QUEEN MARY UNIVERSITY OF LONDON

Session Types, Concurrent Separation Logic & Algebra

by

Akbar Hussain

Submitted for the degree of Doctor of Philosophy

School of Electronic Engineering and Computer Science

March 2013

Declaration of Authorship

I, Akbar Hussain, declare that this dissertation titled, ‘Session Types, Separation Logic and Algebra’ and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at Queen Mary University of London.
- Where I have consulted the published work of others, this has fully been acknowledged in accordance with the standard referencing practices of the discipline.
- Where the dissertation is based on work done by myself jointly with others, I have made clear what was done by others and what I have contributed myself.
- Parts of this work referenced with [21] is published as:
  C.A.R Hoare, A. Hussain, B. Moller, P.W. O’Hearn, R.L. Petersen and G. Struth, On Locality and the Exchange Law for Concurrent Processes. In CONCUR 2011.

Akbar Hussain

1st March 2013

In the name of Allah the most merciful, the especially merciful.

"He brings the living out of the dead and brings the dead out of the living and brings to life the earth after its lifelessness. And thus will you be brought out.

And of His signs is that He created you from dust; then, suddenly you were human beings dispersing [throughout the earth].

And of His signs is that He created for you from yourselves mates that you may find tranquility in them; and He placed between you affection and mercy. Indeed in that are signs for a people who give thought.

And of His signs is the creation of the heavens and the earth and the diversity of your languages and your colours. Indeed in that are signs for those of knowledge.

And of His signs is your sleep by night and day and your seeking of His bounty. Indeed in that are signs for a people who listen.

And of His signs is [that] He shows you the lightning [causing] fear and aspiration, and He sends down rain from the sky by which He brings to life the earth after its lifelessness. Indeed in that are signs for a people who use reason.

And of His signs is that the heaven and earth stand [i.e., remain] by His command. Then when He calls you with a [single] call, from the earth, immediately you will come forth."

[The Qur'an: Chapter 30, Verses 19-25]

"The intelligent man is the one who subjugates himself and works for what will come after death. The stupid man is the one who follows his own whims and hopes that his desires will be gratified by Allah."

[Saying of Muhammad, the Messenger of Allah, *peace be upon him*. (at-Tirmidhi)]

Abstract

This dissertion explores the relation between two formalisms and one algebraic framework for concurrency. Session Types and Concurrent Separation Logic are formalisms that support independent reasoning about concurrent processes, and our motivating question is whether their modularity springs from the same source despite the distance between their models. We first translate a small language we call Baby Session Types (BST), into a 'basic' version of Concurrent Separation Logic (BCSL), and we show that the translation is sound. We then describe a model for Separation Logic (SL) based on Actions, which exhibits some of the structure of a Concurrent Kleene Algebra, an algebra where operators for parallel and sequential composition are linked by a version of the exchange law from category theory. The model connects the algebraic notions to locality concepts that underlie Separation Logic. We then move on to provide a more general construction of an algebra model of BCSL, which can be built from (Baby) Session Types.

Thus, we end up with a model that brings together concepts from all of Session Types, Separation Logic, and Concurrent Kleene Algebra. Thus, the model links diverse models of concurrency. In addition to this it suggests alterations of the algebraic axioms as well as the foundational models underlying Separation Logic. It is hoped that, apart from these specific results, this dissertation can in some modest way contribute to unification in concurrency theory, a theory (or theories) based presently on diverse models.

Acknowledgements

In the name of Allah, the most merciful, the especially merciful

Verily all praise is for Allah (the one worthy of all worship), we praise Him and seek His aid and ask for His forgiveness, and we seek refuge with Allah from the evils of ourselves and our evil actions. Whomsoever Allah guides there is none who can misguide him, and whomsoever Allah misguides there is none who can guide him. And I bear witness that none has the right to be worshipped except Allah Alone, having no partner, and I bear witness that Muhammad is His slave and His Messenger.

Verily the most truthful speech is the Word of Allah and the best guidance is the guidance of Muhammad (may ever increasing peace and blessings be upon him, his family and his companions), and the worst of affairs are the novelties (in religion) and every novelty is an innovation and every innovation (in religion) is a going astray and every going astray is in the Hell fire.

After thanking Allah, I would like to thank my mother and late father who have always fully supported and trusted me in the decisions I have undertaken in life. Although my father passed away while I was only a teenager, the lessons he taught me have certainly played a big part in everything that I have achieved. I pray that Allah preserves my mother and gives her the best of this life and hereafter and forgives any faults of my father and grants him a high status in Heaven, Ameen.

I am ever so grateful to Allah, who enabled me to complete this Thesis and facilitated it for me by placing me amongst two outstanding people - Professor Peter W. O'Hearn and Dr Rasmus L. Petersen - without whom I would not have been able to complete this work. The generosity of their nature and dedication to their students has always been apparent to me and I consider myself to be very fortunate to work under them and learn from them. Rasmus continually explained and tolerantly bore the countless number of questions that I had and I thank him for his patience. I am deeply grateful to Peter's support as my advisor. I cannot say in words how much I appreciate his support and patience throughout the past years, I will not even try. His belief and confidence in my abilities from my first year of undergraduate study has always spurred me on to achieve. I can only sincerely pray that Allah guides both Peter and Rasmus to the success in this life and the life to come.

My sister Shofna greatly eased my affairs by looking after my daughter while both my wife and I went to study, even though she her self suffers from ill health.

A key person who has supported me emotionally and mentally is my wife Saajidah who has long-suffered with patience the many hours that I have been away from the family even whilst at home. Her hard work and dedication in single handedly looking after our daughters whilst completing her own very demanding teaching qualification has been inspirational and amazing.

Finally, I would like to thank the apples of my eye, Maryam and Khadeejah, who have been for me a form of stress relief, giving me a mental break from the challenges that I faced throughout this work. Maryam being the elder of the two has endured hardship; having to move from place to place whilst she was being looked after by others. Although they may not understand now, I hope they know that I love them both dearly and truly appreciate the hardship that both have suffered as my time and my mind have been occupied by my tasks. I now hope that I can spend more time with my daughters and make up for the lost time!

This research was financially supported by the Engineering and Physical Sciences Research Council and the School of Electronic Engineering and Computer Science, Queen Mary, University of London.

Contents

1 Introduction **12**

1.1 General Context 12

1.2 Reasoning About Concurrency 14

1.3 This Work 20

1.4 Contribution 26

1.5 Outline 27

1.6 Related Work 28

2 Formalisms **29**

2.1 Session Types (ST) & Message-Passing 30

2.2 Baby Session Types (BST) 32

2.2.1 Operational Semantics and Subject Reduction 39

2.3 Concurrent Separation Logic & Shared-Memory 42

2.4 Basic Concurrent Separation Logic (BCSL) 47

2.4.1 Heap Model Instantiation 48

2.5 Algebra 49

2.5.1 Concurrent Kleene Algebra 49

2.5.2 Trace Model [21] 50

2.5.3 Interpretation of Triples 50

3 Translation from BST to BCSL **53**

3.1 Session Instantiation of BCSL: BCSL/ST 54

3.2 Translation. 56

3.2.1 Recursion 63

3.3 Discussion 67

4 Exchange & Locality **68**

4.1 State Transformers 69

4.2 Exchange 74

4.3 Local Action 78

4.3.1 Localisation 82

4.4 Relating Algebraic & Semantic Triples 85

4.4.1 Hoare Triples 85

4.4.2 Plotkin Triples 87

4.4.3 Counterexample to Associativity of $*$ 88

4.5 Predicate Transformers: The Resource Model 89

4.5.1 Local Elements 93

4.5.2 Summary 95

5 A Generalised Model of BCSL **96**

5.1 Generalised Predicate Transformer Model 97

5.1.1 Properties of the Model 101

5.1.2 Dijkstra's Healthiness Condition 108

5.1.3 Completeness Results: Connecting Various Triples 112

5.1.4 Connection to the Resource Model 117

5.2 On the Session Types Instantiation 119

5.3 Lattice Structure 120

5.4 Summary of the Algebraic Structure 122

6 Locality Bimonoid **123**

7 Conclusion **127**

Bibliography **129**

List of Figures

1.1 Concurrent Processes 12
1.2 Possible Interleaving of the Program in Figure 1.1 13
1.3 Adder Program 15
1.4 Parallel Pointer Program 16
1.5 Parallel Pointer Program Proof 17
1.6 Communicating Processes 19
1.7 Concurrency & Frame from CSL 23
1.8 Formalisms 23
1.9 Models 25

2.1 Proof Rules for Baby Session Types 34
2.2 Ownership Transfer Example in BST 36
2.3 Inconsistent Context Derivation 38
2.4 Sequential Composition Example in SL 43
2.5 Proof Tree for example 2.4 44
2.6 Concurrent Composition Example in CSL 45
2.7 Buffer Sharing Program 46
2.8 Proof of Buffer Sharing Program in CSL 46
2.9 Proof Rules for Basic Concurrent Separation Logic 48

3.1 Specialised Rules of Session Instantiation 55
3.2 Translation from BST to BCSL/ST 56
3.3 Derivation of a program in BCSL/ST not derivable in BST. 58
3.4 Part A of the derivation in Figure 3.3 59
3.5 Part B of the derivation in Figure 3.3 59
3.6 Derivation of a program in BCSL/ST not derivable in BST with weakening . . . 61
3.7 Deadlock derivable example in BST 62
3.8 BST Recursion Rules 64

3.9 Type derivation of a recursive program in BST 65

3.10 Hoare Logic Procedure Rules 66

4.1 Non-Local & Local Action 82

4.2 Resource Model Operators 89

5.1 Predicate Transformer Commands 100

5.2 Connection Between RM and GPTM 117

For

Saajidah, Maryam & Khadeejah

Chapter 1

Introduction

1.1 General Context

Great emphasis has been given to correctness proofs for sequential programs but there is still a great need for similar work to be done with parallel programs. This is due to the complexity involved with processes executing in parallel, where the results can depend on the unpredictable order in which the actions from parallel processes are executed. For example the following processes in Figure 1.1 can interact in six different ways resulting in the value of y having four different possible outcome shown in Figure 1.2.

Process 1	Process 2
$x := 1;$	$x := 2;$
$y := x + 1;$	$y := 5 * x;$

Figure 1.1: Concurrent Processes

Traditionally, it is because of what Hoare [20] refers to as “fantastic number of combinations involved in arbitrary interleaving” that concurrent programs are difficult to design and verify. Even program testing becomes more difficult for concurrent than sequential programs, because during testing it might not be easy to reproduce the conditions which led to a particular interleaving, which led to a particular program error. In his classic text on operating systems, Brinch Hansen summed up the matter as follows.

> "The main difficulty of multiprogramming is that concurrent activities can interact in a time-dependent manner which makes it practically impossible to locate programming errors by systematic testing. Perhaps, more than anything else, this explains the difficulty of making operating systems reliable." [16]

$x := 1;$	$x := 1;$	$x := 1;$
$y := x + 1;$	$x := 2;$	$x := 2;$
$x := 2;$	$y := x + 1;$	$y := 5 * x;$
$y := 5 * x;$	$y := 5 * x;$	$y := x + 1;$
$(\mathbf{x = 2},\ \mathbf{y = 10})$	$(\mathbf{x = 2},\ \mathbf{y = 10})$	$(\mathbf{x = 2},\ \mathbf{y = 3})$
$x := 2;$	$x := 2;$	$x := 2;$
$x := 1;$	$x := 1;$	$y := 5 * x;$
$y := x + 1;$	$y := 5 * x;$	$x := 1;$
$y := 5 * x;$	$y := x + 1;$	$y := x + 1;$
$(\mathbf{x = 1},\ \mathbf{y = 5})$	$(\mathbf{x = 1},\ \mathbf{y = 2})$	$(\mathbf{x = 1},\ \mathbf{y = 2})$

Figure 1.2: Possible Interleaving of the Program in Figure 1.1

Much work has been done aimed at combating these problems. In language design we have such concepts as *semaphores*, *monitors* and *message-passing* [2], all of which are aimed (at least partly) at giving the programmer abstractions to control time-dependent errors. In verification there have been techniques such as partial order reduction in model checking [13] and compositional proof techniques (more on those below) to combat the complexity of interleavings mentioned by Hoare. But research on these problems continues, and no comprehensive solution has been found.

These problems we have mentioned go back to the 1960s: They are enduring problems in programming and in verification. However, currently we are witnessing a surge in the development of parallelism in computer hardware which makes the problems all the more acute, and even more practically relevant.

Sutter and Larus [33, 49], discuss this revolution in concurrency and mention the reality that we face.

> "Concurrency has long been touted as the "next big thing" and "the way of the future," but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software." [33]

Due to this, it is expected that programmers will need to increasingly program with concurrency in mind in the future. As a result, questions are being raised about the capabilities of programming languages and there is an increased need for programming languages to cater for concurrency. Of course, the fundamental problems of concurrency remain and there are no quick fixes; it is just that in industry and the broader scientific community, there is a growing awareness of the importance of these problems, as Sutter and Laurus (among others) have emphasised. The ripple effect continues further to theoreticians who now are under extreme pressure to understand concurrency better and hopefully provide tools to allow software to react to the demands and challenges posed by new hardware, without compromising the quality of software that is produced.

This provides the general context for the work in this dissertation on reasoning about concurrency. In this dissertation I will be studying the problem of modular reasoning about concurrency. Before I move on to talk about my own contributions, the next section describes some of the more specific context on modular reasoning.

1.2 Reasoning About Concurrency

Early approaches to reasoning about concurrency were the axiomatic works of Hoare [20] and Owicki-Gries [43, 45], with Jones [30] suggesting a compositional way of verifying parallel programs in the rely-guarantee method (also, [48]). More recently, the development of Separation Logic [6, 40] has given a new perspective on a compositional way of verifying *shared-memory* parallel programs. Methods of modular or compositional reasoning, which can treat program components in isolation, are needed if we are to have proof or analysis techniques that scale, or that can be applied to program components without considering an entire program.

In [20] Hoare introduces some axioms to prove concurrent programs that are disjoint, where no variables that are changed in one process are accessed by any other process running in parallel except where the variable is considered a resource to which access is forced to have mutual exclusion properties. I assume the reader to have knowledge of Hoare Logic and I will only be

referring to the parallel aspects. The basic inference rule for parallel processes extending Hoare Logic (sequential) would be:

$$\frac{\{P_1\}\,S_1\,\{Q_1\}\cdots\{P_n\}\,S_n\,\{Q_n\}}{\{P_1\ \wedge\cdots\ \wedge\ P_n\}\,S_1\parallel\cdots\parallel\ S_n\,\{Q_1\ \wedge\cdots\ \wedge\ Q_n\}}$$

where P and Q are pre-conditions and post-conditions for the statement S providing that the processes do not interact with each other. This notion of 'do not interact with each other' is formalised using side conditions on variable access where any variable written to in one process is neither read from nor written to in another. This has the effect, for example, of ruling out programs like $x := x+1 \parallel x := x \times 2$.

Disjoint processes of this variety are obviously very restrictive. To allow disciplined interaction between processes, Hoare [20] uses the notion of critical region (illustrated by Figure 1.3) whereby mutual exclusion is forced on the critical regions of all parallel processes. An invariant is required for the resource to make sure that the state of the resource is in a valid state outside of any critical regions.

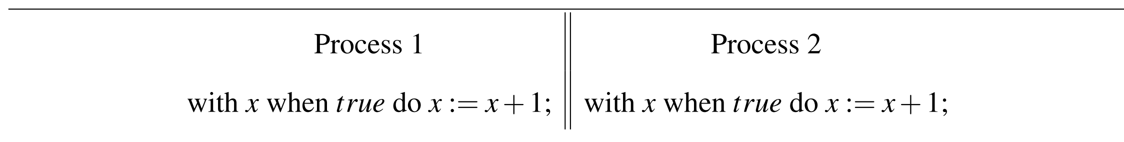


Figure 1.3: Adder Program

Hoare's approach has two major limitations and these are as follows.

1. No interference at all is allowed, except in critical regions.

2. Proofs of programs using shared variables in assertions are not allowed.

The problem with limitation 1 is that it rules out many natural parallel programs, including classic examples like Dekker's algorithm [2]. A reaction to limitation 1 is given by Owicki-Gries [44] who provide a more powerful version of Hoares parallel axiom, where the two conditions that are required by Hoare are replaced by the notion of *interference freeness* with regards to the processes. The basic idea is that each process is proved separately and then checked to see if the subcommands in one process interfere with the assertions used in the proof of the other process. The details of how to check for *interference freeness* is given in [44]. A reaction to limitation 2 is also given by Owicki- Gries in a separate work, where they adopt a restricted approach closer to [20], and include an additional inference rule for *Auxiliary Variables*; further details can be found in [43, 45].

Although the work by Owicki-Gries can be used to prove simple programs, due to the complexity of parallelism this technique may result in very long proofs and the length would increase for larger programs. This can be seen from the number of *interference free* checks that must be done, which is exponential in the length of the program. This approach is also thought to be non-compositional because of the interference checking. That is, you cannot know you have a proof in isolation, without knowing about the other processes.

The non-compositional nature of the Owicki-Gries method of interference checking was addressed in the work of Cliff Jones through the Rely-Guarantee proof method [30]. In brief, Jones keeps extra information, in addition to pre-conditions and post-conditions, to keep track of interactions with the environment. The idea is very simple: Just as a pre-condition records assumptions the developer can make about initial state when designing an implementation, a *rely-condition* records assumptions that can be made about interference from other processes. Similarly just as a post-condition records commitments which must be fulfilled by the implementation, the interference which can be generated by the implementation is captured by giving a *guarantee-condition*. In Jones's approach the specification for a statement S is now written $\{P,R\}\,S\,\{G,Q\}$ where P is the pre-condition, R is the *rely-condition*, G is the *guarantee-condition* and Q is the post-condition.

We will not discuss the Rely-Guarantee method further in this dissertation, except to acknowledge its rightful place amongst compositional proof techniques. The other important work we mention is by Stirling [48], who shows how Owicki-Gries can be reformulated in a compositional manner, following the ideas of Rely-Guarantee. O'Hearn [40] and Vafeiadis [53] give further discussion of the relation of Rely-Guarantee to Separation Logic and to Hoare's ideas from [20] (and therefore, to the work that concerns us in this dissertation).

So far only assignments to simple variables with no indirect addressing have been considered and the above mentioned approaches do not cater for programs that use pointers. Take, for example the following program in Figure 1.4.

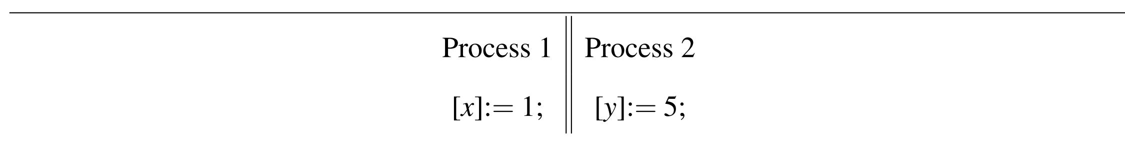


Figure 1.4: Parallel Pointer Program

In Figure 1.4 we have a program where x and y could be aliases. As a result there would be a *race* between the two processes and this would not be detected with the syntactic restrictions for process interaction in the work by Hoare [20] and Owicki-Gries [43, 45] discussed earlier.

Concurrent Separation Logic is a development of the ideas of Hoare [20] and one of the two Owicki-Gries works [45]. Its main difference is that it uses the separating conjunction connective to partition the spate between processes, instead of syntactic restrictions. To see the basic idea, consider the following program proof (or proof outline) in Figure 1.5.

Process 1	Process 2
$\{x \mapsto 3 * y \mapsto 3\}$	
$\{x \mapsto 3\}$	$\{y \mapsto 3\}$
$[x] := 1;$	$[y] := 5;$
$\{x \mapsto 1\}$	$\{y \mapsto 5\}$
$\{x \mapsto 1 * y \mapsto 5\}$	

Figure 1.5: Parallel Pointer Program Proof

Here, we are using $[x]$ to refer to a position in RAM memory corresponding to address x. In this example x and y could be aliases in which case there would be a *race* between the two processes. Generally, a *race* occurs when one process tries to write to a memory address when another is trying to either read from it or write to it (i.e., not separated by synchronisation). The proof rule

$$\frac{\{P_1\}\, S_1\, \{Q_1\} \cdots \{P_n\}\, S_n\, \{Q_n\}}{\{P_1 \wedge \cdots \wedge P_n\}\, S_1 \parallel \cdots \parallel S_n\, \{Q_1 \wedge \cdots \wedge Q_n\}}$$

of Hoare discussed earlier [20] relies on, among other things, ensuring that the different processes S_i do not *race* with one another. This can be achieved using static checks when only assignments to simple variables are present: but with indirect addressing $[x]$, or other forms of pointer indirection, it is difficult to do such *race* checking statically.

