ParR π	w	$\widehat{=}$	safe $t_2\ \pi\ w$
safe	(Inject (Ret _))	InjRet	w	$\widehat{=}$	true
safe	(Inject (t:tree $\mathcal{V}\ A$))	InjStep π	w	$\widehat{=}$	injects $\mathcal{V}\ \mathcal{U}\ \wedge w = w_V \cup w_0$ $\wedge$ safe $t\ \pi\ w_V$
safe	(Hide _ _ (Ret _))	HideRet	w	$\widehat{=}$	true
safe	(Hide Φ g (t:tree $\mathcal{V}\ A$))	HideStep π	w	$\widehat{=}$	hides $\mathcal{U}\ \mathcal{V}\ \Phi$ $\Phi(g)(w, w_V)\ \wedge$ safe $t\ \pi\ w_V$
safe	t	π	w	$\widehat{=}$	true

The definition of safe traverses the path and the tree, and in those cases when the tree and the path match, collects the side-conditions on the *initial states* of the operational semantics rules. Notice that Ret and Unfinished trees do not step, but they are however always safe. In the Act a case, it will be always safe to step in a state w which satisfies the internal precondition of the action, $w \in \mathsf{Pre}_a$. We relate this two notions through a *progress* property for the operational semantics on action trees.

Lemma 3.53 (Progress on action trees)**.** *Given a resource $\mathcal{U}$, a state $w \in \mathsf{Coh}_U$, an action tree t : $\mathsf{tree}\ \mathcal{U}\ A$, and a path π, if $\mathsf{safe}\ t\ \pi\ w$, then either of the following holds:*

- $\neg\ \mathsf{good}\ t\ \pi$;
- $\exists w'\ t',\ w; t \xrightarrow{\pi} w'; t'$.

The lemma above shows that if we can match the tree t with a path π and thus manage to pick a reductions (*i. e.*, $\mathsf{good}\ t\ \pi$), then the safety predicate implies the side conditions of the inference rule determined by π. This lemma allows for safety and the operational stepping to be defined independently of each other. In turn, defining safety independently from stepping will enable us to develop a semantics in the style of Milner whereby we only give meaning to programs that are well-proved in FCSL, analogously to how Milner-style only give semantic meaning to well-typed programs. We close this sub-section with the statement of several properties about trees and their operational semantics.

Lemma 3.54 (Properties of $\xrightarrow{\pi}$)**.** *The stepping relation $\xrightarrow{\pi}$ satisfies the following properties:*

Guarantee	:	*if* $w; t \xrightarrow{\pi} w'; t'$ *then* $w_\mathsf{o} = w'_\mathsf{o}$
Coherence	:	*if* $w; t \xrightarrow{\pi} w'; t'$ *then if* $w \in \mathsf{Coh}_\mathcal{U}$ *then* $w' \in \mathsf{Coh}_\mathcal{U}$
Stability	:	*if* $w; t \xrightarrow{\pi} w'; t'$ *then* $(w, w') \in \mathsf{R}_\mathcal{U}$
Determinism	:	*if* $w; t \xrightarrow{\pi} w'; t'$ *and* $w; t \xrightarrow{\pi} w''; t''$ *then* $t' = t'' \ \wedge\ w' = w''$
Locality	:	*if* $(w \triangleleft p); t \xrightarrow{\pi} w'; t'$ *then there exists* w''*such that* $w' = w'' \triangleleft p$ *and* $(w \triangleright p); t \xrightarrow{\pi} (w'' \triangleright p); t'$
SafetyMono	:	*if* $\mathsf{safe}\ t\ \pi\ (w \triangleleft p)$*then* $\mathsf{safe}\ t\ \pi\ (w \triangleright p)$
Framability	:	*if* $\mathsf{safe}\ t\ \pi\ (w \triangleleft p)$ *and* $(w \triangleright p); t \xrightarrow{\pi} w'; t'$ *then there exists* w'' *such thatw'* $= w'' \triangleright p$ *and* $(w \triangleright p); t \xrightarrow{\pi} (w'' \triangleright p); t'$

Most properties above lift properties of transitions and actions, already presented in Section 3.3, to action trees. The Framability property is direct consequence of the Locality property of $\xrightarrow{\pi}$, and it is the FCSL equivalent of a customary property from abstract separation logic (Calcagno *et al.* , 2007).

Denotational model of FCSL commands The denotational model of FCSL programs is given in terms of sets of action trees. Given a FCSL type ascription $e : \{P\}\ A\ \{Q\}@\ \mathcal{U}$, the denotational model of e, $[\![e]\!] : \mathsf{Prog}_\mathcal{U}\ A$, is a *set* T of trees of increasing precision including the Unfinished tree, which is the coarsest possible approximation of any program. The set may be infinite, as some programs may only be reached in the limit, after infinitely many finite approximations.

$$
\begin{aligned}
[\![\mathbf{return}\ v]\!] &\ \widehat{=}\ \{\mathsf{Unfinished}, \mathsf{Ret}\ v\} \\
[\![\mathbf{bind}\ c_1\ c_2]\!] &\ \widehat{=}\ \{\mathsf{Unfinished}\} \cup \{\mathsf{Seq}\ t_1\ k \mid t_1 \in [\![c_1]\!], \forall x.\, k\ x \in [\![c_2]\!]\} \\
[\![c_1 \parallel c_2]\!] &\ \widehat{=}\ \{\mathsf{Unfinished}\} \cup \{\mathsf{Par}\ t_1\ t_2\ (\lambda v.\mathsf{Ret}\ v) \mid t_1 \in [\![c_1]\!], t_2 \in [\![c_2]\!]\} \\
[\![\mathbf{act}\ a]\!] &\ \widehat{=}\ \{\mathsf{Unfinished}, \mathsf{Act}\ a\} \\
[\![\mathbf{inject}\ c]\!] &\ \widehat{=}\ \{\mathsf{Unfinished}\} \cup \{\mathsf{Inject}\ t \mid t \in [\![c]\!]\} \\
[\![\mathbf{hide}_{\Phi,g}\ c]\!] &\ \widehat{=}\ \{\mathsf{Unfinished}\} \cup \left\{\mathsf{Hide}\ \widehat{\Phi}\ g\ t \mid t \in [\![c]\!]\right\} \\
[\![\mathbf{fix}\ f]\!] &\ \widehat{=}\ \mathsf{lfp}\ (\lambda x.\ [\![f\ x]\!])^{\uparrow}
\end{aligned}
$$

Figure 3.9: Denotational model of FCSL commands as sets of trees. The operator $f^{\uparrow}$ constructs the monotone completion of f.

Definition 3.55 ($\mathsf{Prog}_{\mathcal{U}}\ A$)**.** Given a type A, we define the type of FCSL commands denotations $\mathsf{Prog}_{\mathcal{U}}\ A$ as the following dependent record:

$$\mathsf{Prog}_{\mathcal{U}}\ A\ \widehat{=}\ \{T\ : \mathcal{P}\,(\mathsf{tree}\ \mathcal{U}\ A) \mid\ \mathsf{Unfinished} \in T\}$$

Figure 3.9 presents the denotational semantics of FCSL commands. For **bind**, we employ the fact that the translation $[\![c_2]\!]$ can produce *open* values (*i. e.*, trees t_2), where a variable x is not bound. We subsequently close them by constructing a continuation k as a CiC function, in an indirect way, that is, quantifying over all possible inputs, hence $\forall x, k\ x \in [\![c_2]\!]$. In the case of **hide**, we appeal to Lemma 3.47, which gives a constructive way to define an elaboration predicate $\widehat{\Phi}$ from a given abstraction function Φ. In order to give denotational meaning to the fix-point operator **fix**, we take advantage of the fact that the type $\mathsf{Prog}_{\mathcal{U}}\ A$ (and also $\mathsf{ST}_{\mathcal{U}}\ s$) is a complete lattice. Then, we use Knaster-Tarski fix-point combinator lfp to compute the denotational semantic model of the *monotone completion* of the argument.

Definition 3.56 (Monotone Completion)**.** Given a *lattice* $(A, \preceq)$ and a function $f : A \to A$, the *monotone completion* of f, $f^{\uparrow}$, is defined as follows:

$$f^{\uparrow}\ a\ \widehat{=}\ \mathsf{sup}\{t \mid\ \exists\ b.\ b \preceq a\ \wedge\ t = f\ b\}$$

Theorem 3.57 (Hoare-types are lattices)**.** *The datatypes* $A :\ \mathsf{ST}_{\mathcal{U}}\ s$ *and* $\mathsf{Prog}_{\mathcal{U}}\ A$ *define* complete *lattices. Moreover, given a complete lattice type* T*, the dependent-function space* $\Pi\, x{:}A.\, T$ *is also a complete lattice.*

We have carried out these proofs in our Coq implementation (FCSL, 2017). The domain theory library which implements, among others, the theories of lattices, the Knaster-Tarski fix-point operator lfp, and their properties is inherited from HTT (Nanevski *et al.* , 2010).

[10] Given a lattice $(A, \curlywedge, \curlyvee, \top, \bot)$, we say it is *complete iff* $\forall\ a, b \in A.\, (a \curlywedge b) \in A \wedge (a \curlyvee b) \in A$. That is, a lattice is complete *iff* every subset of the carrier set is also complete.

3.4.2 Predicate Transformers

In this section we define the predicate transformers always and after which ensure that the continuous stepping over a tree t:tree $\mathcal{U}$ A preserves *safety* and adherence to an invariance predicate over configurations R : state $\to$ tree $\mathcal{U}$ A $\to$ Prop. These transformers will later be used to instrument the denotational semantics of Hoare-types ST s $\mathcal{U}$ in the sequel.

Definition 3.58 (Modal Predicates)**.** The predicates $\mathsf{always}^{\zeta}_{\mathcal{U}}\ w\ t\ P$, $\mathsf{always}_{\mathcal{U}}\ w\ t\ P$ and $\mathsf{after}_{\mathcal{U}}\ w\ t\ Q$ are defined relative to a schedule ζ, concurrent resource $\mathcal{U}$, state w, A-returning tree t : tree $\mathcal{U}$ A and arbitrary predicates P : state $\to$ tree $\mathcal{U}$ A $\to$ Prop and Q : $A \to$ state $\to$ Prop:

$$
\begin{aligned}
\mathsf{always}^{\zeta}_{\mathcal{U}}\ w\ t\ P \ \widehat{=}\ & \mathsf{if}\ \zeta = \pi :: \zeta' \\
& \mathsf{then}\ \forall w_2.(w, w_2) \in \mathsf{R}_U \implies \\
& \qquad \mathsf{safe}\ t\ \pi\ w_2\ \wedge \\
& \qquad \Big(\forall w_3\ t_2.\ \ w_2; t \xrightarrow{\pi} w_3; t_2 \implies \mathsf{always}^{\zeta'}_{\mathcal{U}}\ w_3\ t_2\ P\Big) \\
& \mathsf{else}\ \forall w_2.(w, w_2) \in \mathsf{R}_U \implies P\ w_2\ t
\end{aligned}
$$

$$
\mathsf{always}_{\mathcal{U}}\ w\ t\ P \ \widehat{=}\ \forall \zeta.\mathsf{always}^{\zeta}_{\mathcal{U}}\ w\ t\ P
$$

$$
\mathsf{after}_{\mathcal{U}}\ w\ t\ Q \ \widehat{=}\ \mathsf{always}_{\mathcal{U}}\ w\ t\ \ (\lambda\ w'\ t.\ \forall v'.\, t' = \mathsf{Ret}\ v' \implies Q\ v'\ w')
$$

$$
\mathsf{always\text{-}safe}_{\mathcal{U}}\ w\ t \ \widehat{=}\ \mathsf{always}_{\mathcal{U}}\ w\ t\ \ (\lambda\ \_\ \_.\ .\mathsf{True})
$$

The predicate $\mathsf{always}_{\mathcal{U}}\ w\ t\ P$ expresses the fact that starting from the state w, the tree t remains memory-safe and the predicate P holds of all intermediate configurations made by a state and a tree, for any schedule ζ and under any environment of the fine-grained resource $\mathcal{U}$. The helper predicate $\mathsf{always}^{\zeta}_{\mathcal{U}}\ w\ t\ P$ is defined by induction on ζ. First, the fine-grained resource $\mathcal{U}$ is allowed to make arbitrary rely-steps from w to w_2. In the resulting configuration the predicate $P\ w_2\ t$ holds and, moreover, if the schedule ζ is non-empty (*i. e.* $\zeta = \pi::\zeta'$), then the resulting configuration must be safe, safe $t\ \pi\ w_2$ and thus, by Lemma 3.53, it can step. Consequently, if t steps to w_3 and t_2, then the predicate recurses on ζ', w_3 and t_2. Notice that if the predicate $\mathsf{always}^{\zeta}_{\mathcal{U}}\ w\ t\ P$ holds, it automatically implies "preservation" of the safe predicate at each step of reduction of the tree t. This is captured by the predicate $\mathsf{always\text{-}safe}_{\mathcal{U}}\ w\ t$, which weakens $\mathsf{always}_{\mathcal{U}}\ w\ t\ P$ just to assert the safety of all configurations.

The predicate $\mathsf{after}_{\mathcal{U}}\ w\ t\ Q$ encodes that t is safe; however, $Q\ v'\ w'$ only holds if t steps completely to Ret v' in configuration w'. In other words, the predicate after asserts that a certain postcondition holds of the *final* configuration after interleaving all the steps of a scheduler and the environment. Thus, we use after

to assert the fact that the postcondition of a triple holds once—and only if—all the paths prescribed by the scheduler have been taken.

We gather some derived lemmas for the predicate transformers described above.

Proposition 3.59 (Properties of Predicate Transformers)**.** *Given a fine-grained resource $\mathcal{U}$, a tree $t : \mathsf{tree}\ \mathcal{U}\ A$, and a state w, the modal predicate transformers satisfy the following properties:*

$$
\begin{array}{lll}
\textsc{Safety} & : & \mathsf{always}^{\pi::\zeta}_{\mathcal{U}}\ w\ t\ P \implies \mathsf{safe}\ w\ t\ \pi \\
\textsc{Weakening} & : & \mathsf{always}_{\mathcal{U}}\ w\ t\ P \implies \mathsf{always\text{-}safe}\ w\ t \\
\textsc{Coherence} & : & \mathsf{always}_{\mathcal{U}}\ w\ t \implies w \in \mathsf{Coh}_{\mathcal{U}} \\
\textsc{Universal} & : & \mathsf{always\text{-}safe}_{\mathcal{U}}\ w\ t \implies \\
 & & \quad (\mathsf{always}_{\mathcal{U}}\ w\ t\ (\lambda\, w'\ t'.\, \forall x.\, P\ x\ w'\ t') \\
 & & \qquad \iff \forall x.\, \mathsf{always}_{U}\ w\ t\ (\lambda w'\ t'.\, P\ x\ w'\ t')) \\
\textsc{Implication} & : & \mathsf{after}_{\mathcal{U}}\ w\ t\ Q_1 \implies \\
 & & \quad (\forall\, v,\, w \in \mathsf{Coh}_{\mathcal{U}}.\, Q_1\ v\ w \implies Q_2\ v\ w) \implies \mathsf{after}_{\mathcal{U}}\ w\ t\ Q_2 \\
\textsc{Framing} & : & \mathsf{after}_{\mathcal{U}}\ (w \triangleright p)\ t\ Q \implies \\
 & & \quad \mathsf{after}_{\mathcal{U}}\ (w \triangleleft p)\ t\ (\lambda v\ w'.\, \exists w''.\, w' = w'' \triangleleft p\ \wedge\ Q\, v\ (w'' \triangleright p)) \\
\textsc{SeqClosure} & : & t_2 \in \{\mathsf{Seq}\ t_1\ k \mid \forall x.\, k\ x \in K\ x\} \implies \\
 & & \quad \mathsf{after}_{\mathcal{U}}\ w_1\ t_1\ (\lambda\, v\ w.\ \forall\, t.\ (t \in K\ v) \implies \mathsf{after}_{\mathcal{U}}\ w\ t\ Q) \implies \\
 & & \qquad \mathsf{after}_{\mathcal{U}}\ w_1\ t_2\ Q \\
\textsc{ParClosure} & : & \mathsf{after}_{\mathcal{U}}\ ([s_1 \mid j \mid o] \triangleright s_2)\ t_1\ Q_1 \implies \\
 & & \quad \mathsf{after}_{\mathcal{U}}\ ([s_2 \mid j \mid o] \triangleright s_1)\ t_2\ Q_2 \implies \\
 & & \qquad \mathsf{after}_{\mathcal{U}}\ ([s_1 \circ s_2 \mid j \mid o])\ (\mathsf{Par}\ t_1\ t_2\ (\lambda x.\, \mathsf{Ret}\ x)) \\
 & & \qquad\quad \left(\begin{array}{ll} \lambda v'\ w'. & \exists s'_1\ s'_2\ j'\ o'.\ w' = [s'_1 \circ s'_2 \mid j' \mid o'] \\ & \quad \wedge\ P_1\ v'._1\ [s'_1 \mid j' \mid o'] \triangleright s'_2 \\ & \quad \wedge\ P_2\ v'._2\ [s'_2 \mid j' \mid o'] \triangleright s'_1 \end{array}\right) \\
\textsc{InjClosure} & : & w_v \cup w_O \in \mathsf{Coh}_{\mathcal{U}} \implies \mathsf{after}_{\mathcal{V}}\ w_V\ t\ Q \implies \\
 & & \quad \mathsf{after}_{\mathcal{U}}\ (w_V \cup w_O)\ (\mathsf{Inject}\ t) \\
 & & \qquad \left(\begin{array}{ll} \lambda\ v'\ w'. & \exists w'_V\ w'_0.\, w' = w'_V \cup w'_0\ \wedge\ w'_V \in \mathsf{Coh}_{\mathcal{V}} \\ & \quad \wedge\ (w_0, w'_0) \in \mathsf{R}_{\mathcal{W}} \wedge\ Q\ v'\ w'_V \end{array}\right) \\
\textsc{HideClosure} & : & \widehat{\Phi}(g)(w_U, w_V) \implies \mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi \implies \mathsf{after}_{\mathcal{V}}\ w_V\ t\ Q \implies \\
 & & \quad \mathsf{after}_{\mathcal{U}}\ w_U\ (\mathsf{Hide}\ \widehat{\Phi}\ g\ t) \\
 & & \qquad\quad (\lambda v\ w'_U.\ \exists w'_V\ g'.\, \widehat{\Phi}(g')(w'_U, w'_V) \wedge Q\ v\ w'_V)
\end{array}
$$

These lemmas are fundamental to proving the soundness of FCSL (Theorem 3.64). In particular, the closure lemmas are necessary to prove the soundness of the rules for FCSL commands. The proofs of these lemmas can be found in the Coq implementation (FCSL, 2017).

The Safety,Weakening,and Coherence properties reflect the internals of the modal predicate always. The Universal lemma states that the modal predicate always commutes with universal quantification. The always-safe hypothesis enables the property to hold even when the quantification over x is vacuous. The Implication lemma corresponds to postcondition weakening, which is necessary for the soundness of the Conseq rule. The Framing lemma states that if we can assert after that a postcondition Q holds in a small configuration with a explicit frame: $w \triangleright p$, then we could have enlarged the state with p, $w \triangleleft p$ and still prove that the postcondition holds in the enlarged state.

The *closure* lemmas state the algebraic properties of after with regard to FCSL commands, and they are used to justify the latter soundness (*cf.* Section **??**). SeqClosure states that Q holds at the end of the composed tree if final configuration of the prefix t_1 can be used as an initial configuration for the suffix to show Q holds after. ParClosure holds intuitively because when (an approximation t_2 of) c_2 takes a step over its private and shared state, it amounts to R_U environment interference on (an approximation t_1 of) c_1, and *vice versa*. The pattern of shuffling subjective self and other components is inherited from the Par rule: the parallel composition uses $[s_1 \circ s_2 \mid j \mid o]$ whereas the left and right child threads use $[s_1 \mid j \mid o] \triangleleft s_2$ and $[s_2 \mid j \mid o] \triangleleft s_1$, respectively.

The last two lemmas address the closure of after with regard to *injection hiding*. These lemmas require to check auxiliary side conditions related to the *entanglement* of the resources involved. InjClosure states that if after holds for some predicate Q on a state w_V from a "small" concurroid $\mathcal{V}$, it will hold in a large concurroid $\mathcal{U} = \mathcal{V} \rtimes \mathcal{W}$ with an additional state $w_O \in \mathsf{Coh}_{\mathcal{W}}$ reachable by $\mathsf{R}_{\mathcal{W}}$. HideClosure entails that after is closed over *hiding*, *i. e.* over resource scoping. Given the abstraction function Φ, we need to show that the *elaboration* $\widehat{\Phi}$ (Definition 3.45) holds over the initial states from the coarser state w_U to the more fine-grained w_U. We also need to show that Φ is a proper refinement for *hiding*, $\mathsf{hides}\,\mathcal{U}\ \mathcal{V}\ \Phi$ (Definition 3.48). Finally, we notice that the final state w'_V of the refined execution in $\mathcal{V}$ is related to the final state w'_U of the coarse execution in $\mathcal{U}$ by the same refinement function $\widehat{\Phi}$, but with a different seed g' for the auxiliary state, hence the conjunct $\widehat{\Phi}(g')(w'_U, w'_V)$.

3.4.3 Denotational Semantics of Hoare Types

Hoare Types The definition of the $\sqsubseteq$ ordering (Definition 3.3) on Hoare-specs evidences the fact that the Hoare triple $[\bar{v}].\{P\}\ A\ \{Q\}@\ \mathcal{C}$ is essentially syntactic sugar for a different kind of Hoare-type $\mathsf{ST}_{\mathcal{C}}\ s$, translated as follows:

Definition 3.60 (Hoare-Triples)**.** FCSL Hoare-triples $[\bar{v}].\{P\}\ A\ \{Q\}@\ \mathcal{C}$ are translated to Hoare-types $\mathsf{ST}_{\mathcal{C}}\ s$, unfolding the context of logical variables $[\bar{v}]$ as follows:

$$[\bar{v}].\{P\}\ A\ \{Q\}@\ \mathcal{C}\ \widehat{=}\ \ \mathsf{ST}_{\mathcal{C}}\ (\lambda w.\, \exists \bar{v}.\, w \models p, \lambda r\, i\ m.\, \forall \bar{v}.\, i \models p\ \Longrightarrow\ m \models q)$$

In these Hoare-types, postconditions are *binary*[11]predicates $q : \mathsf{state} \to A \to \mathsf{state} \to \mathsf{Prop}$. That is, they are relations ranging over input and output states w and w', following the VDM-style (Bjørner & Jones, 1978) of specifications we have already used in Chapter 2 for HTTcc. The advantage of this implementation of Hoare types is that the logical variables are explicitly bound, making their scoping explicit. We use this formulation of Hoare triples in in our Coq implementation of FCSL model (FCSL, 2017). Unlike the case for HTTcc in Chapter 2, we do implement the $[\bar{v}].\{P\}\, A\, \{Q\}@\, \mathcal{C}$ types in Coq as well and use the latter to implement clients and case studies. This is because, unlike the case for HTTcc, FCSL logical variable contexts are uniform and the translation 3.60 can be implemented right away.

Figure 3.10 presents the typing-rules for FCSL commands. Notice that since we embrace binary post-conditions, we recover a more HTT-like approach to the specification of **bind** in the Seq rule. In fact, the pre- and postcondition pair denotes the same specification presented in Chapter 2 for sequential composition—and also the original HTT rule from (Nanevski *et al.* , 2010). This rule, which does not require the annotation of the intermediate assertion R (*cf.* Seq, Figure 3.5), enables Coq's type-checker to infer weakest pre- and strongest postconditions from sequential compositions.

The rules Ret and Inject ditch the explicit R frames we presented in Figure 3.5. Instead, having access in the postcondition to the initial state allows us to *stabilize* the assertion explicitly thus stating that the initial and ending states are connected via environment steps: $(i, m) \in \mathsf{R}_{\mathcal{U}}$ for the full states in the case of Ret, and $(i_2, m_2) \in \mathsf{R}_{\mathcal{V}}$ in Inject for the state from the resource $\mathcal{V}$ which is entangled with the *smaller* state $\mathcal{U}$. The Act rule also makes explicit that the effect of an atomic action takes place after stabilization both in pre- and post-states.

The rule Par is slightly more involved, as the self-other *shuffling* of the subjective state required by $\circledast$ is made explicit in both the initial and ending states of the post-condition. Thus, we introduce the following definition to make the post-condition of Par more palatable:

Definition 3.61 ($[\_, \_] \circledast\!\Rightarrow [\_, \_]$)**.** The operator $[\_, \_] \circledast\!\Rightarrow [\_, \_]$ in the post-condition of the PAR rule in Figure 3.10 is defined as follows:

$$
\begin{aligned}
&(i, r, m) \models [p_1, p_2] \circledast\!\Rightarrow [q_1, q_2] \iff \\
&\quad \forall\, i_1\, i_2.\, i = [i_1 \circ i_2 \mid i_{\mathsf{j}} \mid i_{\mathsf{o}}] \implies p_1\ [i_1 \mid i_{\mathsf{j}} \mid i_{\mathsf{o}} \circ i_2] \implies p_2\ [i_2 \mid i_{\mathsf{j}} \mid i_{\mathsf{o}} \circ i_1] \implies \\
&\qquad \exists\, m_1\, m_2.\ m = [m_1 \circ m_2 \mid m_{\mathsf{j}} \mid m_{\mathsf{o}}] \\
&\qquad\qquad \wedge\ q_1\ [i_1 \mid i_{\mathsf{j}} \mid i_{\mathsf{o}} \circ i_2]\ r.1\ [m_1 \mid m_{\mathsf{j}} \mid m_{\mathsf{o}} \circ m_2] \\
&\qquad\qquad \wedge\ q_2\ [i_2 \mid i_{\mathsf{j}} \mid i_{\mathsf{o}} \circ i_1]\ r.2\ [m_2 \mid m_{\mathsf{j}} \mid m_{\mathsf{o}} \circ m_1]
\end{aligned}
$$

[11]We turn a blind eye and on the the type of postconditions and will not "count" the result argument A. Then, by calling the postconditions *binary*, we intended to make a contrast with more traditional approaches where postconditions are predicates over the ending-state—or, the ending-state plus the return value.

$$\frac{\Gamma \vdash e_1 :\ \mathsf{ST}_{\mathcal{C}}\ (p, q) \qquad \Gamma \vdash e_2 : \Pi\, x{:}B.\ \mathsf{ST}_{\mathcal{C}}\ (s_2\ x)}{\Gamma \vdash \mathbf{bind}\ e_1\, e_2 : \mathsf{ST}_{\mathcal{C}}\ \ (\lambda\, i.\ p\ i \wedge\ \forall\, r\, j.\ q\ i\, r\, j\ \Longrightarrow\ (s_2\ r)\,.1\ j,\\ \lambda\, i\, r\, m.\ \exists\, y\, k.\ q\ i\, y\, k \wedge\ (s_2\ y)\,.2\ k\, r\, m)}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{ret}\ v :\ \mathsf{ST}_{\mathcal{C}}\ (\lambda\, \_.\ \mathsf{True},\ \lambda\, i\, r\, m.\ (i, m) \in \mathsf{R}_{\mathcal{C}} \wedge r = v)}\ \textsc{Ret}$$

$$\frac{\Gamma \vdash e :\ \mathsf{ST}_{\mathcal{C}}\ s_1 \qquad s_1 \sqsubseteq s_2}{\Gamma \vdash \mathbf{do}\ e :\ \mathsf{ST}_{\mathcal{C}}\ s_2}\ \textsc{Conseq}$$

$$\frac{\Gamma \vdash f : (\Pi\, x{:}A.\ \mathsf{ST}_{\mathcal{C}}\ (s\ x)) \to \Pi\, x{:}A.\ \mathsf{ST}_{\mathcal{C}}\ (s\ x)}{\Gamma \vdash \mathbf{fix}\, f : \Pi\, x{:}A.\ \mathsf{ST}_{\mathcal{C}}\ (s\ x)}\ \textsc{Fix}$$

$$\frac{\Gamma \vdash e_1 :\ \mathsf{ST}_{\mathcal{C}}\ (p_1, q_1) \qquad \Gamma \vdash e_2 :\ \mathsf{ST}_{\mathcal{C}}\ (p_2, q_2)}{\Gamma \vdash e_1 \parallel e_2 : \mathsf{ST}_{\mathcal{C}}\ (p_1 \circledast p_2, [p_1, p_2] \circledast\Rightarrow [q_1, q_2])}\ \textsc{Par}$$

$$\frac{\Gamma \vdash e :\ \mathsf{ST}_{\mathcal{U}}\ (p, q) \qquad \mathsf{injects}\ \mathcal{U}\ \mathcal{V}}{\Gamma \vdash \mathbf{inject}\ e : \mathsf{ST}_{\mathcal{V}}\ (\lambda\, i.\ \exists\, i_1\, i_2.\ i = i_1 \cup i_2 \wedge\ i_1 \in \mathsf{Coh}_{\mathcal{U}} \wedge\ p\ i_1,\\ \lambda\, i\, r\, m.\ \forall\, i_1\, i_2.\, i = i_1 \cup i_2\ \Longrightarrow\ i_1 \in \mathsf{Coh}_{\mathcal{U}}\ \Longrightarrow\\ \exists\, m_1\, m_2.\, m = m_1 \cup m_2 \wedge\ q\ i_1\, r\, m_1 \wedge (i_2, m_2) \in \mathsf{R}_{\mathcal{W}})}\ \textsc{Inject}$$

$$\frac{\Gamma \vdash e :\ \mathsf{ST}_{(\mathcal{P} \rtimes \mathcal{U}) \rtimes \mathcal{V}}\ (p, q) \qquad \mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi}{\Gamma \vdash \mathbf{hide}_{\Phi, g}\ e\ :\mathsf{ST}_{(\mathcal{P} \rtimes \mathcal{U})}\ \ (\lambda\, i.\ \exists\, j.\ \widehat{\Phi}(g)(i, j) \wedge\ p\ j,\\ \lambda\, i\, r\, m.\ \forall\, j.\, \widehat{\Phi}(g)(i, j)\ \Longrightarrow\ p\ j\ \Longrightarrow\\ \exists\, k\, g'.\ \widehat{\Phi}(g')(m, k) \wedge\ q\ j\, r\, k)}\ \textsc{Hide}$$

$$\frac{}{\Gamma \vdash \mathbf{act}\ a\ \ :\mathsf{ST}_{\mathcal{U}}(\lambda\, i.\ \forall\, j'.\ (i, j) \in \mathsf{R}_{\mathcal{U}}\ \Longrightarrow\ j \in \mathsf{Pre}_a,\\ \lambda\, i\, r\, m.\ \exists\, j\, k.\ (i, j) \in \mathsf{R}_{\mathcal{U}} \wedge\ (j, k, r) \in \mathsf{Rel}_a \wedge\ (k, m) \in \mathsf{R}_{\mathcal{U}})}\ \textsc{Action}$$

Figure 3.10: Typing rules for FCSL commands. A spect (p, q) in a Hoare-types $\mathsf{ST}_{\mathcal{C}}\ (p, q)$ consists of an *unary* precondition p and a *binary* postcondition q. The postcondition $[p_1, p_2] \circledast\Rightarrow [q_1, q_2]$ in Par is explained in Definition 3.61.

Intuitively, the assertion captures asserts that for any split $i = [i_1 \circ i_2 \mid i_\mathsf{j} \mid i_\mathsf{o}]$ of the initial state which satisfies the pre-condition $p_1 \circledast p_2$ of the rule, there exists a split of the post-state $m = [m_1 \circ m_2 \mid m_\mathsf{j} \mid m_\mathsf{o}]$ such that

$$(q_1\,[i_1 \mid i_\mathsf{j} \mid i_\mathsf{o} \circ i_2]\,r_{\cdot 1}) \circledast (q_2\,[i_2 \mid i_\mathsf{j} \mid i_\mathsf{o} \circ i_1]\,r_{\cdot 2})$$

It should be noticed that if p_1 and p_2 are precise, and therefore determine a unique split of the state, then the definition translates to:

$$(\lambda\, i\, r\, m.\, p_1\; i \implies q_1\; i\, r\, m) \circledast\circledast (\lambda\, i\, r\, m.\, p_2\; i \implies q_2\; i\, r\, m)$$

where $\circledast\circledast$ is the point-wise lifting of $\circledast$ to binary postconditions.

Given the definition above, it becomes clear that the Par rule is just applying the translation of unary Hoare-triples into binary Hoare-types (Definition 3.60). The intuition that e_1 and e_2 are executed independently of under $\circledast$-shuffling, and then that their self-contributions are combined back in a way that satisfy the preconditions is preserved.

Verify predicate We use after to define the verify predicate transformer we introduced in Section 3.2.2 to present FCSL structural rules.

Definition 3.62 (verify predicate transformer)**.** Given $e :: \mathsf{ST}_\mathcal{C}\ A$ and a predicate $R : A \to \mathsf{state} \to \mathsf{Prop}$, the verify predicate

$$\mathsf{verify}_\mathcal{U}\ e\ R\ w \;\widehat{=}\; w \in \mathsf{Coh}_\mathcal{U} \;\wedge\; \forall t \in [\![e]\!]\,.\ \mathsf{after}_\mathcal{U}\ w\ t\ R$$

As we have mentioned before when we presented the structural rules, the verify predicate realizes FCSL verification framework. Figure 3.11 presents a series of lemmas, which are derived from the properties of the after predicate, whose role is to allow the user to *step* through a FCSL proof obligation, in a similar way one steps over a Hoare-logic proof outline on the whiteboard. We inherit the habit of referring to these lemmas as Floyd-style rules from (Nanevski *et al.* , 2010).

The vrfRet lemma states that if $Q\ r$ holds in the initial states, then the ending state of ret r satisfies $Q\ r$; in other words, ret r does not change the state and just returns r. To account for the possibility that the environment threads may change the state, we stabilize $Q\ r$ in the premise by means of the assertion:

$$(\forall\, m.\ (w, m) \in \mathsf{R}_\mathcal{U} \implies Q\ v\ m)$$

Thus, rather than asserting the validity of the predicate $(Q\ v)$ on w, we allow for the environment to do *any* number of steps from w to any state m with $(w, m) \in \mathsf{R}_\mathcal{U}$, and then asserts whether $(Q\ v)$ still holds of that final state m.

The vrfSeq lemma implements the customary Dijkstra-style rule for computing a predicate transformer of a sequential composition, by sequentially nesting two

$$
\begin{array}{lcl}
\textsc{vrfRet} & : & w \in \mathsf{Coh}_{\mathcal{U}} \implies (\forall\, m.\ (w,m) \in \mathsf{R}_{\mathcal{U}} \implies Q\ v\ m) \implies \\
& & \qquad \mathsf{verify}_{\mathcal{U}}\ (\mathbf{ret}\ v)\ Q\ w \\
\textsc{vrfSeq} & : & \mathsf{verify}_{\mathcal{U}}\ e_1\ (\lambda\, r\, m.\, \mathsf{verify}_{\mathcal{U}}\ (e_2\, r)\ (Q\ r\, m)\ w) \implies \\
& & \qquad \mathsf{verify}_{\mathcal{U}}\ (x \leftarrow e_1; e_2\, x)\ Q\ w \\
\textsc{vrfPar} & : & w = [w_1 \circ w_2 \mid w_{\mathsf{j}} \mid w_{\mathsf{o}}] \implies \mathsf{verify}_{\mathcal{U}}\ e_1\ Q_1\ ([w_1 \mid w_{\mathsf{j}} \mid w_{\mathsf{o}} \circ w_2] \implies \\
& & \qquad \mathsf{verify}_{\mathcal{U}}\ e_2\ Q_2\ ([w_2 \mid w_{\mathsf{j}} \mid w_{\mathsf{o}} \circ w_1] \implies \\
& & \qquad\qquad \mathsf{verify}_{\mathcal{U}}\ (e_1 \parallel_2)\ (\lambda\, r.\ Q_1\ r.{}_1 \circledast Q_2\ r.{}_2)\ w \\
\textsc{vrfAct} & : & (\forall\, j.\, (w,j) \in \mathsf{R}_{\mathcal{U}} \implies j \in \mathsf{Pre}_a \wedge \\
& & \quad \forall\, r\, k\, m.\ (j,k,r) \in \mathsf{Rel}_a \implies (k,m) \in \mathsf{R}_{\mathcal{U}} \implies Q\ r\, m) \implies \\
& & \qquad \mathsf{verify}_{\mathcal{U}}\ (\mathbf{act}\ a)\ Q\ w \\
\textsc{vrfInject} & : & w_u \in \mathsf{Coh}_{\mathcal{U}} \implies \\
& & \quad \mathsf{verify}_{\mathcal{U}}\ e_1\ (\lambda\ r\, k.\ \forall\ w.\ (k \cup m) \in \mathsf{Coh}_{\mathcal{U} \rtimes \mathcal{V}} \implies \\
& & \qquad\qquad (w_v, w) \in \mathsf{R}_{\mathcal{V}} \implies Q\ r\, (k \cup m))\ w_u \implies \\
& & \qquad \mathsf{verify}_{(\mathcal{U} \rtimes \mathcal{V})}\ (\mathbf{inject}\ e)\ Q\ (w_u \cup w_v) \\
\textsc{vrfHide} & : & \mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi \implies \widehat{\Phi}(g)(w_u, w_v) \implies \\
& & \quad \mathsf{verify}_{\mathcal{V}}\ e\ (\lambda\ r\, w_2.\ \forall\ f\, w_1.\ \widehat{\Phi}(f)(w_1, w_2) \implies Q\ r\, w_2)\ w_v \implies \\
& & \qquad \mathsf{verify}_{\mathcal{U}}\ (\mathbf{hide}_{\Phi,g}\ e)\ Q\ w_u
\end{array}
$$

Figure 3.11: FCSL *Floyd-style* lemmas for proof transformation.

applications of verify. Thus it allows us to step through sequential computations in a proof outline. The VRFACT lemma says that if Q is a postcondition for an action a in the pre-state w, then the return value r and the final state m should be such that are related by Rel_a (i.e., such that $(w, m, r) \in \mathsf{Rel}_a$) and $Q\ r\ m$. However, we need to allow for environment steps before and after applying the effect of the atomic action. Thus, we need to stabilize the assertion twice. The first step, which takes us from w to an intermediate state j, needs to be a *rely* step that preserves the *precondition* for the atomic action, thus $jin\mathsf{Pre}_a$. Then, we stabilize the ending state, thus we demand that the ending state m is obtained from environment steps $(k, m) \in \mathsf{R}_\mathcal{U}$ after any state k resulting from the effect of atomic action on the intermediate state j, *i. e.* $(j, k, r) \in \mathsf{Rel}_a$.

The VRFPAR lemma encodes the usual property of separation logics that if the initial state w can be split into w_1 and w_2, such that e_1 executes in w_1 to obtain postcondition Q_1, and e_2 executes in w_2 to obtain postcondition Q_2, then the ending state of $e_1 \parallel e_2$ can be split in the same way. However, the *subjective* nature of FCSL's PAR rule, witnessed by the use of $\circledast$ (Definition 3.2), entails that such split is made in particular way. The *self*-components of the children states divide the *self*-component of the parent ($w_\mathsf{s} = w_1 \circ w_2$). At the same time, the *other*-component of e_1 adds the *self*-components of e_2 ($w_\mathsf{o} \circ w_2$) to capture the fact that e_2 becomes part of the concurrent environment of e_1, and *vice versa*. The joint component w_j represents shared state, so it is propagated to both children without changing. Finally, the end-result of $e_1 \parallel e_2$ is a pair $r = (r.1, r.2)$ of type $A_1 \times A_2$, combining the return results of e_1 and e_2. Thus, the postcondition of $e_1 \parallel e_2$ splits r and passes each projection to Q_1 and Q_2.

The VRFINJECT rule allows us to verify a post-condition Q of an enlarged state $w_u \cup w_v$, when *lifting* a computation $e : \mathsf{ST}_\mathcal{U}\ (P, Q)$ from a smaller resource $\mathcal{U}$. The premise requires that w_u is a valid state for the smaller resource $\mathcal{U}$, and that on the smaller state we can verify a "smaller" version of the post-condition Q, which behaves as Q on the $\mathcal{U}$ part and it is stabilized on the w_v addendum:

$$(\lambda\ r\ k. \forall\ w.\ (k \cup m) \in \mathsf{Coh}_{\mathcal{U} \bowtie \mathcal{V}} \implies (w_v, w) \in \mathsf{R}_\mathcal{V} \implies Q\ r\ (k \cup m))$$

Finally, the VRFHIDE lemma allows us to verify a post-condition Q after the *hiding* of the resource $\mathcal{V}$ with Φ. Given two states w_u, w_v such that the elaboration $\widehat{\Phi}(g)(w_u, w_v)$ holds and, moreover, $\mathsf{hides}\ \mathcal{U}\ \mathcal{V}\ \Phi$, we can verify that Q holds on w_u if we can verify a variant of Q holds on w_v. The latter variant of Q is akin to the stabilizations used before, but connects the states through the $\widehat{\Phi}$ abstraction function rather than environment steps.

Denotational Semantics of Hoare-Types We use the after predicate (Definition 3.58) to ensure that the tree approximations which constitute the model of our commands are memory safe, respect mutual exclusion, and satisfy their FCSL specifications. Thus, we define the denotational meaning of Hoare-types as follows:

Definition 3.63 (Denotational Semantics of FCSL's Hoare Types)**.** Given a resource $\mathcal{U}$ and a specification (p, q), the Hoare-type $\mathsf{ST}_{\mathcal{U}}\ (p, q)$ is defined as:

$$\mathsf{ST}_{\mathcal{U}}\ (p,q) \quad \widehat{=} \quad \left\{ T : \mathsf{Prog}_{\mathcal{U}}\ A \;\middle|\; \begin{array}{r} \forall\, w\, t.\, t \in T \implies w \in \mathsf{Coh}_{\mathcal{U}} \implies \\ w \models p \implies \mathsf{after}_{\mathcal{U}}\ w\ t\ (q\ w) \end{array} \right\}$$

That is, the denotational meaning of Hoare-types is given by a dependent record consisting of a set of trees T and a proof that, for all trees $t \in T$ and all valid states $w \in \mathsf{Coh}_{\mathcal{U}}$ which satisfy the precondition p, *after* every execution of t from the initial state w the postcondition $(q\, w)$ holds.

Then, the intuition for the denotational meaning of a FCSL type ascription $e : \{P\}\ A\ \{Q\}@\,\mathcal{U}$ becomes clear: $[\![e]\!]$ denotes a set of action trees for the resource $\mathcal{U}$ such that all trees can be *stepped* from a state satisfying the precondition p, until a final result value v is obtained in a state that satisfies the post-condition q. The "operational flavour" of this definition which connects denotations through a predicate-transformer implementing pool-semantics under all possible *fair* schedules is reminiscent of the operational semantics of CSL-like program logics *e. g.* Deny-Guarantee (Dodds *et al.* , 2009). The "modal flavour", denoted by the after, and always predicate transformers which enforce *persistent* validity of an assertion, over *always-safe* states, is reminiscent of step-indexed logics like Iris (Jung *et al.* , 2015), where the denotation of triples is given explicitly in terms of a $\Box$ modality.

Soundness We close this section by stating the main soundness result of FCSL. It consists of the proof that if a FCSL judgement $\Gamma \vdash e : T$ holds, then $[\![\Gamma]\!] \vdash_{CiC} [\![e]\!] : [\![T]\!]$. As we have mentioned before[12], FCSL is implemented as a *shallow embedding* in Coq's underlying metatheory, CiC. Here, this entails two particular benefits. First, the denotation of contexts $[\![\Gamma]\!]$ is trivially defined as the pointwise translation of FCSL ascriptions in the contexts to their CiC counterparts. Second, and foremost, that we do not need to establish the soundness of *pure* type ascriptions $e : A$, as they are lifted from Coq. Thus we tackle only judgements $e : \mathsf{ST}_{\mathcal{U}}\ A$. The main soundness result follows.

Theorem 3.64 (Soundness of FCSL)**.** *If* $\Gamma \vdash e : \mathsf{ST}_{\mathcal{U}}\ A$ *then* $[\![\Gamma]\!] \vdash_{CiC} [\![e]\!] : [\![\mathsf{ST}_{\mathcal{U}}\ A]\!]$.

Proof. The proof is discharged by induction on the structure of FCSL commands, as presented in Figure 3.10. For each of these commands, we need to show that their underlying denotational semantics $[\![e]\!]$, defined in Figure 3.9, together with their specifications indeed satisfy the after assertion in the definition of Hoare-types (Definition 3.63). These proofs are discharged appealing to the predicate-transformer's *closure* lemmas from Proposition 3.59, *e. g.* SeqClosure,

[12] *Ad nauseam*, by now.

InjClosure, *etc.*. The only special case is that of the fixed-point operator **fix**, whose soundness arises from the fact that Hoare-types define a complete lattice (Theorem 3.57), and the appeal to the Knaster-Tarski fixed point theorem. The full details of these proofs can be found in our Coq sources (FCSL, 2017). □

3.5 Related Work

Fine-Grained Resources FCSL builds on the previous work on subjective auxiliary state and SCSL logic (Ley-Wild & Nanevski, 2013). The SCSL logic contained the distinction between self and other views, which was essential for compositional implementation of *auxiliary state*. However, it contained exactly one coarse-grained resource, with no ability to create and dispose new resources. In contrast, FCSL can introduce any number of fine-grained resources in a scoped way. Feng's Local Rely-Guarantee (LRG) (Feng, 2009) is, to the best of our knowledge, the first work that reconciled fine-grained reasoning in the style of RG with framing and hiding at the level of transitions (similar to our Inject and Hide). We differ from LRG in that we introduce communication and subjectivity into the mix; thus our injection and hiding rules take *self* and *other* views into account. The latter are a compositional form of auxiliary state, whereas LRG in practice has to use the classical, non-compositional form of auxiliary state (Owicki & Gries, 1976; Ley-Wild & Nanevski, 2013).

Shared Regions & Protocols The work on Concurrent Abstract Predicates (CAP) (Dinsdale-Young *et al.* , 2010) originally introduced a notion of *shared region* that serves a similar purpose as concurroids, in that regions circumscribe a chunk of shared heap with a protocol governing its evolution. A *protocol* is defined by a set of atomic actions, which are RG-style transitions on private state and a region. In addition to heaps, regions may contain abstract capabilities that identify enabled actions. Thus there is a subtle mutual recursion in a protocol definition between an action and the capability to perform the action. A recurring pattern for this approach is quantification over *all* possible capabilities and placing them in a shared region, to be used up if needed in the execution of the protocol.

The CAP framework could atomically change only one region; a restriction lifted in the more recent works on Views (Dinsdale-Young *et al.* , 2013), HOCAP (Svendsen *et al.* , 2013), iCAP (Svendsen & Birkedal, 2014), and Iris (Jung *et al.* , 2015, 2016; Krebbers *et al.* , 2017b,a). Following the Views framework, these logics introduce *view shifts*—or a similar mechanism—to synchronize changes across several regions. Once allocated, CAP's regions have dynamically-scoped lifetime, and they can be disposed by a particular thread if it collects all corresponding region's capabilities. To the best of our knowledge, neither HOCAP nor iCAP nor Iris allow the removal or scoped hiding of a shared region.

In contrast with CAP and their successors, FCSL does not require capabilities

to perform actions, as these are naturally represented in the *self* and *other* views associated with a resource (and can also be seen as auxiliary state). Such auxiliary state is simpler than capabilities; it is not subject to ownership transfer, and there is no need to quantify over all capabilities. In our experience, this simplicity extends to the specification of invariants and transitions, and to the proofs of stability. In FCSL, synchronizing changes over a number of concurroids is achieved directly at the level of transitions by means of entanglement, and at the level of programs by allowing actions to be defined over any concurroid, including entangled ones. Thus, no view shifts are required.

The burden of stability proofs is further reduced in FCSL by formulating private heaps as a separate resource that one may, but need not, entangle with. Thus, when an action manipulates only the internal state of a resource, the attendant stability proofs can ignore private heaps, *e. g.*, the `take` action of a ticketed lock (Nanevski *et al.* , 2014b; FCSL, 2017) and the resource $\mathcal{J}$ for Jayanti's snapshot we will present in the next Chapter. Moreover, the communication in FCSL makes it possible for concurroids to pass heaps between each other directly, rather than going through private state.

Higher-order auxiliary code. FCSL auxiliary code is first order and the transitions of a resource are closed, in the sense that resources cannot[13]be extended with transitions from other resources, if not by combining acquire and release transitions. In contrast, the work by Jacobs and Piessens (Jacobs & Piessens, 2011) relies on the parameterization a program and its proof with auxiliary code that works over the state of other resources and that is invoked by the program as a kind of callback. A well-known challenge of parameterized a program by an auxiliary function is exhibited when the point at which to execute the auxiliary function can be determined only after the program has already terminated. In contrast, FCSL does not have this restriction, as we will see in the next Chapter.

STSs & Contextual Refinement CaReSL (Turon *et al.* , 2013a) uses the same notion of shared region as CAP, though it specifies the transitions in a manner closer to FCSL, namely by means of STS's. However, the logic does not directly provide subjective *self* and *other* views of a resource, but it provides a notion of *tokens*, whose ownership is exchanged between a thread and its environment. CaReSL assertions explicitly allow statements only about self-owned tokens, not *other*-owned ones. Thus, reasoning about the lack of logical changes to environment-owned data has to be encoded with a level of indirection, potentially quantifying over all tokens, similar to CAP's quantification over capabilities. A frequent side condition in CaReSL rules is that various assertions are *token-pure*, which does not have a direct correspondent in FCSL. Similar to CAP, CaReSL

[13]Currently, at least. This restriction is addressed in the new version of FCSL in development. I will update this manuscript with the appropriate references in time.

allows actions that work over only a single region. Moreover, CaReSL does not consider removal or scoped hiding of shared regions, although it can be emulated by introducing an empty "final" protocol state.

CaReSL can reason about fine-grained data structures by means of contextual refinement (a generalization of linearizability). In contrast, in FCSL we reason about fine-grained data structures using history based auxiliary state (Sergey *et al.* , 2015b,a; Delbianco *et al.* , 2017a). Chapter 4 will showcase a particular design of *relinkable histories*, which enable reasoning about fine-grained objects with non-fixed linearization points.

Subjective State and PCMs Similarly to FCSL, Iris makes use of PCMs as a generic mechanism to describe state, which can be split between parallel threads. However, neither iCAP nor Iris have explicit subjective dichotomy of the auxiliary state, which makes encoding of thread-specific contributions in them less direct comparing to FCSL. CoLoSL (Raad *et al.* , 2015) defines a different notion of thread-local views to a shared resource, and uses *overlapping conjunctions* (Hobor & Villard, 2013) to reconcile the permissions and capabilities, residing in the shared state between parallel threads. Overlapping conjunctions afford a description of the shared structure mirroring the recursive calls in the structure's methods. In FCSL, such machinery is not required, as *self* and *other* suffice to represent thread-specific views, and *joint* state does not need to be divided.

Abstract atomicity FCSL atomic actions are restricted by the operational semantic to occur in a unique step. TaDA (da Rocha Pinto *et al.* , 2014), introduced a novel approach, which introduces a new judgement form, $\langle P \rangle\ e\ \langle Q \rangle$, capturing that e has a precondition P and postcondition Q, but is also *abstractly atomic*: e and its concurrent environment will maintain the validity of P through the execution until at one point e makes an atomic step that makes Q hold. After that point, Q may be invalidated, either by future steps of e, or by the environment. The specification style of TaDA is close to ours in the sense that it employs *atomic tracking resources*, that are reminiscent of history auxiliary state (*cf.* Chapter 4). However, the metatheory of TaDA does not support ownership transfer. In contrast, FCSL can verify fine-grained data-structures with *helping* using standard auxiliary state (Sergey *et al.* , 2015b). Iris has recently adopted a flavour of abstract atomicity which differs from TaDA's in that it is encoded using the support for higher-order state available in Iris. Otherwise, the fragment of the proof theory that handles abstract atomicity is almost identical in both logics.