Concurrent Separation Logic (CSL) addresses this issue by using a logical connective, the separating conjunction, to manage dynamic rather than static separation of resources between processes. The separating conjunction $P * Q$ is true when the memory or RAM can be divided into parts, one of which satisfies P and the other satisfies Q. Given this, the proof rule for parallel composition becomes:

$$\frac{\{P_1\}\, S_1\, \{Q_1\} \cdots \{P_n\}\, S_n\, \{Q_n\}}{\{P_1 * \cdots * P_n\}\, S_1 \parallel \cdots \parallel S_n\, \{Q_1 * \cdots * Q_n\}}$$

Returning to our example program which involves potential pointer aliasing, an assertion of the form

$$x \mapsto 3 * y \mapsto 3$$

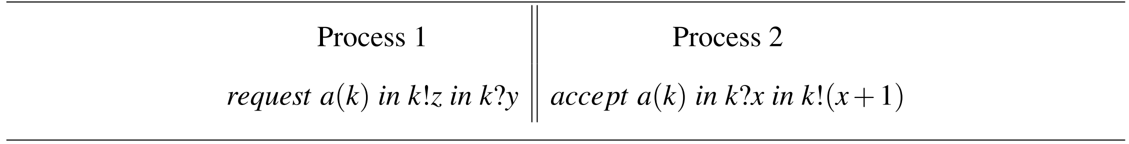
says that both x and y point to 3, but they do so in separate areas of memory: they are not aliases. This assertion is in the form necessary to use as a pre-condition in the conclusion of the parallel rule and, indeed, can be used in a program proof as indicated in the proof outline in Figure 1.5.

O'Hearn [40] has shown that the use of a logical connective, instead of syntactic conditions, to track resource separation is very powerful, and he has given a sequence of examples (parallel mergesort, pointer-copying buffer, a memory manager) that previously resisted modular proofs.

This concludes our brief review of program logics for modular reasoning about concurrency. Alongside the axiomatic methods, there has been some development in type systems for controlling concurrency, particularly in the increasingly influential theory of Session Types (ST) [28, 54, 57]. Session Types provide a modular way of reasoning about *message-passing* programs. They are based on the extremely powerful form of *message-passing* popularised by pi-calculus, where channels or end-points of channels can themselves be passed as the contents of messages.

Parallel programming is commonly used for communicating processes such as message-passing client/server systems. Communicating processes have their own class of problems such as communication mismatch or interference from a third party when two processes are communicating with each other. An early session typing system is presented by Honda and others [28, 50]. The language consists of a basic grammar for communication such as *data passing*, *receiving*, *delegation*, *labeling* and *branching*. The type system provide communication patterns for each process and their interaction with other processes. The type system is used to claim that well typed programs are free from run time errors and communication mismatch.

Interference from a third party on two communicating processes is tackled by the use of hidden channels via which the two processes communicate, hence a third party cannot interfere due to not having access to the channel. The communications are called sessions and the communication pattern of each process is its *type*. Take for example the following program in Figure 1.6 where one process sends a number and receives its successor, calculated and sent by the reciprocating process:



Process 1	Process 2
request $a(k)$ *in* $k!z$ *in* $k?y$	*accept* $a(k)$ *in* $k?x$ *in* $k!(x+1)$

Figure 1.6: Communicating Processes

Here *accept* and *request* create a new fresh channel that only these two processes can use. $k!$ and $k?$ denote sending and receiving on channel k respectively. From Figure 1.6 the following types are constructed:

$$\text{Process 1} \ = ![nat];\ ?[nat]$$

$$\text{Process 2} \ = ?[nat];\ ![nat]$$

These types are compatible in the sense that one process has in-out structure while the other has an opposite out-in structure. They are called *co-types* in session types work. Using the concept of *co-type* the issue of miscommunication is handled, and the session type system is able to detect, for example, if one process is waiting to receive a value of type $[bool]$ while the other process sends a value of type $[nat]$.

Session Types also allow for multiple processes to access the same service, for example in a server/client set up where there is one server and many clients. Each client requests a service from the server for which the server and client initiate a session with their own unique channel; hence each client's communication with the server is done privately without any interference from another process.

The main sequent of the typing system has the form:

$$P \triangleright \Delta$$

which reads "a process P has typing Δ". Where the typing Δ specifies P's behaviour. The subsequent concurrent typing rule takes the following form:

$$\frac{P \triangleright \Delta \quad Q \triangleright \Delta'}{P \mid Q \triangleright \Delta \circ \Delta'}(\Delta \asymp \Delta')$$

where $(\Delta \asymp \Delta')$ corresponds to the compatibility of the types as mentioned previously and $\Delta \circ \Delta'$ is the combined typing context of the communication process.

Having introduced Concurrent Separation Logic and Session Types, the reader might have noticed a pattern. It is easy to see if we place the main inference rules side-by-side:

$$[\text{CSL Par}]\ \frac{\{X_1\}\,c_1\,\{Y_1\} \quad \{X_2\}\,c_2\,\{Y_2\}}{\{X_1 * X_2\}\,c_1 \parallel c_2\,\{Y_1 * Y_2\}} \qquad\qquad [\text{ST Par}]\ \frac{P_1 \rhd \Delta_1 \quad P_2 \rhd \Delta_2}{P_1 \parallel P_2 \rhd \Delta_1 \circ \Delta_2}$$

Session Types combines typing contexts with $\circ$, whereas CSL is combining assertions with $*$. Even without delving further into the details of $*$ or $\circ$, we can see a formal similarity, and we can see how parallel processes are reasoned about independently, in isolation from one another in each system. Furthermore, when we delve more deeply into the systems we will see more similarities, in their control over resources or avoidance of races. It is these similarities that sparked the work in this dissertation. We wondered: is this surface similarity just a coincidence, or something deeper?

This explains two of the subjects studied in this dissertation: Session Types and Concurrent Separation Logic. Then, while we were conducting our first study of connecting these formalisms, a new theory appeared: Concurrent Kleene Algebra [23, 26]. We say more on that below and in Chapter 2. Suffice it to say for now that Concurrent Kleene Algebra seemed like a promising unifying theory, which could be useful in linking diverse formalisms. We started working with the theory, with the intention of simply adopting it to support our study of Concurrent Separation Logic and Session Types. That was a rewarding choice, but in actuality it turned out that we could not simply adopt Concurrent Kleene Algebra, but rather we needed to alter it to fit the demands of connecting Separation Logic and Session Types. In the end, our work involved connecting all three of Concurrent Separation Logic, Session Types, and (an alteration of) Concurrent Kleene Algebra. In the next section we say more about our contribution.

1.3 This Work

As already mentioned, Session Types (ST) and Concurrent Separation Logic (CSL) are formalisms for modular reasoning about concurrent programs. A number of works advance ideas related to both CSL and ST in modular reasoning about *message-passing* programs [7, 27, 34, 55]. We mention also a significant line of work on typestate checking; see, e.g., [4] and its references. But as far as we know, no formal linkage between CSL and ST has previously been established. That is the first problem considered in this dissertation.

A difficulty in linking the two formalisms is that they are far apart in the languages they typically use to advance their ideas. CSL was defined for *shared-memory* concurrency, where the program store is used to share information between processes. It uses classic imperative programming constructs such as *critical regions* or *semaphores* to represent synchronisation between processes. In contrast, as we have already mentioned, Session Types use a communication model coming from pi-calculus, where *message-passing* is the only means of communication. There are no semaphores or critical regions, and no computer store.

Of course, it is possible to encode *message-passing* concurrency in *shared-memory* and conversely, but (at the current stage of knowledge) this encoding has not been shown to give rise to useable logics. For example, in his fundamental book [35], Milner shows how to encode imperative languages in Calculus of Communicating Systems (CCS), and he separately provides a modal logic for CCS. But, as far as we know, no one has used this modal logic to prove substantial imperative programs. Similarly, *message-passing* is often implemented using lower-level *shared-memory* mechanisms such as locks. But, such implementations would give rise to much more cumbersome proofs than for a dedicated logic of *message-passing*; for example, there are dedicated logics for pi-calculus [38] but, as far as we know, an implementation of pi primitives in terms of locks has not been used to reason about pi-calculus.

In any case, many researchers have noticed an intuitive similarity between Session Types and Concurrent Separation Logic: Each achieves independent reasoning about processes by controlling the usage of resources, be they message channels or heap memory. Both formalisms are modular and they look similar in some respects, but different in others. This prompted us to wonder if it might be possible to pin down some of the similarities or if maybe their modularity springs from the same source, if we look through the syntactic dissimilarities to the core of what is happening.

To ease comparison, we use a stripped-down version of Session Types which we call Baby Session Types (BST). BST retains the ability to pass end-points of channels in messages and it uses the resource-oriented typing characteristic of Session Types, but it removes some of the pi-calculus surroundings of Session Types formalisms (e.g., *replication*). We also utilise a simultaneously restricted and generalised form of Concurrent Separation Logic, which we call Basic Concurrent Separation Logic (BCSL). BCSL retains the concurrency proof rule of CSL while avoiding constructs such as *critical regions* or *semaphores*, and it treats pre-conditions and

post-conditions in a general way which does not require that they be interpreted semantically as in typical models of Separation Logic (e.g., as elements of a boolean residuated commutative monoid).

This then makes it possible to relate the two formalisms, and we do so by translating BST into BCSL; that is the subject matter of Chapter 3. The translation is sound in that typings are sent to Hoare triples in a way that preserves provability. The translation is, however, not complete. This can be seen on examples of deadlock, which are ruled out by BST but allowed by BCSL. The translation in this dissertation goes from BST to BCSL, and not the other way round, for the simple reason that Session Types do not contain post-conditions: they are in a sense a formalism of continuations (hence, pre-condition only, as in the 'Hoare doubles' used in logic for continuations [51]). If we were to retain expressive-enough structure in BST, then we might be able to provide a translation in the opposite direction. Translations have been done from functions to processes, using a variation on continuation-parsing style [36]. Pi-calculus makes use of replication and many to many rather than end to end channels. If we have all these then we might be able to do translation by including post-conditions to BST. However, we leave this question unanswered.

After this material on syntactic translations, we turn our attention to models and algebraic structure, taking our lead from the recent Concurrent Kleene Algebra (CKA, [22]). In general, what we do is look at models of Concurrent Separation Logic and see if they satisfy some of the properties found in CKA, or whether CKA can shed new light on the models and the logic. The most striking law of CKA is an ordered version of the Exchange Law from 2-categories or bi-categories [32].

$$\text{(EXCHANGE)} \quad (p * r);(q * s) \sqsubseteq (p;q) * (r;s)$$

In category theory, the exchange law is usually stated with an equivalence or isomorphism in place of the order: it is bi-directional, where in the version of the law that we use it is uni-directional. It can easily be seen why the exchange law holds in an interleaving model of concurrency. Suppose we have a trace t which is composed of traces t_1, t_2 such that $t = t_1;t_2$ where t_1 is an interleaving of two traces t_p and t_r, and t_2 is an interleaving of t_q and t_s. Then clearly t is also an interleaving of $t_p;t_q$ and $t_r;t_s$. Furthermore, as shown in [22], the Exchange Law validates the *Concurrency* and *Frame* rules from Concurrent Separation Logic (CSL) [6, 40], Figure 1.7, the origins of which was from a different model based on separation of resources.

$$\text{[Concurrency]}\ \frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P_1 * P_2\}\,C_1 \| C_2\,\{Q_1 * Q_2\}} \qquad \text{[Frame]}\ \frac{\{P\}\,C\,\{Q\}}{\{P * F\}\,C\,\{Q * F\}}$$

Figure 1.7: Concurrency & Frame from CSL

To derive these rules from the structure of CKA [26] is straightforward as will be shown in Section 2.5.3. Consequently, this means that the modular reasoning that these rules support may most likely be applicable more generally. This is a noteworthy situation to uncover, since historically the *Concurrency* and *Frame* rules have relied on subtle locality properties of the semantics of programs [9, 56], where the manner in which programs access resources is restricted. However, these laws which are based on these restrictions seem to easily hold in a CKA where there exists no such locality conditions for accessing resources. Furthermore, in [26] the authors refer to the validity of Hoare logic laws as being due to a cheat because of the fact that they use the same model for assertions and programs. This probed us to investigate whether the easy validity of Hoare Logic laws in CKA is necessarily based on trickery or whether they are valid for the same reasons as in CSL.

Figure 1.8 summarises the relationship between the formalisms discussed in this dissertation. The links from ST to BST and CSL to BCSL are ones of influence and not formal. BST and BCSL are adapted formalisms from ST and CSL respectively, used in the dissertation to aid comparison, while keeping some of the properties of their parents. On the other hand, the translation and models of BCSL are defined in detail. The models of BST is implicitly obtained, by composition with the translation, because the translation is sound.

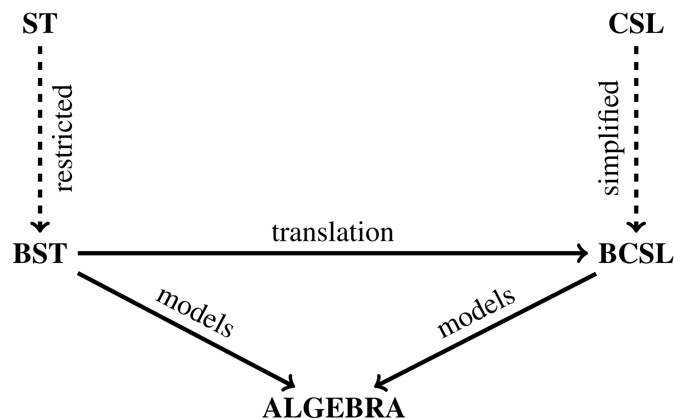


Figure 1.8: Formalisms

As our first step in probing into models and algebra, in Chapter 4 we consider a concrete model, the Resource Model, based on the primitive of resource separation used in CSL. We actually consider two such models, one based on the Local Action model of Separation Logic described in [9] and the other based on predicate transformers [11]. The Local Action model is the classic model of Separation logic. We show how to interpret $*$ of Action in such a way that validates the Exchange Law. This work goes some way to answering our questions about CKA and the proof rules for modularity, but for one flaw: in this model, $*$ is not associative. In fact, associativity of parallel composition is not forced by CSL, and the model does have some interesting structure (and it provides an improved understanding of local actions, as we shall see). The non-associativity is irritating, and we seek to remedy it. This is where the model we call Resource Model based on predicate transformers come in.

Using the Resource Model, which is a model of Concurrent Separation Logic and contains notions of parallel and sequential composition satisfying the Exchange Law, we show that two algebraic interpretations of Hoare triples, namely Hoare and Plotkin triples (introduced in Chapter 2) agree with the usual interpretations of pre-condition and post-condition specification for predicate transformers. This result confirms that there is no cheating in the sense that the various logical laws of Hoare and Concurrent Separation Logic mean the same thing when the algebra is instantiated to this standard and independently existing model of Concurrent Separation Logic.

Through this investigation we find that the models only have some of the structure of a CKA and this leads us to suggest an alternate algebraic structure called Locality Bimonoid, the subject matter of Chapter 6. The structure of Locality Bimonoid is that of two ordered monoids on the same poset linked by the Exchange Law. Furthermore, we discover a pleasant way of expressing the notion of locality using the equation $f = f * \mathsf{skip}$, where skip is the unit of the sequence operator, compared to the characterisation of a semantic definition of locality used in Separation Logic [9]. We end with an algebra that satisfies the Exchange Law but not Locality. The locality equation serves as a healthiness condition (in the sense of Dijkstra and He/Hoare [11, 25]) used to pick out the local elements constituting a sub-algebra. The function $(-) * \mathsf{skip}$ form part of a Galois connection between local and non-local elements and acts as a localiser. In contrast to our expectations we also find that the Exchange Law does not rely on any locality conditions, therefore the law is valid with the non-local algebra. As a result the non-local algebra validates only the *Concurrency* rule from CSL, whereas a CKA validated the *Frame* rule also.

In their work on CKA [22], Hoare and others noted that the rules of Basic CSL can be derived from the structure of a CKA. In Chapter 5 we show here something of a converse: instead of starting from a CKA and deriving the CSL rules, we start from CSL and derive part of the CKA structure (a Locality Bimonoid). The way we obtain our model is again using predicate transformers, but this time of a generalised variety. Usually, the notion of predicate transformer begins with the concept of a set of states, and the transformers work on the powerset. Instead of starting with a set, the generalised model, which we call the Generalised Resource Model, starts with an ordered commutative monoid, and in place of the powerset we consider the set of down-closed subsets. It is like working with a model of intuitionistic rather than classical logic (a Kripke model), enriched as well with a monoid operation. Indeed, taking the monoid operator into account means that the predicates are obtained from the model-theoretic structure of (intuitionistic) Bunched Logic [41, 46].

The generality in this predicate transformer model has two benefits. First, it lets us construct a model from the proof theory of BCSL, by taking the propositions under entailment as the beginning ordered monoid. This lets us prove a completeness theorem, connecting provability with validity in a particular Locality Bimonoid: this gives us the converse of the previous observations on how CSL laws held in an algebraic structure. Second, the generalisation from sets to an ordered monoid enables us to consider an instantiation where the starting point is an ordered monoid of typing contexts taken from (Baby) Session Types. Figure 1.9 highlights the models discussed which are instances of the algebra.

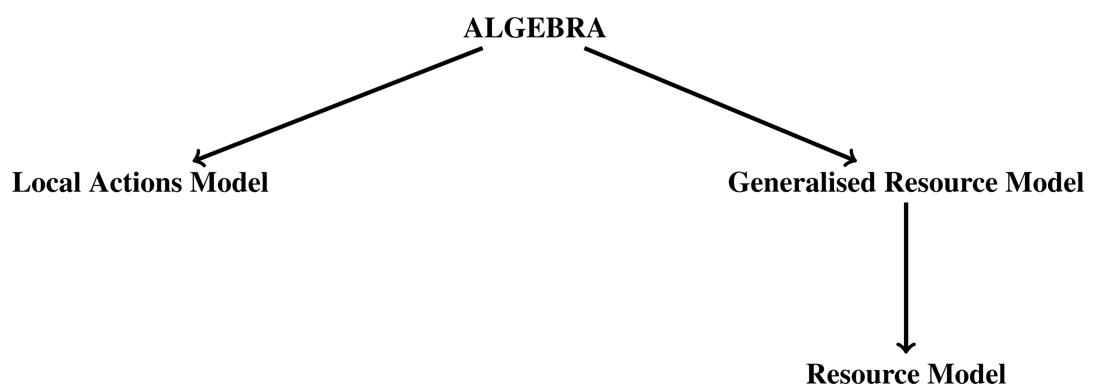


Figure 1.9: Models

This rounds off the dissertation, by giving us our connection between all three formalisms. It shows two conceptually distant instances of the same algebraic structure, based on *shared-memory* and *message-passing*, which have independently been shown through numerous examples and papers to provide modular reasoning about concurrent programs. We might suggest that, perhaps, some degree of unification has been obtained in the theory. We will say more on that in the conclusion (Chapter 7).

1.4 Contribution

The main contributions of the dissertation can be summarised as follows.

1. Sound translation from Baby Session Types to Basic Concurrent Separation Logic.
2. Investigation of the connection between locality in models of Separation Logic and in algebra, based on a state transformer model (Local Action) and a predicate transformer model (Resource Model).
3. Identification of the concept of Locality Bimonoid, as a weakened form of algebraic structure.
4. Study of a Generalised Predicate Transformer Model, which results in a completeness theorem connecting provability in Basic Concurrent Separation Logic and validity in the model, and which allows an instantiation using concepts from Session Types.

Of these contributions, the technical results under 1 and 4 were carried out by myself. 2 and 3 were joint work with R.L. Petersen, P.W. O'Hearn, C.A.R. Hoare and G. Struth where generally we collaborated on finding the algebraic structure. In more detail the technical results under 2 in Chapter 4 was mainly where the collaboration took place. The technical proofs for Lemma 4.3.3, Lemma 4.3.4, Lemma 4.3.5, Proposition 4.3.6, Lemma 4.5.1, Proposition 4.5.2, Lemma 4.5.3 and Lemma 4.5.5 were solved solely by myself and all other technical proofs in Chapter 4 were carried out jointly with R.L. Petersen.

1.5 Outline

Chapter 2: We begin the dissertation by introducing the formalisms that we aim to make a link between; namely, Session Types (ST), Concurrent Separation Logic (CSL), and Concurrent Kleene Algebra (CKA). We also describe how we modify Concurrent Separation Logic and Session Types to aid the process of making the formal link easier.

Chapter 3: In this chapter we provide a session instantiation of a generalised Basic Concurrent Separation Logic. We then make a syntactic translation from Baby Session Types to Basic Concurrent Separation Logic, and prove that the translation is sound.

Chapter 4: In this chapter we study the algebraic structure of the Local Action model of Separation Logic (SL) [9], which is one of the standard models of the logic. We start from the model and derive part of a Concurrent Kleene Algebra. We then link the identified algebraic notions to the semantics of Separation Logic. As a result of this study we get a better understanding of local actions and show that locality can be characterised in a very simple way. We also gain some insight on the algebraic structure of locality, which we put to use later. Finally, we identify a slight defect in our interpretation of parallelism in the Local Action model, which we remedy by defining an alternate model based on predicate transformers.

Chapter 5: This chapter provides a Generalised Predicate Transformer model for Basic Concurrent Separation Logic (BCSL), hence Baby Session Types (BST) as well. The formulation is influenced by our finding in Chapter 4, but it is more general in that it considers predicates as built from a pre-order rather than a set. This allows a BST instantiation in which predicates are built from the proof theory of Session Types formalism. The model then allows us to connect the proof theories of BST and BCSL to the model: We obtain a soundness and completeness result linking the semantics and proof theory. Additionally, the Generalised Predicate Transformer model has a number of the logical laws from Concurrent Kleene Algebra, and we are able to connect the algebraic views of Hoare and Plotkin triples to the more traditional one of predicate transformers (which we call the Dijkstra triple) and also to the proof theory. Thus, in this chapter we connect up all of proof theory, particular models, and algebraic laws.

Chapter 6: In this chapter we summarise the algebraic notions that we have identified and studied. Having given several specific models we describe an algebraic notion that generalises them, which we call a 'Locality Bimonoid'.