Linearizability Reasoning Although we have not established a formal connection between FCSL specifications and *linearizability*, FCSL has been used to verify both *linearizable* fine-grained concurrent data structures (Sergey *et al.* , 2015b,a; Delbianco *et al.* , 2017a) and other data-structures which are *not*

linearizable (Sergey *et al.* , 2016) and whose correctness is given by further correctness criteria, *e. g. Concurrency-Aware-Linearizability* (Hemed & Rinetzky, 2014; Hemed *et al.* , 2015). This issue will be central in the next chapter, where we present a novel technique for specifying and verifying fine-grained data structures with non-fixed linearization points implemented on top of FCSL.

3.6 Summary

This chapter presented *fine-grained concurrent resources*, a novel model for scalable shared-variable concurrency verification, based on communicating state transition systems (STS); and FCSL—a logic and verification framework which enables proof modular verification of complex concurrent data structures. In this logic, client proofs are developed in FCSL out of the specs, and not the code of programs, therefore we can substitute different implementations of different algorithms in clients, without disturbing the clients' proofs.

A fine-grained resource is specified by a resource invariant, as in the CSL family of program logics, but it also adds transitions in the form of relations between resource states, following the influence of R/G and its derivatives. The result is a powerful logic, which gets the best of both worlds. In addition to the case studies presented in this chapter, FCSL has been used to verify a number of non-trivial fine-grained concurrent data structures (Sergey *et al.* , 2015b,a, 2016).

FCSL provides a flexible framework in which clients of the logic can specify their own fine-grained concurrent resources (including the structure for the auxiliary state), adjusting to the demands of the task at hand. All of the auxiliary state features we visited in this chapter are *user-defined* concepts, and therefore they are not hardwired into the semantics of the logic. This will prove a remarkable advantage in the next chapter, where we will use FCSL to implement, *off-the-shelf*, a novel technique to reason about fine-grained concurrent objects with contrived correctness arguments.

We will show how the rich specification language of FCSL suffices to encode the correctness criteria of these data structures as a part of the resource invariants, without having to modify the meta-theory of FCSL. We illustrate the method by verifying (mechanically in Coq) an intricate optimal snapshot algorithm due to Jayanti (Jayanti, 2005), together with some concurrent clients of the snapshot data structure. This algorithm features future-dependent, non-local and non-regional linearization points, making it very intricate to reason with. Most program logics aimed at similar verification tasks require specific meta-theory (Chakraborty *et al.* , 2015; Liang & Feng, 2013). Surprisingly, using FCSL we can do without.

4

Concurrent Data Structures Linked in Time

Linearizability (Herlihy & Wing, 1990) is the de facto correctness criterion for reasoning with concurrent fine-grained data-structures. It works by relating the concurrent history of a program with its sequential behaviour. More precisely, for each concurrent history of an object, linearizability requires that there exists a mapping to a sequential history, such that the ordering of matching call/return pairs is preserved either if they are performed by the same thread, or if they do not overlap. To prove linearizability, one usually has to identify linearization points in programs or object methods, showing that this particular point is the single, atomic, point where the effect of the operation occurs.

However, for certain complex concurrent objects, proving them to be linearizable is not a straightforwards task: the linearization points of their methods are not fixed by the structure of the programs themselves, but rather depend on intricate interactions with the environment. Traditionally, verifying such objects requires a dedicated metatheory, e.g. supporting prophecy variables, capable of reasoning about their highly speculative nature.

In contrast, in this chapter we propose a new method for reasoning about concurrent objects with such linearization points, based on an existing separation logic, FCSL (Nanevski *et al.* , 2014a). The method embraces the dynamic nature of linearization points, and encodes it as part of the structure's *auxiliary state*, so that it can be dynamically changed by *auxiliary code*, *in place*.

We name the idea *linking in time*, because we reason about temporal ordering of events using the same logical features that one would use to reason about spatial linking via pointers in separation logic. Moreover, relying on separation logic to reason about time enables that linearization points of a procedure can be specified locally, even when they may be positioned in other procedures in the classical approach. Furthermore, the auxiliary state provides a convenient way to concisely express the properties essential for reasoning about clients of such concurrent objects.

We will illustrate our technique by presenting the mechanization in FCSL of an

optimal concurrent-snapshot algorithm originally introduced by Jayanti (Jayanti, 2005), whose correctness itself has a highly-speculative nature, relying on non-trivial arguments about its non-fixed linearization points.

4.1 Introduction

Formal verification of concurrent objects commonly requires reasoning about linearizability (Herlihy & Wing, 1990). This is a standard correctness criterion whereby a concurrent execution of an object's procedures is proved equivalent, via a simulation argument, to some sequential execution. The clients of the object can be verified under the sequentiality assumption, rather than by inlining the procedures and considering their interleavings. Linearizability is often established by describing the *linearization points* (LP) of the object, which are points in time where procedures take place, *logically*. In other words, even if the procedure physically executes across a time interval, exhibiting its linearization point enables one to pretend, for reasoning purposes, that it occurred instantaneously (*i. e.*, atomically); hence, an interleaved execution of a number of procedures can be reduced to a sequence of atomic events.

Reasoning about linearization points can be tricky. Many times, a linearization point of a procedure is not *local*, but may appear in another procedure or thread. Equally bad, linearization points' place in time may not be determined statically, but may vary based on the past, and even future, *run-time* information, thus complicating the simulation arguments. A particularly troublesome case is when run-time information influences the logical order of a procedure that has already terminated. This chapter presents a novel approach to specification of concurrent objects, in which the dynamic and non-local aspects inherent to linearizability can be represented in a procedure-local and thread-local manner.

The starting point of our idea is to realize what are the shortcomings of linearizability as a canonical specification method for concurrent objects. Consider, for instance, the following two-threaded program manipulating a correct implementation of stack by invoking its `push` and `pop` methods, which are atomic, *i. e.*, linearizable:

```
push(3);        ||  push(4)
t1 := pop();    ||  t2 := pop();
```

Assuming that the execution started in an empty stack, we would like to derive that it returns an empty stack and `(t1, t2)` is either `(3, 4)` or `(4, 3)`. Linearizability of the stack guarantees that the overall trace of `push`/`pop` calls is coherent with respect to a sequential stack execution. However, it does not capture *client*-specific partial knowledge about the *ordering* of particular `push`/`pop` invocations in sub-threads, which is what allows one to prove the desired result as a composition of separately-derived partial specifications of the left and the right thread.

This thread-local information, necessary for compositional reasoning about clients, can be captured in a form of *auxiliary state* (Owicki & Gries, 1976) (a generalization of *history variables* (Abadi & Lamport, 1988)), widely used in Hoare-style specifications of concurrent objects (Sergey *et al.* , 2015b; Ley-Wild & Nanevski, 2013; Jung *et al.* , 2015, 2016). A testament of expressivity of Hoare-style logics for concurrency with rich auxiliary state are the recent results in verification of fine-grained data structures with helping (Sergey *et al.* , 2015b), concurrent graph manipulations (Sergey *et al.* , 2015a), barriers (Jung *et al.* , 2016; Dodds *et al.* , 2016), and even *non-linearizable* concurrent objects (Sergey *et al.* , 2016).

Although designed to capture information about events that happened concurrently *in the past* (hence the original name *history variables*), auxiliary state is known to be of little use for reasoning about data structures with *speculative* executions, in which the ordering of past events may depend on other events happening in the *future*. Handling such data structures requires specialized metatheory (Liang & Feng, 2013) that does not provide convenient abstractions such as auxiliary state for client-side proofs. This is one reason why the most expressive client-oriented concurrency logics to date avoid reasoning about speculative data structures altogether (Jung *et al.* , 2016).

The surprising result we present in this chapter is that by allowing certain *internal* (*i. e.*, not observable by clients) manipulations with the auxiliary state, we can use an existing program logic for concurrency, like, *e. g.*, FCSL (Nanevski *et al.* , 2014a; Sergey *et al.* , 2015a), to specify and verify algorithms whose linearizability argument requires speculations, *i. e.*, depends on the *dynamic reordering* of events based on run-time information from the future. To showcase this idea, we provide a new specification (spec) and the first formal proof of a very sophisticated snapshot algorithm due to Jayanti (Jayanti, 2005), whose linearizability proof exhibits precisely such kind of dependence.

While we specify Jayanti's algorithm by means of a separation-style logic, the spec nevertheless achieves the same general goals as linearizability, combined with the benefits of compositional Hoare-style reasoning. In particular, our Hoare triple specs expose the logical atomicity of Jayanti's methods (Section 4.3), while hiding their true fine-grained and physically non-atomic nature. The approach also enables that the separation logic reasoning is naturally applied to clients (Section 4.4). Similarly to linearizability, our clients can reason out of procedures' spec, not code. We can also ascribe the same spec to different snapshot algorithms, without modifying client's code or proof.

In more detail, our approach works as follows. We use shared auxiliary state to record, as a list of timed events (*e. g.*, writes occurring at a given time), the logical order in which the object's procedures are perceived to execute, each instantaneously (Section 4.5). Tracking this time-related information through state enables us to specify its dynamic aspects. We can use *auxiliary code* to mutate the logical order *in place*, thereby permuting the logical sequencing of the

```
1  write (p, v) {
2    p := v;
3    b ← read(S);
4    if b
5    then (fwd p) := v}

   fwd (p : ptr) {
     return (p = x) ? fx : fy }

6  scan : (A × A) {
7      S := true;
8      fx := ⊥;
9      fy := ⊥;
10     vx ← read(x);
11     vy ← read(y);
12     S := false;
13     ox ← read(fx);
14     oy ← read(fy);
15     rx ← if (ox ≠ ⊥) then ox else vx;
16     ry ← if (oy ≠ ⊥) then oy else vy;
17     return (rx, ry)}
```

Figure 4.1: Jayanti's single-scanner/single-writer snapshot algorithm.

procedures, as may be needed when some run-time event occurs (Sections 4.6 and 4.8). This mutation is similar to updating pointers to reorder a linked list, except that it is executed over auxiliary state storing time-related data, rather than over real state. This is why we refer to the idea as *linking-in-time*.

Encoding temporal information by way of representing it as mutable state allows us to use FCSL off-the-shelf to verify example programs. In particular, FCSL has been implemented in the proof assistant Coq, and we have fully mechanized the proof of Jayanti's algorithm (Delbianco *et al.* , 2017b).

4.2 Verification challenge and main ideas

Jayanti's snapshot algorithm (Jayanti, 2005) provides the functionality of a shared array of size m, operated on by two procedures: `write`, which stores a given value into an element, and `scan`, which returns the array's contents. We use the *single-writer/single-scanner* version of the algorithm. which assumes that at most one thread writes into an element, and at most one thread invokes the scanner, at any given time. In other words, there is a scanner lock and m per-element locks. A thread that wants to scan, has to acquire the scanner lock first, and a thread that wants to write into element i has to acquire the i-th element lock. However, scanning and writing into different elements can proceed concurrently. This is the simplest of Jayanti's algorithms, but it already exhibits linearization points of dynamic nature. We also restrict the array size to $m = 2$ (*i. e.*, we consider two pointers x and y, instead of an array). This removes some tedium from verification,

```
l:  write (x,2);
    write (y,1)   ||  c:  scan ()  ||  r:  write (x,3)
```

(a) Parallel composition of three threads `l, c, r`.

```
 1  c: S := true                      11  l: y := 1
 2  c: fx := ⊥                        12  l: read(S)  // b <- true
 3  c: fy := ⊥                        13  l: fy := 1
 4  c: read(x)  // vx <- 5            14  l: return ()
 5  c: read(y)  // vy <- 0            15  c: S := false
 6  l: x := 2                         16  r: read(S)  // b <- false
 7  l: read(S)  // b <- true          17  r: return ()
 8  l: fx := 2                        18  c: read(fx) // ox <- 2
 9  l: return ()                      19  c: read(fy) // oy <- 1
10  r: x := 3                         20  c: return (2,1)
```

(b) A possible interleaving of the threads in (a).

Figure 4.2: An example leading to a scanner miss.

but exhibits the same conceptual challenges.

The difficulty in this snapshot algorithm is ensuring that the scanner returns the most recent snapshot of the memory. A naïve scanner, which simply reads x and y in succession, is unsound. To see why, consider the following scenario, starting with $x = 5$, $y = 0$. The scanner reads x, but before it reads y, another thread preempts it, and changes x to 2 and, subsequently, y to 1. The scanner continues to read y, and returns $x = 5, y = 1$, which was never the contents of the memory. Moreover, (x, y), changed from $(5, 0)$ to $(2, 0)$ to $(2, 1)$ as a result of distinct non-overlapping writes; thus, it is impossible to find a linearization point for the scan because linearizability only permits reordering of overlapping operations.

To ensure a sound snapshot, Jayanti's algorithm internally keeps additional *forwarding pointers* fx and fy, and a Boolean *scanner bit* S. The implementation is given in Figure 4.1[1]. A writer storing v into p (line 2), will additionally store v into the forwarding pointer for p (line 5), provided S is set. If the scanner missed the write and instead read the old value of p (lines 10–11), it will have a chance to catch v via the forwarding pointer (lines 13–14). The scanner bit S is used by writers (line 3) to detect a scan in progress, and forward v.

As Jayanti proves, this implementation *is* linearizable. Informally, every overlapping calls to `write` and `scan` can be rearranged to appear as if they

[1]Following Jayanti, we simplify the presentation and omit the locking code that ensures the single-writer/single-scanner setup. We make this locking explicit in our Coq development (Delbianco *et al.* , 2017b). It should be clarified that such a locking is intended to restrict the number of concurrent threads calling the methods, *i. e.* it does not lock nor synchronize the main memory, and thus it preserves the fine-grained nature of the algorithm.

occurred sequentially. To illustrate, consider the program in Figure 4.2a, and one possible interleaving of its primitive memory operations in Figure 4.2b. The threads `l`, `c`, and `r`, start with $x = 5, y = 0$. The thread `c` is scheduled first, and through lines 1–5 sets the scanner bit, clears the forwarding pointers, and reads $x = 5, y = 0$. Then `l` intervenes, and in lines 6–9, overwrites x with 2, and seeing S set, forwards 2 to fx. Next, `r` and `l` overlap, writing 3 into x and 1 into y. However, while 1 gets forwarded to fy (line 13), 3 is not forwarded to fx, because S was turned off in line 15 (*i. e.*, the scan is no longer in progress). Hence, when `c` reads the forwarded values (lines 18, 19), it returns $x = 2, y = 1$.

While $x = 2, y = 1$ was never the contents of the memory, returning this snapshot is nevertheless justified because we can *pretend* that the scanner *missed* `r`'s write of 3. Specifically, the events in Figure 4.2b can be *reordered* to represent the following sequential execution:

$$\texttt{write}\,(\texttt{x}, 2);\ \texttt{write}\,(\texttt{y}, 1);\ \texttt{scan}\,();\ \texttt{write}\,(\texttt{x}, 3) \tag{4.1}$$

Importantly, the client programs have no means to discover that a different scheduling actually took place in real time, because they can access the internal state of the algorithm only via interface methods, `write` and `scan`.

This kind of temporal reordering is the most characteristic aspect of linearizability proofs, which typically describe the reordering by listing the linearization points of each procedure. At a linearization point, the procedure's operations can be spliced into the execution history as an uninterrupted chunk. For example, in Jayanti's proof, the linearization point of `scan` is at line 12 in Figure 4.1, where the scanner bit is unset. The linearization point of `write`, however, may vary. If `write` starts before an overlapping `scan`'s line 12, and moreover, the `scan` misses the `write`—note the dynamic and future-dependent nature of this property—, then `write` should appear after `scan`; that is, the `write`'s linearization point is right after `scan`'s linearization point at line 12. Otherwise, `write`'s linearization point is at line 2. In the former case, `write` exactly has a non-local and future-dependent linearization point, because the decision on the logical order of this `write` depends on the execution of `scan` in a different thread. This decision takes effect on lines 13–14, which can take place *after* the execution of `write` has terminated. For instance, in Figure 4.2b the execution of `write` in `r` terminates at step 17, yet, in Jayanti's proof, the decision to linearize this `write` after the overlapping `scan` is taken at line 18, when the `scan` reads the value from the previous `write`.

Obviously, the high-level pattern of the proof requires tracking the *logical ordering* of the `write` and `scan` events, which differs from their *real-time ordering*. As the logical ordering is inherently dynamic, depending on properties such as `scan` missing a `write`, we formalize it in Hoare logic, by keeping it as a list of events in auxiliary state that can be dynamically reordered as needed. For example, Figure 4.3 shows the situation in the execution of `scan` that we reviewed above. We start with the (initializing) writes of 5 and 0 already executed, and

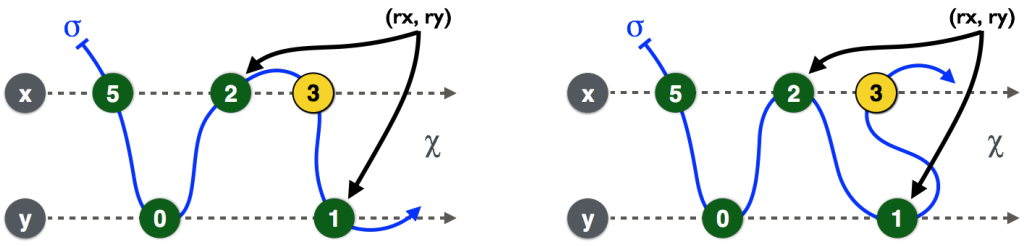

Figure 4.3: Changing the logical ordering (solid line ζ) of write events from (5, 0, 2, 3, 1) in (a) to (5, 0, 2, 1, 3) in (b), to reconcile with `scan` returning the snapshot $x = 2, y = 1$, upon missing the write of 3. Dashed lines χ represent real-time ordering.

our program performs the writes of 2, 3 and 1 in the real time order shown by the position of the events on the dashed lines. In Figure 4.3a, the logical order ζ coincides with real-time order, but is unsound for the snapshot $x = 2, y = 1$ that `scan` wants to return. In that case, the auxiliary code with which we annotate `scan`, will change the sequence ζ in-place, as shown in Figure 4.3b.

Our specification and verification challenge then lies in reconciling the following requirements. First, we have to posit specs that say that `write` performs a write, and `scan` performs a scan of the memory, with the operations executing in a single logical moment. Second, we need to implement the event reordering discipline so that a method call only reorders events that overlap with it; the logical order of the past events should be preserved. This will be accomplished by introducing yet further structures into the auxiliary state and code. Finally, the specs must hide the specifics of the reordering discipline, which should be internal to the snapshot object. Different snapshot implementations should be free to implement different re-orderings, without changing the method specs.

4.3 Specification

General considerations. For the purposes of specification and proof, we record a history of the snapshot object as a set of entries of the form $t \mapsto (p, v)$. The entry says that at time t (a natural number), the value v was written into the pointer p. We thus identify a write event with a *single* moment in time t, enabling the specs of `write` and `scan` to present the view that write events are logically atomic. Moreover, in the case of snapshots, we can ignore the scan events in the histories. The latter do not modify the state in a way observable by clients who can access the shared pointers only via interface methods `write` and `scan`.

We keep three auxiliary history variables. The history variables $\chi_{\mathtt{s}}$ and $\chi_{\mathtt{o}}$ are

local to the specified thread, and record the *terminated* write events carried out by the specified thread, and that thread's interfering environment, respectively. We refer to $\chi_{\mathtt{s}}$ as the *self*-history, and to $\chi_{\mathtt{o}}$ as the *other*-history (Ley-Wild & Nanevski, 2013; Nanevski *et al.* , 2014a; Nanevski, 2016; Sergey *et al.* , 2015b). The role of $\chi_{\mathtt{o}}$ is to enable the spec of `write` to situate the performed write event within the larger context of past and ongoing writes, and the spec of `scan` to describe how it logically reordered the writes that overlapped with it. The third history variable $\chi_{\mathtt{j}}$ records the set of write events that are in progress. These are events that have been initiated, time-stamped, and have executed their physical write to memory, but have not terminated yet. It is an important component of our auxiliary state design that when a write event terminates, it is moved from $\chi_{\mathtt{j}}$ to the invoking thread's $\chi_{\mathtt{s}}$, to indicate the *ownership* of the write by the invoking thread. We name by χ the union $\chi_{\mathtt{s}} \cup \chi_{\mathtt{o}} \cup \chi_{\mathtt{j}}$, which is the global history of the data structure. As common in separation logic, the union is *disjoint*, *i. e.*, it is undefined if the components contain duplicate timestamps. By the semantics of our specs, χ is always defined, thus $\chi_{\mathtt{s}}$, $\chi_{\mathtt{o}}$ and $\chi_{\mathtt{j}}$ never duplicate timestamps.

The real-time ordering of the time-stamped events is the natural numbers ordering on the timestamps. To track the *logical* ordering, we need further auxiliary notions. The first is the auxiliary variable ζ, whose type is a mathematical sequence. The sequence ζ is a permutation of timestamps from χ showing the logical ordering of the events in χ. We write $t_1 \leq_\zeta t_2$, and say that t_1 is logically ordered before t_2, if t_1 appears before t_2 in ζ. The sequence ζ resides in joint state, and can be dynamically modified by any thread. For example, the execution of the scanner may reorder ζ, as shown in Figure 4.3b. Because ζ is a sequence, the order $\leq_\zeta$ is linear.

Because sequence ζ changes dynamically under interference, it is not appropriate for specifications. Thus, our second auxiliary notion is the *partial* order Ω, a suborder of $\leq_\zeta$ that is *stable* in the following sense. It relates the timestamps of events whose logical order has been determined, *and will not change in the future*. Thus Ω can grow over time, to add new relations between previously unrelated timestamps, but cannot change the old relations.

To illustrate the distinction between the two orders, we refer to Figure 4.3a. There, ζ represents the linear order $5{-}0{-}2{-}3{-}1$, which changes in Figure 4.3b to $5{-}0{-}2{-}1{-}3$. Since 1 and 3 exchange places, the stable order Ω cannot initially relate the two. Thus, in Figure 4.3a, Ω is represented by the Hasse diagram $5{-}0{-}2{<}\begin{smallmatrix}1\\3\end{smallmatrix}$. In Figure 4.3b, the relation $1{-}3$ is added to this partial order, making it the linear order $5{-}0{-}2{-}1{-}3$. Note how the previous relations remain unchanged.

The third auxiliary notion is the set $\mathsf{scanned}\,\Omega$ of timestamps. A write's timestamp is placed in $\mathsf{scanned}\,\Omega$, if that write has been observed by some scanner; that is, the written value is returned in some snapshot, or has been rewritten

$$\texttt{write}\ (p, v): \quad \{\chi_{\mathsf{s}} = \emptyset\}\ ()$$
$$\{\exists t.\, \chi'_{\mathsf{s}} = t \mapsto (p, v) \wedge \mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downarrow t\}@\mathcal{J}$$

$$\texttt{scan}: \quad \{\chi_{\mathsf{s}} = \emptyset\}\ (A \times A)$$
$$\{r.\, \exists t.\, \chi'_{\mathsf{s}} = \emptyset \wedge r = \mathsf{eval}\ t\, \Omega'\, \chi' \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t \wedge t \in \mathsf{scanned}\ \Omega'\}@\mathcal{J}$$

Figure 4.4: Snapshot methods specification in FCSL.

by another value that is returned in some snapshot. To illustrate, in the above example, $\{5, 0, 2\} \subseteq \mathsf{scanned}\,\Omega$. Intuitively, we want to model that after a write has been observed, the ordering of the events logically preceding the write must be stabilized, and moreover, must be a sequence. Thus, $\mathsf{scanned}\,\Omega$ is a *linearly ordered subset* of Ω.[2] The set $\mathsf{scanned}\,\Omega$ can also be seen as *representing all the scans that have already been executed*. Such representation of scans allows us to avoid tracking scan events directly in the history.

In the sequel, we concretize Ω and $\mathsf{scanned}\,\Omega$ in terms of ζ and other auxiliary state. However, we keep the notions abstract in the method specs and in client reasoning. This enables the use of different snapshot algorithms, with the same specs, without invalidating the client proofs. We also mention that ζ, Ω and $\mathsf{scanned}\,\Omega$ can be encoded as user-level concepts in FCSL, and require no new logic to be developed.

Snapshot specification. Figure 4.4 presents our specs for `scan` and `write`. These are partial correctness specs that describe how the methods change the state from the precondition (first braces) to the postcondition (second braces), possibly influencing the value r that the procedure returns. We use VDM-style notation with unprimed variables for the state before, and primed variables for the state after the method executes. We use Greek letters for state-dependent values that can be mutated by the method, and Latin letters for immutable variables. The component $\mathcal{J}$ is fine-grained resource (as defined in Chapter 3) that describes the state space of the algorithm, i.e, the invariants on the auxiliary and real state, and the transitions, i.e., the allowed atomic mutations of the state. For now, we keep $\mathcal{J}$ abstract, but will define it in Sections 4.5 and 4.6. We denote by $\Omega \downarrow t$ the downward-closed set of timestamps $\Omega \downarrow t = \{s \mid s\ \Omega\ t\}$. Let $\Omega \downarrow\!\downarrow t = (\Omega \downarrow t) \setminus \{t\}$.

The spec for `write` says the following. The precondition starts with the empty self history χ_{s}, indicating that the procedure has not made any writes. In the postcondition, a new write event $t \mapsto (p, v)$ has been placed into χ'_{s}. Thus, a call to `write` wrote v into pointer p. The timestamp t is fresh, because χ' does not contain duplicate timestamps. Moreover, the write appears as if it occurred

[2] In terminology of linearizability, one may say that $\mathsf{scanned}\,\Omega$ is the set of "linearized" writes.

atomically at time t, thus capturing the logical atomicity of `write`.

The next conjunct, $\mathsf{dom}(\chi_{\mathtt{o}}) \cup \mathsf{scanned}\,\Omega \subseteq \Omega' \downarrow t$, positions the write t into the context of other events. In particular, if $s \in \mathsf{dom}(\chi_{\mathtt{o}})$, *i. e.*, if s finished prior to invoking `write`, then s is logically ordered strictly before t. In other words, `write` cannot reorder prior events that did not overlap with it. The definition of linearizability contains a similar prohibition on reordering non-overlapping events, but here, we capture it using a Hoare-style spec. For similar reasons, we require that $\mathsf{scanned}\,\Omega \subseteq \Omega' \downarrow t$. As mentioned before, $\mathsf{scanned}\,\Omega$ represents all the scans that finished prior to the call to `write`. Consequently, they do not overlap with `write` in real time, and have to be logically ordered before t.

Notice what the spec of `write` *does not prevent*. It is possible that some event, say with a timestamp s, finishes in real time before the call of `write` at time t. Events s and t do not overlap, and hence cannot be reordered; thus $s\ \Omega\ t$ always. However, the relationship of s with other events that ran concurrently with s, may be fixed only later, thus supporting implementation of "future-dependent" nature, such as Jayanti's.

In the case of `scan`, we start and terminate with an empty $\chi_{\mathtt{s}}$, because `scan` does not create any write events, and we do not track scan events. However, when `scan` returns the pair $r = (r_x, r_y)$, we know that there exists a timestamp t that describes when the scan took place. This t is the timestamp of the last write preceding the call to `scan`.

The postcondition says that t is the moment in which the snapshot was logically taken, by the conjunct $r = \mathsf{eval}\ t\ \Omega'\ \chi'$. Here, eval is a pure, specification-level function that works as follows. First, it reorders the entire real-time post-history χ' according to logical post-ordering Ω'. Then, it computes and returns the values of x and y that would result from executing the write events of such reordered history up to the timestamp t. For example, if t is the timestamp of event 1 in Figure 4.3b, then $\mathsf{eval}\ t\ \Omega'\ \chi'$ would return $(2, 1)$. Hence, the conjunct says that `scan` performed a scan of x and y, consistent with the ordering Ω', and returned the read values into r. The scan appears as if it occurred atomically, immediately after time t, thus capturing the atomicity of `scan`.

The next conjunct, $\mathsf{dom}(\chi) \subseteq \Omega' \downarrow t$, says that the scanner returned a snapshot that is current, rather than corresponding to an outdated scan. For example, referring to Figure 4.3, if `scan` is invoked after the events 2 and 1 have already executed, then `scan` should not return the pair $(5, 0)$ and have t be the timestamp of the event 0, because that snapshot is outdated. Specifically, the conjunct says that the write events from χ are ordered no later than t, similar to the postcondition of `write`. However, while in `write` we constrained the events from $\mathsf{dom}(\chi_{\mathtt{o}}) \cup \mathsf{scanned}\,\Omega$, here we constrain the full global history $\chi = \chi_{\mathtt{o}} \cup \chi_{\mathtt{j}}$. The addition of $\chi_{\mathtt{j}}$ shows that the scanner will observe and order all of the write events that have been time-stamped and recorded in $\chi_{\mathtt{j}}$ (and thus, that have written their value to memory), prior to the invocation of `scan`.

Lastly, the conjunct $t \in \mathsf{scanned}\,\Omega'$ explicitly says that t has been observed by

the just finished call to scan.

Again, it is important what the spec does not prevent. It is possible that the timestamp t identified as the moment of the scan, corresponds to a write that has been initiated, but has not yet terminated. Despite being ongoing, t is placed into $\mathsf{scanned}\,\Omega'$ (*i. e.*, t is "linearized"). Also, notice that the postcondition of `scan` actually specifies the "linearization" order of events that are initiated by another method, namely `write`, thus supporting implementations of "non-local" nature, such as Jayanti's.

We close the section with a brief discussion of how the specs are used. Because $\mathcal{J}$, Ω and $\mathsf{scanned}$ are abstracted from the clients, we need to provide an interface to work with them. The interface consists of a number of properties showing how various assertions interact, summarized in the statements below.

The first statement presents the invariants on the transitions of $\mathcal{J}$, often referred to as 2-state invariants. Another way of working with such invariants is to include them in the postcondition of every method.[3] For simplicity, here we agglomerate the properties, and use them implicitly in proofs as needed.

Invariant 4.1 (Transition invariants)**.** In any program respecting the transitions of $\mathcal{J}$, the following properties hold:

1. $\chi \subseteq \chi'$, $\chi_{\mathsf{s}} \subseteq \chi'_{\mathsf{s}}$, and $\chi_{\mathsf{o}} \subseteq \chi'_{\mathsf{o}}$.
2. $\Omega \subseteq \Omega'$ and $\mathsf{scanned}\,\Omega \subseteq \mathsf{scanned}\,\Omega'$.
3. For every $s \in \mathsf{scanned}\,\Omega$, $\Omega \downarrow s = \Omega' \downarrow s$.

Invariant 4.1.1 says that histories only grow, but does not insist that $\chi_{\mathsf{j}} \subseteq \chi'_{\mathsf{j}}$, as timestamps can be removed from χ_{j} and transferred to χ_{s}. Invariant 4.1.2 states that Ω is monotonic, and the same applies for $\mathsf{scanned}\,\Omega$. This is a fundamental stability requirement for our system: no transition from $\mathcal{J}$ can change the relations between write events in Ω and, moreover, write events which have been observed by the scanner— and thus are in $\mathsf{scanned}\,\Omega$— cannot be unobserved. Invariant 4.1.3 says that if a new event is added to increase Ω to Ω', that event appears logically later than any $s \in \mathsf{scanned}\,\Omega$. In other words, once events are observed by a scanner, and placed into $\mathsf{scanned}\,\Omega$ in a certain order, we cannot insert new events among them to modify the past observation.

The second statement exposes the properties of Ω, $\mathsf{scanned}$, and eval that are used for client reasoning:

Invariant 4.2 (Relating $\mathsf{scanned}$ and snapshots)**.** The set $\mathsf{scanned}\ \Omega$ satisfies the following properties:

1. if $t_1 \in \mathsf{scanned}\,\Omega$ and $t_2 \in \mathsf{scanned}\,\Omega$, then $t_1\ \Omega\ t_2 \vee t_2\ \Omega\ t_1$ (linearity).

[3] In fact, this is what we currently do in our Coq files.

2. if $t_2 \in \mathsf{scanned}\,\Omega$ and $t_1\ \Omega\ t_2$, then $t_1 \in \mathsf{scanned}\,\Omega$ (downward closure).

3. if $t \in \mathsf{scanned}\,\Omega$, $\chi \subseteq \chi'$, $\Omega \subseteq \Omega'$, $\mathsf{scanned}\,\Omega \subseteq \mathsf{scanned}\,\Omega'$, and $\Omega{\downarrow}t = \Omega'{\downarrow}t$ then $\mathsf{eval}\,t\,\Omega\,\chi = \mathsf{eval}\,t\,\Omega'\,\chi'$. (snapshot preservation).

The first two properties merely state that the subset $\mathsf{scanned}\,\Omega$ is totally-ordered (4.2.1) and also downward closed (4.2.1). The last property is the most interesting: it entails that once a snapshot is observed by `scan`, its validity will not be compromised by future or ongoing calls to `write`. Thus, snapshots returned from previous calls to `scan` are still valid and observable in the future.

4.4 Client reasoning

Comparison with linearizability specifications. In linearizability one would specify `write` and `scan` by relating them, via a simulation argument, to sequential programs for writing and scanning, respectively. On the face of it, such specs are indeed simpler than ours above, as they merely state that `write` writes and `scan` scans. Our specs capture this property with one conjunct in each postcondition. The remainders of the postconditions describe the relative order of the atomic events, *observed* by threads, including explicit prohibition on reordering non-overlapping events, which is itself inherent in the definition of linearizability.

However, the additional specifications are not pointless, and they become useful when it comes to reasoning about clients. Linearizability tells us that we can simplify a fine-grained client program by replacing the occurrences of `write` and `scan` with the atomic and sequential equivalents, thus turning the client into an equivalent coarse-grained concurrent program. However, linearizability is not directly concerned with verifying that coarse-grained equivalent itself. Then, if one is interested in proving client properties which involve timing and/or ordering properties of such events, it is likely that the simple sequential spec described above do not suffice, and extra auxiliary state is still required.

On the other hand, if one wants to reason about such clients using a Hoare logic, then our specs are immediately useful. Moreover, in our setting, client reasoning depends solely on the API for scan and write, regardless of the different linearizations of a program. In the sequel, we illustrate this claim by deriving interesting client timing properties out of the specs of `write` and `scan`.

Moreover, because we use separation logic, our approach easily supports reasoning about programs with a dynamic number of threads, and about programs that transfer state ownership. In fact, as we already commented in Section 4.3, our proofs rely on transferring write events from $\chi_{\mathtt{j}}$ (joint ownership) to $\chi_{\mathtt{s}}$ (private ownership), upon `write`'s termination. This is immediate in FCSL, as reasoning about histories inherits the infrastructure of the ordinary heap-based separation logic, such as framing and, in this case, ownership transfer. In

contrast, Linearizability is usually considered for a fixed number of threads, and its relationship with ownership transfer is more subtle (Gotsman & Yang, 2012; Cerone *et al.* , 2014).

An additional benefit of specifying the event orders by Hoare triples at the user level, is that one can freely combine methods with different event-ordering properties, that need not respect the constraints of linearizability (Sergey *et al.* , 2016).

Example clients. We first consider the client e, defined as follows:

$$\textbf{do}\ \left(\begin{array}{l}\texttt{write}\ (x,2);\\ \texttt{write}\ (y,1)\end{array}\ \middle\|\ \texttt{scan}\ ()\ \middle\|\ \texttt{write}\ (x,3)\right)$$

It is our running example from Figure 4.2a, implemented no in FCSL notation. Thus, we use the command **do** to acribe the specification below. In the rest of this Chapter, we will omit the concurrent resource $\mathcal{J}$, as it never changes.

$$\begin{array}{l} e\ :\ \{\chi_{\mathsf{s}} = \emptyset\}\ \ (\mathsf{nat} \times \mathsf{nat}) \\ \quad \{r.\, \exists t_1\, t_2\, t_3\, t_s.\ \ \chi'_{\mathsf{s}} = t_1 \mapsto (y,1) \cup t_2 \mapsto (x,2) \cup t_3 \mapsto (x,3) \\ \qquad\qquad\qquad \wedge\, \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t_s \wedge \mathsf{dom}(\chi_{\mathsf{o}}) \subseteq \Omega' \downarrow t_2, \Omega' \downarrow t_3 \\ \qquad\qquad\qquad \wedge\, t_2\ \Omega'\ t_1 \wedge r = \mathsf{eval}\ t_s\ \Omega'\ \chi'\} \end{array}$$

The spec of e states that (1) `write` $(x,2)$, timestamped t_2, occurs sequentially before `write` $(y,1)$ which is timestamped t_1, (2) the remaining write, timestamped t_3, and the scan, timestamped t_s, are not temporally constrained, and (3) the writes that terminated before the client started are ordered before t_2 (and thus before t_1), t_3 and t_s. The example illustrates how to track timestamps and their order, but does not utilize the properties of $\mathsf{scanned}\,\Omega$. We illustrate the latter in another example at the end of this section.

We will show that e satisfies the specification given above. We will first split the composition, and proceed to verify the following subprograms separately:

$$\texttt{scan}\ ()\ \|\ \texttt{write}\ (x,3) \quad \text{and} \quad \texttt{write}\ (x,2);\ \texttt{write}\ (y,1)$$

Then, we will combine them in order to obtain the full proof for e. As proof outlines show intermediate states, in addition to pre- and post-states, we cannot quite utilize VDM notation in them. As a workaround, we explicitly introduce logical variables h and h_o to name (subsets of) the initial global and other history.

1 $\{\chi_{\mathsf{s}} = \emptyset \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$

2a $\{\chi_{\mathsf{s}} = \emptyset \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$ || 2b $\{\chi_{\mathsf{s}} = \emptyset \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$

3a `scan` $()$ || 3b `write` $(x, 3)$

4a $\{r.\ \exists t_s.\ \chi_{\mathsf{s}} = \emptyset \wedge \mathsf{dom}(h) \subseteq \Omega\downarrow t_s \wedge r = \mathsf{eval}\, t_s\, \Omega\, \chi\}$ || 4b $\{\exists t_3.\ \chi_{\mathsf{s}} = t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h_o) \subseteq \Omega \downarrow t_3\}$

5 $\{r.\ \exists t_3\, t_s.\ \chi_{\mathsf{s}} = t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h_o) \subseteq \Omega \downarrow t_3 \wedge \mathsf{dom}(h) \subseteq \Omega\downarrow t_s \wedge r = \mathsf{eval}\ t_s\ \Omega\ \chi\}$

The proof applies the rule PAR for parallel composition of FCSL (Figure 3.5). We have already described this rule in Section 3.1. Here, we just mention that, upon forking, the rule distributes the value of χ_{s} of the parent thread, to the χ_{s} values of its children; in this case, all these are $\emptyset$. Dually, upon joining, the χ_{s} values of the children in lines 4a and 4b, are collected, in line 5, into that of the parent. The other assertions in 4a and 4b directly follow from the specs of `scan` and `write` and the Invariants 4.1.1 and 4.1.2, and directly transfer to line 5. While the proof outline does not establish how `scan` and `write` interleaved, it does however establish the fact that t_3 and t_s both appear after any write event prior to the call to the client.

1 $\{\chi_{\mathsf{s}} = \emptyset \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$
2 `write` $(x, 2)$;
3 $\{\exists t_2.\ \chi_{\mathsf{s}} = t_2 \mapsto (x, 2) \wedge \mathsf{dom}(h_{\mathsf{o}}) \subseteq \Omega \downarrow t_2\}$
4 `write` $(y, 1)$
5 $\{\exists t_1\, t_2.\ \chi_{\mathsf{s}} = t_1 \mapsto (y, 1) \cup t_2 \mapsto (x, 2) \wedge \mathsf{dom}(h_{\mathsf{o}}) \subseteq \Omega \downarrow t_2 \wedge t_2\ \Omega\ t_1\}$

The second proof outline starts with the same precondition. Then, line 3 follows directly from the spec of `write`, using $h_o \subseteq \chi_{\mathsf{o}}$. To proceed, we need to apply FCSL's *framing*: the precondition of `write` mandates an empty self-history $\chi_{\mathsf{s}} = \emptyset$, but we have $\chi_{\mathsf{s}} = t_2 \mapsto (x, 2)$.

The frame rule was introduced in Chapter 3. For the purpose of the proof at hand, the frame rule allows us to generalize the specifications of `write` and `scan` from Figure 4.4. In that figure, both procedures start with the precondition that $\chi_{\mathsf{s}} = \emptyset$. But what do we do if the procedures are invoked by another one which has already completed a number of writes, and thus its χ_{s} is non-empty? By $\circledast$-ing with the frame predicate $R \mathrel{\widehat{=}} (\chi_{\mathsf{s}} = k)$, the frame rule allows us to generalize these specs into ones where the input history equals an arbitrary k:

`write` (p, v) : $\{\chi_{\mathsf{s}} = k\}$ $()$
$\{\exists t.\ \chi'_{\mathsf{s}} = h \cup t \mapsto (p, v) \wedge \mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downarrow t\}$
`scan` : $\{\chi_{\mathsf{s}} = k\}$ $(\mathsf{nat} \times \mathsf{nat})$
$\{r.\ \exists t.\ \chi'_{\mathsf{s}} = k \wedge r = \mathsf{eval}\ t\, \Omega'\, \chi' \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t \wedge t \in \mathsf{scanned}\, \Omega'\}$

Thus, *framing* alllows us to modify the spec of `write` by joining $t_2 \mapsto (x, 2)$ to χ_s, χ'_s *and* χ_o as follows.

$$\begin{aligned}\texttt{write}\ (p, v):\ & \{\chi_\mathsf{s} = t_2 \mapsto (x, 2)\}\ ()\\ & \{\exists t.\, \chi'_\mathsf{s} = t \mapsto (p, v) \cup t_2 \mapsto (x, 2)\\ & \qquad \wedge \mathsf{dom}(\chi_\mathsf{o} \cup t_2 \mapsto (x, 2)) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downarrow t\}\end{aligned}$$

Such a framed spec for `write` gives us that after line 4: (1) $\chi_\mathsf{s} = t_1 \mapsto (y, 1) \cup t_2 \mapsto (x, 2)$, and (2) $\mathsf{dom}(h_\mathsf{o} \cup t_2 \mapsto (x, 2)) \subseteq \Omega \downarrow t_1$. From Invariants 4.1, we also obtain that (3) $\mathsf{dom}(h_\mathsf{o}) \subseteq \Omega \downarrow t_2$, which simply transfers from line 3. Now, in the presence of (2), we can simplify (3) into $t_2\ \Omega\ t_1$, thus obtaining the postcondition in line 5.

The final step applies the rule for parallel composition to the two derivations, splitting χ_s upon forking, and collecting it upon joining:

$$\begin{aligned}e:\ & \{\chi_\mathsf{s} = \emptyset \wedge h \subseteq \chi \wedge h_o \subseteq \chi_\mathsf{o}\}\ (\mathsf{nat} \times \mathsf{nat})\\ & \{r.\, \exists\, t_1\, t_2\, t_3\, t_s.\, \chi_\mathsf{s} = t_1 \mapsto (y, 1) \cup t_2 \mapsto (x, 2) \cup t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t_s\\ & \qquad \wedge \mathsf{dom}(h_\mathsf{o}) \subseteq \Omega \downarrow t_2, \Omega \downarrow t_3 \wedge t_2\ \Omega\ t_1 \wedge r = \mathsf{eval}\, t_s\, \Omega\, \chi\}\end{aligned}$$

From here, the VDM spec of e is derived by priming the Greek letters in the postcondition, and choosing $h = \chi$ and $h_o = \chi_\mathsf{o}$.

The spec of e can be further used in various contexts. For example, to recover the context from Section 4.2, where e is invoked with $x = 5$, $y = 0$, we can frame e wrt. $\chi_\mathsf{s} = t_5 \mapsto (x, 5) \cup t_0 \mapsto (y, 0)$ to make explicit the events that initialize x and y. Then, it is possible to derive in FCSL that if e executes without interference (*i. e.*, if $\chi = \chi_\mathsf{o} = \chi' = \chi'_\mathsf{o} = \emptyset$), then the result at the end must be $r \in \{(5, 0), (2, 0), (3, 0), (2, 1), (3, 1)\}$. As expected, $r \neq (5, 1)$, because the write of 2 sequentially precedes the write of 1.

We next illustrate the use of Invariants 4.2, which are required for clients that use `scan` in *sequential composition*. We consider the program

$$e' = \mathbf{do}\ \ (r \leftarrow \texttt{scan};\ \texttt{write}\,(x, v);\ \mathbf{return}\, r)$$

and prove that e' can be ascribed the following spec:

$$\begin{aligned}e':\ & \{\chi_\mathsf{s} = \emptyset\}\ (\mathsf{nat} \times \mathsf{nat})\\ & \{\exists\, t_s\ t_x.\ \chi'_\mathsf{s} = t_x \mapsto (x, v) \wedge t_s \in \Omega' \downarrow t_x \wedge r = \mathsf{eval}\ t_s\ \Omega'\ \chi'\}\end{aligned}$$

The spec says that the write event (t_x) is subsequent to the scan (t_s), as one would expect. In particular, the snapshot r remains valid, *i. e.*, the write does not change the order Ω and history χ in a way that makes r cease to be a valid

snapshot in Ω' and χ'. The proof outline follows, with the explanation of the critical steps.

1 $\{\chi_{\mathtt{s}} = \emptyset\}$
2 $r \leftarrow$ `scan`;
3 $\{\exists\, t_s, w'(=\Omega), h'(=\chi).\ \chi_{\mathtt{s}} = \emptyset \wedge t_s \in \mathsf{scanned}\, w' \wedge r = \mathsf{eval}\ t_s\ w'\ h'\}$
4 `write` (x, v);
5 $\{\exists\, t_s\, t_x.\ \chi_{\mathtt{s}} = t_x \mapsto (x, v) \wedge t_s \in \Omega \downarrow t_x \wedge t_s \in \mathsf{scanned}\, \Omega \wedge r = \mathsf{eval}\ t_s\ \Omega\ \chi\}$
6 **return** r

Line 3 is a direct consequence of the spec of `scan`, where we omitted the conjunct $\mathsf{dom}(\chi) \subseteq \Omega' \downarrow t_s$, as we do not need it for the subsequent derivation. We also introduce explicit names w' and h' for the current values of Ω and χ. Now, to derive line 5, by the spec of `write`, we know there exists a timestamp t_x corresponding to the write, such that (1) $\chi_{\mathtt{s}} = t_x \mapsto (x, v)$, which is a conjunct in line 5, and also (2) $\mathsf{dom}(\chi_{\mathtt{o}}) \cup \mathsf{scanned}\, w' \subseteq \Omega \downarrow t_x$. Furthermore, (3) $t_s \in \mathsf{scanned}\, w'$, and (4) $r = \mathsf{eval}\ t_s\ w'\ h'$, simply transfer from line 3. From (2) and (3), we infer that $t_s \in \Omega \downarrow t_x$. To complete the derivation of line 5, it remains to show that $t_s \in \mathsf{scanned}\, \Omega$ and $r = \mathsf{eval}\ t_s\ \Omega\ \chi$. For this, we use (3), (4) and the Invariants 4.1 and 4.2, as follows. First, by Invariant 4.1.3, and because $t_s \in \mathsf{scanned}\, w'$, we get $w' \downarrow t_s = \Omega \downarrow t_s$. By Invariant 4.1.2, this gives us $t_s \in \mathsf{scanned}\, \Omega$ as well. By Invariant 4.1.1, $h' \subseteq \chi$, and then by Invariant 4.2.3, $r = \mathsf{eval}\ t_s\ w'\ h' = \mathsf{eval}\ t_s\ \Omega\ \chi$, completing the deduction of line 5.

Observe that the main role of $\mathsf{scanned}$ in proofs is to enable showing *stability* of values obtained by eval, using Invariant 4.2.3. The remaining Invariants 4.2.1 and 4.2.2 allow us to replace a number of conjuncts about $\mathsf{scanned}$ by a single one that expresses the membership of the largest timestamp in the current $\mathsf{scanned}$ set.

4.5 Internal auxiliary state

In order to verify the *implementations* of `write` and `scan`, we require further auxiliary state that does *not* feature in the specifications, and is thus hidden from the clients.

First, we track the point of execution in which `write` and `scan` are, but instead of line numbers, we use datatypes to encode extra information in the constructors. For example, the scanner's state is a triple (S_s, S_x, S_y). S_s is drawn from $\{\mathsf{S_{On}}, \mathsf{S_{Off}}\ t\}$. If $\mathsf{S_{On}}$, then the scanner is in lines 7–11 in Figure 4.1. If $\mathsf{S_{Off}}\ t$, the the scanner reached line 12 at "time" t, and is now in 13–17. S_x is a Boolean bit, set when the scanner clears fx in line 8, and reset upon scanner's termination (dually for S_y and fy). Writers' state for x is tracked by the auxiliary W_x (dually, W_y). These are drawn from $\{\mathsf{W_{Off}}, \mathsf{New}\ t\, v, \mathsf{Fwd}\ t\, v, \mathsf{Done}\ t\, v\}$, where t marks the

beginning of the write and v is the value written to pointer p. If $\mathsf{W}_{\mathsf{Off}}$, then no write is in progress. If $\mathsf{New}\ t\, v$, then the writer is in line 2. If $\mathsf{Fwd}\ t\, v$, then b has been set in line 3, triggering forwarding. If $\mathsf{Done}\ t\, v$, the writer is free to exit.

Second, like in linearizability, we record the ending times of terminated events, using an auxiliary variable τ. τ is a function that takes a timestamp identifying the beginning of some event, and returns the ending time of that event, and is undefined if the event has not terminated. However, we do not *generate* fresh timestamps to mark event ending times. Instead, at the end of `write`, we simply read off the last used timestamp in χ, and use it as the ending time of `write`. This is a somewhat non-standard way of keeping time, but it suffices to prove that events t_1 and t_2 which are non-overlapping (*i. e.*, $\tau(t_1) < t_2$ or $\tau(t_2) < t_1$) are never reordered. The latter is required by the postconditions of `write` and `scan`, as we discussed in Section 4.3. Formally, the following is an *invariant* of the snapshot object; *i. e.*, one of the properties that define the the state space $\mathsf{Coh}_{\mathcal{J}}$ of the fine-grained resource $\mathcal{J}$ from Figure 4.4, preserved by $\mathcal{J}$'s transition.

Invariant 4.3. The logical order $<_\zeta$ preserves the real time order of non-overlapping events: $\forall t_1 \in \mathsf{dom}(\tau), t_2 \in \mathsf{dom}(\chi)$, if $\tau(t_1) < t_2$ then $t_1 <_\zeta t_2$.

Third, we track the rearrangement status of write events wrt. an ongoing *active* scan, by *colours*. A scan is *active* if it has cleared the forwarding pointers in lines 8 and 9, and is ready to read x and y. We keep the auxiliary variable κ, which is a function mapping each timestamp in χ to a colour, as follows.

- Green timestamps identify write events whose position in the logical order is fixed in the following sense: if $\kappa(t_1) = \mathsf{green}$ and $t_1 <_\zeta t_2$, then $t_1 <_{\zeta'} t_2$ for every ζ' to which ζ may step by auxiliary code execution (Section 4.6). For example, since we only reorder overlapping events, and only the scanner reorders events, every event that finished before the active scan started will be green. Also, a green timestamp never changes its colour.

- Red timestamps identify events whose order is not fixed, but which will *not* be manipulated by the active scan, and are left for the next scan.