Chapter 7: This chapter concludes the dissertation and discusses some possible future work.

1.6 Related Work

This dissertation builds on the work on Concurrent and Abstract versions of Separation Logic [9, 10, 40], Concurrent Kleene Algebra [22, 23, 26], and Session Types [28, 54, 57]. Although this dissertation builds on these works, there is no other closely related work in the sense of trying to relate these formalisms, particularly bringing Session Types into the picture. We see our work as a step forward in understanding the connections between formalisms, and hopefully as informing a developing line of work on algebraic principles of concurrency being pushed by Hoare, Möller and others.

Apart from the works on Concurrent Kleene Algebra and Concurrent Separation Logic, other related work have been on the Exchange Law in concurrency where the structure of a pair of monoids having a shared unit and satisfying the Exchange Law is known as a Concurrent Monoid in [26]. This structure was earlier studied by Gischer in the 1980s [12], who considered a pomset model. Later in the 1990's Bloom and Ésik who considered what we know as the Trace model (see Section 2.5.2). Those papers did not study locality as we do here, but they did show further examples of the Exchange Law in concurrency. The relation between these works and this dissertation is similar to the relation of this dissertation to the work on Concurrent Kleene Algebra. We present a more general structure which we connect to a standard pre-condition and post-condition specification model, that compliments the models in [5, 12, 26] based on traces.

Chapter 2

Formalisms

A concurrent system consists of two or more processes that may or may not interact with each other. The two main forms of communication between processes are through the use of shared memory or channels. In a system where processes interact through *shared-memory*, processes communicate by writing and reading to a shared memory location. In a system using channel-based communication the processes communicate by sending/receiving values via channels. (Note: In Unix terminology it is 'threads' that may share memory, where 'processes' normally do not. We are not following such a terminological distinction in this work, and use 'process' as a generic term to denote entities that may communicate either using shared memory or channels.)

This dissertation is driven mainly by three formalisms; Session Types (ST), Concurrent Separation Logic (CSL) and algebraic laws for concurrency. This chapter aims to familiarise the reader with the theoretical background needed on these subjects for the material in later chapters. ST and CSL originally focused on *message-passing* and *shared-memory*, respectively. However, we wish to compare the two formalisms and in order to make this comparison we take a minimalist approach where we strip as much surrounding detail from ST and generalise CSL as much as possible, leaving simple (even simplistic) basic notions that can be compared technically.

This chapter starts with 2.1 introducing Session Types followed by 2.2 describing a stripped down version of Session Types called Baby Session Types (BST). Next, 2.3 introduces Concurrent Separation Logic and in 2.4 we give a generalised version of CSL, Basic Concurrent Separation Logic (BCSL). Lastly, in 2.5 we identify an algebraic formalism (Concurrent Kleene Algebra) and show some algebraic laws for concurrency.

2.1 Session Types (ST) & Message-Passing

Message passing is one way of achieving interprocess communication. In concurrent systems, processes often interact via exchanging messages as part of a pre-established protocol, which specifies the sequence and form of messages that are communicated between them in a reciprocal manner. For the desired communication to occur it is essential that these protocols are obeyed. Process calculi such as Communicating Sequential Processes (CSP) [19], Calculus of Communicating Systems (CCS) [35] and pi-calculus [37] are based on *message-passing*. We focus a little on pi-calculus since it is where the basic communication ideas for Session Types stem from. Pi-calculus is a calculus where channels are used between processes enabling the processes to interact by sending and receiving messages whether they be values or channels. The process constructs in pi-calculus include the following:

$$P \quad ::= \quad x(y).P \quad | \quad \overline{x}\langle y\rangle.P \quad | \quad (P \mid P) \quad | \quad (\nu x)P \quad | \quad 0 \quad | \quad \ldots$$

Here, $x, y \in X$, where X is a set of entities called names. The input prefixing $x(y).P$ is a process waiting for a message to be received on a communication channel named x before proceeding as P, binding the received message to the name y. The output prefixing $\overline{x}\langle y\rangle.P$ describes a process where the message y is emitted on a channel named x before proceeding as P. The concurrent composition is $P \mid Q$, where P and Q are two processes or threads executed concurrently. $(\nu x)P$ is the creation of a new name, which may be seen as a process allocating a new channel name x within P. The nil process, 0, is a process that has completed.

Below is a small example of a program which consists of 3 parallel processes:

$$(\nu x)\left(\overline{x}\langle 3\rangle.\overline{x}\langle 3\rangle.0 \quad | \quad x(y).\overline{z}\langle x\rangle.0\right) \quad | \quad z(v).v(w).0$$

Channel x is initially only known to the two leftmost processes; it allows them to communicate, where an integer 3 is sent from the left to the middle process, and bound to y. This communication is one step of execution, which results in the process:

$$(\nu x)(\overline{x}\langle 3\rangle.0 \quad | \quad \overline{z}\langle x\rangle.0) \quad | \quad z(v).v(w).0$$

Now the two rightmost processes can communicate on channel z, and the name x becomes bound to v. The next step in the process is now:

$$(\nu x)(\overline{x}\langle 3\rangle.0 \quad | \quad 0 \quad | \quad x(w).0)$$

Note that since the local name x has been output, the scope of x is extended to cover the right process as well. This is the phenomenon called 'scope extension' in pi-calculus. Finally, channel x is now used to send integer 3 to the right process.

From the example above we can think of two processes that use a common channel to be the two ends of the channel. The communication protocol between them consists of sequences of actions in one and reciprocal actions in the other (e.g., send paired with receive). Session Types use this idea and provide a type system whereby the specification of the protocol of each process is expressed as a typing context. A Session Type specifies not only the data types of individual messages, but also the state transition of the protocol and hence describes the allowable sequence of messages. Using the types of processes it becomes possible to verify, at compile-time, whether communication between processes is in accordance with the specified protocol. This ensures that well-typed programs behave in the intended manner without any run-time errors due to the inconsistency of communication patterns between the processes.

We leave the formal treatment of Session Types for the next section where we introduce a restricted version of Session Types. Here we give an informal description and discuss the basic ideas behind Session Types through an example. For the program

$$(\nu x)\left(\overline{x}\langle 3\rangle.\overline{x}\langle 3\rangle.0 \quad | \quad x(y).\overline{z}\langle x\rangle.0\right) \quad | \quad z(v).v(w).0$$

we used before, the following types would be associated with each process:

$$\begin{aligned} \textit{left process}: &\quad \overline{x}: !int.!int.end \\ \textit{middle process}: &\quad x: ?int.?int.end,\ \overline{z}: ![?int.end].end \\ \textit{right process}: &\quad z: ?[?int.end].end \end{aligned}$$

where ? and ! represent input and output respectively and $[-]$ denotes a channel type. Each type describes the sequence of action performed on each channel. The typing context of the left process is trivial where the channel simply outputs an integer in succession. The middle process is slightly complicated, where in the program the middle process only performs one action on x (receives an integer), however in the typing there are two actions on x. The subtlety is that after the left and the middle processes communicate the type of x becomes $?int.end$ which is then sent to the right process on $\overline{z}$ to be then matched with the second output on $\overline{x}$ by the left process.

At first glance the type of the right process may seem odd since in the program the process receives a channel then performs another action of receiving an integer. This is not so obvious from the type of the process where only the receiving of a channel is shown. The next action is actually to receive on z, which matches the fact that in the program x is received by z then x receives an integer. We also note a slight limitation of session type where the types do not capture some channel dependencies. In the program z receives x before performing an action on x, however the typing of the right process does not show this, since there is no mention of x. From the typings we can see that x has a reciprocal type to $\bar{x}$,

$$\begin{aligned} \bar{x} : &\quad !int.!int.end \\ x : &\quad ?int.?int.end \end{aligned}$$

where whenever $\bar{x}$ sends a message of type *int* x receives a message of type *int* and when $\bar{x}$ completes so does x. Furthermore, we can see that to obtain the type of x from the type of $\bar{x}$ we just simply need to replace ! with ? and vice-versa. Similarly, $\bar{z}$ has a reciprocal type to z. We say that that the two types are *dual* or *co-types* of each other, a notion central to session types. The three typing contexts give us three session types, one for each process. However, the type of the middle process is not dual to either the left process nor the right process. Subtyping allows us to specialise the type of the middle process to that of the left process, $x : ?int.?int.end$, or to that of the right process, $\bar{z} : ![?int.end].end$, as required by duality.

Session Types include more features such as *selection*, *branching*, *replication*, *recursion* and other useful types which allow to reason about more realistic programs. To retain the focus on technically comparing the modularity of Session Types and Concurrent Separation Logic we will not include these features.

2.2 Baby Session Types (BST)

Baby Session Types is a simplified version of Session Types [28, 54, 57]. The language consists of a basic grammar for communication such as data passing and receiving. Session Types provide communication patterns for each process and its interaction with other processes. The type system is used to claim that well typed programs are free from run time errors and communication mismatch. As we have mentioned, to ease comparison with Concurrent Separation Logic, focusing on the resource control aspect, we remove some of the constructs of Session Types such as *selection* and *branching*. Below is the basic grammar and typing rules for BST:

Programs. A BST program is a parallel composition of processes that communicate by synchronous message-passing.

$$P \;::=\; k?j.P \;\mid\; k!j.P \;\mid\; P \parallel P \;\mid\; \texttt{inact}$$

As before $k?j.P$ and $k!j.P$ receive and send a channel j on channel k respectively followed by P. $P \parallel P$ is parallel composition and `inact` is the inactive or stopped process. The term $k?j.P$ binds variable j in the continuation P, while the term $k!j.P$ performs no binding. Following standard practice, we will work with terms up to α-conversion of bound variables. Throughout the dissertation we will follow Barendregt's variable convention, which is:

> "Variable Convention: If $M_1,\ldots,M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables." ([3], Page 26)

We refer the reader to [52] for further discussion. As a consequence, we tacitly assume that bound variables are fresh when stating proof rules.

Types. Metavariables α and β range over types, given by the following grammar.

$$\alpha,\beta \;::=\; ![\alpha];\beta \;\mid\; ?[\alpha];\beta \;\mid\; \texttt{end}$$

Here, $![\alpha];\beta$ describes a channel on which you can send an α, after which the channel behaves as described by β. Similarly, $?[\alpha];\beta$ means 'receive α, then continue as β'. Finally `end` indicates that the channel has completed all its actions and has no further actions to perform. The *co-type* $\overline{\alpha}$ is obtained by switching ! and ? at the outermost level (not within $[\alpha]$), and by setting $\overline{\texttt{end}} = \texttt{end}$.

$$\overline{![\alpha];\beta} = ?[\alpha];\overline{\beta} \quad \overline{?[\alpha];\beta} = ![\alpha];\overline{\beta} \quad \overline{\texttt{end}} = \texttt{end}$$

Typing Contexts. The metavariable Δ ranges over finite multi-sets of variable/type pairs. We say that a context Δ is *consistent* if any channel occurs at most twice, and when it occurs twice the occurrences must be *co-types* of one another. The *consistent* contexts are those that require channel-ends to be used in a consistent manner, as two end-points in a point-to-point communication. A typing context is called *completed* if `end` is the only type that appears within it. The metavariable Φ is often used to range over *completed* contexts. $\Delta_1 \circ \Delta_2$ is multi-set union. We write $\Delta_1 \asymp \Delta_2$ to mean that $\Delta_1 \circ \Delta_2$ is *consistent*. We will use the following notion of entailment:

$$\Delta_1 \vdash \Delta_2 \quad\Longleftrightarrow\quad \Delta_1 \text{ is } \textit{inconsistent} \text{ or } \exists\Phi.\Delta_1 = \Delta_2 \circ \Phi.$$

This entailment encapsulates the idea that an inconsistency implies anything, and that we can forget about *completed* contexts (or we can add them, as pre-conditions). It is used in the *Consequence* rule of BST.

Typing. The main sequent of the typing system is $P \triangleright \Delta$ which says that process P has typing Δ. The typing rules can be found in Figure 2.1.

$$[\text{Consequence}] \; \frac{\Delta_1 \vdash \Delta_2 \quad P \triangleright \Delta_2}{P \triangleright \Delta_1}$$

$$[\text{Inact}] \; \frac{}{\texttt{inact} \triangleright \emptyset} \qquad [\text{Par}] \; \frac{P_1 \triangleright \Delta_1 \quad P_2 \triangleright \Delta_2}{P_1 \parallel P_2 \triangleright \Delta_1 \circ \Delta_2}$$

$$[\text{Receive}] \; \frac{P \triangleright \Delta \circ k\colon \beta \circ j\colon \alpha}{k?j.P \triangleright \Delta \circ k\colon ?[\alpha];\beta} \qquad [\text{Send}] \; \frac{P \triangleright \Delta \circ k\colon \beta}{k!j.P \triangleright \Delta \circ k\colon ![\alpha];\beta \circ j\colon \alpha}$$

Figure 2.1: Proof Rules for Baby Session Types

The formulation of BST above takes its lead from the recent [54], but what is left out of the language is almost as notable as what is included. We have avoided recursion, thread and session creation, communication of non-channel data values (e.g., integers), and the variants of record matching and selection often used in session type systems. Also, because there are no 'non-linear' channels, we cannot type-check a program in which two or more processes compete for a service (unless the typing context is *inconsistent*). In short, we have left out most of the things that might be useful in a useful language. What is left over includes mobility of channel ends, *ownership* transfer, *race* avoidance, and independent reasoning about processes, some of the more original (and challenging to model) aspects of Session Types, and what we want to concentrate on.

A few further remarks on different formulations of Session Types are in order. In some works, the composition $(k : \alpha) \circ (k : \overline{\alpha})$ is defined to be $k : \bot$, where $\bot$ is a special additional type that represents hidden communication. The treatment in [54] instead unions the typings together, choosing to do hiding in the types in a way that matches binding-time in the programs rather than matching parallel composition. We follow [54] in using unioning and avoiding the $\bot$ type, but we do not have an outermost binding construct (written $\nu xy.P$ in [54]) which establishes two ends of a channel, so we simply allow a channel variable to appear twice. We make these remarks to be clear in that BST is not intended to be in any way original, and is just following ideas in some of the formulations of Session Types.

Basic Examples. As a very simple example of *race* avoidance, notice that it is not possible to find any *consistent* typing *raceType* such that

$$(k!x.\mathtt{inact}) \parallel (k!y.\mathtt{inact}) \triangleright \mathit{raceType}$$

is a derivable typing. The reason is that, to type this, in the *Par* rule we would need to assign the ability to send on k to both processes, and this would make $\Delta_1 \circ \Delta_2$ *inconsistent* in the rule.

Here, we are using a 'race' by analogy with *shared-memory*. There, two processes *race* if one tries to write to a location while the other is trying to read from or write to it, with no intervening synchronisation. Race-avoidence is a concern in programming [1]. In the context of *message-passing*, we can say by analogy that the two processes attempting to concurrently send on the same channel are racing for this ability to send. One of the remarkable features of Concurrent Separation Logic is that it rules out races in the traditional *shared-memory* sense [6, 40]. The way that Session Types rule out such channel races is an indication of some similarity, which partly spurred us to look further.

We also find that the ability to send or receive on a channel is not stationary, once and for all lying within a particular process. Here is a simple example where processes exchange these abilities.

$$\begin{array}{c} k!z.\,k?y.\,\mathtt{inact} \quad \parallel \quad k?x.\,k!x.\,\mathtt{inact} \\ \triangleright \\ z:\mathtt{end} \circ k:![\mathtt{end}];?[\mathtt{end}];\mathtt{end} \;\circ\; k:?[\mathtt{end}];![\mathtt{end}];\mathtt{end} \end{array}$$

Here, the exchange of abilities takes place due the two occurrences of k in the typing context having types dual to each other. The left process starts off by sending z on k which is received by the right process bound to x then the right process sends x on k which is received by the left process bound to y.

A slightly more intricate example in Figure 2.2 shows channel permissions flowing amongst processes. We depict the information used in typing the processes as intermediate assertions in the code. This is intended to be suggestive of how Hoare logic proof outlines are often given, and of the way that proofs of concurrent processes have been depicted in CSL where sequential proofs are done independently for the processes.

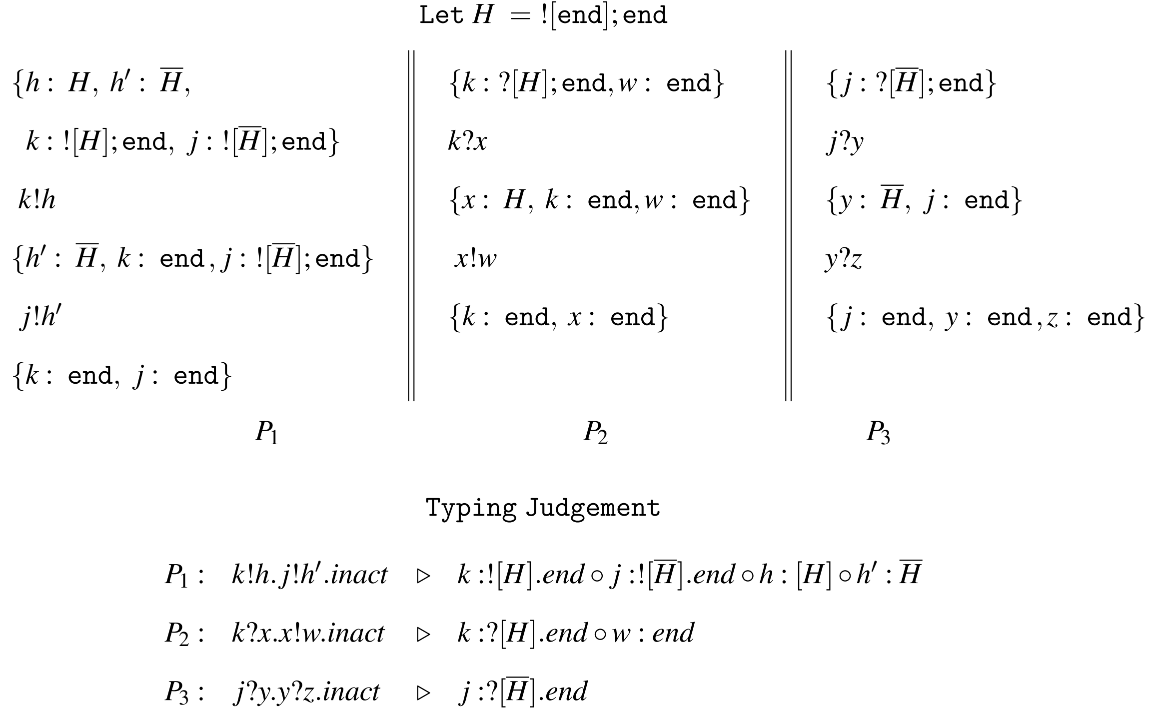


Figure 2.2: Ownership Transfer Example in BST

You can think of the intermediate typing contexts in Figure 2.2 as describing the permissions (i.e. the channels) that each process holds or owns at any point of execution. The k's in P_1 and P_2 have *co-type* of each other, indicating that the two ends of channels can communicate privately without external interference. The same goes for the js in P_1 and P_3. P_1 completes before P_2 and P_3 because the later two are waiting to acquire the relevant permissions. The ownership of permissions is transferred using the sending and receiving commands, where each time a send is executed, the channel that is passed is removed from the post-condition. The opposite is true when receive is executed, where the received channel is then added to the post-condition. Once P_1 completes, both P_2 and P_3 have completed their first step of receiving a channel from P_1 and are then allowed to transfer channels between each other.

Aside: On Inconsistent Contexts. The inclusion of *inconsistent* as well as *consistent* typing contexts might seem strange at first. In an early version of this work, we attempted a formulation of BST using *consistent* contexts only, but a technical difficulty (pointed out by Vasco Vasconcelos) arose. Consider the judgement (Figure 2.3):

$$k!j.\mathtt{inact} \parallel k?x.x!z.j?m.\mathtt{inact} \ \triangleright\ k\colon ![\alpha];\mathtt{end} \circ k\colon ?[\alpha];\mathtt{end} \circ j\colon \alpha \circ j\colon \overline{\alpha} \circ z\colon \mathtt{end}$$

where $\alpha = ![end]; end$.

This judgement is typable in BST using a derivation that uses *consistent* contexts at every step. However, if we take one step of execution, we obtain the contractum $j!z.j?m.\mathtt{inact}$, and subject reduction (that types are preserved when terms reduce, defined formally in section 2.2.1) would dictate that it should be typable as

$$j!z.j?m.\mathtt{inact} \;\triangleright\; j:\alpha \circ j:\overline{\alpha} \circ z:\mathtt{end}$$

which it is, and this context is *consistent*, but the derivation uses an *inconsistent* context at the step $j?m.\mathtt{inact} \;\triangleright\; j:\mathtt{end} \circ j:\overline{\alpha}$, below is part of the derivation from Figure 2.3 with x instantiated with j.

$$\cfrac{\cfrac{\cfrac{j:\mathtt{end}\circ j:\mathtt{end}\circ m:\mathtt{end}\vdash\emptyset \qquad \cfrac{}{\mathtt{inact}\triangleright\emptyset}\;[\text{Inact}]}{\mathtt{inact}\triangleright j:\mathtt{end}\circ j:\mathtt{end}\circ m:\mathtt{end}}\;[\text{Consequence}]}{j?m.\mathtt{inact}\triangleright j:\mathtt{end}\circ j:\overline{\alpha}}\;[\text{Receive}]}{j!z.j?m.\mathtt{inact}\triangleright j:\alpha\circ j:\overline{\alpha}\circ z:\mathtt{end}}\;[\text{Send}]$$

We will study subject reduction formally in the next section but this illustrates that if we leave the *inconsistent* contexts out, we will obtain undesirable properties. It is because we have included *inconsistent* contexts that we included the *Consequence* rule. Often in session type formalisms, one uses a rule

$$[\text{Inact}]\;\frac{\Phi \text{ is } \textit{completed}}{\mathtt{inact}\triangleright\Phi}$$

and weakening by *completed* contexts Φ

$$\frac{P\triangleright\Delta}{P\triangleright\Delta\circ\Phi}$$

is an admissible rule. We want weakening to take into account inconsistency as well as completedness, and it was simpler to just include the *Consequence* rule.