- Yellow timestamps identify events whose order is not fixed yet, but which *may* be manipulated by the ongoing active scan, as follows. The scan can *push* a yellow timestamp in logical time, *past* another green or yellow timestamp, but not past a red one. *This is the only way the logical ordering can be modified.*

There are a number of invariants that relate colours and timestamps. We next list the ones that are most important for understanding our proof. We use χ_p to denote the sequence of writes into the pointer p that appear in the history χ,

sorted by their order in ζ[4].

Invariant 4.4 (Colors)**.** The colours of χ_p are described by the regular expression $\mathbf{g}^+\mathbf{y}^?\mathbf{r}^*$: there is a non-empty prefix of green timestamps, followed by *at most* one yellow, and arbitrary number of reds.

By the above invariant, the yellow colour identifies the write event into the pointer p, that is the *unique* candidate for reordering by the ongoing active scan. Moreover, all the writes into p prior to the yellow write, will have already been coloured green (and thus, fixed in time), whether they overlapped with the scanner or not.

Invariant 4.5 (Color of forwarded values)**.** Let $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}$, and $p \in \{x, y\}$, and $S_p = \mathsf{True}$ (*i. e.*, scanner is in lines 13–16), and $v \neq \bot$ has been forwarded to p; *i. e.*, $\texttt{fwd}\ p \mapsto v$. Then the event of writing v into p is in the history, *i. e.*, there exists t such that $t \mapsto (p, v) \in \chi_p$. Moreover, t is the last green, or the yellow timestamp in χ_p.

The above invariant restricts the set of events that could have forwarded a value to the scanner, to only two: the event with the (unique) yellow timestamp, or the one corresponding to the last green timestamp. By Invariant 4.4, these two timestamps are consecutive in χ_p.

Invariant 4.6 (Red zone)**.** If $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, then χ satisfies the $(\mathbf{g}|\mathbf{y})^+\mathbf{r}^*$ pattern. Moreover, for every $t \in \mathsf{dom}(\chi)$:

- $\kappa(t) = \mathsf{green} \implies t \leq t_{\mathsf{off}}$
- $\kappa(t) = \mathsf{yellow} \implies t \leq t_{\mathsf{off}} \leq \tau(t)$
- $\kappa(t) = \mathsf{red} \implies t_{\mathsf{off}} < t$

This invariant restricts the global history χ (not the pointer-wise projections χ_p). First, the red events in χ are consecutive, and cannot be interspersed among green and yellow events. Thus, when a scanner pushes a yellow event past a green event, or past another yellow event, it will not "jump over" any reds. Second, the invariant relates the colours to the time t_{off} at which the scanner was turned off (in line 12, Figure 4.1). This moment is important for the algorithm; *e. g.*, it is the linearization point for `scan` in Jayanti's proof (Jayanti, 2005). We will use the above inequalities wrt. t_{off} in our proofs, to establish that the events reordered by the scanner *do* overlap, as per Invariant 4.3.

We can now define the stable logical order Ω, and the set $\mathsf{scanned}\,\Omega$, using the internal auxiliary state of colours and ending times.

[4]For reasoning purposes, it serves us better to think of χ_p as sub-histories, with an *e*xternal ordering given by ζ. We do, however, implement χ_p as a list filter: $\chi_p = \mathsf{filter}\ (\lambda t.\, t \mapsto (p, \_) \in \chi)\ \zeta$.

Definition 4.7 (Logical order Ω and $\mathsf{scanned}\,\Omega$)**.** We define the stable order Ω, and the $\mathsf{scanned}\,\Omega$ assertion as follows:

1. $t_1\,\Omega\,t_2 \;\widehat{=}\; (t_1 = t_2) \vee (\tau(t_1) < t_2) \vee (t_1 <_\zeta t_2 \wedge \kappa(t_1) = \mathsf{green})$

2. $\mathsf{scanned}\,\Omega = \{t \mid \Omega{\downarrow}t = {\leq_\zeta}{\downarrow}t \wedge \forall s \in \Omega{\downarrow}t.\, \kappa(s) = \mathsf{green}\}$.

From the definition of Ω, notice that $t_1\,\Omega\,t_2$ is stable (*i. e.*, invariant under interference), since threads do not change the ending times τ, the colour of green events, or the order of green events in $<_\zeta$, as we already discussed. From the definition of $\mathsf{scanned}\,\Omega$, notice that for every $t \in \mathsf{scanned}\,\Omega$, it must be that $\Omega{\downarrow}t$ is a linearly-ordered set wrt. Ω, because it equals a prefix of the *sequence* ζ.

We close this section with a few technical invariants that we use in the sequel.

Invariant 4.8 (Last write)**.** Let pointer $p \in \{x, y\}$, and $\mathsf{last}_\zeta\ \chi_p$ be the timestamp in χ_p that is largest wrt. the logical order $\leq_\zeta$. Then the contents of p equals the value written by the event associated with $\mathsf{last}_\zeta\ \chi_p$. That is, $p \mapsto \chi_p(\mathsf{last}_\zeta \chi_p)$.

Invariant 4.9 (Joint history)**.** Let pointer $p \in \{x, y\}$. If the writer for p is active *i. e.* $W_p \neq \mathsf{W_{Off}}$, then the write event that it is performing is time-stamped and placed into joint history χ_j. Dually, if $t \in \mathsf{dom}(\chi_\mathsf{j})$, then the event t is performed by the active writer for p:

$$t \mapsto (p, v) \in \chi_\mathsf{j} \iff W_p = \mathsf{New}\ t\ v \vee W_p = \mathsf{Fwd}\ t\ v \vee W_p = \mathsf{Done}\ t\ v$$

Invariant 4.10 (Terminated events)**.** Histories χ_o and χ_s store only terminated events, *i. e.*, events whose ending times are recorded in τ. Moreover, the co-domain of τ is bounded by the maximal timestamp, in real time, in $\mathsf{dom}(\chi)$:

1. $\mathsf{dom}(\tau) = \mathsf{dom}(\chi_\mathsf{s}) \uplus \mathsf{dom}(\chi_\mathsf{o})$.

2. $\forall a \in \mathsf{dom}(\tau).\ \tau(a) \leq \mathsf{max}\ (\mathsf{dom}(\chi))$.

Lemma 4.11 (Green/yellow read values)**.** *Let $p \in \{x, y\}$. If the scanner state is $S_s = \mathsf{S_{On}}, S_p = \mathsf{True}$,* i. e.*, the scanner is between lines 10–11 in Figure 4.1, and $p \mapsto v$ in the physical heap, then exists t such that $t \mapsto (p, v) \in \chi_p$. Moreover, t is the last green or the yellow timestamp in χ_p.*

Lemma 4.12 (Chain)**.** *If $t \in \mathsf{dom}(\chi)$ and $\kappa({\leq_\zeta}{\downarrow}t) = \mathsf{green}$, then $\Omega{\downarrow}t \;=\; {\leq_\zeta}{\downarrow}t$.*

.

```
1   write (p, v) {
2     ⟨ p := v; register(p, v)⟩;
3     ⟨ b ← read(S);  check(p, b)⟩;
4     if b
5     then ⟨ fwd p := v;  forward(p)⟩;
5'    ⟨ finalize(p)⟩}

6   scan() : (A × A) {
7     ⟨ S := true;  set(true)⟩;
8     ⟨ fx := ⊥;  clear(x)⟩;
9     ⟨ fy := ⊥;  clear(y)⟩;
10    vx ← ⟨read(x)⟩;
11    vy ← ⟨read(y)⟩;
12    ⟨ S := false;  set(false)⟩;
13    ox ← ⟨read(fx)⟩;
14    oy ← ⟨read(fy)⟩;
15    rx ← if (ox ≠ ⊥) then ox else vx;
16    ry ← if (oy ≠ ⊥) then oy else vy;
17    ⟨ relink(rx, ry);  ret (rx, ry) ⟩}
```

Figure 4.5: Snapshot procedures annotated with auxiliary code.

4.6 Auxiliary code implementation

Figure 4.5 annotates Jayanti's procedures with auxiliary code (typed in *italic*), with ⟨angle braces⟩ denoting that the enclosed real and auxiliary code execute *simultaneously* (*i. e.*, atomically). The auxiliary code builds the histories, evolves the sequence ζ, and updates the colour of various write events, while respecting the invariants from Section 4.3. Thus, it is the *constructive* component of our proofs. Each atomic command in Figure 4.5 represents one *transition* of the STS C from Figure 4.4.

The auxiliary code is divided into several procedures, all of which are sequences of reads followed by updates to auxiliary variables. We present them as Hoare triples in Figure 4.6, with the unmentioned state considered unchanged. The bracketed variables preceding the triples (*e. g.*, $[t, v]$) are logical variables used to show how the pre-state value of some auxiliary changes in the post-state. To symbolize that these triples *define* an atomic command, rather than merely stating the command's properties, we enclose the pre- and postcondition in angle brackets $\langle - \rangle$.

4.6.1 Auxiliary code for `write`.

In line 2, *register*(p, v) creates the write event for the assignment of v to p. It allocates a *fresh* timestamp t, inserts the entry $t \mapsto (p, v)$ into χ_j, and adds t to the end of ζ, thus registering t as the currently latest write event. The fresh timestamp t is computed out of the history χ; we take the largest natural number

$$
\begin{array}{lll}
\mathit{register}(p,v) & : & \langle W_p = \mathsf{W_{Off}}\rangle \\
& & \langle \zeta' = \mathsf{snoc}\ \zeta\ t,\ \chi'_{\mathsf{j}} = \chi_{\mathsf{j}} \uplus t \mapsto (p,v),\ W'_p = \mathsf{New}\ t\ v, \\
& & \ \kappa' = \text{if } (S_s = \mathsf{S_{On}})\&S_p \text{ then } \kappa[t \mapsto \mathsf{yellow}] \text{ else } \kappa[t \mapsto \mathsf{red}]\rangle \\
& & \text{where } t = \mathsf{fresh}\ \chi = \mathsf{max}\,(\mathsf{dom}(\chi)) + 1 \\
\mathit{check}\,(p,b) & : [t,v].\ \langle W_p = \mathsf{New}\ t\ v\rangle\ \langle W'_p = \text{if } b \text{ then } \mathsf{Fwd}\ t\ v \text{ else } \mathsf{Done}\ t\ v\rangle \\
\mathit{forward}\,(p) & : [t,v].\ \langle W_p = \mathsf{Fwd}\ t\ v\rangle \\
& & \langle W'_p = \mathsf{Done}\ t\ v,\ \kappa' = \text{if } (S_s = \mathsf{S_{On}})\&S_p \text{ then } \kappa[t \mapsto \mathsf{green}] \text{ else } \kappa\rangle \\
\mathit{finalize}(p) & : [t,v].\ \langle W_p = \mathsf{Done}\ t\ v, t \mapsto (p,v) \in \chi_{\mathsf{j}}\rangle \\
& & \langle W'_p = \mathsf{W_{Off}},\ \chi'_{\mathsf{s}} = \chi_{\mathsf{s}} \uplus t \mapsto (p,v), \\
& & \ \chi'_{\mathsf{j}} = \chi_{\mathsf{j}} \setminus \{t\},\ \tau' = \tau \uplus t \mapsto \mathsf{max}\,(\mathsf{dom}(\chi))\rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{set}(b) & : & \langle S_s = \text{if } b \text{ then } \mathsf{S_{Off}}\,(\_) \text{ else } \mathsf{S_{On}},\ S_x = \neg\, b, S_y = \neg\, b\rangle \\
& & \langle S'_s = \text{if } b \text{ then } \mathsf{S_{On}} \text{ else } \mathsf{S_{Off}}\,(\mathsf{last}\ \chi),\ S'_x = \neg\, b, S'_y = \neg\, b\rangle \\
\mathit{clear}(p) & : & \langle S_s = \mathsf{S_{On}},\ S_p = \mathsf{False}\rangle \\
& & \langle S'_s = \mathsf{S_{On}},\ S'_p = \mathsf{True},\ \kappa' = \kappa[\chi_p \mapsto \mathsf{green}]\rangle \\
\mathit{relink}(r_x, r_y) & : [t_x, t_y]. & \langle S_s = \mathsf{S_{Off}}(\_), t_x \mapsto (x, r_x), t_y \mapsto (y, r_y) \in \chi, S_x = S_y = \mathsf{True}, \\
& & \ \forall p \in \{x,y\}.\, \mathsf{lastGY}\ p\, t_p\rangle \\
& & \langle S'_s = S_s, S'_x = S'_y = \mathsf{False}, \kappa' = \kappa[t_x, t_y \mapsto \mathsf{green}], \\
& & \ \zeta' = \text{if } (d = \mathsf{Yes}\ x\ s) \text{ then } \mathit{push}\ s\ t_y\ \zeta \\
& & \qquad \text{else if } (d = \mathsf{Yes}\ y\ s) \text{ then } \mathit{push}\ s\ t_x\ \zeta \text{ else } \zeta\rangle \\
& & \ \text{where } d = \mathit{inspect}\ t_x\, t_y\, \zeta\, \kappa
\end{array}
$$

Figure 4.6: Auxiliary procedures for `write` and `scan`. Bracketed variables (*e. g.*, $[t,v]$) are logical variables that scope over precondition and postcondition.

occurring as a timestamp in χ, and increment it by 1. The variable W_p updates the writer's state to indicate that the writer finished line 2 with the timestamp t allocated, and the value v written into p. The col-or of t is set to yellow (*i. e.*, the order of t is left undetermined), but only if $(S_s = \mathsf{S_{On}})\&S_p$ (*i. e.*, an active scanner is in line 10). Otherwise, t is coloured red, indicating that the order of t will be determined by a future scan.

In line 3, $\mathit{check}(p,b)$, depending on b, sets the writer state to Fwd, indicating that a scan is in progress, and the writer should forward, or to Done, indicating that the writer is ready to terminate.

In line 5, *forward* colours the allocated timestamp t green, if an active scanner has passed lines 8–9 and is yet to reach line 12, because such a scanner will definitely see the write, either by reading the original value in lines 10–11, or by reading the forwarded value in lines 13–14. Thus, the logical order of t becomes

fixed. In fact, it is possible to derive from the invariants in Section 4.3, that this order is the same one t was assigned at registration, *i. e.*, the linearization point of this write is line 2.

In line 5', *finalize* moves the write event t from the joint history χ_{j} to the thread's self history χ_{s}, thus acknowledging that t has terminated. The currently largest timestamp of χ is recorded in τ as t's ending time. By definition of Ω, all the writes that terminated before t in real time, will be ordered before t in Ω.

4.6.2 Auxiliary code for `scan`.

The method *set* toggles the scanner state S_s on and off. When executed in line 12, it returns the timestamp t_{off} that is currently maximal in real time, as the moment when the scanner is turned off. Note that this does not create a fresh timestamp, but rather selects that of the last write event in χ.

The procedure *clear*(p) is executed in lines 8–9 simultaneously with clearing the forwarding pointer for p. In addition to recording that the scanner passed lines 8 or respectively 9, by setting the S_p bit, it colours the sub-history χ_p green. Thus, by definition of $\mathsf{scanned}\,\Omega$, the ongoing one and all previous writes to p are recorded as scanned, and thus linearized.

Finally, the key auxiliary procedure of our approach is *relink*. It is executed at line 17 just before the scanner returns the pair (r_x, r_y). Its task is to modify the logical order of the writes, to make (r_x, r_y) *appear* as a valid snapshot. This will always be possible under the precondition of *relink* that the timestamps t_x, t_y of the events that wrote r_x, r_y respectively, are either the last green or the yellow ones in the respective histories χ_x and χ_y, and *relink* will consider all four cases. This precondition holds after line 16 in Figure 4.5, as one can prove from Invariants 4.4 and 4.5. In the precondition we introduce the following abbreviation:

$$\mathsf{lastGY}\ p\,t \ \widehat{=}\ t = \mathsf{last\_green}_{\zeta}\,\chi_p \vee \kappa(t) = \mathsf{yellow} \tag{4.2}$$

Relink uses two helper procedures *inspect* and *push*, to change the logical order. *Inspect* decides if the selected t_x and t_y determine a valid snapshot, and *push* performs the actual reordering. The snapshot determined by t_x and t_y is valid if there is no event s such that $t_x <_\zeta s <_\zeta t_y$ and s is a write to x (or, symmetrically $t_y <_\zeta s <_\zeta t_x$, and s is a write to y). If such s exists, *inspect* returns $\mathsf{Yes}\ x\ s$ (or $\mathsf{Yes}\ y\ s$ in the symmetric case). The reordering is completed by *push*, which moves s right after t_y (after t_x in the symmetric case) in $\leq_\zeta$. Finally, *relink* colours t_x and t_y green, to fix them in Ω. We can then prove that (r_x, r_y) is a valid snapshot wrt. Ω, and remains so under interference. Notice that the timestamp s returned by *inspect* is always uniquely determined, and yellow. Indeed, since t_x and t_y are not red, no timestamp between them can be red either (Invariant 4.6). If $t_x <_\zeta s <_\zeta t_y$ and s is a write to x (and the other case is symmetric), then t_x must be the last green in χ_x, forcing s to be the unique yellow timestamp in χ_x, by

Invariant 4.4.

To illustrate, in Figure 4.3a we have $r_x = 2$, $r_y = 1$, t_x and t_y are both the last green timestamp of χ_x and χ_y, respectively, and $t_x <_\zeta t_y$. However, there is a yellow timestamp s in χ_x coming after t_x, encoding a write of 3. Because $t_x <_\zeta s <_\zeta t_y$, the pair (r_x, r_y) is not a valid snapshot, thus *inspect* returns $\mathsf{Yes}\ x\ s$, after which *push* moves 3 after 1.

We have omitted the definitions of *inspect* and *push* for the time being. These are presented as an aside (Section 4.7). Rather, we prefer to bring forwards at this point the main properties of *relink*, whose proofs can be found in our Coq files (Delbianco *et al.* , 2017b).

Lemma 4.13 (Main property of *relink*)**.** *Let the precondition of relink hold,* i. e., $S_s = \mathsf{S_{Off}}(\_)$, $t_x \mapsto (x, r_x), t_y \mapsto (y, r_y) \in \chi$, $S_x = S_y = \mathsf{True}$, *and* $\forall p \in \{x, y\}.\, \mathsf{lastGY}\ p\ t_p$. *Then the ending state of relink satisfies the following:*

1. *For all* $p \in \{x, y\}$, $t_p = \mathsf{last\_green}_{\zeta'}\ \chi'_p$.
2. *Let* $t = \mathsf{max}_{\zeta'}(t_x, t_y)$. *Then for every* $s \leq_{\zeta'} t$, $\kappa'(s) = \mathsf{green}$.

4.7 Aside: Auxiliary definitions for *relink*

In Section 4.6, we described briefly the implementation of *relink*, without giving much details on the auxiliary helper functions *inspect* and *push*. We give here their definitions, together with some associated properties:

Definition 4.14 (inspect)**.** Given two timestamps t_x, t_y then $\mathit{inspect}\ t_x\ t_y\ \zeta\ \kappa$ is defined as follows:

$$\mathit{inspect}\ t_x\ t_y\ \kappa\ \widehat{=}\ \begin{cases} \mathsf{Yes}\ x\ t_z & \text{if } t_x <_\zeta t_y,\ t_x = \mathsf{last\_green}_\zeta\ \chi_x, \\ & t_z = \mathsf{yellow\_timestamp}_\zeta\ \chi_x, \text{ and } t_z <_\zeta t_y \\ \mathsf{Yes}\ y\ t_z & \text{if } t_y <_\zeta t_x,\ t_y = \mathsf{last\_green}_\zeta\ \chi_y, \\ & t_z = \mathsf{yellow\_timestamp}_\zeta\ \chi_y, \text{ and } t_z <_\zeta t_x \\ \mathsf{No} & \text{otherwise} \end{cases}$$

Definition 4.15 (push)**.** *push* is a surgery operation defined on ζ as follows:

$$\text{Let } \zeta = \zeta_{<_i} +\!\!+\ i\ +\!\!+\ \zeta_{i..j} +\!\!+\ j\ +\!\!+\ \zeta_{>_j}, \text{ then } \mathit{push}\ i\ j\ \zeta = \zeta_{<_i} +\!\!+\ \zeta_{i..j} +\!\!+\ j\ +\!\!+\ i\ +\!\!+ \zeta_{>_j}$$

The definition of *inspect* works under the assumption that t_x and t_y are, respectively, the last green or yellow timestamp in χ_x and χ_y. This latter fact is recovered in the definition of *relink* in Figure 4.6 and reinforced in line 21 in the proof of `scan` in Figure 4.8. When *inspect* returns $\mathsf{Yes}\ p\ t_z$, ζ' is computed by pushing some i timestamp past another timestamp j in ζ. The definition of *push*

above shows that this operation is an algebraic manipulation on sequences. In fact, we implement it using standard *surgery* operations on lists: $+\!\!+$, take, *etc.*.

In Section 4.6, we have mentioned that the correctness aspect of auxiliary code involves proving that the code preserves the auxiliary state invariants from Section 4.5. For example, the correctness proof of *relink*, relies on the following helper lemmas. The first lemma asserts that *inspect* correctly determines the "offending" timestamp; the second and the third lemma assert that *push* modifies ζ in a way that allows us to prove (in Section 4.8), that the pair (r_x, r_y) a valid snapshot.

Lemma 4.16 (Correctness of *inspect*)**.** *If t_x, t_y are timestamps for write events of r_x, r_y, then inspect t_x t_y ζ κ correctly determines that (r_x, r_y) is a valid snapshot under ordering $<_\zeta$ and colours κ, or otherwise returns the "offending" timestamp. More formally, if $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, and for each $p \in \{x, y\}$, $t_p \mapsto (p, r_p) \in \chi$ and $\mathsf{lastGY}\ p\, t_p$, the following are exhaustive possibilities.*

1. *If $t_x <_\zeta t_y$ and $\kappa(t_x) = \mathsf{yellow}$, then inspect t_x t_y ζ $\kappa = \mathsf{No}$. Symmetrically for $t_y <_\zeta t_x$.*
2. *If $t_x <_\zeta t_y$, $t_x = \mathit{last\_green}\ \chi_x$, and $\forall s \in \chi_x.\ t_x <_\zeta s \implies t_y <_\zeta s$, then inspect $t_x\, t_y\, \zeta\, \kappa = \mathsf{No}$. Symmetrically for $t_y <_\zeta t_x$.*
3. *If $t_x <_\zeta t_y$, $t_x = \mathit{last\_green}\ \chi_x$, $s \in \chi_x$, and $t_x <_\zeta s <_\zeta t_y$, it follows that inspect $t_x\, t_y \zeta \kappa = \mathsf{Yes}\, x\, s$ and $\kappa(s) = \mathsf{yellow}$. Symmetrically for $t_y <_\zeta t_x$.*

Lemma 4.17 (Push Mono)**.** *Given elements a, b, i, j, all in ζ, and $\zeta' = \mathit{push}\ i\ j\ \zeta$, then:*

1. *If $a <_\zeta i$ then $a <_\zeta b \implies a <'_\zeta b$.*
2. *If $j <_\zeta b$ then $a <_\zeta b \implies a <'_\zeta b$.*
3. *If $a \neq i$ then $a <_\zeta b \implies a <'_\zeta b$*

Lemma 4.18 (Correctness of *push*)**.** *Given $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = S_y = \mathsf{True}$, and for $p \in \{x, y\}$, we have $t_p \mapsto (p, r_p) \in \chi$, $\mathsf{lastGY}\ p\, t_p$, and inspect t_x t_y ζ $\kappa = \mathsf{Yes}\ p\ t_s$. If we name $t_z \in \{t_x, t_y\}$, with $p \neq z$, and $\zeta' = \mathit{push}\ t_s\ t_z\ \zeta$, then:*

1. *relink satisfies the 2-state invariants from Invariant 4.1.*
2. *$\chi', \zeta', \tau', \kappa'$ satisfies all the resource invariants from Section 4.5,* i. e. *Invariants 4.3–4.10.*

In our mechanization, these three lemmas allow us to prove Lemma 4.13, *relink*'s main property.

1 $\{\chi_{\mathsf{s}} = \emptyset \land w \subseteq \Omega \land h \subseteq \chi \land h_o \subseteq \chi_{\mathsf{o}}\}$
2 $\{\chi_{\mathsf{s}} = \emptyset \land W_p = \mathsf{W_{Off}} \land w \subseteq \Omega \land h \subseteq \chi \land h_o \subseteq \chi_{\mathsf{o}}\}$
3 $\langle p := v; \mathit{register}(v)\rangle;$
4 $\{\exists t\,.\ \chi_{\mathsf{s}} = \emptyset \land W_p = \mathsf{New}\ t\ v \land t \mapsto (p, v) \in \chi_{\mathsf{j}} \land$
$\mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\ w \subseteq \Omega \downarrow t\}$
5 $\langle b \leftarrow \mathsf{read}(S);\ \mathit{check}(p, b)\rangle;$
6 $\{\exists t.\ \chi_{\mathsf{s}} = \emptyset \land W_p = \mathsf{if}\ b\ \mathsf{then}\ \mathsf{Fwd}\ t\ v\ \mathsf{else}\ \mathsf{Done}\ t\ v \land t \mapsto (p, v) \in \chi_{\mathsf{j}} \land$
$\mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\ w \subseteq \Omega \downarrow t\}$
7 **if** b **then** $\langle$ `fwd` $p := v;\ \mathit{forward}(p, v)\rangle;$
8 $\{\exists t.\ \chi_{\mathsf{s}} = \emptyset \land W_p = \mathsf{Done}\ t\ v \land t \mapsto (p, v) \in \chi_{\mathsf{j}} \land$
$\mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\ w \subseteq \Omega \downarrow t\}$
9 $\langle \mathit{finalize}(i, v)\rangle$
10 $\{\exists t.\ \chi_{\mathsf{s}} = t \mapsto (p, v) \land \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\ w \subseteq \Omega \downarrow t\}$

Figure 4.7: Proof outline for `write`.

4.8 Correctness

We can now show that `write` and `scan` satisfy the specifications from Figure 4.4. As before, we avoid VDM notation in proof outlines by using logical variables.

Proof outline for `write` The proof outline for `write` is presented in Figure 4.7. Line 1 introduces logical variables w, h and h_o, which name the initial values of Ω, χ, and χ_{o}. Line 2 adds the knowledge that the writer for the pointer p is turned off ($W_p = \mathsf{W_{Off}}$). This follows from our implicit assumption that there is only one writer in the system, which, in the Coq code, we enforce by locks.

Line 3 is the first command of the program, and the most important step of the proof. Here *register* allocates a fresh timestamp t for the write event, puts t into χ_{j}, colouring it yellow or red, and changes W_p to $\mathsf{New}\ t\ v$, simultaneously with the physical update of p with v (see Figure 4.6). The importance of the step shows in line 4, where we need to establish that t is placed into the logical order after all the other finished or scanned events (*i. e.*, $\mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega \downarrow t$). This information is the most difficult part of the proof, but once established, it merely propagates through the proof outline.

Why does this inclusion hold? From the definition, we know that *register* appends t to the end of the list ζ (the clause $\zeta' = \mathsf{snoc}\ \zeta\ t$ in the definition of *register* in Figure 4.6). Thus, after the execution of line 3, we know that for every other timestamp s, $s <_\zeta t$. In particular, $s \neq t$, so it suffices to prove $s\ \Omega\ t$. We consider two cases: $s \in \mathsf{dom}(h_o)$ and $s \in \mathsf{scanned}\,\Omega$. In the first case, by

Invariant 4.10, $s \in \mathsf{dom}(\tau)$. By freshness of t wrt. global history h (which includes h_o), we get $\tau(s) < t$, and then the desired $s\ \Omega\ t$ follows from the definition of Ω. In the second case, by definition of $\mathsf{scanned}$, $\kappa(s) = \mathsf{green}$. Since $s <_\zeta t$, the result again follows by definition of Ω.

Still regarding line 4, we note that $t \in \mathsf{dom}(\chi_\mathtt{j})$ holds despite the interference of other threads. This is ensured by the Invariant 4.9, because no other thread but the writer for p, can modify W_p. Thus, this property will continue to hold in lines 6 and 8.

In line 6, the writer state W_p is updated following the definition of the auxiliary procedure *check*. The conjunct on $\mathsf{dom}(h_o) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t$ propagates from line 4, by monotonicity of Ω (Invariant 4.1). Similarly, in line 8, W_p is changed following the definition of *forward*, and the the other conjunct propagates. *Forward* further colours a number of timestamps green, but this is done in order to satisfy the state space invariants from Section 4.3, and is not exposed in the proof of `write`. Finally, in line 10, *finalize* moves $t \mapsto (p, v)$ from $\chi_\mathtt{j}$ to $\chi_\mathtt{s}$, thus completing the proof.

Proof outline for `scan`. Finally, the proof outline for is given in Figure 4.8. Line 1 introduces the logical variable h to name the initial χ. Line 2 adds the knowledge that $S_s = \mathsf{S}_{\mathsf{Off}\ \_}$ and $S_x = S_y = \mathsf{False}$, *i. e.*, that there are no other scanners around, which is enforced by locking in our Coq files.

Line 3 is the first line of the code; it simply sets the scanner bit S, and the auxiliaries S_x and S_y, following the definition of *set*. The conjunct $h \subseteq \chi$ follows from monotonicity by Invariant 4.1. The first important property comes from the lines 5 and 7. In these lines, *clear* sets the values of S_x and S_y, but, importantly, also colours the events from h green, first colouring x-events, and then y-events. This will be important at the end of the proof, where the fact that h is all green will enable inferring the postcondition. Moreover, because green events are never re-coloured, we propagate this property to subsequent lines without commentary.

The read from x in line 9, and from y in line 11, must return the last green, or the yellow event of their pointer, if no values are forwarded in *fx* and *fy*, respectively. This holds by Lemma 4.11, and is reflected by the conjuncts $\mathsf{fwdLastGY}\ x\ t_x\ vx$ and $\mathsf{fwdLastGY}\ x\ t_x\ vy$ in line 12, where:

$$\mathsf{fwdLastGY}\ p\ t\ v\ \widehat{=}\ \mathtt{fwd}\ p \mapsto \bot \implies \mathsf{lastGY}\ p\ t \wedge t \mapsto (p, v) \in \chi$$

The implication guard $\mathtt{fwd}\ p \mapsto \bot$ will be stripped away in the future, if and when the reads of forwarding pointers in lines 15 and 17 observe that no forwarding values exist.

In line 13, the scanner unsets the bit S and records the ending time of the scanner into the variable t_{off} in line 14. The conjuncts $\mathsf{fwdLastGY}\ x\ t_x\ vx$ and $\mathsf{fwdLastGY}\ y\ t_y\ vy$ from line 12 transfer to line 14 directly. This is so because *set* does not change any colours. Moreover, any writes that may run concurrently

1 $\{\chi_{\mathsf{s}} = \emptyset \land h \subseteq \chi\}$
2 $\{\chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{Off}\ \_} \land S_x = S_y = \mathsf{False} \land h \subseteq \chi\}$
3 $\langle S := \texttt{true};\ \mathit{set}(\texttt{true})\rangle;$
4 $\{\chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{On}} \land S_x = S_y = \mathsf{False} \land h \subseteq \chi\}$
5 $\langle fx := \bot;\ \mathit{clear}(x)\rangle;$
6 $\{\chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{On}} \land S_x = \mathsf{True} \land S_y = \mathsf{False} \land h \subseteq \chi \land \kappa(\mathsf{dom}(h_x)) = \mathsf{green}\}$
7 $\langle fy := \bot;\ \mathit{clear}(y)\rangle;$
8 $\{\chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{On}} \land S_x = S_y = \mathsf{True} \land h \subseteq \chi \land \kappa(\mathsf{dom}(h)) = \mathsf{green}\}$
9 $vx \leftarrow \langle \mathsf{read}(x)\rangle;$
10 $\{\exists\, t_x.\ \chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{On}} \land S_x = S_y = \mathsf{True} \land$
$h \subseteq \chi \land \kappa(\mathsf{dom}(h)) = \mathsf{green} \land \mathsf{fwdLastGY}\ x\, t_x\, vx\}$
11 $vy \leftarrow \langle \mathsf{read}(y)\rangle;$
12 $\{\exists\, t_x\, t_y.\ \chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{On}} \land S_x = S_y = \mathsf{True} \land h \subseteq \chi \land$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \land \mathsf{fwdLastGY}\ x\, t_x\, vx \land \mathsf{fwdLastGY}\ x\, t_x\, vy\}$
13 $\langle S := \texttt{false};\ \mathit{set}(\texttt{false})\rangle;$
14 $\{\exists\, t_x\, t_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{Off}}\, t_{\mathsf{off}} \land S_x = S_y = \mathsf{True} \land h \subseteq \chi \land$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \land \mathsf{fwdLastGY}\ x\, t_x\, vx \land \mathsf{fwdLastGY}\ y\, t_y\, vy\}$
15 $ox \leftarrow \langle \mathsf{read}(fx)\rangle;$
16 $\{\exists\, t_y\, t'_x\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{Off}}\ t_{\mathsf{off}} \land S_x = S_y = \mathsf{True} \land h \subseteq \chi \land$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \land \mathsf{fwdLastGY}\ y\, t_y\, vy \land$
$\mathsf{lastGYHist}\ x\ t'_x\ (\text{if } r = \bot \text{ then } vx \text{ else } r)\}$
17 $oy \leftarrow \langle \mathsf{read}(fy)\rangle;$
18 $\{\exists\, t'_x\, t'_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{Off}}\ t_{\mathsf{off}} \land S_x = S_y = \mathsf{True} \land h \subseteq \chi \land$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \land \mathsf{lastGYHist}\ x\ t'_x\ (\text{if } ox = \bot \text{ then } vx \text{ else } ox) \land$
$\mathsf{lastGYHist}\ y\ t'_y\ (\text{if } oy = \bot \text{ then } vy \text{ else } oy)\}$
19 $rx \leftarrow \textbf{if}\ (ox \neq \bot)\ \textbf{then}\ ox\ \textbf{else}\ vx;$
20 $ry \leftarrow \textbf{if}\ (oy \neq \bot)\ \textbf{then}\ oy\ \textbf{else}\ vy;$
21 $\{\exists\, t'_x\, t'_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \emptyset \land S_s = \mathsf{S}_{\mathsf{Off}}\ t_{\mathsf{off}} \land S_x = S_y = \mathsf{True} \land h \subseteq \chi \land$
$\kappa(\mathsf{dom}(h)) = \mathsf{green} \land \mathsf{lastGYHist}\ x\ t'_x\ rx \land \mathsf{lastGYHist}\ y\ t'_y\ ry\}$
22 $\langle\, \mathit{relink}(rx, ry);\ \textbf{return}\ (rx, ry)\,\rangle$
23 $\{r.\ \exists t.\ \chi_{\mathsf{s}} = \emptyset \land r = \mathsf{eval}\ t\ \Omega\ \chi \land \mathsf{dom}(h) \subseteq \Omega{\downarrow}t \land t \in \mathsf{scanned}\,\Omega\}$

Figure 4.8: Proof outline for `scan`.

with this scan cannot invalidate the conjuncts. To see this, assume that we had a concurrent `write` to x (reasoning is symmetric for y). Such a `write` may add a new yellow timestamp s, but only if t_x itself is the last green, in accord with Invariant 4.4. In that case, t_x remains the last green timestamp, and $\mathsf{fwdLastGY}\ x\ t_x\ vx$ remains valid. The concurrent `write` may change the colour of s to green, by invoking *forward* (Figure 4.5, line 5), but then fx becomes non-$\bot$, thus making $\mathsf{fwdLastGY}\ x\ t_x\ vx$ hold trivially.

In lines 15 and 17, `scan` reads from the forwarding pointers fx and fy and stores the obtained values into ox and oy, respectively. By Invariant 4.5, we know that if $ox \neq \bot$, there exists t'_x s.t. $t'_x \mapsto (x, ox) \in \chi$, and t'_x is the last green or yellow write event of χ_x. In case $ox = \bot$, we know from the $\mathsf{fwdLastGY}$ conjunct preceding the read from fx, that such last green or yellow event is exactly t_x. The consideration for fy is symmetric, giving us the assertion in line 18, where:

$$\mathsf{lastGYHist}\ p\ t\ v\ \widehat{=}\ \mathsf{lastGY}\ p\ t \wedge t \mapsto (p, v) \in \chi$$

Next, line 19 merely names by rx the value of vx, if ox equals $\bot$, and similarly for ry in line 20, leading to line 21. Finally, on line 22, the method finishes by invoking $\langle\, \mathit{relink}(rx, ry);\ \mathbf{return}\ (rx, ry)\,\rangle$. Thus, it returns the selected snapshot (r_x, r_y) and relinks the events so that the Ω justifies the choice of snapshots.

We prove that the final state satisfies the postcondition in line 23, by using the main property of *relink* (Lemma 4.13). First, we pick $t = \mathsf{max}_\zeta(t'_x, t'_y)$. Then $r = \mathsf{eval}\ t\ \Omega\ \chi$ holds, by the following argument. By Lemma 4.13.1, rx is the value of the last green timestamp in χ_x. By Lemma 4.13.2, all the timestamps below t are green, thus rx is the value of the *last* timestamp in χ_x that is smaller or equal to t. By a symmetric argument, the same holds of ry. But then, the pair $r = (rx, ry)$ is the snapshot at t, *i. e.*, equals $\mathsf{eval}\ t\ \Omega\ \chi$.

The conjunct $t \in \mathsf{scanned}\,\Omega$ is proved as follows. Unfolding the definition of $\mathsf{scanned}$, we need to show $\Omega{\downarrow}t = {\leq_\zeta}{\downarrow}t$, and $\forall s \in \Omega{\downarrow}t.\, \kappa(s) = \mathsf{green}$. The first conjunct follows from Lemma 4.12. The second immediately follows from the first by Lemma 4.13.2.

To establish $\mathsf{dom}(h) \subseteq \Omega{\downarrow}t$, we proceed as follows. Let $s \in \mathsf{dom}(h)$. From line 21, we know $\kappa(s) = \mathsf{green}$. Because t'_x and t'_y are last green (by ζ) or yellow events, by Invariant 4.4 it must be $s \leq_\zeta t'_x, t'_y$, and thus $s \leq_\zeta t$. However, we already showed that $\Omega{\downarrow}t = {\leq_\zeta}{\downarrow}t$. Thus, $s\ \Omega\ t$, finally establishing the postcondition.

4.9 Discussion

Comparison with linearizability, revisited As we argued in Section 4.3, our specifications for the snapshot methods directly capture that the method calls can be placed in a linear sequence, in a way that preserves the order of non-overlapping calls. This is precisely what linearizability achieves as well, but

by technically different means. We here discuss some similarities and differences between our method and linearizability.

The first distinction is that linearizability is a property of a concurrent object, whereas our specifications are ascribed to individual methods, as customary in Hoare logic. This immediately enables us to use an of-the-shelf Hoare logic, such as FCSL, for specification.

Second, linearizability draws its power from the connection to contextual refinement (Filipovic *et al.* , 2010): one can substitute a potentially complex method A in a larger context, by a simpler method B, to which A linearizes. In our setting, such a property is enabled by a general substitution principle, which says that programs with the same spec can be interchanged in a larger context, without affecting the larger context's proof. Moreover, contextual refinement (and thus linearizability) is defined for general programs, without regard to their preconditions and postconditions. However, it is often the case that the refinement only holds if the substituted programs satisfy some Hoare logic spec. In this sense, our setting is more expressive, since the substitution principle is given relative to a Hoare logic spec.

Finally, while our specification of the snapshot methods are motivated by linearizability, there is no requirement—and hence no proof—that an FCSL specification implies linearizability. But this is a feature, rather than a bug. It enables us to specify and combine, in one and the same logic, programs that are linearizable, with those that are not. We refer to (Sergey *et al.* , 2016) for examples of how to specify and verify non-linearizable programs in FCSL.

Alternative snapshot implementations. FCSL's substitution principle can be exploited further in an orthogonal way: it allows us to re-use the specs for `write` and `scan` in Figure 4.4, ascribing them to a different concurrent snapshot algorithm. For that matter, we re-visit the previous verification in FCSL of the pair-snapshot algorithm (Sergey *et al.* , 2015b). We present only `scan` in Figure 4.9, as `write` is trivial.

```
1  scan () : (A × A) {
2    (cx, vx) ← read(x);
3    (cy, _) ← read(y);
5    (_, tx) ← read(x);
5    if vx = tx
6    then return (cx, cy)
7    else scan (); }
```

Figure 4.9: `scan` using version numbers.

In this example, the snapshot structure consists of pointers x and y storing tuples (c_x, v_x) and (c_y, v_y), respectively. c_x and c_y are the payload of x and y, whereas v_x and v_y are version numbers, internal to the structure. Writes to x and y increment the version number, while `scan` reads x, y and x again, in succession. Snapshot inconsistency is avoided by restarting if the two version numbers of x differ. In the notation used throughout

this chapter, the specification proved for `scan` in (Sergey *et al.* , 2015b) reads:

$$\texttt{scan} : \{\chi_{\mathsf{s}} = \emptyset\}\ \{\exists t.\, \chi'_{\mathsf{s}} = \emptyset \wedge r = \mathsf{eval}\ t\ \chi' \wedge \mathsf{dom}(\chi) \subseteq \chi' {\downarrow} t\}$$

This spec is indeed very similar to the one of `scan` in Figure 4.4, but exhibits that the algorithm does not require dynamic modification to the event ordering. Thus, by defining Ω to be the natural ordering on timestamps in the global history χ (so that $\Omega' {\downarrow} t = \chi' {\downarrow} t$), and taking $\mathsf{scanned}\,\Omega$ to be the set of all timestamps in χ (so that $t \in \mathsf{scanned}\,\Omega$ is trivially true and can be added to the postcondition above), the above spec directly weakens into that of Figure 4.4. Since client proofs are developed in FCSL out of the specs, and not the code of programs, we can substitute different implementations of snapshot algorithms in clients, without disturbing the clients' proofs. This is akin to the property that programs that linearize to the same sequential code are interchangeable in clients.

Scalability of the Linking in Time approach. Even when we have presented here, only one formalized case study, we can distill how the proposed verification technique of *relinking in time* would scale to other more complex data structures. As we have mentioned throughout this chapter, the crux of our technique is the ability to change atomically the apparent order of past events as long as they satisfy the invariants of the data-structure and the *real-time* order of non-overlapping events inherited from *linearizability*. An informal taxonomy of the auxiliary code methods can be given, dividing them into three families according on how they relate with the logical ordering of events:

register Auxiliary code methods whose role is to witness the existence of a new atomic event. In Jayanti's snapshot case, we have only one: *register*. This needs not be the start of the method, but rather the atomic moment when it becomes visible to the environment: the pointer is written, the value is enqueued, dequeued, pushed, popped, *etc.*.

forward This class of auxiliary code methods update the auxiliary state in order to reflect certain dynamic events. Their purpose is to contribute/refine the evidence the *relink* method needs to assess whether the order of events is correct, and if it is not the case, how to fix it. In our case study: *checkS*, *forward*, *finalize*, *set*, and *clear*.

relink auxiliary code methods which change the order to justify the correctness of a result. In our case study *relink* implements such behavior.

The concurrent resource we presented in this chapter for Jayanti's snapshot algorithm considers only *write* events in the history, and computes snapshots by evaluating scanned segments of the history. Thus, there could only be one *register*-class auxiliary method. If we had encoded scan sevents as well—a needless

and painful exercise, as per our argument above—there would be a another method of this kind. In a similar way, it is easy to envision that data-structures with more than one *effectful* method like queues or stacks, will define two auxiliary state code *register*-like operations of this class, one per thread. Or at least, two instances of such.

As for *relink*, in the case of a snapshot data structure we argue that it is optimal to have only one of such methods, and place it in the `scan` method before returning the snapshot, just as we have done in this paper. Checking the correctness of this auxiliary code and, the stability of its specification and proving it satisfies the 2-step invariants defined in Invariant 4.1 constitutes a significant part of the proof burden in our mechanization. If we had gone for a design in which a *relink*-like auxiliary code operation was happening, for instance, together with each of the reading loops in `scan`, we would have increased significantly the size and complexity of our mechanization.

We have identified a potential candidates for applying our technique in the future, such as the TS stack (Dodds *et al.* , 2015) and the ever ubiquitous Herlihy–Wing (Herlihy & Wing, 1990; Schellhorn *et al.* , 2012; Henzinger *et al.* , 2013a; Khyzha *et al.* , 2017) queue. Our preliminary understanding is that our approach would be to add one *relink* auxiliary code operation in those cases as well: respectively at the end of the the pop and dequeue methods. Of course, we do not have any evidence to claim this hints to a general rule. Nor we believe it should be the case. A tailored suit fits better than a *one-size-fits-all* jacket.

Relation to Jayanti's original proof. Finally, we close this section by noting that our proof of Jayanti's algorithm seems very different from Jayanti's original proof. Jayanti relies on so-called *forwarding principles*, as a key property of the proof. For example, Jayanti's First Forwarding Principle says (in paraphrase) that if `scan` misses the value of a concurrent write through lines 10–11 of Figure 4.1, but the write terminates before the scanner goes through line 12 (the linearization point of `scan`), then the scanner will catch the value in the forwarding pointers through lines 13–14. Instead of forwarding principles, we rely on colors to algorithmically construct the status of each write event as it progresses through time, and express our assertions using formal logic. For example, though we did not use the First Forwarding Principle, we nevertheless can express a similar property, whose proof follows from the auxiliary state Invariants introduced in Section 4.5:

Proposition 4.19. *If* $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}$ *and* $S_x = S_y = \mathsf{True}$*—i.e., the scanner is in lines 13–16 and it has unset* S *in line 12 at time* t_{off}*—then:* $\forall t \in \chi.\ t \leq \tau(t) < t_{\mathsf{off}} \implies \kappa(t) = \mathsf{green}$.

4.10 Related work

Program logics for linearizability. The proof method for establishing linearizability of concurrent objects based on the notion of linearization points has been presented in the original paper by Herlihy and Wing (Herlihy & Wing, 1990). The first Hoare-style logic, employing this method for compositional proofs of linearizability was introduced in Vafeiadis' PhD thesis (Vafeiadis *et al.* , 2006; Vafeiadis, 2008). However, that logic, while being inspired by the combination of Rely-Guarantee reasoning and Concurrent Separation logic (Vafeiadis & Parkinson, 2007) with syntactic treatment of linearization points (Vafeiadis *et al.* , 2006), did not connect reasoning about linearizability to the verification of client programs that make use of linearizable objects in a concurrent environment.

Both these shortcomings were addressed in more recent works on program logics for linearizability (Liang & Feng, 2013; Khyzha *et al.* , 2016), or, equivalently, *observational refinement* (Filipovic *et al.* , 2010; Turon *et al.* , 2013a). These works provided semantically sound methodologies for verifying refinement of concurrent objects, by encoding atomic commands as resources (sometimes encoded via a more general notion of *tokens* (Khyzha *et al.* , 2016)) directly into a Hoare logic. Moreover, the logics (Liang & Feng, 2013; Turon *et al.* , 2013a) allowed one to give the objects standard Hoare-style specifications. However, in the works (Liang & Feng, 2013; Turon *et al.* , 2013a), these two properties (*i. e.*, linearizability of a data structure and validity of its Hoare-style spec) are established separately, thus doubling the proving effort. That is, in those logics, provided a proof of linearizability for a concurrent data structure, manifested by a spec that suitably handles a *command-as-resource*, one should then devise a declarative specification that exhibits temporal and spatial aspects of executions (akin to our history-based specs from Figure 4.4), required for verifying the client code.

Importantly, in those logics, determining the linearization order of a procedure is tied with that procedure "running" the command-as-resource within its execution span. This makes it difficult to verify programs where the procedure terminates before the order is decided on, such as `write` operation in Jayanti's snapshot. The problem may be overcome by extending the scope of *prophecy variables* (Abadi & Lamport, 1988) or *speculations* beyond the body of the specified procedure. However, to the best of our knowledge, this has not been done yet.

Hoare-style specifications as an alternative to linearizability. A series of recent Hoare logics focus on specifying concurrent behaviour *without* resorting to linearizability (Sergey *et al.* , 2015b, 2016; Svendsen & Birkedal, 2014; da Rocha Pinto *et al.* , 2014; Jung *et al.* , 2015). The developments presented in this chapter continues the same line of thinking, building on (Sergey *et al.* , 2015b), which explored patterns of assigning Hoare-style specifications with self/other auxiliary histories to concurrent objects, including *higher-order* ones (e.g., flat combiner (Hendler *et al.* , 2010)), and *non-linearizable* ones (Sergey

et al. , 2016) in FCSL (Nanevski *et al.* , 2014a), but has not considered non-local, future-dependent linearization points, as required by Jayanti's algorithm.

Alternative logics, such as Iris (Jung *et al.* , 2015, 2016) and iCAP (Svendsen & Birkedal, 2014), employ the idea of "ghost callbacks" (Jacobs & Piessens, 2011), to identify precisely the point in code when the callback should be invoked. Such a program point essentially corresponds to a local linearization point. Similarly to the logical linearizability proofs, in the presence of future-dependent LPs, this method would require speculating about possible future execution of the callback, just as commented above, but that requires changes to these logics' metatheory, in order to support speculations, that have not been carried out yet.

The specification style of TaDA (da Rocha Pinto *et al.* , 2014) is closer to ours in the sense that it employs *atomic tracking resources*, that are reminiscent of our history entries. However, the metatheory of TaDA does not support ownership transfer of the atomic tracking resources, which is crucial for verifying algorithms with non-local linearization points. As demonstrated by the technique introduced in this chapter and also previous works (Sergey *et al.* , 2015b, 2016), history entries can be subject to ownership transfer, just like any other resources.

The key novelty of the current work with respect to previous results on Hoare logics with histories (Fu *et al.* , 2010; Liang & Feng, 2013; Gotsman *et al.* , 2013; Bell *et al.* , 2010; Sergey *et al.* , 2015b; Hemed *et al.* , 2015) is the idea of representing logical histories as auxiliary state, thus enabling constructive reasoning, by *relinking*, about dynamically changing linearization points. Since relinking is just a manipulation of otherwise standard auxiliary state, we were able to use FCSL *off the shelf*, with no extensions to its metatheory. Furthermore, we expect to be able to use FCSL's higher-order features to reason about higher-order (*i. e.*, parameterized by another data structure) snapshot-based constructions (Petrank & Timnat, 2013). Related to our result, O'Hearn *et al.* have shown how to employ history-based reasoning and Hoare-style logic to *non-constructively* prove the existence of linearization points for concurrent objects out of the data structure invariants (O'Hearn *et al.* , 2010); this result is known as *the Hindsight Lemma*. The reasoning principle presented in this work generalizes that idea, since the Hindsight Lemma is only applicable to "pure" concurrent methods (*e. g.*, a concurrent set's `contains` (Heller *et al.* , 2006)) that do not influence the position of other threads' linearization points. In contrast, our history relinking handles such cases, as showcased by Jayanti's construction, where the linearization point of `write` depends on the (future) outcome of `scan`.

Semantic proofs of linearizability. There has been a long line of research on establishing linearizability using forward-backwards simulations (Schellhorn *et al.* , 2012; Colvin *et al.* , 2006, 2005). These proofs usually require a complex simulation argument and are not modular, because they require reasoning about the entire data structure implementation, with all its methods, as a monolithic

state-transition system.