We are not claiming any originality or significance in the definition of Baby Session Types. It was formed with Petersen and O'Hearn to give a simple basis for further work, not as something significant in and of itself. However, the remarks on inconsistency are just one instance of the many subtleties, particularly surrounding subject reduction (preserving of typing by reduction) in formalisms for session types, as discussed in [57]), and it became important at some point for us to do some sanity checking regarding this formalism. In the next subsection I include a proof, due to Rasmus Petersen, about subject reduction for this formalism; it can be skipped without loss of continuity.

$$
\dfrac{
\dfrac{
\dfrac{k : end \vdash \emptyset \quad \dfrac{}{\texttt{inact} \triangleright \emptyset}\,[\text{Inact}]}{\texttt{inact} \triangleright k : end}\,[\text{Consequence}]
}{k!j.\texttt{inact} \triangleright k : ![\alpha];\texttt{end} \circ j : \alpha}\,[\text{Send}]
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{k : \texttt{end} \circ x : \texttt{end} \circ j : \texttt{end} \circ m : \texttt{end} \vdash \emptyset \quad \dfrac{}{\texttt{inact} \triangleright \emptyset}\,[\text{Inact}]}{\texttt{inact} \triangleright k : \texttt{end} \circ x : \texttt{end} \circ j : \texttt{end} \circ m : \texttt{end}}\,[\text{Consequence}]
}{j?m.\texttt{inact} \triangleright k : \texttt{end} \circ x : \texttt{end} \circ j : \overline{\alpha}}\,[\text{Receive}]
}{x!z.j?m.\texttt{inact} \triangleright k : \texttt{end} \circ x : \alpha \circ j : \overline{\alpha} \circ z : \texttt{end}}\,[\text{Send}]
}{k?x.x!z.j?m.\texttt{inact} \triangleright k : ?[\alpha];\texttt{end} \circ j : \overline{\alpha} \circ z : \texttt{end}}\,[\text{Receive}]
}{k!j.\texttt{inact} \parallel k?x.x!z.j?m.\texttt{inact} \triangleright k : ![\alpha];\texttt{end} \circ k : ?[\alpha];\texttt{end} \circ j : \alpha \circ j : \overline{\alpha} \circ z : \texttt{end}}\,[\text{Par}]
$$

where $\alpha = ![\texttt{end}];\texttt{end}$

Figure 2.3: Inconsistent Context Derivation

2.2.1 Operational Semantics and Subject Reduction

In light of the subtleties noted in the last section we include here an account of subject reduction.

Structural congruence on programs is the transitive, symmetric relation generated by the following three rules:

$$P \parallel \texttt{inact} \cong P \qquad P \parallel Q \cong Q \parallel P \qquad (P \parallel Q) \parallel R \cong P \parallel (Q \parallel R)$$

and reduction is the relation generated by the following three rules:

$$\begin{array}{rcll} k!j.P \parallel k?j'.Q & \xrightarrow{k} & P \parallel Q[j/j'] & [Com\ (k)] \\ P \xrightarrow{k} P' & \Longrightarrow & P \parallel Q \xrightarrow{k} P' \parallel Q & [Par] \\ Q \cong Q' \wedge Q' \xrightarrow{k} P' \wedge P' \cong P & \Longrightarrow & Q \xrightarrow{k} P & [Str] \end{array}$$

Here, $Q[j/j']$ denotes the result of substituting j for j' in Q. In the substitution lemma below we use a similar notation $\Delta[j/j']$ for substitution into a typing context. Our treatment of subject reduction will involve changing the types as we change the program.

Definition 2.2.1. *If Δ is consistent and $\Delta = \Delta' \circ k :![\alpha];\beta \circ k :?[\alpha];\overline{\beta}$ then we define $\Delta/k = \Delta' \circ k : \beta \circ k : \overline{\beta}$ otherwise the symbol is undefined.*

We use lemmas which account for the structure of *consistent* typings for sending and receiving.

Lemma 2.2.2. *If $k!j.P \triangleright \Delta$, with Δ consistent, then there exists α, β and Δ' such that $\Delta = \Delta' \circ k : ![\alpha];\beta \circ j : \alpha$ and $P \triangleright \Delta' \circ k : \beta$.*

Proof: Obviously, the equality is modulo reordering of Δ. The proof is by induction of the typing derivation, only two cases are relevant:

Case *Consequence*: As Δ is *consistent* then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also *consistent*. Then, by induction hypothesis, there is α, β and Δ'' such that $\Delta_2 = \Delta'' \circ k :![\alpha];\beta \circ j : \alpha$ and $P \triangleright \Delta'' \circ k : \beta$. Now, by the rule of consequence, α, β and $\Delta' = \Delta'' \circ \Phi$ have the required properties.

Case *Send*: Follows directly from the rule.

∎

Corollary 2.2.3. *If $k!j.P \triangleright \Delta$, with Δ consistent, then there exists α, β and Δ' such that $\Delta = \Delta' \circ k : ![\alpha];\beta$.*

Lemma 2.2.4. *If $k?j.P \triangleright \Delta$, with Δ consistent, then there exists α, β and Δ' with j not free in Δ', such that $\Delta = \Delta' \circ k :?[\alpha];\beta$ and $P \triangleright \Delta' \circ k : \beta \circ j : \alpha$.*

Proof: Obviously, the equality is modulo reordering of Δ. The proof is by induction of the typing derivation, only two cases are relevant:

Case *Consequence***:** As Δ is *consistent* then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also *consistent*. Then, by induction hypothesis, there is α, β and Δ'' with j not free in Δ'', such that $\Delta_2 = \Delta'' \circ k : ![\alpha];\beta$ and $P \triangleright \Delta'' \circ k : \beta \circ j : \alpha$. Now, by the rule of consequence, α, β and $\Delta' = \Delta'' \circ \Phi$ have the required properties. (If j is free in Φ, then we α-rename $k?j.P$ to $k?l.P$ for some fresh l which is then free neither in Δ'' nor in Φ[1].)

Case *Receive***:** Follows directly from the rule.

■

Lemma 2.2.5. *If $k!j.P \parallel k?j.Q \triangleright \Delta$, with Δ consistent, then there exists α, β and Δ' such that $\Delta = \Delta' \circ k :![\alpha];\beta \circ k :?[\alpha];\overline{\beta}$.*

Proof: Obviously, the equality is modulo reordering of Δ. The proof is by induction of the typing derivation, only two cases are relevant:

Case *Consequence***:** As Δ is *consistent* then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also *consistent*. Then, by induction hypothesis, there is α, β and Δ'' such that $\Delta_2 = \Delta'' \circ k :![\alpha];\beta \circ k :?[\alpha;\overline{\beta}]$. Now α, β and with $\Delta' = \Delta'' \circ \Phi$ have the required properties.

Case *Par***:** So $\Delta = \Delta_1 \circ \Delta_2$ and if Δ is *consistent* then so are both Δ_1 and Δ_2. Thus

- By Corollary 2.2.3 there is α_1, β_1 and Δ'_1 such that $\Delta_1 = \Delta'_1 \circ k :![\alpha_1];\beta_1$.
- By Lemma 2.2.4 there is α_2, β_2 and Δ'_2 such that $\Delta_2 = \Delta'_2 \circ k :?[\alpha_2];\beta_2$.

Since $\Delta_1 \asymp \Delta_2$ we conclude that $\alpha_2 = \alpha_1$ and that $\beta_2 = \overline{\beta_1}$. Then $\alpha = \alpha_1$, $\beta = \beta_1$ and $\Delta' = \Delta'_1 \circ \Delta'_2$ have the desired properties.

■

[1] A formulation of the lemma that takes alpha renaming into account would be: If $k?j'.P \triangleright \Delta$, with Δ consistent, then there exists α, β, Δ' and j with j not free in Δ', such that $\Delta = \Delta' \circ k :?[\alpha];\beta$ and $P[j/j'] \triangleright \Delta' \circ k : \beta \circ j : \alpha$.

Lemma 2.2.6 (Substitution). *If $P \triangleright \Delta$ then $P[j/j'] \triangleright \Delta[j/j']$.*

Proof: Proof is by induction on the derivation of $P \triangleright \Delta \circ j' : \alpha$.

For the rule of consequence we have to note that if Δ is *inconsistent* then so is $\Delta[j/j']$.

For the receive rule we note that if j is the channel received then it is bound and the substitution has no effect (? is a binder and the substitution is capture avoiding).

■

Lemma 2.2.7 (Subject Reduction). *If $P \triangleright \Delta$ with Δ consistent, and $P \xrightarrow{k} P'$ then Δ/k is defined and consistent and $P' \triangleright \Delta/k$.*

Proof: Proof is by case on the rule used to prove $P \xrightarrow{k} P'$, then, if necessary, by induction on the typing derivation for $P \triangleright \Delta$.

Case $Com(k)$**:** By Lemma 2.2.5, Δ/k is defined and since Δ is *consistent* then so is Δ/k. That $P' \triangleright \Delta/k$ is shown by induction on the derivation for $P \triangleright \Delta$; since $P = k!j.P_1 \parallel k?j'.P_2$, only two cases apply:

Case *Consequence***:** If Δ is *consistent* then $\Delta = \Delta_2 \circ \Phi$ and so Δ_2 is also *consistent*. Then, by induction hypothesis, $P' \triangleright \Delta_2/k$. Now, since Δ_2/k is defined and $\Delta_2 \asymp \Phi$, Φ cannot mention k. Thus $\Delta/k = (\Delta_2 \circ \Phi)/k$ is defined and equal to $(\Delta_2/k) \circ \Phi$. This means that $\Delta/k \vdash \Delta_2/k$ and so, by the rule of consequence, $P' \triangleright \Delta/k$.

Case *Par***:** So $\Delta = \Delta_1 \circ \Delta_2$ and if Δ is *consistent* then so are both Δ_1 and Δ_2. Thus

- by Lemma 2.2.2 there is α_1, β_1 and Δ_1' such that $\Delta_1 = \Delta_1' \circ k :![\alpha_1];\beta_1 \circ j : \alpha$ and $P_1 \triangleright \Delta_1' \circ k : \beta_1$.
- by Lemma 2.2.4 there is α_2, β_2 and Δ_2' with j' not free in Δ_2', such that $\Delta_2 = \Delta_2' \circ k :?[\alpha_2];\beta_2$ and $P_2 \triangleright \Delta_2' \circ k : \beta_2 \circ j' : \alpha$.

By Lemma 2.2.4, $(\Delta_2' \circ k : \beta_2 \circ j' : \alpha)[j/j'] = \Delta_2' \circ k : \beta_2 \circ j : \alpha$ and by Lemma 2.2.6, $P_2[j/j'] \triangleright \Delta_2' \circ k : \beta_2 \circ j : \alpha$ and so the parallel rule gives $P_1 \parallel P_2[j/j'] \triangleright (\Delta_1 \circ \Delta_2)/k$.

Case *Par***:** We know that $P \parallel Q \triangleright \Delta$ and $\Delta = \Delta_1 \circ \Delta_2$. Since Δ is *consistent* then so is Δ_1 thus by the induction hypothesis we know that Δ_1/k is defined and *consistent* and $P' \triangleright \Delta_1/k$. Then we know that $P' \parallel Q \triangleright \Delta/k$ since $(\Delta_1/k) \circ \Delta_2 = (\Delta_1 \circ \Delta_2)/k$ when $\Delta_1 \asymp \Delta_2$.

Case Str**:** Structural congruence preserves typing.

■

2.3 Concurrent Separation Logic & Shared-Memory

In *shared-memory* concurrency, the executions of concurrent processes are intended to interact via a shared resource. It is very difficult to ascertain absence of runtime errors such as a *race*, and *dangling pointers*, which may occur through the accessing or deallocation of the same storage by two processes simultaneously. The difficulty lies in the fact that these phenomena may cause unpredictable or irreproducible behaviour.

Separation Logic [42, 47] is an extension of Hoare Logic [17] which permits a local way of reasoning about programs that alter memory. The basic idea was to extend the assertions by introducing the separating conjunction operator $*$, which permit a state to be split into disjoint parts, allowing reasoning of a part of a state in a way that automatically guarantees that the other part of the state remains unaffected [9]. Part of the extension is the inclusion of the *Frame* rule:

$$\frac{\{P\}\,C\,\{Q\}}{\{P*R\}\,C\,\{Q*R\}}$$

which codifies the intuition behind this local reasoning. The idea is that the pre-condition P of a given Hoare triple contains all the resources that command C will access. The consequence of this is that anything outside of this state (e.g R) remains unchanged; so R can be added on to the pre-condition and the post-condition.

As the aim of this section is to provide the context for work to follow we will discuss the main ideas in an informal way. To give some examples we refer to the heap model where $Heaps$ is the set $\mathbb{N} \rightharpoonup_f \mathbb{N}$ of finite partial functions from natural numbers to natural numbers, and $X * Y$ is defined as follows. First, let $h \bullet h'$ denote the union of heaps with disjoint domain, which is undefined when the domains overlap. Then, if X and Y are sets of heaps (elements of $P(Heaps)$), we define their separating conjunction by:

$$X * Y = \{h_X \bullet h_Y \mid h_X \in X \wedge h_Y \in Y \wedge h_X \bullet h_Y \downarrow\}$$

where $h_X \bullet h_Y \downarrow$ means that $h_X \bullet h_Y$ is defined.

A typical predicate is the points-to fact $n \mapsto m$, where n and m are natural numbers. It is the singleton set $\{h\}$ where h is the heap that maps n to m and which is undefined on all numbers other than n. Another predicate is $n \mapsto -$, which is $\bigvee_m n \mapsto m$. A typical command is the mutation statement $[n] := m$ where n and m are natural numbers. Associated with it is the following Hoare triple axiom.

$$\frac{}{\{n \mapsto -\}[n] := m\{n \mapsto m\}}$$

Consider the program $[10] := 23; [11] := 44$ with the initial state $\{10 \mapsto - \ \wedge 11 \mapsto -\}$. Figure 2.4 gives an example proof (or proof outline) for this program in SL.

$$
\begin{array}{c}
\{10 \mapsto - * 11 \mapsto -\} \\
\{10 \mapsto -\} \\
[10] := 23 \\
\{10 \mapsto 23\} \\
\{10 \mapsto 23 * 11 \mapsto -\} \\
\{11 \mapsto -\} \\
[11] := 44 \\
\{11 \mapsto 44\} \\
\{10 \mapsto 23 * 11 \mapsto 44\}
\end{array}
$$

Figure 2.4: Sequential Composition Example in SL

The proof outline shows the use of the $*$ operator, which replaces the $\wedge$ connective, splitting the state into two disjoint portions $\{10 \mapsto -\}$ and $\{11 \mapsto -\}$. Since the first command only accesses the portion containing $\{10 \mapsto -\}$, we only need to focus on that portion without worrying about the other portion, which is added on via the *Frame* rule. A similar description can be given for the second command, which focuses on the portion of state containing $\{11 \mapsto -\}$. This annotated program is a shorthand for a proof which uses the *Frame* rule twice and the sequencing rule once, the corresponding proof tree is given in Figure 2.5.

Separation Logic was first applied to sequential code. However, because of the way the $*$ operator breaks the state into portions, it naturally paved a way to reason about *shared-memory* concurrency, where different portions of state now could be assigned to different processes. O'Hearn [40] introduces concurrency to Separation Logic using the concept of ownership and spatial separation using the $*$ operator. The basic idea is that programs are only allowed to manipulate resources that they own at a given time, this also allows reasoning about processes independently. Concurrent Separation Logic (CSL) extends Separation Logic by the addition of the *Concurrency* rule:

$$
\frac{\{P_1\}\, C_1\, \{Q_1\} \quad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 * P_2\}\, C_1 \parallel C_2\, \{Q_1 * Q_2\}}
$$

which states that processes that operate on separate portions can be reasoned about independently. CSL allows us to prove concurrent programs compositionally by separating the two disjoint part of the heap and proving each separate program with respect to what each program uses in the heap, this concept is known as *ownership*. Figure 2.6 gives example proof (or proof outlines) in CSL for the concurrent program $[10] := 23 \parallel [11] := 44$ based on the heap model.

$$\dfrac{\dfrac{\{10 \mapsto -\}\,[10] := 23\,\{10 \mapsto 23\}}{\{10 \mapsto - * 11 \mapsto -\}\,[10] := 23\,\{10 \mapsto 23 * 11 \mapsto -\}}\ [\text{Frame}] \qquad \dfrac{\{11 \mapsto -\}\,[11] := 44\,\{11 \mapsto 44\}}{\{10 \mapsto 23 * 11 \mapsto -\}\,[11] := 44\,\{10 \mapsto 23 * 11 \mapsto 44\}}\ [\text{Frame}]}{\{10 \mapsto - * 11 \mapsto -\}\,[10] := 23; [11] := 44\,\{10 \mapsto 23 * 11 \mapsto 44\}}\ [\text{Sequence}]$$

Figure 2.5: Proof Tree for example 2.4

$$\{10 \mapsto - \ * \ 12 \mapsto -\}$$
$$\{10 \mapsto -\} \qquad \{12 \mapsto -\}$$
$$[10] := 23 \quad \| \quad [12] := 44$$
$$\{10 \mapsto 23\} \qquad \{12 \mapsto 44\}$$
$$\{10 \mapsto 23 \ * \ 12 \mapsto 44\}$$

Figure 2.6: Concurrent Composition Example in CSL

Brookes has proven a fundamental result that CSL for heaps cannot prove any racy programs [6]; or more precisely, there can never be a *race* in a proven program, starting from a state satisfying the pre-condition. As an example, for the program:

$$[10] := 23 \quad \| \quad [10] := 44$$

we cannot find any consistent pre-condition, the reason being is that each process needs a pre-condition $10 \mapsto -$ according to the axiom for $[n] := m$ given above, and $10 \mapsto - \ * \ 10 \mapsto -$ is *false* since $*$ operator requires that its conjuncts be assigned disjoint subheaps, and there is no way to split up a given heap so that one cell (10, in this case) is sent to both disjoint subheaps.

The heap model of CSL also allows for the transfer of ownership of heap buffers between processes. The first example used to show this was a ‘pointer transferring buffer’, where a cell is allocated in one process, placed in a buffer and read out by a second process, and then safely disposed in that second process. This is achieved by the concept of ownership transfer whereby shared resources are owned by a particular process at a given time. Consider the following buffer sharing program [40] in Figure 2.7.

The $put()$ and $get()$ procedures place or get the content that is in the buffer, they use the semaphore $full$ to know if the buffer is full or empty. The semaphore also acts as the resource owner by which ownership of resources are transferred. Hence, at any one point only on program accesses the buffer or owns the buffer.

Figure 2.8 gives the proof in CSL [40] where we would have to separately verify the pre-condition and post-condition specifications for the $put()$ and $get()$ operations. Here, the knowledge that x is allocated disappears from the left process in the $put()$ operation, to be materialised after the $get()$ operation in the second process. This is, evidently, somewhat analogous to the example of ownership transfer using Session Types in Figure 2.2 of Section 2.2.

$$
\begin{array}{lcl}
\mathit{full} := \texttt{false}; & & \\
\texttt{resource}\ \mathit{buf}(\texttt{c}, \mathit{full}); & & \\
\texttt{get}(\texttt{y}) & = & \texttt{with}\ \mathit{buf}\ \texttt{when}\ \mathit{full}\ \texttt{do} \\
 & & y\ :=\ c;\ \mathit{full} := \texttt{false} \\
 & & \texttt{endwith}; \\
\texttt{put}(\texttt{x}) & = & \texttt{with}\ \mathit{buf}\ \texttt{when}\ \neg \mathit{full}\ \texttt{do} \\
 & & c\ :=\ x;\ \mathit{full} := \texttt{true} \\
 & & \texttt{endwith}; \\
x := \texttt{cons}(\texttt{a},\texttt{b}); & \| & \mathit{get}(y); \\
\mathit{put}(x); & & \mathit{use}(y); \\
 & & \mathit{dispose}(y);
\end{array}
$$

Figure 2.7: Buffer Sharing Program

$$
\begin{array}{ccc}
 & \{\mathit{emp}\} & \\
 & \{\mathit{emp} * \mathit{emp}\} & \\
\{\mathit{emp}\} & & \{\mathit{emp}\} \\
x := \texttt{cons}(\texttt{a},\texttt{b}) & \| & \mathit{get}(y) \\
\{x \mapsto -,-\} & & \{y \mapsto -,-\} \\
\mathit{put}(x); & & \mathit{use}(y); \\
\{\mathit{emp}\} & & \{y \mapsto -,-\} \\
 & & \mathit{dispose}(y); \\
 & & \{\mathit{emp}\} \\
 & \{\mathit{emp} * \mathit{emp}\} & \\
 & \{\mathit{emp}\} &
\end{array}
$$

Figure 2.8: Proof of Buffer Sharing Program in CSL

We have treated this last example rather informally, but we hope the reader can see from the discussion in this section and the previous one, some of the apparent similarities between Session Types and Separation Logic, which led us to be curious as to the actual relationship between them.