Recent works (Henzinger *et al.* , 2013a; Chakraborty *et al.* , 2015; Dodds *et al.* , 2015) describe methods for establishing linearizability of sophisticated implementations (such as the Herlihy–Wing queue (Herlihy & Wing, 1990) or the time-stamped stack (Dodds *et al.* , 2015)) in a modular way, via *aspect-oriented* proofs. This methodology requires devising, for each class of objects (*e. g.*, queues or stacks), a set of specification-specific conditions, called *aspects*, characterizing the observed executions, and then showing that establishing such properties implies its linearizability. This approach circumvents the challenge of reasoning about future-dependent linearization points, at the expense of (a) developing suitable aspects for each new data structure class and proving the corresponding "aspect theorem", and (b) verifying the aspects for a specific implementation. Even though some of the aspects have been mechanized and proved adequate (Dodds *et al.* , 2015), currently, we are not aware of such aspects for snapshots.

Our approach is based on program logics and the use of STSs to describe the state-space of concurrent objects. Modular reasoning is achieved by means of separately proving properties of specific STS transitions, and then establishing specifications of programs, composed out of well-defined atomic commands, following the transitions, and respecting the STS invariants.

Proving linearizability with forward simulations In a recent work developed concurrently with ours, (Bouajjani *et al.* , 2017) show that *forward simulations* suffice for two classes of data structures with non-fixed linearization points: (1) queues similar to the Herlihy–Wing queue which have a fixed `dequeue` method and a non-fixed future dependent `enqueue` method, and (2) stacks similar to the time-stamped stack with a `pop` method with fix *commitment points* and a `push` method with non-fixed ones. Jayanti's snapshot method fits *a priori* in this dual pattern of having one method with a fix linearization point (scan) and another with a future-dependent one (write). However, the authors have not consider snapshot data-structures. Thus, it would be interesting to investigate a relationship between their method and our *Linking in Time* technique.

Proving linearizability using partial orders. Concurrently with us, Khyzha *et al.* (Khyzha *et al.* , 2017) have developed a proof method for proving linearizability, which can handle certain class of data structures with similar future dependent behaviour. The method works by introducing a partial order of events for the data structure as auxiliary state, which in turn defines the abstract histories used for satisfying the sequential specification of the data structure. Relations are added to this partial order at *commitment points* of the instrumented methods, which the verifier has to identify.

The ultimate goal of this method is to assert the linearizability of a concurrent data structure. As we have shown in Section 4.4, FCSL goes beyond as it provides

a logical framework to carry out formal proofs about the correctness of a concurrent data structure and its clients.

The proof technique also tracks the ordering of events differently from ours. Where we keep a single witness for the current total ordering of events at all stages of execution, their technique requires keeping many witnesses. Their main theorem requires a proof that all linearizations of the abstract histories—*i. e.* all possible linear extensions of the partial order into a total order—satisfy the sequential specification of the data structure.

Through personal communication we learned that the technique cannot apply, for instance, to the verification of the *time-stamped* (TS) stack (Dodds *et al.* , 2015). This is because a partial order does not suffice to characterize the abstract histories required to verify the data structure. In contrast, given the flexibility of FCSL in designing and reasoning with auxiliary state, we believe that our technique would not suffer such shortcomings.

4.11 Summary

This chapter illustrates a new approach allowing one to specify that the execution history of a concurrent data structure can be seen as a *sequence of atomic events*. The approach is thus similar in its goals to linearizability, but is carried out exclusively using a separation-style logic to uniformly represent the state and time aspects of the data structure and its methods.

Reasoning about time using separation logic is very effective, as it naturally supports *dynamic and in-place updates* to the temporal ordering of events, much as separation logic supports dynamic and in-place updates of spatially linked lists. The need to modify the ordering of events frequently appears in linearizability proofs, and has been known to be tricky, especially when the order of a terminated event depends on the future. In our approach, the modification becomes a conceptually simple manipulation of auxiliary state of histories of coloured timestamps.

We have carried out and mechanized our proof of Jayanti's algorithm (Jayanti, 2005) in FCSL, without needing any additions to the logic. Such development, together with the fact that FCSL has previously been used to verify a number of non-trivial concurrent structures (Sergey *et al.* , 2015b,a, 2016), gives us confidence that the approach will be applicable, with minor modifications, to other structures whose linearizations exhibit dynamic dependence on the future (Dodds *et al.* , 2015; Morrison & Afek, 2013; Hoffman *et al.* , 2007).

One modification that we envision will be in the design of the data type of time-stamped histories. In the current paper, a history of the snapshot object needs to keep only the `write` events, but not the `scan` events. In contrast, in the case of stacks, a history would need to keep both events for push and pop operations. But in FCSL, histories are a *user-defined* concept, which is not

hardwired into the semantics of the logic. Thus, the user can choose any particular notion of history, as long as it satisfies the properties of a Partial Commutative Monoid (Ley-Wild & Nanevski, 2013; Nanevski *et al.* , 2014a). Such a history can track pushes and pops, or any other auxiliary notion that may be required, such as, *e. g.*, specific ordering constraints on the events.

Part III

Conclusions

5

Conclusions & Future Work

This thesis presents two advances in the field of formal verification of stateful computer programs in the presence of higher-order control effects. In this chapter, we revisit these contributions and discuss briefly potential lines of future research and open questions.

5.1 Conclusions

The first contribution of this thesis is a higher-order type theory, HTTcc, whose aim is the verification of higher-order stateful programs featuring `call/cc` and `abort` control operators. The implemented theory, presented in Chapter 2, supports mutable state in the style of Separation logic, and, to the best of our knowledge, is the first Hoare logic or type theory to support the combination of higher-order functions, dynamic, mutable state and higher-order control operators. Moreover, HTTcc features *algebraic* control operators, initially introduced by Jaskelioff (Jaskelioff, 2009), which we here adapt to persistent state, *i. e.* without rollbacks. We have observed through experimentation that the use of algebraic control operators entails the need for less user-provided program annotations, when compared to the case of traditional, *non-algebraic*, control operators.

Another interesting feature of this work is the use of *dependent record types*—specifically, the $\Sigma\bullet\mathsf{SK}\,A$ type—to capture closures (potentially capturing or aborting to execution contexts) which can be returned and executed later. This, together with the fact that our shallow embedding in Coq allows for higher-order assertions about such closures, enables HTTcc to reason about *backwards* or *re-entrant* jumps. The **inc3** example in Section 2.2 and **ping–pong** in Section 2.6 illustrate this phenomenon. This is a fundamental contribution of HTTcc, and it is unlike most existing Hoare-style logics targeted at the verification of imperative programs with first order jumps (Arbib & Alagic, 1979; Clint & Hoare, 1972; Audebaud & Zucca, 1999), which only allow for *exit* jumps. Moreover, this is also unique with

regard to previously existing *pure*—*i. e.* without state—higher-order logics for higher-order control flow (Crolard & Polonowski, 2012; Berger, 2009).

Following the tradition of previous Hoare-Type theories, we have implemented HTTcc as a domain specific language in Coq, and verified a number of characteristic example programs that use **callcc** (Delbianco & Nanevski, 2017). A selected survey of this examples were presented in Section 2.6 and several more can be found in the Coq sources.

The second contribution of this thesis is *Linking in Time*, a new approach allowing one to specify that the execution history of a concurrent data structure can be seen as a *sequence of atomic events*. The approach is thus similar in its goals to linearizability, but is carried out exclusively using a separation-style logic to uniformly represent the state and time aspects of the data structure and its methods. Reasoning about time using separation logic is very effective, as it naturally supports *dynamic and in-place updates* to the temporal ordering of events, much as separation logic supports dynamic and in-place updates of spatially linked lists. The need to modify the ordering of events frequently appears in linearizability proofs, and has been known to be tricky, especially when the order of a terminated event depends on the future. In our approach, the modification becomes a conceptually simple manipulation of auxiliary state of histories of coloured timestamps.

We have illustrated the approach on a snapshot algorithm by Jayanti (Jayanti, 2005), whose linearizability proof exhibits exactly such dependence on the future. We have carried out the mechanization of the technique's infrastructure as well as the proof of correctness of Jayanti's snapshot construction *off-the-shelf* in FCSL, without needing any additions to the logic. This is an considerable advantage with regard to previously existing methods for Hoare-style verification of this class of data structures (Liang & Feng, 2013; Turon *et al.* , 2013a), which require special meta-theoretical devices *e. g.* prophecy variables (Abadi & Lamport, 1988). To the best of our knowledge, this establishes FCSL as the first program logic capable of mechanically proving full functional correctness of data structures whose linearizability proofs exhibit non-locality *and* dependence on future and *non-regional* events.

5.2 Open Questions & Future Work

HTTcc As we have mentioned elsewhere, HTTcc is the first high-order separation-like logic for a language with higher-order control effects and dynamic mutable state. As it usually happens with any *first thing*, there is still plenty to be done. Together with my co-authors we have discussed the possibility of extending this work from the meta-theoretical to the practical, all the way through the (re-)implementation of (successors of) HTTcc: there are practical improvements to be carried out in the implementation, many possible extensions of HTTcc, and

also the applications of the logic to other verification domains.

When it comes to extensions of, or improvements to, HTTcc's meta-theory, there are two possible and orthogonal directions for future work. A first axis would correspond to state-reasoning. As we have mentioned before in Chapter 2, the state in our logic is given by a first-order heap. This is a consequence of the implementation of the heap data-type—basically, a finite map from *non-null* addresses ($\mathbb{N}^+$) to values of *dynamic type*, *i. e.* values of type $\Sigma\,(X{:}\mathsf{Type}).\,X$. Given this model, it is impossible to accommodate for a higher-order heap in the current implementation of HTTcc. As we have discussed previously on Chapter 2, attempting to store a value of type $\mathsf{SK}\,A$ or $\mathsf{Kont}\,A$ into the heap would raise a *universe inconsistency* error in Coq. A traditional way of circumventing this issue has been to restrict the embedding of the assertion logic (Ni & Shao, 2006; Gotsman *et al.* , 2007; Chlipala, 2013), by *encoding* the *small* datatypes which can be allocated in the heap. We believe, however, that this is not a good idea for HTTcc, as it would forbid the use of high-order specifications as the ones needed for describing the behaviour of backwards jumps, as we have done for **inc3** in Figure 2.1, and for **ping**–**pong** in Figure 2.8.

There are two alternatives moving forward: either to design and implement an *impredicative* version of HTTcc, following the previous work done for HTTs (Svendsen *et al.* , 2011; Petersen *et al.* , 2008); or to look further into the recent[1] developments of *universe polymorphism* in Coq (Sozeau & Tabareau, 2014; The Coq Development Team, 2016) that, in theory could allow for this sort of type instances. The extension of HTTcc with *high-order state*, would enable, for instance, the implementation and verification of coroutines and other concurrency primitives *à la* CML (Reppy, 1999), *i. e.* implemented in terms of *callcc* and *abort* keeping a *queue* of continuations as a scheduler for pending threads. However, this would not be a straight-forward task: it does not suffice to be able to implement the concurrency primitives using `call/cc` in HTTcc, one would have to come with a *sensible* specification to the pending coroutines. How would one specify *re-entrance*? If the user of the library—*i. e.* client code—is allowed to `yield`, then how would the specs acknowledge the interference from the environment when the client resumes control?

Moreover, it is natural to desire to extend HTTcc to cope with *delimited or composable* (Felleisen *et al.* , 1987, 1988; Ariola *et al.* , 2009; Wadler, 1994) continuations. To the best of our knowledge, there is no verification framework that can reason with *delimited* control effects and dynamic mutable state. A first hurdle would be reconcile models for delimited control (which are non-monadic (Wadler, 1994)) with the underlying foundations of Hoare Type Theory—that is, *indexed* Hoare monads.

[1] At least, more recent than the original designs of both HTTcc and FCSL.

Linking in Time The immediate and obvious future work here is applying the lightweight technique described in Chapter 4 to further data structures with non-trivial linearization points. One modification that we envision will be in the design of the data type of time-stamped histories. In the current implementation of the concurrent resource for Jayanti's snapshot construction, a history of the snapshot object needs to keep only the `write` events, but not the `scan` events. In contrast, in the case of other data structures such as stacks, a history would need to keep both events for push and pop operations. But in FCSL, histories are a *user-defined* concept, which is not hardwired into the semantics of the logic. Thus, the user can choose any particular notion of history, as long as it satisfies the properties of a Partial Commutative Monoid (Ley-Wild & Nanevski, 2013; Nanevski *et al.* , 2014a). Such a history can track pushes and pops, or any other auxiliary notion that may be required, such as, *e. g.*, specific ordering constraints on the events.

That said, we have identified several linearizable data structures whose linearization points exhibit similar future dependent or non-regional behaviour (Herlihy & Wing, 1990; Dodds *et al.* , 2015; Morrison & Afek, 2013; Hoffman *et al.* , 2007), and we are looking forwards to verify them in FCSL by applying the technique introduced in this thesis. The Herlihy-Wing queue (Herlihy & Wing, 1990) is, perhaps, the most immediately obvious of them: this is a paradigmatic case of a data-structure with non-fixed linearization point, and it has become a 101 example for introducing a new logic or technique for proving linearizability for such a class of data structures (Schellhorn *et al.* , 2012; Henzinger *et al.* , 2013a; Khyzha *et al.* , 2017). At a first glance, the instrumentation of the algorithm and the proof would be structured in similar way to that of Jayanti's snapshot construction: *enqueue* and *dequeue* events would be registered at the beginning of their respective methods, and a atomic *relink* ghost-code operation would be added at the end of the dequeue method in order to fix the order when dequeue succeeds, by changing the order of conflicting pending *enqueues*.

Beyond Linearizability As we have mentioned before, Linearizability has become the golden standard by which concurrent objects are designed, specified and proven correct. However, designing efficient concurrent objects often requires abandoning the standard specification technique of *linearizability* in favour of more relaxed correctness conditions. In order to reason about such objects, several novel conditions have been developed: concurrency-aware linearizability (CAL) (**?**), quiescent consistency (QC) (Aspnes *et al.* , 1994; Derrick *et al.* , 2014), quasi-linearizability (QL) (Afek *et al.* , 2010), quantitative relaxation (Henzinger *et al.* , 2013b), quantitative quiescent consistency (QQC) (Jagadeesan & Riely, 2014), and local linearizability (Haas *et al.* , 2016), to name a few. However, this explosion of criteria is not satisfactory, especially when each of them require a specific logic or proof method to reason with them.

Instead, a radically different approach would be to use the general design idea behind the work presented in Chapter 4: encoding the consistency/correctness criteria and its verification through the invariants and operations of a FCSL concurrent resource. Our aim is to show that, the to perform the verification of different data structures in a powerful and unified setting for Hoare-style reasoning, rather than having to design, implement, and prove correct yet another new logic for a small class of concurrent objects that extend beyond the boundaries of linearizability reasoning. Moreover, one could devote the effort to prove interesting properties about complex clients.

Consequently, we have already done some steps in this regard, by verifying *non-linearizable* concurrent objects in FCSL (Sergey *et al.* , 2016). In that work, we present the verification of two non-linearizable concurrent objects and their clients in FCSL, which have so far been specified only by non-standard conditions of *concurrency-aware linearizability*, *quiescent*, and *quantitative quiescent consistency*. In the future, we look forwards to extend this approach to other history-based correctness criteria, such as the ones mentioned above.

FCSL and `fork/join` concurrency In FCSL, concurrency is introduced by means of the parallel composition operator ||, or *par*. Given two programs $p1$ and $p2$, *par* composes them concurrently, creating a new program $p1 \mid\mid p2$ where resources are shared among two threads, with $p1$ and $p2$ potentially racing for a shared resource. This syntactic, *well-bracketed* approach to introducing concurrency in a language is preferred from a logic design perspective because it favours reasoning inductively and fosters modular reasoning. However, in most *real-world* concurrent programming environments (Herlihy & Shavit, 2008; Raynal, 2013; Butenhof, 1997), threads are spawned dynamically using *fork* primitives. This practice enables more patterns for concurrent programming than ||, but also hinders the modular verification of programs. An immediate future research direction is to build upon FCSL in order to develop new logics that can tackle the modular verification of the unstructured `fork/join` primitives present in massively used libraries such as POSIX threads (*a. k. a. pthreads*) (Butenhof, 1997).

Several CSL-like program logics have dealt with `fork/join` concurrency, achieving different degrees of expressive power when it comes to which programs can be forked, and how are they joined. Some logics support reasoning about storable, yet *first-order* threads (Gotsman *et al.* , 2007; Dodds *et al.* , 2009), others restrict *a priori* the thread-structure of the clients (Jacobs & Piessens, 2011), or some of them deal only with `fork` primitives (Jung *et al.* , 2015, 2016). We believe that, by extending FCSL notion of *concurrent resources* in order to be able to identify each thread's specific contributions, we could develop a logic that *truly* implements `fork/join` concurrency in a thread-modular manner, and that can also verify *unstructured* forking and joining patterns such as *revisions* and *isolation* types (Burckhardt *et al.* , 2010; Leijen *et al.* , 2011).

Free/Algebraic Hoare Type Theories Traditionally, all existing Hoare Type Theories—including those presented in this thesis—have been implemented *monolithically*, *i. e.* giving a unique denotational semantics definition for the SK type in Chapter 2 or the FCSL triples $\{P\}e\{Q\}@C$ in Part II. An open research question is how to combine different HTTs to develop new ones, akin to the work it has been carried out in the computational effects community about combining effects. Even when monads have been successfully used to give semantics to computational effects for quite some time, monads *do not compose well* and therefore combining monadic effects modularly has been an open question for some time. Several alternatives have been proposed through time to solve this issue, with *monad transformers* (Liang *et al.* , 1995) being the established way to program with combined monads in Haskell. A more recent trend towards user definable effects is to consider *algebraic effects* (Plotkin & Power, 2003, 2002; Jaskelioff, 2009) or *extensible effects* (Kiselyov *et al.* , 2013; Brady, 2013b), that is, effects whose denotational definitions arise from their operations. Ahman *et al.* have done the first steps in this regard for Dijkstra monads and F* (Ahman *et al.* , 2017), but there is no similar precedent for HTTs. A starting point would be to study the categorical models of HTT by Jacobs (Jacobs, 2015) and look into how to define proper combinations or compositions in this setting following the techniques mentioned above. Perhaps, it would be however more interesting to consider designing HTTs on top of—or inspired from those of—existing extensible effects type systems, such as those provided by Eff (Bauer & Pretnar, 2015) and Idris (Brady, 2013a,b).

Bridging the gap This thesis presents two independent contributions. We believe there are several worthy interconnections to pursue in order to build bridges between them. For example, it would be worthwhile to combine HTTcc higher-order control features with a FCSL-like notion of subjective state to develop logics based on other models of concurrency such as *coroutines* (Conway, 1963; Abadi & Plotkin, 2009) or futures (Flanagan & Felleisen, 1995, 1999) A *future* represents the result of an asynchronous computation which cannot be accessed until is completion. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. *Coroutines* are language primitives that enable cooperation between asynchronous, resumable threads. Given it is possible to implement these concurrency models in a setting with continuations and *higher-order store*, i.e. one where computations can be stored in the heap, designing a programming logic for such features would be a natural extension of the work proposed. Moreover, it would be a suitable bridge between the two pillars of this thesis: Hoare-style reasoning for continuations and shared memory concurrency.

Bibliography

Abadi, Martín, & Lamport, Leslie. 1988. The Existence of Refinement Mappings. *Pages 165–175 of: Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88)*. IEEE Computer Society.

Abadi, Martín, & Plotkin, Gordon D. 2009. A model of cooperative threads. *Pages 29–40 of:* Shao, Zhong, & Pierce, Benjamin C. (eds), *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. ACM.

Aceto, Luca, & Ingólfsdóttir, Anna. 2007. Characteristic Formulae: From Automata to Logic. *Bulletin of the EATCS*, **91**, 58–75.

Afek, Yehuda, Korland, Guy, & Yanovsky, Eitan. 2010. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. *Pages 395–410 of: OPODIS*. Springer.

Ahman, Danel, Hritcu, Catalin, Maillard, Kenji, Martinez, Guido, Plotkin, Gordon D., Protzenko, Jonathan, Rastogi, Aseem, & Swamy, Nikhil. 2017. Dijkstra monads for free. *Pages 515–529 of:* Castagna, Giuseppe, & Gordon, Andrew D. (eds), *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. ACM.

Appel, Andrew W. 1992. *Compiling with Continuations*. Cambridge University Press.

Arbib, Michael A., & Alagic, Suad. 1979. Proof Rules for Gotos. *Acta Inf.*, **11**, 139–148.

Ariola, Zena M., Herbelin, Hugo, & Sabry, Amr. 2009. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, **22**(3), 233–273.

Aspnes, James, Herlihy, Maurice, & Shavit, Nir. 1994. Counting Networks. *J. ACM*, **41**(5), 1020–1048.

Audebaud, Philippe, & Zucca, Elena. 1999. Deriving Proof Rules from Continuation Semantics. *Formal Asp. Comput.*, **11**(4), 426–447.

Avigad, Jeremy, & Harrison, John. 2014. Formally Verified Mathematics. *Commun. ACM*, **57**(4), 66–75.

Barras, Bruno, & Bernardo, Bruno. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. *Pages 365–379 of:* Amadio, Roberto M. (ed), *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008. Proceedings*. Lecture Notes in Computer Science, vol. 4962. Springer.

Barthe, Gilles, & Uustalu, Tarmo. 2002. CPS translating inductive and coinductive types. *Pages 131–142 of:* Thiemann, Peter (ed), *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02)*. ACM.

Bauer, Andrej, & Pretnar, Matija. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, **84**(1), 108–123.

Bell, Christian J., Appel, Andrew W., & Walker, David. 2010. Concurrent Separation Logic for Pipelined Parallelization. *Pages 151–166 of:* Cousot, Radhia, & Martel, Matthieu (eds), *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*. Lecture Notes in Computer Science, vol. 6337. Springer.

Berger, Martin. 2009. Program Logics for Sequential Higher-Order Control. *Pages 194–211 of:* Arbab, Farhad, & Sirjani, Marjan (eds), *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 5961. Springer.

Bertot, Yves, & Castéran, Pierre. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Bjørner, Dines, & Jones, Cliff B. (eds). 1978. *The Vienna Development Method: The Meta-Language*. Lecture Notes in Computer Science, vol. 61. Springer.

Bornat, Richard, Calcagno, Cristiano, O'Hearn, Peter W., & Parkinson, Matthew J. 2005. Permission accounting in separation logic. *Pages 259–270 of:* Palsberg, Jens, & Abadi, Martín (eds), *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*. ACM.

Bouajjani, Ahmed, Emmi, Michael, Enea, Constantin, & Mutluergil, Suha Orhun. 2017. Proving linearizability using forward simulations. *In: Computer Aided Verification - 24th International Conference, CAV 2017. Proceedings (To Appear)*. A preliminary version is available on arXiv from http://arxiv.org/abs/1702.02705.

Brady, Edwin. 2013a. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, **23**(5), 552–593.

Brady, Edwin. 2013b. Programming and reasoning with algebraic effects and dependent types. *In:* (Morrisett & Uustalu, 2013).

Brookes, Stephen. 2007. A Semantics for Concurrent Separation Logic. *Theor. Comput. Sci.*, **375**(1-3), 227–270.

Brookes, Stephen, & O'Hearn, Peter W. 2016. Concurrent separation logic. *SIGLOG News*, **3**(3), 47–65.

Burckhardt, Sebastian, Baldassin, Alexandro, & Leijen, Daan. 2010. Concurrent programming with revisions and isolation types. *Pages 691–707 of:* Cook, William R., Clarke, Siobhán, & Rinard, Martin C. (eds), *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. ACM.

Butenhof, David R. 1997. *Programming with POSIX threads*. Addison-Wesley Professional.

Calcagno, Cristiano, O'Hearn, Peter W., & Yang, Hongseok. 2007. Local Action and Abstract Separation Logic. *Pages 366–378 of: 22nd IEEE Symposium on Logic in Computer Science (LICS 2007), Proceedings*. IEEE Computer Society.

Cerone, Andrea, Gotsman, Alexey, & Yang, Hongseok. 2014. Parameterised Linearisability. *Pages 98–109 of:* Esparza, Javier, Fraigniaud, Pierre, Husfeldt, Thore, & Koutsoupias, Elias (eds), *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Proceedings, Part II*. Lecture Notes in Computer Science, vol. 8573. Springer.

Chakraborty, Soham, Henzinger, Thomas A., Sezgin, Ali, & Vafeiadis, Viktor. 2015. Aspect-oriented linearizability proofs. *LMCS*, **11**(1).

Charguéraud, Arthur. 2010. *Characteristic Formulae for Mechanized Program Verification*. Ph.D. thesis, Université Paris-Diderot.

Chlipala, Adam. 2013. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. *In:* (Morrisett & Uustalu, 2013).

Clint, Maurice, & Hoare, C. A. R. 1972. Program Proving: Jumps and Functions. *Acta Inf.*, **1**.

Colvin, Robert, Doherty, Simon, & Groves, Lindsay. 2005. Verifying Concurrent Data Structures by Simulation. *Electr. Notes Theor. Comput. Sci.*, **137**(2), 93–110.

Colvin, Robert, Groves, Lindsay, Luchangco, Victor, & Moir, Mark. 2006. Formal Verification of a Lazy Concurrent List-Based Set Algorithm. *Pages 475–488 of:* Ball, Thomas, & Jones, Robert B. (eds), *Computer Aided Verification, 18th International Conference, CAV 2006, Proceedings*. Lecture Notes in Computer Science, vol. 4144. Springer.

Conway, Melvin E. 1963. Design of a Separable Transition-diagram Compiler. *Commun. ACM*, **6**(7), 396–408.

Crolard, Tristan, & Polonowski, Emmanuel. 2012. Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. *J. Log. Algebr. Program.*, **81**(3).

da Rocha Pinto, Pedro, Dinsdale-Young, Thomas, & Gardner, Philippa. 2014. TaDA: A Logic for Time and Data Abstraction. *Pages 207–231 of:* Jones, Richard (ed), *ECOOP 2014 - Object-Oriented Programming - 28th European Conference. Proceedings*. Lecture Notes in Computer Science, vol. 8586. Springer.

Danvy, Olivier, & Filinski, Andrzej. 1992. Representing Control: A Study of the CPS Transformation. *MSCS*, **2**(4).

Delbianco, Germán Andrés, & Nanevski, Aleksandar. 2013. Hoare-style reasoning with (algebraic) continuations. *In:* (Morrisett & Uustalu, 2013).

Delbianco, Germán Andrés, & Nanevski, Aleksandar. 2017 (April). *Hoare-style Reasoning with (Algebraic) Continuations: Coq Formalization Source Files*. Available from: http://delbian.co/HTTcc.

Delbianco, Germán Andrés, Sergey, Ilya, Nanevski, Aleksandar, & Banerjee, Anindya. 2017a. Concurrent Data Structures Linked in Time. *Pages 8:1–8:30 of:* Müller, Peter (ed), *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Leibniz International Proceedings in

Informatics (LIPIcs), vol. 74. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Delbianco, Germán Andrés, Sergey, Ilya, Nanevski, Aleksandar, & Banerjee, Anindya. 2017b. Concurrent Data Structures Linked in Time (Artifact). *DARTS*, **3**(2), 04:1–04:4. Available from: https://doi.org/10.4230/DARTS.3.2.4.

Derrick, John, Dongol, Brijesh, Schellhorn, Gerhard, Tofan, Bogdan, Travkin, Oleg, & Wehrheim, Heike. 2014. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. *Pages 200–214 of:* Jones, Cliff B., Pihlajasaari, Pekka, & Sun, Jun (eds), *FM 2014: Formal Methods - 19th International Symposium. Proceedings*. Lecture Notes in Computer Science, vol. 8442. Springer.

Dijkstra, Edsger W. 1968. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM*, **11**(3), 147–148.

Dinsdale-Young, Thomas, Dodds, Mike, Gardner, Philippa, Parkinson, Matthew J., & Vafeiadis, Viktor. 2010. Concurrent Abstract Predicates. *Pages 504–528 of:* D'Hondt, Theo (ed), *ECOOP 2010 - Object-Oriented Programming, 24th European Conference. Proceedings*. Lecture Notes in Computer Science, vol. 6183. Springer.

Dinsdale-Young, Thomas, Birkedal, Lars, Gardner, Philippa, Parkinson, Matthew J., & Yang, Hongseok. 2013. Views: compositional reasoning for concurrent programs. *In:* (Giacobazzi & Cousot, 2013).

Dodds, Mike, Feng, Xinyu, Parkinson, Matthew J., & Vafeiadis, Viktor. 2009. Deny-Guarantee Reasoning. *Pages 363–377 of:* Castagna, Giuseppe (ed), *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009. Proceedings*. Lecture Notes in Computer Science, vol. 5502. Springer.

Dodds, Mike, Haas, Andreas, & Kirsch, Christoph M. 2015. A Scalable, Correct Time-Stamped Stack. *Pages 233–246 of:* Rajamani, Sriram K., & Walker, David (eds), *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. ACM.

Dodds, Mike, Jagannathan, Suresh, Parkinson, Matthew J., Svendsen, Kasper, & Birkedal, Lars. 2016. Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic. *ACM Trans. Program. Lang. Syst.*, **38**(2), 4:1–4:72.

Dreyer, Derek, Neis, Georg, & Birkedal, Lars. 2010. The impact of higher-order state and control effects on local relational reasoning. *Pages 143–156 of:* Hudak, Paul, & Weirich, Stephanie (eds), *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010*. ACM.

FCSL. 2017 (April). *FCSL: Fine-grained Concurrent Separation Logic*. Project Website: http://software.imdea.org/fcsl/.

Felleisen, Matthias, Friedman, Daniel P., Kohlbecker, Eugene E., & Duba, Bruce F. 1986. Reasoning with Continuations. *Pages 131–141 of: Proceedings of the Symposium on Logic in Computer Science (LICS '86)*. IEEE Computer Society.

Felleisen, Matthias, Friedman, Daniel P., Duba, Bruce, & Merrill, John. 1987. *Beyond Continuations*. Tech. rept. 216. Indiana University.

Felleisen, Matthias, Wand, Mitchell, Friedman, Daniel P., & Duba, Bruce F. 1988. Abstract Continuations: A Mathematical Semantics for Handling Full Jumps. *Pages 52–62 of: LISP and Functional Programming*.

Feng, Xinyu. 2009. Local rely-guarantee reasoning. *Pages 315–327 of:* Shao, Zhong, & Pierce, Benjamin C. (eds), *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. ACM.

Feng, Xinyu, Ferreira, Rodrigo, & Shao, Zhong. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. *Pages 173–188 of:* Nicola, Rocco De (ed), *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Proceedings*. Lecture Notes in Computer Science, vol. 4421. Springer.

Filinski, Andrzej. 1994. Representing Monads. *In: POPL*.

Filipovic, Ivana, O'Hearn, Peter W., Rinetzky, Noam, & Yang, Hongseok. 2010. Abstraction for concurrent objects. *Theor. Comput. Sci.*, **411**(51-52), 4379–4398.

Flanagan, Cormac, & Felleisen, Matthias. 1995. The Semantics of Future and Its Use in Program Optimizations. *Pages 209–220 of:* Cytron, Ron K., & Lee, Peter (eds), *POPL*. ACM Press.

Flanagan, Cormac, & Felleisen, Matthias. 1999. The Semantics of Future and an Application. *J. Funct. Program.*, **9**(1), 1–31.

Friedman, Daniel P., & Felleisen, Matthias. 1996. *The seasoned schemer*. MIT Press.

Fu, Ming, Li, Yong, Feng, Xinyu, Shao, Zhong, & Zhang, Yu. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. *Pages 388–402 of: CONCUR*. Lecture Notes in Computer Science, vol. 6269. Springer.

Giacobazzi, Roberto, & Cousot, Radhia (eds). 2013. *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM.

Gonthier, Georges, Mahboubi, Assia, & Tassi, Enrico. 2008. *A Small Scale Reflection Extension for the Coq system*. Tech. rept. 6455. INRIA.

Gonthier, Georges, Asperti, Andrea, Avigad, Jeremy, Bertot, Yves, Cohen, Cyril, Garillot, François, Roux, Stéphane Le, Mahboubi, Assia, O'Connor, Russell, Biha, Sidi Ould, Pasca, Ioana, Rideau, Laurence, Solovyev, Alexey, Tassi, Enrico, & Théry, Laurent. 2013. A Machine-Checked Proof of the Odd Order Theorem. *Pages 163–179 of:* Blazy, Sandrine, Paulin-Mohring, Christine, & Pichardie, David (eds), *Interactive Theorem Proving - 4th International Conference, ITP 2013. Proceedings*. Lecture Notes in Computer Science, vol. 7998. Springer.

Gotsman, Alexey, & Yang, Hongseok. 2012. Linearizability with Ownership Transfer. *Pages 256–271 of: CONCUR*. Lecture Notes in Computer Science, vol. 7454. Springer.

Gotsman, Alexey, Berdine, Josh, Cook, Byron, Rinetzky, Noam, & Sagiv, Mooly. 2007. Local Reasoning for Storable Locks and Threads. *Pages 19–37 of:* Shao, Zhong (ed), *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Proceedings*. Lecture Notes in Computer Science, vol. 4807. Springer.

Gotsman, Alexey, Rinetzky, Noam, & Yang, Hongseok. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. *Pages 249–269 of:* Felleisen, Matthias, & Gardner, Philippa (eds), *Proceedings of the 22nd European Symposium on Programming Languages and Systems (ESOP 2013)*. Lecture Notes in Computer Science, vol. 7792. Rome, Italy: Springer.

Griffin, Timothy. 1990. A Formulae-as-Types Notion of Control. *In: POPL*.

Haas, Andreas, Henzinger, Thomas A., Holzer, Andreas, Kirsch, Christoph M., Lippautz, Michael, Payer, Hannes, Sezgin, Ali, Sokolova, Ana, & Veith, Helmut. 2016. Local Linearizability for Concurrent Container-Type Data Structures. *Pages 6:1–6:15 of:* Desharnais, Josée, & Jagadeesan, Radha (eds), *27th International Conference on Concurrency Theory, CONCUR 2016, Proceedings*. LIPIcs, vol. 59. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Heller, Steve, Herlihy, Maurice, Luchangco, Victor, Moir, Mark, III, William N. Scherer, & Shavit, Nir. 2006. A Lazy Concurrent List-Based Set Algorithm. *Pages 3–16 of: OPODIS*. Lecture Notes in Computer Science, vol. 3974. Springer.

Hemed, Nir, & Rinetzky, Noam. 2014. *Brief announcement*: Concurrency-Aware Linearizability. *Pages 209–211 of: PODC*. ACM.

Hemed, Nir, Rinetzky, Noam, & Vafeiadis, Viktor. 2015. Modular Verification of Concurrency-Aware Linearizability. *Pages 371–387 of: DISC*. Lecture Notes in Computer Science, vol. 9363. Springer.

Hendler, Danny, Incze, Itai, Shavit, Nir, & Tzafrir, Moran. 2010. Flat combining and the synchronization-parallelism tradeoff. *Pages 355–364 of: SPAA*. ACM.

Henzinger, Thomas A., Sezgin, Ali, & Vafeiadis, Viktor. 2013a. Aspect-Oriented Linearizability Proofs. *Pages 242–256 of:* D'Argenio, Pedro R., & Melgratti, Hernán C. (eds), *CONCUR*. Lecture Notes in Computer Science, vol. 8052. Springer.

Henzinger, Thomas A., Kirsch, Christoph M., Payer, Hannes, Sezgin, Ali, & Sokolova, Ana. 2013b. Quantitative relaxation of concurrent data structures. *In:* (Giacobazzi & Cousot, 2013).

Herlihy, Maurice, & Shavit, Nir. 2008. *The art of multiprocessor programming*. M. Kaufmann.

Herlihy, Maurice, & Wing, Jeannette M. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, **12**(3), 463–492.

Hoare, C. A. R. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM*, **12**(10), 576–580.

Hobor, Aquinas, & Villard, Jules. 2013. The ramifications of sharing in data structures. *In:* (Giacobazzi & Cousot, 2013).

Hoffman, Moshe, Shalev, Ori, & Shavit, Nir. 2007. The Baskets Queue. *Pages 401–414 of: OPODIS*. LNCS. Springer-Verlag.

Hyland, Martin, Levy, Paul Blain, Plotkin, Gordon D., & Power, John. 2007. Combining algebraic effects with continuations. *Theor. Comput. Sci.*, **375**(1-3).

Jacobs, Bart. 2015. Dijkstra and Hoare monads in monadic computation. *Theor. Comput. Sci.*, **604**, 30–45.

Jacobs, Bart, & Piessens, Frank. 2011. Expressive modular fine-grained concurrency specification. ACM.

Jagadeesan, Radha, & Riely, James. 2014. Between Linearizability and Quiescent Consistency - Quantitative Quiescent Consistency. *Pages 220–231 of:* Esparza, Javier, Fraigniaud, Pierre, Husfeldt, Thore, & Koutsoupias, Elias (eds), *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Proceedings, Part II*. LNCS, vol. 8573. Springer.

Jaskelioff, Mauro. 2009. Modular Monad Transformers. *Pages 64–79 of:* Castagna, Giuseppe (ed), *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009. Proceedings*. Lecture Notes in Computer Science, vol. 5502. Springer.

Jayanti, Prasad. 2005. An optimal multi-writer snapshot algorithm. *Pages 723–732 of: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, STOC 2005*. ACM.

Jensen, Jonas Braband, Benton, Nick, & Kennedy, Andrew. 2013. High-level separation logic for low-level code. *In: POPL*.

Jones, Cliff B. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, **5**(4), 596–619.

Jung, Ralf, Swasey, David, Sieczkowski, Filip, Svendsen, Kasper, Turon, Aaron, Birkedal, Lars, & Dreyer, Derek. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *Pages 637–650 of:* Rajamani, Sriram K., & Walker, David (eds), *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. ACM.

Jung, Ralf, Krebbers, Robbert, Birkedal, Lars, & Dreyer, Derek. 2016. Higher-order ghost state. *Pages 256–269 of:* Garrigue, Jacques, Keller, Gabriele, & Sumii, Eijiro (eds), *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*. ACM.

Khyzha, Artem, Gotsman, Alexey, & Parkinson, Matthew J. 2016. A Generic Logic for Proving Linearizability. *Pages 426–443 of: FM*.

Khyzha, Artem, Dodds, Mike, Gotsman, Alexey, & Parkinson, Matthew J. 2017. Proving Linearizability Using Partial Orders. *Pages 639–667 of:* Yang, Hongseok (ed), *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017. Proceedings*. Lecture Notes in Computer Science, vol. 10201. Springer.

Kiselyov, Oleg, Sabry, Amr, & Swords, Cameron. 2013. Extensible effects: an alternative to monad transformers. *Pages 59–70 of:* Shan, Chung-chieh (ed), *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Haskell 2013*. ACM.

Kleymann, Thomas. 1999. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, **11**, 541–566.

Kowaltowski, Tomasz. 1977. Axiomatic Approach to Side Effects and General Jumps. *Acta Inf.*, **7**.

Krebbers, Robbert, Jung, Ralf, Bizjak, Ales, Jourdan, Jacques-Henri, Dreyer, Derek, & Birkedal, Lars. 2017a. The Essence of Higher-Order Concurrent Separation Logic. *Pages 696–723 of:* Yang, Hongseok (ed), *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Proceedings*. Lecture Notes in Computer Science, vol. 10201. Springer.

Krebbers, Robbert, Timany, Amin, & Birkedal, Lars. 2017b. Interactive proofs in higher-order concurrent separation logic. *Pages 205–217 of:* Castagna, Giuseppe, & Gordon, Andrew D. (eds), *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. ACM.

Krishnaswami, Neelakantan R. 2011. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. Ph.D. thesis, Carnegie Mellon University.

Lamport, Leslie. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.*, **3**(2), 125–143.

Leijen, Daan, Fähndrich, Manuel, & Burckhardt, Sebastian. 2011. Prettier concurrency: purely functional concurrent revisions. *Pages 83–94 of:* Claessen, Koen (ed), *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011*. ACM.

Ley-Wild, Ruy, & Nanevski, Aleksandar. 2013. Subjective auxiliary state for coarse-grained concurrency. *In:* (Giacobazzi & Cousot, 2013).

Liang, Hongjin, & Feng, Xinyu. 2013. Modular verification of linearizability with non-fixed linearization points. *Pages 459–470 of:* Boehm, Hans-Juergen, & Flanagan, Cormac (eds), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013*. ACM.

Liang, Sheng, Hudak, Paul, & Jones, Mark P. 1995. Monad Transformers and Modular Interpreters. *Pages 333–343 of:* Cytron, Ron K., & Lee,

Peter (eds), *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press.

Moggi, Eugenio. 1989. Computational Lambda-Calculus and Monads. *Pages 14–23 of: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89)*. IEEE Computer Society.

Morrisett, Greg, & Uustalu, Tarmo (eds). 2013. *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*. ACM.

Morrison, Adam. 2016. Scaling synchronization in multicore programs. *Commun. ACM*, **59**(11), 44–51.

Morrison, Adam, & Afek, Yehuda. 2013. Fast Concurrent Queues for x86 Processors. *Pages 103–112 of: PPoPP*. New York, NY, USA: ACM.

Nanevski, Aleksandar. 2016. *Separation Logic and Concurrency*. Oregon programming languages summer school. http://software.imdea.org/~aleks/oplss16/notes.pdf.

Nanevski, Aleksandar, Morrisett, Greg, & Birkedal, Lars. 2006. Polymorphism and separation in hoare type theory. *Pages 62–73 of:* Reppy, John H., & Lawall, Julia L. (eds), *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*. ACM.

Nanevski, Aleksandar, Morrisett, J. Gregory, & Birkedal, Lars. 2008a. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, **18**(5-6), 865–911.

Nanevski, Aleksandar, Morrisett, Greg, Shinnar, Avraham, Govereau, Paul, & Birkedal, Lars. 2008b. Ynot: dependent types for imperative programs. *Pages 229–240 of:* Hook, James, & Thiemann, Peter (eds), *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008*. ACM.

Nanevski, Aleksandar, Vafeiadis, Viktor, & Berdine, Josh. 2010. Structuring the verification of heap-manipulating programs. *Pages 261–274 of:* Hermenegildo, Manuel V., & Palsberg, Jens (eds), *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. ACM.

Nanevski, Aleksandar, Ley-Wild, Ruy, Sergey, Ilya, & Delbianco, Germán Andrés. 2014a. Communicating State Transition Systems for Fine-Grained Concurrent Resources. *Pages 290–310 of:* Shao, Zhong (ed), *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014. Proceedings*. Lecture Notes in Computer Science, vol. 8410. Springer.

Nanevski, Aleksandar, Ley-Wild, Ruy, Sergey, Ilya, & Delbianco, Germán Andrés. 2014b. *Communicating State Transition Systems for Fine-Grained Concurrent Resources (Extended Version).* Available from http://software.imdea.org/fcsl/papers/histories-extended.pdf. Extended version of (Nanevski *et al.* , 2014a).

Ni, Zhaozhong, & Shao, Zhong. 2006. Certified assembly programming with embedded code pointers. *Pages 320–333 of:* Morrisett, J. Gregory, & Jones, Simon L. Peyton (eds), *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006.* ACM.

O'Hearn, Peter W. 2007. Resources, concurrency, and local reasoning. *Th. Comp. Sci.*, **375**(1-3), 271–307.

O'Hearn, Peter W., Reynolds, John C., & Yang, Hongseok. 2001. Local Reasoning about Programs that Alter Data Structures. *Pages 1–19 of:* Fribourg, Laurent (ed), *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Proceedings.* Lecture Notes in Computer Science, vol. 2142. Springer.

O'Hearn, Peter W., Rinetzky, Noam, Vechev, Martin T., Yahav, Eran, & Yorsh, Greta. 2010. Verifying linearizability with hindsight. ACM.

Owicki, Susan S., & Gries, David. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM*, **19**(5), 279–285.

Park, David Michael Ritchie. 1981. Concurrency and Automata on Infinite Sequences. *Pages 167–183 of:* Deussen, Peter (ed), *Theoretical Computer Science, 5th GI-Conference, Proceedings.* Lecture Notes in Computer Science, vol. 104. Springer.

Petersen, Rasmus Lerchedahl, Birkedal, Lars, Nanevski, Aleksandar, & Morrisett, Greg. 2008. A Realizability Model for Impredicative Hoare Type Theory. *Pages 337–352 of:* Drossopoulou, Sophia (ed), *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008. Proceedings.* Lecture Notes in Computer Science, vol. 4960. Springer.

Petrank, Erez, & Timnat, Shahar. 2013. Lock-Free Data-Structure Iterators. *Pages 224–238 of:* Afek, Yehuda (ed), *Distributed Computing - 27th International Symposium, DISC 2013. Proceedings.* Lecture Notes in Computer Science, vol. 8205. Springer.

Plotkin, Gordon D., & Power, A. John. 2004. Computational Effects and Operations: An Overview. *Electr. Notes Theor. Comput. Sci.*, **73**.

Plotkin, Gordon D., & Power, John. 2002. Notions of Computation Determine Monads. *In: FoSSaCS*.

Plotkin, Gordon D., & Power, John. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures*, **11**(1).

Raad, Azalea, Villard, Jules, & Gardner, Philippa. 2015. CoLoSL: Concurrent Local Subjective Logic. *Pages 710–735 of:* Vitek, Jan (ed), *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015. Proceedings*. Lecture Notes in Computer Science, vol. 9032. Springer.

Raynal, Michel. 2013. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer.

Reppy, John H. 1999. *Concurrent Programming in ML*. Cambridge, England: Cambridge University Press.

Reynolds, John C. 1993. The Discoveries of Continuations. *LISP and Symbolic Computation*, **6**(3-4).

Reynolds, John C. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. *Pages 55–74 of: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), Proceedings*. IEEE Computer Society.

Saabas, Ando, & Uustalu, Tarmo. 2007. A compositional natural semantics and Hoare logic for low-level languages. *Theor. Comput. Sci.*, **373**(3), 273–302.

Sabry, Amr, & Felleisen, Matthias. 1993. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, **6**(3-4), 289–360.

Schellhorn, Gerhard, Wehrheim, Heike, & Derrick, John. 2012. How to Prove Algorithms Linearisable. *Pages 243–259 of:* Madhusudan, P., & Seshia, Sanjit A. (eds), *Computer Aided Verification - 24th International Conference, CAV 2012. Proceedings*. Lecture Notes in Computer Science, vol. 7358. Springer.

Schwinghammer, Jan, Birkedal, Lars, Reus, Bernhard, & Yang, Hongseok. 2011. Nested Hoare Triples and Frame Rules for Higher-order Store. *Logical Methods in Computer Science*, **7**(3:21), 1–42.

Sergey, Ilya, Nanevski, Aleksandar, & Banerjee, Anindya. 2015a. Mechanized verification of fine-grained concurrent programs. *Pages 77–87 of:* Grove, David, & Blackburn, Steve (eds), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*. ACM.

Sergey, Ilya, Nanevski, Aleksandar, & Banerjee, Anindya. 2015b. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. *Pages 333–358 of:* Vitek, Jan (ed), *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015. Proceedings.* Lecture Notes in Computer Science, vol. 9032. Springer.

Sergey, Ilya, Nanevski, Aleksandar, Banerjee, Anindya, & Delbianco, Germán Andrés. 2016. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. *Pages 92–110 of:* Visser, Eelco, & Smaragdakis, Yannis (eds), *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016.* ACM.

Sozeau, Matthieu, & Tabareau, Nicolas. 2014. Universe Polymorphism in Coq. *Pages 499–514 of:* Klein, Gerwin, & Gamboa, Ruben (eds), *Interactive Theorem Proving - 5th International Conference, ITP 2014. Proceedings.* Lecture Notes in Computer Science, vol. 8558. Springer.

Springer, George, & Friedman, Daniel P. 1989. *Scheme and the Art of Programming.* MIT Press and McGraw-Hill.

Støvring, Kristian, & Lassen, Søren B. 2007. A complete, co-inductive syntactic theory of sequential control and state. *Pages 161–172 of:* Hofmann, Martin, & Felleisen, Matthias (eds), *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007.* ACM.

Strachey, Christopher, & Wadsworth, Christopher P. 2000. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symbolic Computation*, **13**(1/2), 135–152.

Svendsen, Kasper, & Birkedal, Lars. 2014. Impredicative Concurrent Abstract Predicates. *Pages 149–168 of:* Shao, Zhong (ed), *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Proceedings.* Lecture Notes in Computer Science, vol. 8410. Springer.

Svendsen, Kasper, Birkedal, Lars, & Nanevski, Aleksandar. 2011. Partiality, State and Dependent Types. *Pages 198–212 of:* Ong, C.-H. Luke (ed), *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011. Proceedings.* Lecture Notes in Computer Science, vol. 6690. Springer.

Svendsen, Kasper, Birkedal, Lars, & Parkinson, Matthew J. 2013. Modular Reasoning about Separation of Concurrent Data Structures. *Pages 169–188 of:* Felleisen, Matthias, & Gardner, Philippa (eds), *Proceedings of the 22nd European Symposium on Programming Languages and Systems*

(ESOP 2013). Lecture Notes in Computer Science, vol. 7792. Rome, Italy: Springer.

Swamy, Nikhil, Weinberger, Joel, Schlesinger, Cole, Chen, Juan, & Livshits, Benjamin. 2013. Verifying higher-order programs with the dijkstra monad. *Pages 387–398 of:* Boehm, Hans-Juergen, & Flanagan, Cormac (eds), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013*. ACM.

Swierstra, Wouter. 2009. A Hoare Logic for the State Monad. *Pages 440–451 of:* Berghofer, Stefan, Nipkow, Tobias, Urban, Christian, & Wenzel, Makarius (eds), *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009. Proceedings*. Lecture Notes in Computer Science, vol. 5674. Springer.

Tan, Gang, & Appel, Andrew W. 2006. A Compositional Logic for Control Flow. *Pages 80–94 of:* Emerson, E. Allen, & Namjoshi, Kedar S. (eds), *VMCAI*. Lecture Notes in Computer Science, vol. 3855. Springer.

The Coq Development Team. 2016. *The Coq Proof Assistant Reference Manual*. https://coq.inria.fr/distrib/current/files/Reference-Manual.pdf.

Thielecke, Hayo. 1997. *Categorical Structure of Continuation Passing Style*. Ph.D. thesis, University of Edimburgh, Edimburgh, UK.

Thielecke, Hayo. 2009. Control effects as a modality. *J. Funct. Program.*, **19**(1), 17–26.

Turon, Aaron, Dreyer, Derek, & Birkedal, Lars. 2013a. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. *In:* (Morrisett & Uustalu, 2013).

Turon, Aaron Joseph, Thamsborg, Jacob, Ahmed, Amal, Birkedal, Lars, & Dreyer, Derek. 2013b. Logical relations for fine-grained concurrency. *In:* (Giacobazzi & Cousot, 2013).

Vafeiadis, Viktor. 2008. *Modular fine-grained concurrency verification*. Ph.D. thesis, University of Cambridge, UK.

Vafeiadis, Viktor, & Parkinson, Matthew J. 2007. A Marriage of Rely/Guarantee and Separation Logic. *Pages 256–271 of:* Caires, Luís, & Vasconcelos, Vasco Thudichum (eds), *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Proceedings*. Lecture Notes in Computer Science, vol. 4703. Springer.

Vafeiadis, Viktor, Herlihy, Maurice, Hoare, Tony, & Shapiro, Marc. 2006. Proving correctness of highly-concurrent linearisable objects. *Pages 129–136 of:* Torrellas, Josep, & Chatterjee, Siddhartha (eds), *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006*. ACM.

Wadler, Philip. 1994. Monads and Composable Continuations. *Lisp and Symbolic Computation*, **7**(1), 39–56.

Yoshida, Nobuko, Honda, Kohei, & Berger, Martin. 2008. Logical Reasoning for Higher-Order Functions with Local State. *Logical Methods in Computer Science*, **4**(4).