To prove programs with critical regions like for the $put()$ and $get()$ operations above, we would need to include rules for critical regions. The same would go for other synchronisation constructs such as semaphores or locks. Although important, in the context of *shared-memory* concurrency, we do not need to include these constructs to approach Session Types. So, in the technical part of the thesis we include the concurrency proof rule, but not other rules of CSL for its specialised synchronisation constructs.

2.4 Basic Concurrent Separation Logic (BCSL)

We will take a general view point of CSL which will be referred to as Basic CSL (BCSL). Where BST is defined by reference to particular language constructs, in formulating BCSL we follow the lead of Abstract Separation Logic [9] and define a system which abstracts away from the exact details of primitive commands or pre-condition and post-condition specifications. In fact, we will be considerably more general than Abstract Separation Logic, and this will enable us later to instantiate it to be more like BST.

Structure. We presume we are given:

- A pre-ordered commutative monoid of propositions $(Props, \vdash, *, emp)$;
- A set Com equipped with total binary operations $c_1 \parallel c_2$ and $c_1; c_2$, and with a distinguished element $\mathsf{skip} \in Com$.

We recall that a pre-order is a binary relation on a set that is reflexive and transitive. The order here provides an abstract model of logical entailment, and we include it in order to state the *Consequence* rule in Hoare Logic (in Figure 2.9 below). This is a standard ingredient of program logics.

A commutative monoid is a binary operator that is commutative and associative, together with a unit for that binary operator. In a pre-ordered commutative monoid we only require the associativity, unity and commutativity laws to hold 'up to' the equivalence induced by the order (where p and q are 'equivalent' if and only-if $p \vdash q$ and $q \vdash p$). So, for example, the commutativity law requires $p * q \vdash q * p$ and $p * q \vdash q * p$, but not literally $p * q = q * p$. It is additionally assumed that $*$ is monotone with respect to the order, so that if $p \vdash p'$ and $q \vdash q'$ then $p * q \vdash p' * q'$. Being a pre-ordered commutative monoid is what one expects of the propositions in a logic like Separation Logic: monotonicity corresponds to the typical proof rule for introducing $*$. As

the reader might be aware, in presentations of Separation Logic, propositions are actually taken to have stronger structure. For example, propositions are usually assumed to carry a Boolean algebra structure, and that is not required of BCSL. This extra generality will ease the comparison to Session Types. The proof rules for Basic Concurrent Separation Logic can be found in Figure 2.9.

$$[\text{Skip}]\ \frac{}{\{X\}\,\mathsf{skip}\,\{X\}} \qquad [\text{Frame}]\ \frac{\{X\}\,c\,\{Y\}}{\{X*F\}\,c\,\{Y*F\}}$$

$$[\text{Seq}]\ \frac{\{X\}\,c_1\,\{Y\} \quad \{Y\}\,c_2\,\{Z\}}{\{X\}\,c_1;c_2\,\{Z\}} \qquad [\text{Par}]\ \frac{\{X_1\}\,c_1\,\{Y_1\} \quad \{X_2\}\,c_2\,\{Y_2\}}{\{X_1*X_2\}\,c_1 \parallel c_2\,\{Y_1*Y_2\}}$$

$$[\text{Consequence}]\ \frac{X' \vdash X \quad \{X\}\,c\,\{Y\} \quad Y \vdash Y'}{\{X'\}\,c\,\{Y'\}}$$

Figure 2.9: Proof Rules for Basic Concurrent Separation Logic

We intend that in a specialisation of BCSL there may be additional rules and axioms: Those just given form the core, that may be added to.

2.4.1 Heap Model Instantiation

We briefly recap how CSL works with a heap model. The structure of propositions is obtained by setting

$$(Props, \vdash, *, emp) \;=\; (P(Heaps), \subseteq, *, \{u\})$$

where the material to the right is defined as follows. $Heaps$ is the set $\mathbb{N} \rightharpoonup_f \mathbb{N}$ of finite partial functions from natural numbers to natural numbers, and $P(Heaps)$ is the powerset. u is the empty partial function, and $X * Y$ is defined as before. The specialisation of BCSL to the heap model will include the mutation axiom:

$$\frac{}{\{n \mapsto -\}[n] := m\{n \mapsto m\}}$$

We have already defined the assertion $\{n \mapsto m\}$ and $\{n \mapsto -\}$ earlier in this section, where we said that $\{n \mapsto m\}$ is the singleton set $\{h\}$ where h is the singleton heap that maps n to m, and where $\{n \mapsto -\}$ is the set of those singleton heaps whose domain of definition is precisely n. The example given in Figure 2.6 would also be a proof in BCSL.

2.5 Algebra

Algebra is a great way of looking beyond the syntax of systems and it is a way of abstracting away from the details of a particular language or model, resulting in a more general formalism. In this section we discuss some work on an algebra that has been developed in concurrency theory called Concurrent Kleene Algebra (CKA). Although we do not use the full structure of this algebra, we take some of the important ideas from it.

2.5.1 Concurrent Kleene Algebra

A Concurrent Kleene Algebra [22] is a system that offers, next to choice and iteration, two composition operators, one that stands for sequential execution and the other for concurrent execution. They are related by an in-equational form of the Exchange Law. In order to define Concurrent Kleene Algebra we first provide definitions for two terms which are used in the definition of a CKA; *Semiring* and *Quantale*, taken from [22]:

Definition 2.5.1 (Semiring)**.** *A semiring is a structure* $(S,+,0,\cdot,1)$ *such that* $(S,+,0)$ *is a commutative monoid,* $(S,\cdot,1)$ *is a monoid, multiplication distributes over addition in both arguments and 0 is annihilator for multiplication. A semiring is idempotent if its addition is.*

In the application to concurrency the $+$ operator plays the role of non-deterministic choice in a programming language, and the $\cdot$ operator can play the role of parallel or sequential composition, as will be the case in the definition of CKA. Element 0 is the unit for the $+$ operator and element 1 is the unit for the $\cdot$ operator.

Definition 2.5.2 (Quantale)**.** *A quantale is an idempotent semiring that is a complete lattice under the natural order and in which composition distributes over arbitrary lub. The glb and the lub of a set* T *are denoted by* $\bigsqcap T$ *and* $\bigsqcup T$*, respectively. Their binary variants are* $x \sqcap y$ *and* $x \sqcup y$*.*

A *quantale* ensures preservation of *lubs* which implies Scott-continuity (preservation of ω-chains) and hence the existence of fixed-points. In this work we show that a quantale is not strictly necessary. In Chapters 4 and 5 we show that we can get models with weaker structure and in section 5.3.4 we show that the parallel operator does not preserve *lubs*, hence the structure is not a *quantale*. We now give the definition of a CKA:

Definition 2.5.3 (Concurrent Kleene Algebra [22])**.** *By a Concurrent Kleene Algebra we mean a structure* $(S,+,0,*,;,1)$ *such that* $(S,+,0,*,1)$ *and* $(S,+,0,;,1)$ *are quantales linked by the exchange axiom:*

$$(a*b);(c*d) \sqsubseteq (b;c)*(a;d)$$

In a CKA the following laws hold:

1. $a*b = b*a$
2. $(a*b);(c*d) \sqsubseteq (a;c)*(b;d)$
3. $a;b \sqsubseteq a*b$
4. $(a*b);c \sqsubseteq a*(b;c)$
5. $a;(b*c) \sqsubseteq (a;b)*c$

In this structure we have two quantales one for sequential composition (;) and the other for parallel composition ($*$) with both sharing the same unit (1).

2.5.2 Trace Model [21]

To give an example of a CKA we look at a trace model. Let A be a set. Then $Traces$ is the set of finite sequences over A. The powerset $P(Traces)$ will be the carrier set of this algebra. We have the following two binary operations on $P(Traces)$:

1. $T_1 * T_2$ is the set of those traces obtained by interleaving a trace in T_1 with a trace in T_2;

2. $T_1 ; T_2$ is the set of those traces obtained by concatenating a trace in T_1 with a trace in T_2.

The set $\{\varepsilon\}$ functions as a unit for both ; and $*$, where ε is the empty sequence. The Exchange Law in this model was already remarked in [5]. $P(Traces)$ is a prototypical CKA [26], this means, among other things, that it is a complete lattice in which $*$ and ; distribute through the *glb* $\bigsqcup$, denoting union, and that both $*$ and ; have a left and right zero (the empty set, $\emptyset$).

2.5.3 Interpretation of Triples

The notion of Concurrent Kleene Algebra might seem rather spare, and the trace model perhaps does not suggest a connection to logic. But, remarkably, by regarding assertions (pre-conditions and post-conditions) as the same kinds of things as programs, we can obtain a very economical kind of interpretation of standard Floyd-Hoare logic, and beyond.

Hoare Triple. In [22] an interpretation of Hoare triples is shown in a CKA with the structure $(S, +, 0, *, ;, 1)$. The interpretation of Hoare triples given is:

$$\{p\}\, c\, \{q\} \Leftrightarrow p;c \sqsubseteq q$$

where $p,\ c,\ q \in S$.

A striking aspect of this interpretation is that versions of the *Frame* and *Concurrency* rules from CSL can be proven immediately.

Frame Rule. $\forall p, c, q, r \in S\ \{p\}\, c\, \{q\} \Rightarrow \{p * r\}\, c\, \{q * r\}$

Proof: [22]

$$\begin{array}{rcll} \{p\}\, c\, \{q\} & \Leftrightarrow & p;c \sqsubseteq q & \texttt{definition} \\ & \Rightarrow & r * (p;c) \sqsubseteq r * q & \texttt{monotonicity of } * \\ & \Rightarrow & (r * p);c \sqsubseteq r * q & \texttt{definition 2.4.3 - Law4} \\ & \Leftrightarrow & \{r * p\}\, c\, \{r * q\} & \texttt{definition} \end{array}$$

■

Concurrency Rule. $\forall p, p', c, c', q, q' \in S\ \{p\}\, c\, \{q\} \wedge \{p'\}\, c'\, \{q'\} \Rightarrow \{p * p'\}\, c * c'\, \{q * q'\}$

Proof: [22]

$$\begin{array}{rcll} \{p\}\, c\, \{q\} \wedge \{p'\}\, c'\, \{q'\} & \Leftrightarrow & p;c \sqsubseteq q\ \wedge\ p';c' \sqsubseteq q' & \texttt{definition} \\ & \Rightarrow & (p;c) * (p';c') \sqsubseteq q * q' & \texttt{monotonicity of } * \\ & \Rightarrow & (p * p');(c * c') \sqsubseteq q * q' & \texttt{exchange axiom} \\ & \Leftrightarrow & \{p * p'\}\, c * c'\, \{q * q'\} & \texttt{definition} \end{array}$$

■

Other rules such as *Sequence* rule, *Consequence* rule can also be proved, the proof of which can be found in [22].

Plotkin Triple. An alternate triple has been considered as well in the algebra models

$$P \rightarrow_C Q \Leftrightarrow P \sqsupseteq C;Q$$

This has been dubbed the 'Plotkin triple' by Hoare, as he has shown (in as yet unpublished notes) that it can be used to derive some rules in structural operational semantics. We consider the Plotkin triple as a variant Hoare triple, which is useful for thinking about the future: the intuition is that P describes the future, and over-approximates what C followed by Q describe. With Plotkin triples we are also able to recover the rules of BCSL.

Although we get the BCSL rules formally, at this point we have not shown that they have the same import as in Separation Logic. It is not obvious that the interpretation of these rules in a CKA means anything about resource separation, absence of races, etc. The Traces model of CKA is so far removed from the resource-oriented models of CSL that one might wonder if the validity of these laws in CKA has any importance whatsoever, or whether it might just be a trick. Of course, part of the aim in this dissertation is to show that there is no trickery at all, that the laws mean the same thing as usual, when we look at a model built from the resource primitives of Separation Logic.

Chapter 3

Translation from BST to BCSL

In this chapter we present a translation which takes typing judgements in (our version of) Session Types to a version of Concurrent Separation Logic. In more detail, we begin by presenting a particular instantiation, based on concepts from Session Types, of our BCSL system. This instantiation makes use of the minimal assumptions we made about propositions in BCSL, requiring very little structure of them (an ordered commutative monoid). Then we translate our BST system of baby session types into this instantiation of BCSL. We prove a theorem on the soundness of the translation.

Due to the simplistic nature of both formalisms the translation itself is trivial, but the preservation properties of the translation are not necessarily so (after all, it is possible, e.g., to have non-equivalent type systems on the same syntax). In fact, we find that completeness property of the translation (provability in the target of the translation implies provability in the source) does not hold. The reasons revolve around some of the most essential differences between BST and BCSL. Soundness of the translation, however, goes through straightforwardly.

3.1 Session Instantiation of BCSL: BCSL/ST

To instantiate BCSL, what we need to do is set down the structure of propositions and the structure of commands.

Propositions. We define a pre-ordered commutative monoid $(Props, \vdash, *, emp)$. We simply take *Prop* to be the set of session typing contexts Δ, and $\Delta * \Delta'$ to be $\Delta \circ \Delta'$. *emp* is the empty context $\emptyset$. $X \vdash Y$ is the entailment notion between typing contexts already defined in Section 2.2. Note that propositions here have much less structure than one finds usually in logic, including Separation Logic. There is no boolean algebra, or even conjunction structure. We are using the minimality of the BCSL assumptions here: typing contexts in the BST session typing system do indeed form a pre-ordered commutative monoid, but not (say) a boolean algebra.

Commands. We extend the commands of BCSL with primitives for receiving and sending channels. The former uses a continuation, to account for the binding aspect, but the latter does not use a continuation as we can use sequencing instead.

$$C \quad ::= \quad k?j.C \mid k!j \mid C \parallel C \mid C;C \mid \mathsf{skip}$$

There is an asymmetry here, in that the receiving form $k?j.C$ uses an explicit continuation where the sending form $k!j$ does not. There is no deep reason. We would prefer to do away with all explicit continuations, letting ; do the job. But, if we were not to use a binding form, then to treat receiving we should either include imperative references to store the result or a functional language notation that includes return values. Either of these would be a fine choice in a language, but to keep our formalism as small as possible we made the compromise to include this binding form (so as to avoid return values or references).

As an example you can take nearly any program that receives a value. If you send a value, then that does not change what you can do after, so the remaining commands in the continuation can just forget that you just sent a value. But if you receive a value, then the commands in the continuation may depend on what that value is. It might for instance pass it on or branch on it. So there has to be a way for the continuation to refer to that received value. Either by having a pointer to it or have a bound variable containing the value or a similar mechanism. We chose one that seemed to add the least to our language.

Proof Rules. The proof rules are those of BCSL together with specialised rules in Figure 3.1.

$$\text{[Send]}\ \frac{}{\{k\colon\ ![\alpha];\beta * j\colon\ \alpha\}\, k!j\, \{k\colon\ \beta\}}$$

$$\text{[Receive]}\ \frac{\{A * k\colon\ \beta * j\colon\ \alpha\}\, P\, \{B\}}{\{A * k\colon\ ?[\alpha];\beta\}\, k?j.P\, \{B\}}$$

Figure 3.1: Specialised Rules of Session Instantiation

Note that our tacit assumption that the bound variable j is fresh in the *Receive* rule means that it is not free in A or B and not equal to k.

The following lemma serves as a sanity check of the calculus related to *inconsistent* types. It can be thought of as a relative of the fact that in Hoare logic $\{\mathit{false}\}C\{Q\}$ always holds: *false* as a pre-condition validates any conclusion.

Lemma 3.1.1. *If Δ is inconsistent then $P \triangleright \Delta$ holds for all programs P.*

Proof: Assume Δ is *inconsistent*. Suppose $P \triangleright \Delta'$ and since Δ is *inconsistent* we know that $\Delta \vdash \Delta'$ then we can apply the *Consequence* rule to get $P \triangleright \Delta$. Thus all we need to prove is that

$$\forall P.\ \exists \Delta'.\ P \triangleright \Delta'$$

We show this by induction on the structure of P:

Case $k?j.P$**:** By induction hypothesis, there is Δ'' such that $P \triangleright \Delta''$. By the *Consequence* rule $P \triangleright \Delta'' \circ k : \mathtt{end} \circ j : \mathtt{end}$ (even though this context might be *inconsistent*). By the *Receive* rule we then have $k?j.P \triangleright \Delta'' \circ k :?[\mathtt{end}];\mathtt{end}$.

Case $k!j.P$**:** By induction hypothesis, there is Δ'' such that $P \triangleright \Delta''$. By *Consequence* rule $P \triangleright \Delta'' \circ k : \mathtt{end}$ (even though this context might be *inconsistent*). By the *Send* rule we then have $k!j.P \triangleright \Delta'' \circ k :![\mathtt{end}];\mathtt{end} \circ j : \mathtt{end}$.

Case $P_1 \parallel P_2$**:** Follows from the rule and induction hypothesis, with $\Delta' = \Delta'_1 \circ \Delta'_2$.

Case `inact`**:** Follows from the rule with $\Delta' = \emptyset$.

■

3.2 Translation.

The translation $\langle\langle - \rangle\rangle$ is a straightforward mapping from untyped BST to BCSL/ST programs and is given in Figure 3.2.

$$\begin{aligned}
\langle\langle \texttt{inact} \rangle\rangle &= \texttt{skip} \\
\langle\langle P \parallel Q \rangle\rangle &= \langle\langle P \rangle\rangle \parallel \langle\langle Q \rangle\rangle \\
\langle\langle k?j.P \rangle\rangle &= k?j.\langle\langle P \rangle\rangle \\
\langle\langle k!j.P \rangle\rangle &= (k!j)\ ;\ \langle\langle P \rangle\rangle
\end{aligned}$$

Figure 3.2: Translation from BST to BCSL/ST

We have chosen to translate the send command differently to the receive command. This is because of the aforementioned binding that happens in the receive command. We let the sequential operator of BCSL/ST handle the continuation which gives us a simpler rule for send in BCSL/ST.

Theorem 3.2.1 (Soundness of Translation)**.**

$$P \triangleright \Delta \textit{ in BST} \implies \{\Delta\}\,\langle\langle P \rangle\rangle\,\{emp\} \textit{ in BCSL/ST}$$

Proof:

Soundness follows by showing that all four session type rules are admissible in the translation: the proof is by induction on proofs with a case analysis on the last rule. All cases are straightforward.

Case `Inact`**:**

$$\frac{}{\texttt{inact} \triangleright \emptyset}$$

We are required to show $\{\emptyset\}\,\mathsf{skip}\,\{emp\}$. Since $\emptyset = emp$ we can use the axiom for skip to get the desired result.

Case `Par`**:**

$$\frac{P_1 \triangleright \Delta_1 \quad P_2 \triangleright \Delta_2}{P_1 \parallel P_2 \triangleright \Delta_1 \circ \Delta_2}$$

We are required to show $\{\Delta_1 * \Delta_2\}\,P_1 \parallel P_2\,\{emp\}$ This follows from the induction hypothesis and the *Concurrency* rule

$$\frac{\{\Delta_1\}\,P_1\,\{emp\} \quad \{\Delta_2\}\,P_2\,\{emp\}}{\{\Delta_1 * \Delta_2\}\,P_1 \parallel P_2\,\{emp * emp\}}$$

Case `Receive`**:**

$$\frac{P \triangleright \Delta \circ k\colon \beta \circ j\colon \alpha}{k?j.P \triangleright \Delta \circ k\colon ?[\alpha];\beta}$$

We are required to show

$$\{\Delta * k\colon ?[\alpha];\beta\}\, k?j.\langle\langle P\rangle\rangle\, \{emp\}$$

By induction hypothesis we get $\{\Delta * k\colon \beta * j\colon \alpha\}\, \langle\langle P\rangle\rangle\, \{emp\}$ which together with the rule for *Receive* gives the desired result.

Case `Send`**:**

$$\frac{P \triangleright \Delta \circ k\colon \beta}{k!j.P \triangleright \Delta \circ k\colon ![\alpha];\beta \circ j\colon \alpha}$$

We are required to show

$$\{\Delta * k\colon ![\alpha];\beta * j : \alpha\}\, (k!j)\, ;\, \langle\langle P\rangle\rangle\, \{emp\}$$

By induction hypothesis we get $\{\Delta * k\colon \beta\}\, \langle\langle P\rangle\rangle\, \{emp\}$ and the following instance of the rule for sequencing and the *Send* rule gives us the desired result.

$$\frac{\dfrac{\{k\colon ![\alpha];\beta * j : \alpha\}\, k!j\, \{k\colon \beta\}}{\{\Delta * k\colon ![\alpha];\beta * j : \alpha\}\, k!j\, \{\Delta * k\colon \beta\}} \quad \{\Delta * k\colon \beta\}\, \langle\langle P\rangle\rangle\, \{emp\}}{\{\Delta * k\colon ![\alpha];\beta * j : \alpha\}\, (k!j)\, ;\, \langle\langle P\rangle\rangle\, \{emp\}}$$

Case `Consequence`**:**

$$\frac{\Delta_1 \vdash \Delta_2 \quad P \triangleright \Delta_2}{P \triangleright \Delta_1}$$

We are required to show

$$\{\Delta_1\}\, \langle\langle P\rangle\rangle\, \{emp\}$$

By induction hypothesis we get $\{\Delta_2\}\, \langle\langle P\rangle\rangle\, \{emp\}$. Since $\Delta_1 \vdash \Delta_2$ we can apply the *Consequence* rule to get $\{\Delta_1\}\, \langle\langle P\rangle\rangle\, \{emp\}$.

■

However, completeness, the converse of Theorem 3.2.1 does not hold. To see why, consider that the following triple is derivable in BCSL/ST (see Figure 3.3 for the derivation)

$$\{k : ![\mathtt{end}];\mathtt{end} * k : ?[\mathtt{end}];\mathtt{end} * j : \mathtt{end}\}\, k!j;k?j.\mathsf{skip}\, \{emp\}$$

On the other hand, we cannot derive

$$k!j;k?j.\mathtt{inact} \triangleright k : ![\mathtt{end}];\mathtt{end} \circ k : ?[\mathtt{end}];\mathtt{end} \circ j : \mathtt{end}$$

From the example we see that in BCSL/ST a process can send a channel and receive it, whereas BST does not allow this. So we conclude that completeness does not hold in general.

$$
\dfrac{
\dfrac{
\dfrac{[A]}{\{k :![\mathtt{end}];\mathtt{end} * k :?[\mathtt{end}];\mathtt{end} * j : \mathtt{end}\}\, k!j\, \{k :?[\mathtt{end}];\mathtt{end}\}}\ [\text{Frame}]
\qquad
\dfrac{[B]}{\{k :?[\mathtt{end}];\mathtt{end}\}\, k?j.\mathsf{skip}\, \{k : \mathtt{end}\}}\ [\text{Rec}]
}{
\{k :![\mathtt{end}];\mathtt{end} * k :?[\mathtt{end}];\mathtt{end} * j : \mathtt{end}\}\, k!j;k?j.\mathsf{skip}\, \{k : \mathtt{end}\}
}\ [\text{Seq}]
\qquad
k : \mathtt{end} \vdash \mathit{emp}
}{
\{k :![\mathtt{end}];\mathtt{end} * k :?[\mathtt{end}];\mathtt{end} * j : \mathtt{end}\}\, k!j;k?j.\mathsf{skip}\, \{\mathit{emp}\}
}\ [\text{Cons}]
$$

Figure 3.3: Derivation of a program in BCSL/ST not derivable in BST.

$$\dfrac{\dfrac{\dfrac{}{\{k :![\mathtt{end}];\mathtt{end} * j : \mathtt{end}\}\, k!j\, \{k : \mathtt{end}\}}\,[\text{Send}] \qquad k : \mathtt{end} \vdash emp}{\{k :![\mathtt{end}];\mathtt{end} * j : \mathtt{end}\}\, k!j\, \{emp\}}\,[\text{Cons}]}{\{k :![\mathtt{end}];\mathtt{end} * k :?[\mathtt{end}];\mathtt{end} * j : \mathtt{end}\}\, k!j\, \{k :?[\mathtt{end}];\mathtt{end}\}}\,[\text{Frame}]$$

Figure 3.4: Part A of the derivation in Figure 3.3

$$\dfrac{\dfrac{\dfrac{}{\{k : \mathtt{end} * j : \mathtt{end}\}\, \mathsf{skip}\, \{k : \mathtt{end} * j : \mathtt{end}\}}\,[\text{Skip}] \qquad k : \mathtt{end} \vdash emp}{\{k : \mathtt{end} * j : \mathtt{end}\}\, \mathsf{skip}\, \{k : \mathtt{end}\}}\,[\text{Cons}]}{\{k :?[\mathtt{end}];\mathtt{end}\}\, k?j.\mathsf{skip}\, \{k : \mathtt{end}\}}\,[\text{Rec}]$$

Figure 3.5: Part B of the derivation in Figure 3.3

The question naturally arises of whether we can more precisely understand the source of *incompleteness*, either by adding something to make more BST programs derivable, or considering a restriction on BCSL proofs.

At first glance, one might think to enlarge BST with a weakening rule.

$$\frac{P \triangleright \Delta}{P \triangleright \Delta \circ \Delta'}$$

If we were to do that, then

$$x!y.\mathtt{inact} \triangleright x :![\mathtt{end}];\mathtt{end} * x :?[\mathtt{end}];\mathtt{end} * y : \mathtt{end}$$

would be derivable:

$$\dfrac{\dfrac{\dfrac{x : \mathtt{end} \vdash \emptyset \qquad \dfrac{}{\mathtt{inact} \triangleright \emptyset}\,[\text{Inact}]}{\mathtt{inact} \triangleright x : end}\,[\text{Consequence}]}{x!y.\mathtt{inact} \triangleright x :![\mathtt{end}];\mathtt{end} \circ y : \mathtt{end}}\,[\text{Send}]}{x!y.\mathtt{inact} \triangleright x :![\mathtt{end}];\mathtt{end} \circ x :?[\mathtt{end}];\mathtt{end} \circ y : \mathtt{end}}\,[\text{Weakening}]$$

However, this is not a general solution, as there are other examples not amenable to simple weakening. One is the Hoare triple

$$\{x :![\mathtt{end}]; ![\mathtt{end}]; \mathtt{end} * y : \mathtt{end} * z : \mathtt{end} * x :?[\mathtt{end}]; ?[\mathtt{end}]; \mathtt{end}\}\ (x!y; x!z; \mathsf{skip}) \parallel x?y.\mathsf{skip}\ \{emp\}$$

the derivation of which is given in Figure 3.6. The program in this triple is the translation of a BST term $x!y.x!z.\mathtt{inact} \parallel x?y.\mathtt{inact}$ but the judgement

$$x!y.x!z.\mathtt{inact} \parallel x?y.\mathtt{inact}$$
$$\triangleright$$
$$x :![\mathtt{end}]; ![\mathtt{end}]; \mathtt{end} \circ y : \mathtt{end} \circ z : \mathtt{end} \circ x :?[\mathtt{end}]; ?[\mathtt{end}]; \mathtt{end}$$

is not typable using weakening.

From these examples, it appears that the essence of the difference between BCSL/ST and BST concerns the way that BCSL/ST can reason using entailments involving post-conditions. This cannot be done in BST simply because there are no post-conditions.

Furthermore, in hindsight, we find that the failure of completeness to some extent is natural. Examples we have seen so far intuitively involve deadlock. For example, in

$$\{k :![\mathtt{end}]; \mathtt{end} * k :?[\mathtt{end}]; \mathtt{end} * j : \mathtt{end}\}\ k!j; k?j.\mathsf{skip}\ \{emp\}$$

we have a program that owns both ends of channel k (as expressed by the pre-condition) performing a send on it, so there is no way for any other process to synchronise with $k!j$. From the point of view of partial correctness Hoare logic, where failure to terminate validates a triple, we would expect the triple to hold (and it is provable). The corresponding Session Types judgement is not typable; as is well known, Session Types rule out some deadlocks. Take for example the following program that is derivable in BST (see Figure 3.7 for the derivation).

$$k!a; j?b.\mathtt{inact} \parallel j!c.k?d.\mathtt{inact}$$
$$\triangleright$$
$$k :![\mathtt{end}]; \mathtt{end} \circ j :?[\mathtt{end}]; \mathtt{end} \circ a : \mathtt{end} \circ j :![\mathtt{end}]; \mathtt{end} \circ k :?[\mathtt{end}]; \mathtt{end} \circ c : \mathtt{end}$$

This is a program that deadlocks and yet is typable; so Session Types is not a total correctness (ensuring termination) formalism. Here we find that Session Types is somewhat unusual from the point of view of Hoare logic. Sometimes it rules out deadlock (like the programe $k!j; k?j.\mathsf{skip}$), so divergence does not falsify a triple. On the other hand, in other cases, like this one, it is happy with deadlock. In retrospect one expects a Hoare logic to take a uniform position: deadlock always falsifies a triple (total correctness), or deadlock always satisfies a triple (partial correctness). That is why, perhaps, it would be difficult to find an exact fit with a natural Hoare logic.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{A}\,[\text{Send}] \quad \dfrac{}{B}\,[\text{Send}]
      }{
        \{\Delta_1\}\, x!y; x!z\, \{x : \mathsf{end}\}
      }\,[\text{Seq}]
      \quad
      \dfrac{}{C}\,[\text{Skip}]
    }{
      \{\Delta_1\}\, x!y; x!z; \mathsf{skip}\, \{x : \mathsf{end}\}
    }\,[\text{Seq}]
    \quad
    \dfrac{
      \dfrac{
        \dfrac{}{D}\,[\text{Skip}] \quad x :?[\mathsf{end}]; \mathsf{end} * y : \mathsf{end} \vdash x :?[\mathsf{end}]; \mathsf{end}
      }{
        \{x :?[\mathsf{end}]; \mathsf{end} * y : \mathsf{end}\}\, \mathsf{skip}\, \{x :?[\mathsf{end}]; \mathsf{end}\}
      }\,[\text{Cons}]
    }{
      \{\Delta_2\}\, x?y.\mathsf{skip}\, \{x :?[\mathsf{end}]; \mathsf{end}\}
    }\,[\text{Rec}]
  }{
    \{\Delta_1 * \Delta_2\}\, (x!y; x!z; \mathsf{skip}) \parallel x?y.\mathsf{skip}\, \{\Delta_3\}
  }\,[\text{Par}]
  \quad \Delta_3 \vdash emp
}{
  \{\Delta_1 * \Delta_2\}\, (x!y; x!z; \mathsf{skip}) \parallel x?y.\mathsf{skip}\, \{emp\}
}\,[\text{Cons}]
$$

$A = \{\Delta_1\}\, x!y\, \{x :![\mathsf{end}]; \mathsf{end} * z : \mathsf{end}\}$

$B = \{x :![\mathsf{end}]; \mathsf{end} * z : \mathsf{end}\}\, x!z\, \{x : end\}$

$C = \{x : end\}\, \mathsf{skip}\, \{x : end\}$

$D = \{x :?[\mathsf{end}]; \mathsf{end} * y : \mathsf{end}\}\, \mathsf{skip}\, \{x :?[\mathsf{end}]; \mathsf{end} * y : \mathsf{end}\}$

$\Delta_1 = x :![\mathsf{end}]; ![\mathsf{end}]; \mathsf{end} * y : \mathsf{end} * z : \mathsf{end}$

$\Delta_2 = x :?[\mathsf{end}]; ?[\mathsf{end}]; \mathsf{end}$

$\Delta_3 = x : \mathsf{end} * x :?[\mathsf{end}]; \mathsf{end}$

Figure 3.6: Derivation of a program in BCSL/ST not derivable in BST with weakening

$$\dfrac{k : \mathtt{end} \circ j : \mathtt{end} \vdash \emptyset \qquad \dfrac{}{\mathtt{inact} \triangleright \emptyset}\,[\text{Inact}]}{\mathtt{inact} \triangleright k : \mathtt{end} \circ j : \mathtt{end}}\,[\text{Cons}]$$

$$\dfrac{j : \mathtt{end} \circ k : \mathtt{end} \vdash \emptyset \qquad \dfrac{}{\mathtt{inact} \triangleright \emptyset}\,[\text{Inact}]}{\mathtt{inact} \triangleright j : \mathtt{end} \circ k : \mathtt{end}}\,[\text{Cons}]$$

$$\dfrac{\mathtt{inact} \triangleright k : \mathtt{end} \circ j : \mathtt{end}}{j?b.\mathtt{inact} \triangleright k : \mathtt{end} \circ j :?[\mathtt{end}];\mathtt{end}}\,[\text{Receive}] \qquad \dfrac{\mathtt{inact} \triangleright j : \mathtt{end} \circ k : \mathtt{end}}{k?d.\mathtt{inact} \triangleright j : \mathtt{end} \circ k :?[\mathtt{end}];\mathtt{end}}\,[\text{Receive}]$$

$$\dfrac{j?b.\mathtt{inact} \triangleright k : \mathtt{end} \circ j :?[\mathtt{end}];\mathtt{end}}{k!a.j?b.\mathtt{inact} \triangleright k :![\mathtt{end}];\mathtt{end} \circ j :?[\mathtt{end}];\mathtt{end} \circ a : \mathtt{end}}\,[\text{Send}] \qquad \dfrac{k?d.\mathtt{inact} \triangleright j : \mathtt{end} \circ k :?[\mathtt{end}];\mathtt{end}}{j!c.k?d.\mathtt{inact} \triangleright j : \mathtt{end} \circ k :?[\mathtt{end}];\mathtt{end} \circ c : \mathtt{end}}\,[\text{Send}]$$

$$\dfrac{k!a.j?b.\mathtt{inact} \triangleright k :![\mathtt{end}];\mathtt{end} \circ j :?[\mathtt{end}];\mathtt{end} \circ a : \mathtt{end} \qquad j!c.k?d.\mathtt{inact} \triangleright j : \mathtt{end} \circ k :?[\mathtt{end}];\mathtt{end} \circ c : \mathtt{end}}{k!a.j?b.\mathtt{inact} \parallel j!c.k?d.\mathtt{inact} \triangleright k :![\mathtt{end}];\mathtt{end} \circ j :?[\mathtt{end}];\mathtt{end} \circ a : \mathtt{end} \circ j :![\mathtt{end}];\mathtt{end} \circ k :?[\mathtt{end}];\mathtt{end} \circ c : \mathtt{end}}\,[\text{Par}]$$

Figure 3.7: Deadlock derivable example in BST

3.2.1 Recursion

Earlier we discussed how many things had been left out of Baby Session Types. While we will not consider an entire language as found in typical Session Types papers, one omission is glaring: recursion. If we were not able to treat recursion, then our analysis would not apply to Turing-complete programming languages. Therefore, in this section, we briefly indicate how recursion fits into the above.

To treat recursion in Session Types, we must first do three things:

1. Extend the syntax of terms in BST with extra terms X and $\texttt{Let}\ X = P\ \texttt{in}\ Q$, denoting a process variable and a term within which there is a recursively-defined process.

2. Extend the types to include type variables t and recursive types $\mu t.\alpha$.

3. Extend the typing rules to account for an environment component Θ assigning contexts to process variables. Typing judgement then become of the form $\Theta \vdash P \triangleright \Delta$ instead of only $P \triangleright \Delta$. The metavariable Θ ranges over assignments of typing contexts to process variables. Formally, such an assignment can be represented as a finite partial function from process variables X to session typing contexts Δ. We write $\Theta, X : \Delta$ to represent the partial function like Θ except that X maps to Δ (this assumes that X is not in the domain of Θ). We also write $dom(\Theta)$ to denote the set of process variables on which Θ is defined.

The following is an example of a recursive program

$$\texttt{Let}\ X = in?v.out!v.X\ \texttt{in}\ X$$

which can be thought of as a buffer interspersed between producer and consumer process, where the buffer receives a channel v via channel in and passes the received channel along another channel out, and then recurses. In our formalism, we will obtain a typing judgement

$$\Theta \vdash \texttt{Let}\ X = in?v.out!v.X\ \texttt{in}\ X \triangleright in : \mu t.?[\alpha].t \circ out : \mu t'.![\alpha].t'$$

here Θ is the empty environment and we give an arbitrary channel type α to v. This judgement is obtained by using an environment which assigns a typing context to the process variable X, during the derivation of the body of the recursive procedure. That is

$$X : in : \mu t.?[\alpha].t,\ out : \mu t'.![\alpha].t' \vdash in?v.out!v.X \triangleright in : \mu t.?[\alpha].t \circ out : \mu t'.![\alpha].t'$$

Our type system will be set up to use assignments to process variables in this way in order to type recursive processes.

It is worth remarking what the typing judgement for this program does not say. The context

$$in : \mu t.?[\alpha].t \circ out : \mu t'.![\alpha].t'$$

does not specify that the input on channel *in* happens before the output on *out*. This inability to specify sequentialisation between channels is typical of basic session typing systems.

On the other hand, if we were to input and output on the same channel, as in

$$\texttt{Let}\, X = ch?v.ch!v.X \;\texttt{in}\; X$$

then we could specify the sequentialisation. A typing for this process is

$$\Theta \vdash \texttt{Let}\, X = ch?v.ch!v.X \;\texttt{in}\; X \triangleright ch : \mu t.?[\alpha].![\alpha].t$$

which shows that first there is an input on *ch*, and then an output on *ch*.

Now, we formally make the following definitions of the terms and types. The grammar of terms is now the following

$$P \;::=\; k?j.P \;\mid\; k!j.P \;\mid\; P \parallel P \;\mid\; \texttt{inact} \;\mid\; X \;\mid\; \texttt{Let}\, X = P \;\texttt{in}\; Q$$

where X is a process variable and the term $\texttt{Let}\ X = P\ \texttt{in}\ Q$ denotes a recursive process as mentioned before. The corresponding types are now extended as follows:

$$\alpha, \beta \;::=\; ![\alpha];\beta \;\mid\; ?[\alpha];\beta \;\mid\; \texttt{end} \;\mid\; t \;\mid\; \mu t.\alpha$$

The standard recursive type in Session Types is denoted by $\mu t.\alpha$ which represents the behaviour of starting by performing α and, when t is encountered, recurring. t denotes a type variable. The *co-type* $\overline{\mu t.\alpha}$ is $\mu t.\overline{\alpha}$. Note that we do not consider rules for unfolding of the recursive type. Rather, as in [57], we can take an *equi-recursive* view of types, where we do not distinguish between a type $\mu t.\alpha$ and its unfolding $\alpha[\mu t.\alpha/t]$, considering them to be the same for the purpose of the typing rules.

The typing rules for the process variable and recursive term are given in Figure 3.8.

$$[\text{Var}]\ \frac{}{\Theta, X : \Delta \vdash X \triangleright \Delta} \qquad [\text{Def}]\ \frac{\Theta, X : \Delta' \vdash P \triangleright \Delta' \quad \Theta, X : \Delta' \vdash Q \triangleright \Delta}{\Theta \vdash \texttt{Let}\ X = P\ \texttt{in}\ Q \triangleright \Delta}$$

Figure 3.8: BST Recursion Rules

Besides these rules, the Θ component is treated ‘additively’ (i.e., pointwise, without change) in extending the previous BST rules. We can now give a type derivation proof for the recursive program above, which can be found in Figure 3.9. We note here that since we do not have *selection* and *branching* in our language, we cannot write programs that recurse depending on the kind of messages they receive, examples of such programs can be found in [28].

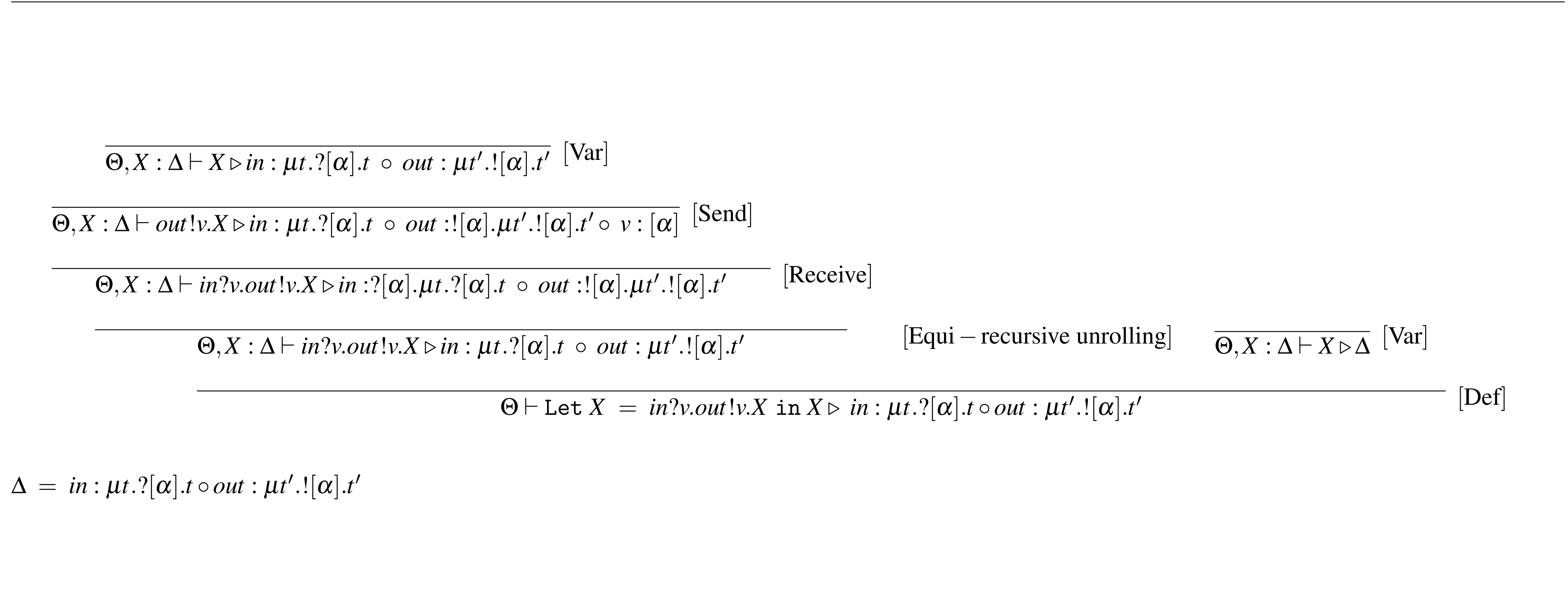


Figure 3.9: Type derivation of a recursive program in BST

Now that we have formally added recursion to BST, we must also do the same for BCSL/ST. For BCSL/ST, we can simply use the standard Hoare Logic proof rules for recursive parameterless procedures [18]. Like in BST, we need to do three things to extend BCSL/ST to handle recursion.

1. Extend the syntax of terms in BCSL/ST with extra terms X and $\texttt{Let}\ X\ =\ P\ \texttt{in}\ Q$, denoting a command variable and a command within which there is a recursively-defined process.

2. Extend the types to include type variables t and recursive types $\mu t.\alpha$. (In fact, we have already done this above in the extension of BST to recursion, as BCSL/ST uses BST contexts as its assertions.)

3. Extend the proof rules to account for an environment component Ω which is a collection of pre-condition and post-condition specification for command variables. Judgement then become of the form $\Omega \vdash \{A\}\,P\,\{B\}$ instead of only $\{A\}\,P\,\{B\}$.

We can formally extend the syntax of BCSL/ST along these lines in the same way that we did for BST. As we expect that this will be clear, we omit the details.

Now, we can give the proof rules for the new commands: these are given in Figure 3.10. Like in BST, the environment part is treated additively (staying the same) for all the other terms. We also use terms up to α-equivalence where $\texttt{Let}\ X\ =\ P\ \texttt{in}\ Q =_\alpha Let\ Y = P[Y/X]\ in\ Q[Y/X]$.

$$\text{[ProcCall]}\ \frac{}{\Omega, \{A\}\,X\,\{B\} \vdash \{A\}\,X\,\{B\}} \qquad \text{[ProcDecl]}\ \frac{\begin{array}{c}\Omega, \{A'\}\,X\,\{B'\} \vdash \{A'\}\,P\,\{B'\} \\ \Omega, \{A'\}\,X\,\{B'\} \vdash \{A\}\,Q\,\{B\}\end{array}}{\Omega \vdash \{A\}\,\texttt{Let}\ X\ =\ P\ \texttt{in}\ Q\,\{B\}}$$

Figure 3.10: Hoare Logic Procedure Rules

We have extended the syntax of session typing and program logic formalisms to deal with recursion. To connect them, we must extend the translation $\langle\langle - \rangle\rangle$ from BST to BCSL/ST in the following way.

$$\begin{array}{lrcl} \textit{Process variables} & \langle\langle X \rangle\rangle & = & X \\ \textit{Context} & \langle\langle \Theta, X : \Delta \rangle\rangle & = & \langle\langle \Theta \rangle\rangle, \{\Delta\}\,\langle\langle X \rangle\rangle\,\{emp\} \\ \textit{Recursion} & \langle\langle \texttt{Let}\ X\ =\ P\ \texttt{in}\ Q \rangle\rangle & = & \texttt{Let}\ \langle\langle X \rangle\rangle\ =\ \langle\langle P \rangle\rangle\ \texttt{in}\ \langle\langle Q \rangle\rangle \end{array}$$

The soundness argument from before extends without trouble.

Theorem 3.2.2 (Soundness of Translation with Recursion)**.**

$$\Theta \vdash P \triangleright \Delta \; in\ BST \Longrightarrow \langle\langle\Theta\rangle\rangle \vdash \{\Delta\}\,\langle\langle P\rangle\rangle\,\{emp\} \; in\ BCSL/ST$$

Proof:

The proof is by induction on the derivation of $\Theta \vdash P \triangleright \Delta$ as in Theorem 3.2.1. We concentrate on the *Var* and *Def* cases only, as all other rules can be proven as in the proof given earlier in Theorem 3.2.1.

Case `Var`**:**

$$\frac{}{\Theta, X : \Delta \vdash X \triangleright \Delta}$$

We are required to show

$$\langle\langle\Theta\rangle\rangle, \{\Delta\}\,\langle\langle X\rangle\rangle\,\{emp\} \vdash \{\Delta\}\,\langle\langle X\rangle\rangle\,\{emp\}$$

The result follows from the *ProcCall* rule..

Case `Def`**:**

$$\frac{\Theta, \mathtt{X} : \Delta' \vdash P \triangleright \Delta' \quad \Theta, \mathtt{X} : \Delta' \vdash Q \triangleright \Delta}{\Theta \vdash \mathtt{Let}\ X \ = \ P \ \mathtt{in}\ Q \triangleright \Delta}$$

We are required to show

$$\langle\langle\Theta\rangle\rangle \vdash \{\Delta\}\,\mathtt{Let}\ \langle\langle X\rangle\rangle \ = \ \langle\langle P\rangle\rangle \ \mathtt{in}\ \langle\langle Q\rangle\rangle\,\{emp\}$$

By induction hypothesis we get

$$\langle\langle\Theta\rangle\rangle, \{\Delta'\}\,\langle\langle X\rangle\rangle\,\{emp\} \vdash \{\Delta'\}\,\langle\langle P\rangle\rangle\,\{emp\}$$

and

$$\langle\langle\Theta\rangle\rangle, \{\Delta'\}\,\langle\langle X\rangle\rangle\,\{emp\} \vdash \{\Delta'\}\,\langle\langle Q\rangle\rangle\,\{emp\}$$

Then we can apply the *ProcDecl* rule to get the desired result.

■

3.3 Discussion

We have succeeded in making a first connection between a formalism inspired by Session Types and a version of Concurrent Separation Logic. We could criticise the result as being heavily tied to the syntax of Session Types, as well, not identifying underlying structure. But it is a start, and now we turn to the relation between models of Separation Logic and algebra, as we search for more structure.

Chapter 4

Exchange & Locality

Where the previous chapter was concerned with syntax, we now turn our attention to (denotational) semantics. We are going to look at a standard model of Separation Logic, and consider the mathematical structure that it has. The hope is that by identifying the abstract structures in the models, we can help to make links between different formalisms. The particular kind of mathematical structure we are looking for comes from the work on Concurrent Kleene Algebra [22], as described earlier in Section 2.5.

In more detail, in this chapter we start from the standard Local Action model of Separation Logic (SL) [9] and attempt to identify the algebraic structures that the model satisfies. The model is built from state transformers, and is equipped with notions of parallel and sequential composition. We find, first, that some, but not all, of the structure of a CKA is obtained in the model. Then, we show how two algebraic interpretations of Hoare triples coincide, in the model, with the 'fault-avoiding interpretation of triples' that underpins the work on Separation Logic [9, 29, 47].

These detailed findings shed some light on the connection between Separation Logic and the ideas behind Concurrent Kleene Algebra. We previously mentioned in section 2.5.3 that CKA give us rules for CSL, however, the authors of [22] think that they use a cheat:

> "The validity of Hoare logic in this weak model is entirely due to a cheat: that we use the same model for our assertions as for our programs. Thus any weakness of the programming model is immediately reflected in the weakness of the assertion

> language and its logic. In fact, conventional assertions mention the current values of single-valued program variables; and this is not adequate for reasoning about general fine-grain concurrency."

Because we have shown that a standard model of Separation Logic in fact satisfies many of the axioms of Concurrent Kleene Algebra, we might say that we have shown that there is in fact no cheat in the connection between algebraic notions and the logic. That is, the connection is stronger and more satisfying than the authors of [22] suggest.

While our initial motivation was to use the concrete model of Local Action to probe the algebraic structures, a secondary effect of this effort has been an improved understanding of the Local Action model itself. We actually end up with two models, one with locality and the other without, and a Galois connection between them. The Galois connection uses a *localising* construction $(-) * \mathsf{skip}$, and it turns out that the locality condition from [9] holds of a function f just when $f = f * \mathsf{skip}$. Thus, locality is now seen in a simpler and more general algebraic form. We also find, in contrast with our initial expectation, that the Exchange Law does not itself rely on locality: the non-local model admits exchange, and as a consequence it validates the *Concurrency* but not the *Frame* rule of SL.

There is however a caveat, that associativity fails for the parallel composition operator in the Local Action model, making the model defective in some respects. However, we include it as it relates very well to the familiar notion of locality from SL. As a response to this defect, we also include a model we call the Resource Model, built from predicate transformers rather than state transformers. It is very similar to the Local Action model, but has associativity. Although, historically the predicate transformer model presented later in the chapter stemmed from the work in Chapter 5, we include this chapter here for technical flow and for it to act as a cushion for what is to come in the forthcoming chapter (where we will have a rather unusual predicate transformer model).

4.1 State Transformers

We begin by examining a standard model of Separation Logic from [9]. The model is defined using certain state transformers, functions of type $\Sigma \mapsto P(\Sigma)^{\top}$, where $\top$ indicates the possibility of a memory fault or *race* condition. The set of states Σ itself comes with an algebraic structure, that of a *Separation Algebra*, that formalises the intuition of resource separation in Separation

Logic, and the state transformers have locality conditions that describe a concept of locality linked to the *Frame* rule of Separation Logic. Our aim is to investigate the algebraic structure of the set $\Sigma \mapsto P(\Sigma)^\top$ of state transformers, and to link it to concepts from Concurrent Kleene Algebra. Here is the notion of *Separation Algebra* from [9].

Definition 4.1.1 (Separation Algebra [9]). *A Separation Algebra is a cancellative, partial commutative monoid* $(\Sigma, \bullet, u)$ *. A partial commutative monoid is given by a partial binary operation where the unity, commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined. The cancellative property says that for each* $\sigma \in \Sigma$*, the partial function* $\sigma \bullet (-) : \Sigma \rightharpoonup \Sigma$ *is injective. The induced Separateness (#) relation and Substate (*$\preceq$*) relation is given by*

$$\begin{aligned} \sigma_0 \# \sigma_1 &\iff \sigma_0 \bullet \sigma_1 \textit{ is defined} \\ \sigma_0 \preceq \sigma_2 &\iff \exists \sigma_1 .\, \sigma_2 = \sigma_0 \bullet \sigma_1 \end{aligned}$$

The cancellativity requirement is used in the proof of Lemma 4.3.5 concerning the lattice structure of the collection of those transformers $\Sigma \mapsto P(\Sigma)^\top$ satisfying a locality condition, which we take from [9]. The next few definitions describe the lattice structure of the topped powerset $P(\Sigma)^\top$ and of the set of state transformers.

Definition 4.1.2 ([9]). *Let* Σ *be a Separation Algebra. Predicates over* Σ *are just elements of the powerset* $P(\Sigma)$*. It has an ordered total commutative monoid structure* $(*, emp)$ *given by*

$$\begin{aligned} p * q &= \{\sigma_0 \bullet \sigma_1 \mid \sigma_0 \# \sigma_1 \wedge \sigma_0 \in p \wedge \sigma_1 \in q\} \\ emp &= \{u\} \end{aligned}$$

Definition 4.1.3 ([9]). $P(\Sigma)^\top$ *is obtained by adding a new greatest element to* $P(\Sigma)$*. It has a total commutative monoid structure, keeping the unit emp the same as in* $P(\Sigma)$*, and extending* $*$ *so that* $p * \top = \top * p = \top$*.*

Definition 4.1.4 (Actions [9]). *We refer to functions* $f : \Sigma \to P(\Sigma)^\top$ *as actions. The function space*

$$[\Sigma \to P(\Sigma)^\top]$$

with pointwise order is a complete lattice.

We now undertake to define operations on the lattice $[\Sigma \mapsto P(\Sigma)^\top]$ corresponding to sequential and parallel composition. These operators give us the data for comparison to Concurrent Kleene Algebra.

The sequential composition $f;g$ of *actions* functionally composes f with the lifting $g^\dagger : P(\Sigma)^\top \to P(\Sigma)^\top$. This means that $\top$ trumps: if $f(\sigma)$ is a set that includes a state which g sends to $\top$, then $(f;g)\sigma = \top$.

Definition 4.1.5 (Lifting [9])**.** *If $f : \Sigma \to P(\Sigma)^\top$ then the lifting $f^\dagger : P(\Sigma)^\top \to P(\Sigma)^\top$ is defined by saying $f^\dagger\top = \top$ and $f^\dagger X = \bigsqcup\{f\sigma \mid \sigma \in X\}$ if $X \neq \top$.*

Definition 4.1.6 (Sequential Composition)**.** *For actions f and g, we define*

$$(f;g)\sigma \quad = \quad \begin{cases} \top & \textit{if } f\sigma = \top \\ \bigsqcup\{g\sigma' \mid \sigma' \in f\sigma\} & \textit{otherwise} \end{cases}$$

; *is easily seen to be monotone in both arguments.*

Definition 4.1.7 (Parallel Composition)**.** *For actions f and g, we define*

$$(f * g)\sigma \quad = \quad \bigsqcap\{f\sigma_1 * g\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\}$$

also $*$ is easily seen to be monotone in both arguments.

The sequential and parallel operators both have units, but they are not the same.

$$\mathsf{skip} \; = \; \lambda\sigma.\{\sigma\}$$

is the unit of ;, and

$$\mathsf{nothing} \; = \; \lambda\sigma.\,\text{if } \sigma = u \text{ then } \{u\} \text{ else } \top$$

is the unit of $*$.

Proposition 4.1.8 (Units)**.**

1. *$f;\mathsf{skip} = f$ and $\mathsf{skip};f = f$.*

   Proof:

   Case $f;\mathsf{skip} = f$: *When $f\sigma = \top$ then the cases are trivial.*

$$\begin{aligned} (f;\mathsf{skip})\sigma &= \bigsqcup\{\mathsf{skip}\sigma' \mid \sigma' \in f\sigma\} \\ &= \bigsqcup\{\{\sigma'\} \mid \{\sigma' \in f\sigma\} \\ &= f\sigma \end{aligned}$$

Case $\mathsf{skip}; f = f$

$$\begin{aligned}(\mathsf{skip}; f)\sigma &= \bigsqcup\{f\sigma' \mid \sigma' \in \mathsf{skip}\sigma\} \\ &= \bigsqcup\{f\sigma' \mid \sigma' \in \{\sigma\}\} \\ &= f\sigma\end{aligned}$$

■

2. $f * \mathsf{nothing} = f$.

Proof:

$$\begin{aligned}(f * \mathsf{nothing})\sigma &= \bigsqcap\{f\sigma_1 * \mathsf{nothing}\ \sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\} \quad (\top \textit{ if } \sigma_2 \neq u) \\ &= \bigsqcap\{f\sigma * \mathsf{nothing}\ u \mid \sigma = \sigma \bullet u\} \\ &= \bigsqcap\{f\sigma * \{u\} \mid \sigma = \sigma \bullet u\} \\ &= \bigsqcap\{f\sigma \mid \sigma = \sigma \bullet u\} \\ &= f\sigma\end{aligned}$$

■

nothing is not the sort of thing that we can define in a programming language, and does not satisfy the locality condition used in the next section. Since the units are distinct for ; and $*$ we immediately identify that *Actions* do not form a CKA.

Before continuing with our technical work, we make some remarks on the definition of $*$ of Action. This interpretation of $*$ is motivated by the proof rules of CSL. We can use the parallel proof rule to generate lots of potential post-conditions, and then the usual *Conjunction* rule of Hoare logic to obtain the most accurate possible post-condition by taking the intersection of all those. To see what we mean by this, using the *Concurrency* rule together with the rule of consequence, one can derive

$$\frac{P \vdash P_1 * P_2 \quad \{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P\}\,C_1 \parallel C_2\,\{Q_1 * Q_2\}}$$

and also

$$\frac{P \vdash P_1' * P_2' \quad \{P_1'\}\,C_1\,\{Q_1'\} \quad \{P_2'\}\,C_2\,\{Q_2'\}}{\{P\}\,C_1 \parallel C_2\,\{Q_1' * Q_2'\}}$$

Therefore, using the usual Hoare logic rule of conjunction on post-conditions

$$\frac{\{P\}\,C\,\{S_1\} \quad \{P\}\,C\,\{S_2\}}{\{P\}\,C\,\{S_1 \wedge S_2\}}$$

one obtains

$$\frac{P \vdash P_1 * P_2 \quad \{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\} \quad P \vdash P'_1 * P'_2 \quad \{P'_1\}C_1\{Q'_1\} \quad \{P'_2\}C_2\{Q'_2\}}{\{P\}C_1 \parallel C_2\{(Q_1 * Q_2) \wedge (Q'_1 * Q'_2)\}}$$

and Definition 4.1.7 is just an infinitary version of this proof.

Note that the *Conjunction* rule has not been included in BCSL, though it is included in the original CSL. It is not needed for making the connection to Session Types later. The remarks on the *Conjunction* rule just given are more to help understand the semantic definition of the $*$ of functions: the intuition behind $*$ in this model is more logical than operational. Also, we remark that this definition of parallel composition abstracts away from intermediate steps of computation, and thus does not provide an accurate model of process interaction. It is a somewhat coarse over-approximation of a more detailed model such as the one in [6]. This over-approximation is related to what is computed in some program analyses for concurrency [8, 15].

The definition of $f * g$ has a mixed angelic/demonic character. Let us say that $\sigma = \sigma_1 \bullet \sigma_2$ is a 'safe splitting' if $f\sigma_0 \neq \top$ and $g\sigma_1 \neq \top$. It is angelic as far as faulting goes, in the sense that if there is any safe splitting, then $(f * g)\sigma$ is not $\top$: the angel tries to avoid $\top$ by searching for a good splitting. It is also a safety first approach, in that only states which are possible outcomes under all splittings are tallied by the *glb* $\bigsqcap$. But it is demonic as far as non-faulting output states go: to have $\sigma' \in (f * g)\sigma$, it must be that $\sigma' \in f\sigma_0 * g\sigma_1$ for every safe splitting; the $*$ operator on predicates filters out the rest.

Example 4.1.9 (Angels and Demons)**.** *Recall from Section 2.4.1 that by the Heap model we mean finite partial functions from naturals to naturals, $\Sigma = N \rightharpoonup_f N$, with $\bullet$ being union of functions with disjoint domains and u being the empty function. With this model we will use commands*

$$\begin{array}{lcl} [n] := m & = & \lambda\sigma.\,\mathsf{if}\ n \in dom(\sigma)\ \mathsf{then}\ \{(\sigma \mid n \mapsto m)\}\ \mathsf{else}\ \top \\ [n] := \mathsf{alloc}() & = & \lambda\sigma.\,\mathsf{if}\ n \in dom(\sigma)\ \mathsf{then}\ \bigcup_{m \notin dom(\sigma), r \in N}\{(\sigma \mid n \mapsto m, m \mapsto r)\}\ \mathsf{else}\ \top \\ \mathsf{dispose}[n] & = & \lambda\sigma.\,\mathsf{if}\ n \in dom(\sigma)\ \mathsf{then}\ \bigcup_{m \in N}\{(\sigma \mid n \mapsto m)\}\ \mathsf{else}\ \top \end{array}$$

which are all functions $\Sigma \to P(\Sigma)^\top$. To illustrate the angelic and demonic aspects of parallel composition, consider the following program:

$$\left(\begin{array}{l} [10] := \mathsf{alloc}(); \\ \mathsf{if}\ [10] = 20\ \mathsf{then}\ [30] := 1\ \mathsf{else}\ [30] := 2; \\ \mathsf{dispose}[10]; \\ [10] := 8; \end{array}\right) \quad * \quad \mathsf{skip}$$

Suppose we run this in start state $[10 \mapsto 8, 20 \mapsto 66, 30 \mapsto 88]$. *Thinking operationally about the global state, the allocator will not allocate cell 20, so we will never have* $[10] = 20$ *in the condition of the left program. Thus,* $[10 \mapsto 8, 20 \mapsto 66, 30 \mapsto 1]$ *is never a final state of the program. In the local-state semantics, however, there is in fact a splitting where* $[10 \mapsto 8, 20 \mapsto 66, 30 \mapsto 1]$ *is a final state. That is the splitting where* $[20 \mapsto 66]$ *is sent to* skip. *But there is another splitting, where* $[]$ *is sent to* skip, *in which* $[10 \mapsto 8, 20 \mapsto 66, 30 \mapsto 1]$ *cannot result. Thus the demon stops this final state from occurring.* ■

4.2 Exchange

In this section we investigate versions of the Exchange Law in the Action model.

Definition 4.2.1 (Exchange Laws)**.** *We consider the following formulas for* $p, q, r, s \in Action$:

$$(p * r)\,;(q * s) \sqsubseteq (p;q) * (r;s) \qquad \textit{full exchange}$$
$$(r * p);q \sqsubseteq r * (p;q) \qquad \textit{small exchange I}$$
$$p;(q * r) \sqsubseteq (p;q) * r \qquad \textit{small exchange II}$$

Interestingly, we will find that the full Exchange Law holds in our model, but not the small laws. In contrast to Concurrent Kleene Algebra, it is possible to have the full while not the small laws here because the units of ; and $*$ are different in our model. From an algebraic perspective, it will be the job of locality, considered in the next subsection, to re-establish the small laws.

It will be helpful to give an example of the full exchange law, and then counter examples to the small versions of it. We re-use the commands defined in example 4.1.9 to give the examples.

Example 4.2.2 (Exchange in Heap Model)**.**

$$([10] := 55 * [20] := 66)\,;([20] := 77 * [10] := 88) \qquad (\neq \top)$$
$$\sqsubseteq$$
$$([10] := 55\,;[20] := 77) * ([20] := 66\,;[10] := 88) \qquad (= \top)$$

The reason that this holds is that the lower term is $\top$, *as there is a race on* 10 *as well as on* 20: *there is no way to split the state in such a way that the result of both sides are non-*$\emptyset$. *On the other hand, the upper term is not* $\top$ *since, given a state whose domain contains both* 10 *and* 20, *we can split the state for the first parallel composition, and again for the second: thus, there is a non* $\top$ *pre-condition for this program.*

Note that this example also shows that the reverse direction of the Exchange Law is not valid in the model. ■

Example 4.2.3 (Invalidity of small exchange in the Heap Model)**.** *To obtain a counterexample to the law*

$$p\,;(q*r) \sqsubseteq (p\,;q)*r,$$

we choose $p = q = \mathsf{nothing}$ *and* $r = \mathsf{skip}$. *We observe that* $\mathsf{nothing}\,;\mathsf{nothing} = \mathsf{nothing}$ *and that the left hand side is then* $\mathsf{nothing}\,;(\mathsf{nothing} * \mathsf{skip}) = \mathsf{nothing}$ *and the right hand side simplifies to* skip. *Clearly it is not the case that* $\mathsf{nothing} \sqsubseteq \mathsf{skip}$.

We let $h = \{10 \mapsto 50\}$ *(the singleton set of a function only defined at 10), and evaluate both sides on h. The left hand side is* $\top$ *while the right hand side contains at least h because* $h = u \bullet h$ *and*

$$(\mathsf{nothing}\,;\mathsf{nothing})u = \{u\} \qquad \text{and} \qquad \mathsf{skip}\ h = \{h\}$$

so $u \bullet h \in (\mathsf{nothing}\,;\mathsf{nothing}) * \mathsf{skip}\ (u \bullet h)$.

The same instantiation yields a counterexample to

$$(r*p)\,;q \sqsubseteq r*(p\,;q).$$

■

We also consider that the reverse inclusion for the small laws and provide counter example.

Example 4.2.4 (Invalidity of small exchange for the reverse order)**.** *To obtain a counter example to*

$$p;(q*r) \sqsupseteq (p;q)*r$$

we choose $p = [x] := alloc()$, $q = \mathsf{nothing}$ *and* $r = dispose[x]$. *Then* $(([x] := alloc(); \mathsf{nothing}) * dispose[x])u = \top$ *while* $([x] := alloc(); (\mathsf{nothing} * dispose[x]))u = \{u\}$.

To obtain a counter example to

$$(p*q);r \sqsupseteq p*(q;r)$$

we choose $p = skip$, $q = \mathsf{nothing}$ *and* $r = dispose[10]$ *then* $(skip*(nothing;dispose[10]))(10 \mapsto 50) = \top$ *and* $((skip*\mathsf{nothing});dispose[10])(10 \mapsto 50) = \{10 \mapsto 50\}$ *where we give u to the right hand side of the* $*$ *and* $10 \mapsto 50$ *to the left hand side.* ■

Proposition 4.2.5 (Exchange Law)**.** *Actions satisfy the Exchange Law*

$$(f_1 * g_1);(f_2 * g_2) \sqsubseteq (f_1;f_2) * (g_1;g_2)$$

In order to prove the Exchange Law, we first need a technical lemma:

Lemma 4.2.6. *Let f_1, f_2, g_1 and g_2 be actions and let σ, σ', σ_1 and σ_2 be elements of Heaps. If*

$$\begin{aligned} & \sigma' \in \bigsqcap\{f_1\sigma_1 * g_1\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\} \\ \text{and} \quad & \sigma_1 \bullet \sigma_2 = \sigma \end{aligned}$$

then

$$\bigsqcap\{f_2\sigma_1' * g_2\sigma_2' \mid \sigma' = \sigma_1' \bullet \sigma_2'\} \sqsubseteq (f_1;f_2)\sigma_1 * (g_1;g_2)\sigma_2$$

Proof:

If either $f_1\sigma_1 = \top$ or $g_1\sigma_2 = \top$ then the right-hand side is $\top$, so assume that neither is $\top$. We can then rewrite the right-hand side:

$$\begin{aligned} (f_1;f_2)\sigma_1 * (g_1;g_2)\sigma_2 &= \bigsqcup\{f_2\sigma_1' \mid \sigma_1' \in f_1\sigma_1\} * \bigsqcup\{g_2\sigma_2' \mid \sigma_2' \in g_1\sigma_2\} \\ &= \bigsqcup\{f_2\sigma_1' * g_2\sigma_2' \mid \sigma_1' \in f_1\sigma_1, \sigma_2' \in g_1\sigma_2\} \end{aligned}$$

The second step here used that $*$ of predicates (i.e., $*$ on $P(\Sigma)^\top$) preserves all *lubs* (it does not require $*$ of *Actions* to preserve *lubs*). Now, since $f_1\sigma_1 \neq \top$ and $g_1\sigma_2 \neq \top$ and $\sigma' \in \bigsqcap\{f_1\sigma_1 * g_1\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\}$, there is $\sigma_1' \in f_1\sigma_1$ and $\sigma_2' \in g_1\sigma_2$ such that $\sigma' = \sigma_1' \bullet \sigma_2'$. And so

$$\bigsqcap\{f_2\sigma_1' * g_2\sigma_2' \mid \sigma' = \sigma_1' \bullet \sigma_2'\} \sqsubseteq \bigsqcup\{f_2\sigma_1' * g_2\sigma_2' \mid \sigma_1' \in f_1\sigma_1, \sigma_2' \in g_1\sigma_2\}$$

■

It is well known that in Separation Logic the $*$ of assertions commutes with arbitrary unions. That is, whenever a partial commutative monoid (a concrete separation algebra) is used to induce a total monoid structure on the powerset, the monoid operator on this powerset preserves all unions [9, 46].

Corollary 4.2.7. *For f_1, f_2, g_1 and g_2 actions we have*

$$\begin{aligned} & \bigsqcup\{\bigsqcap\{f_2\sigma_1' * g_2\sigma_2' \mid \sigma' = \sigma_1' \bullet \sigma_2'\} \mid \sigma' \in \bigsqcap\{f_1\sigma_1 * g_1\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\}\} \\ \sqsubseteq \quad & \bigsqcap\{(f_1;f_2)\sigma_1 * (g_1;g_2)\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\} \end{aligned}$$

Proof:

The inequality is of the form $\bigsqcup A \sqsubseteq \bigsqcap B$. If $A = \emptyset$ (no suitable σ') then the left-hand side is $\emptyset$, while if $B = \emptyset$ (no splitting of σ) then the right-hand side is $\top$, so in either of those cases the inequality holds. If both A and B are non-empty then, by Lemma 4.2.6, all elements of A are less than all elements of B. So all elements of A are less than $\bigsqcap B$ and so $\bigsqcup A \sqsubseteq \bigsqcap B$.

■

We can now give the proof of the proposition.

Proof: [of Exchange Law, Proposition 4.2.5]

Suppose $(f_1 * g_1)(\sigma) = \top$

Then $\forall \sigma_1, \sigma_2.\ \sigma = \sigma_1 \bullet \sigma_2 \Rightarrow (f_1 * g_1)(\sigma) \sqsubseteq f_1\sigma_1 * g_1\sigma_2 \Rightarrow f_1\sigma_1 = \top \vee g_1\sigma_2 = \top$ and so $((f_1; f_2) * (g_1; g_2))\sigma = \top$

Suppose $(f_1 * g_1)(\sigma) \neq \top$

$$
\begin{aligned}
& ((f_1 * g_1); (f_2 * g_2))\sigma \\
= \ & \bigsqcup\{(f_2 * g_2)\sigma' \mid \sigma' \in (f_1 * g_1)\sigma\} \\
= \ & \bigsqcup\{(f_2 * g_2)\sigma' \mid \sigma' \in \bigsqcap\{f_1\sigma_1 * g_1\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\}\} \\
= \ & \bigsqcup\{\bigsqcap\{f_2\sigma_1' * g_2\sigma_2' \mid \sigma' = \sigma_1' \bullet \sigma_2'\} \mid \sigma' \in \bigsqcap\{f_1\sigma_1 * g_1\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\}\} \\
\sqsubseteq \ & \bigsqcap\{(f_1; f_2)\sigma_1 * (g_1; g_2)\sigma_2 \mid \sigma = \sigma_1 \bullet \sigma_2\} \\
= \ & ((f_1; f_2) * (g_1; g_2))\sigma
\end{aligned}
$$

where the next-to-last step uses Corollary 4.2.7.

■

In summary, the Action model satisfies some, but not all, of the properties of a CKA:

Lemma 4.2.8. *In the Action model we have two ordered monoids* $(Action, \sqsubseteq, *, \mathsf{nothing})$ *and* $(Action, \sqsubseteq, ;, \mathsf{skip})$ *which are linked by the exchange law and where* $*$ *is commutative.*

The properties of the Action Model seem a bit odd at first sight. We have a situation where the small exchange laws are invalid, but the full exchange law is valid. In program logic terms this means that the *Concurrency* rule (which expresses a form of modular reasoning) is present while the *Frame* rule (which expresses locality, and corresponds to small exchange laws for Hoare or Plotkin triples) is absent. We now seek to probe this issue.

4.3 Local Action

Now we investigate the locality condition underpinning the *Frame* rule of Separation Logic, and link it to algebra. We obtain a characterisation of locality in Lemma 4.3.6 (Locality Characterisation), which shows a purely algebraic condition that is much more general but which dovetails with the notion used in the specific model. This opens the way to a more abstract understanding of the algebraic structure surrounding locality, eventually leading to a Galois connection between local and non-local Action. Here is the main notion from [9].

Definition 4.3.1 (Local Action [9])**.** *Suppose* Σ *is a Separation Algebra. A Local Action* $f : \Sigma \to P(\Sigma)^\top$ *is a function satisfying the locality condition:*

$$\sigma_1 \# \sigma_2 \ \textit{implies} \ f(\sigma_1 \bullet \sigma_2) \sqsubseteq f\sigma_1 * \{\sigma_2\}.$$

We let LocAct denote the set of local actions, with pointwise order.

The locality condition states that a Local Action f when applied to state σ gives a smaller set of states than when f only uses a portion of the state (σ_1) and leaving another portion (σ_2) untouched, where $\sigma = \sigma_1 \bullet \sigma_2$ and $\sigma_1,\ \sigma_2$ are separate and if $f\sigma = \top$ then $f\sigma_1 * \{\sigma_2\} = \top$.

The locality condition is a way to formalise the intuition that when a program operates without faulting on a small state, then it leaves any extra or separate state unchanged. An important point is that when f does not fault on a small state ($f\sigma_1 \neq \top$), then neither does it on a larger state. The reason is that, if $f\sigma_1 \neq \top$ then $f\sigma_1 * \{\sigma_2\}$ cannot be $\top$ and we require that $f(\sigma_1 \bullet \sigma_2)$ is less than or equal to this. That the result $f(\sigma_1 \bullet \sigma_2)$ is dominated by $f\sigma_1 * \{\sigma_2\}$ captures the intuition that σ_2, the additional or separate state, is left unchanged.

To appreciate the definition of Local Action it is helpful to consider a non-example. In the Heap model, the function $f : \sigma \mapsto \{[10 \mapsto 2]\}$ is the constant function which maps any input state to the single output consisting of a heap with only one allocated cell, 10, with contents 2. Technically, this function is not local because of the following:

$$f([11 \mapsto 4, 12 \mapsto 10]) = \{[10 \mapsto 2]\}$$
$$\not\subseteq$$
$$f(11 \mapsto 4) * \{[12 \mapsto 10]\} = \{[10 \mapsto 2, 12 \mapsto 10]\}$$

Less technically, f violates the intuition of 'locally accessed resources' for the following reason. For local f, if $f([11 \mapsto 4])$ does not deliver $\top$ then we think that it does not 'access' any locations other than 11. Furthermore, it disposes 11 and allocates 10. This should mean that it leaves location 12 alone, if 12 happens to be allocated in the pre-state. However, $f([11 \mapsto 4, 12 \mapsto 10]) = \{[10 \mapsto 2]\}$ disposes both 11 and 12, and certainly does not leave 12 alone. In this sense, f does not access circumscribed resources.

The proof of lemma 4.3.5 (Local Action form a Complete Lattice) and proposition 4.3.9 (Structure of Localisation) makes use of a special property of singleton states called 'precision'. The Precision Characterisation Lemma (Lemma 4.3.4) enables us to use the fact that the $*$ of predicates distributes through $\bigsqcap$ (*glb*) when dealing with *precise* states. We will also find that the connection between $*$, precision and *glb*'s is important for (non) associativity of $*$, in the counter example in Section 4.4.3.

Definition 4.3.2 (Precise Predicates [9]). *A predicate $p \in P(\Sigma)$ is precise if for every $\sigma \in \Sigma$, there exists at most one $\sigma_p \preceq \sigma$ such that $\sigma_p \in p$.*

Intuitively, as explained in [39], a *precise* predicate cuts out a specific part of memory corresponding to a data structure, the unique σ_P within σ (when σ_P exists). For example, the data structure acyclic singly-linked list rooted at x determines a *precise* predicate. Given σ, σ_P consists of all those nodes in σ obtained by following tail links from x, if this traversal of links terminates at nil. If the traversal leads to a dangling pointer or a cycle, there is no such σ_P.

The following sets down some examples in more formal detail.

Example 4.3.3 (Precise and Imprecise Predicates). *We use the heap model defined in Section 2.4.1 to give the following examples.*

1. *The predicate $5 \mapsto 7 \in P(\Sigma)$ is the set $\{\sigma\}$ consisting of a single state σ where σ maps cell 5 to 7 and σ is undefined for all other cells. This predicate is precise.*

2. *The predicate $5 \mapsto 7 * 6 \mapsto 8$ is the set $\{\sigma\}$ where σ is only defined for cells 5 and 6, mapping them to 7 and 8 respectively. This predicate is also precise.*

3. *The predicate $5 \mapsto 7 \sqcup 6 \mapsto 8$ is the set $\{\sigma, \sigma'\}$ (disjunction) where σ maps cell 5 to 7 and σ' maps cell 6 to 8 and undefined otherwise. This predicate is imprecise.*

■

1 and 2 in this example are *precise* because they each define a single state σ' which plays the role of σ_P, for any P consisting of states that contain this σ'. On the other hand, the predicate in 3 is *imprecise* because, considering $\sigma = 5 \mapsto 7 * 6 \mapsto 8$ in the definition of precision, there are two states smaller than σ which are in the predicate, the states $5 \mapsto 7$ and $6 \mapsto 8$, where the notion of precision requires that there be one. Connecting back to the above-stated intuition of precision, the disjunctive predicate does not cut out a unique portion of memory corresponding to a data structure. We will not discuss the conceptual basis for *precise* predicates further in this thesis, and instead refer to [39] and [9] for further discussion.

Lemma 4.3.4 (Precision Characterisation [9])**.**

1. *Every singleton predicate* $\{\sigma\}$ *is precise.*
2. p *is precise if and only-if for all* $X \subseteq P(\Sigma)$*,*

$$\bigsqcap X * p = \bigsqcap \{x * p \mid x \in X\}$$

Lemma 4.3.5 (Local Actions form a Complete Lattice [9])**.** *LocAct is a complete lattice, with meets and joins defined pointwise (and inherited from the function space* $[\Sigma \to P(\Sigma)^\top]$*).*

The definition of Local Action was given in order to validate the *Frame* rule. The local actions can be picked out, amongst the general actions, via the following pleasant characterisation.

Lemma 4.3.6 (Locality Characterisation)**.** *Let* $f : \Sigma \to P(\Sigma)^\top$*. Then* f *is a Local Action in the sense of [9] if and only-if* $f = f * \mathsf{skip}$ *that is*

$$f\sigma = \bigsqcap \{f\sigma_1 * \{\sigma_2\} \mid \sigma = \sigma_1 \bullet \sigma_2\} \iff f = f * \mathsf{skip}.$$

Proof:

FOR THE 'IF' DIRECTION: We use locality to prove one direction

$$\begin{array}{rcll} (f * \mathsf{skip})\sigma & = & \bigsqcap\{f\sigma_1 * \{\sigma_2\} \mid \sigma = \sigma_1 \bullet \sigma_2\} & \\ & \sqsupseteq & \bigsqcap\{f(\sigma_1 \bullet \sigma_2) \mid \sigma = \sigma_1 \bullet \sigma_2\} & \text{(by locality)} \\ & = & f\sigma & \end{array}$$

but we don't need it for the other

$$\begin{array}{rcll} (f * \mathsf{skip})\sigma & = & \bigsqcap\{f\sigma_1 * \{\sigma_2\} \mid \sigma = \sigma_1 \bullet \sigma_2\} & \\ & \sqsubseteq & \bigsqcap\{f(\sigma \bullet u) \mid \sigma = \sigma \bullet u\} & \text{(taking particular } \sigma_1, \sigma_2) \\ & = & f\sigma & \end{array}$$

where u is the unit of $\bullet$.

FOR THE 'ONLY-IF' DIRECTION: Consider a fixed splitting $\sigma = \sigma_1 \bullet \sigma_2$ of state σ. Then

$$f\sigma = (f * \mathsf{skip})\sigma = \bigsqcap\{f\sigma_1' * \{\sigma_2'\} \mid \sigma = \sigma_1' \bullet \sigma_2'\}$$

Since this is a lower bound of $X = \{f\sigma_1' * \{\sigma_2'\} \mid \sigma = \sigma_1' \bullet \sigma_2'\}$, and since $f\sigma_1 * \{\sigma_2\} \in X$ for the fixed splitting $\sigma = \sigma_1 \bullet \sigma_2$, it must be the case that $f(\sigma_1 \bullet \sigma_2) \sqsubseteq f\sigma_1 * \{\sigma_2\}$.

■

Thus Local Action have a shared unit for both operators. As a result the small exchange laws are validated.

Lemma 4.3.7. *Local Action satisfy the small exchange laws.*

The proof is simply to instantiate the exchange law with the shared unit for r and s respectively. Now we have to only verify that ; and $*$ preserve locality.

Lemma 4.3.8 (Locality Preservation)**.** *If f and g are local then so are $f;g$ and $f*g$.*

Proof:

Case $f;g$:

$$\begin{array}{rcll} f;g & = & (f*\mathsf{skip});(g*\mathsf{skip}) & \text{Locality of } f,g \\ & \sqsubseteq & (f;g)*(\mathsf{skip};\mathsf{skip}) & \text{Exchange Law} \\ & = & (f;g)*\mathsf{skip} & \mathsf{skip} \text{ unit of ;} \end{array}$$

Conversely,

$$\begin{array}{rcl} f;g & = & (f;g)*\mathsf{nothing} \\ & \sqsupseteq & (f;g)*\mathsf{skip} \end{array}$$

We get the last step since $\mathsf{nothing} \sqsupseteq \mathsf{skip}$.

Case $f*g$:

$$\begin{array}{rcl} (f*g)\sigma & = & (f*(g*\mathsf{skip}))\sigma \\ & = & \bigsqcap\{f\sigma_1 * \bigsqcap\{g\sigma_3 * \{\sigma_4\} \mid \sigma_2 = \sigma_3 \bullet \sigma_4\} \mid \sigma = \sigma_1 \bullet \sigma_2\} \\ & = & \bigsqcap\{f\sigma_1 * g\sigma_3 * \{\sigma_4\} \mid \sigma = \sigma_1 \bullet \sigma_3 \bullet \sigma_4\} \\ & = & \bigsqcap\{\bigsqcap\{f\sigma_1 * g\sigma_3 * \{\sigma_4\} \mid \sigma' = \sigma_1 \bullet \sigma_3\} \mid \sigma = \sigma' \bullet \sigma_4\} \\ & = & \bigsqcap\{\bigsqcap\{f\sigma_1 * g\sigma_3 \mid \sigma' = \sigma_1 \bullet \sigma_3\} * \{\sigma_4\} \mid \sigma = \sigma' \bullet \sigma_4\} \\ & = & (f*g)*\mathsf{skip} \end{array}$$

We get $(f*g)\sigma = (f*(g*\mathsf{skip}))\sigma$ from the locality of g.

■

4.3.1 Localisation

Local Action can thus be described in purely algebraic terms, as elements satisfying $f = f * \mathsf{skip}$, in a way that is more general than the definiton from [9]. In fact, $(-) * \mathsf{skip}$ can be seen as a function from Action to Local Action, illustrated by the following picture of the space of Action.

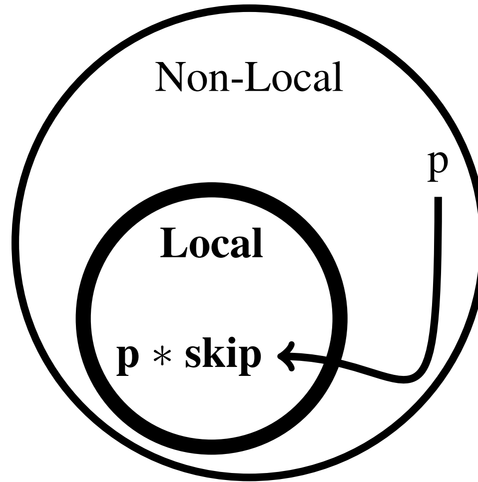


Figure 4.1: Non-Local & Local Action

This localising function has interesting properties, stated in the following result.

Proposition 4.3.9 (Structure of Localisation)**.**

1. $f \sqsubseteq g$ *implies* $f * \mathsf{skip} \sqsubseteq g * \mathsf{skip}$,

2. $(f * \mathsf{skip}) * \mathsf{skip} = f * \mathsf{skip}$,

3. $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$.

4. $f * \mathsf{skip}$ *is the largest Local Action smaller than or equal to* f*. In more detail, for arbitrary* $f, g : \Sigma \to P(\Sigma)^{\top}$

   (a) $f * \mathsf{skip} \sqsubseteq f$*, and*

   (b) if $g \sqsubseteq f$ *and* g *is local then* $g \sqsubseteq f * \mathsf{skip}$.

5. $(-) * \mathsf{skip}$ *is the right adjoint part of a Galois connection between actions and local actions, where its mate is the inclusion function from Local Action to Action. In more detail, if* $f, g : \Sigma \to P(\Sigma)^{\top}$ *and* g *is local, then*

$$g \sqsubseteq f \qquad \Longleftrightarrow \qquad g \sqsubseteq f * \mathsf{skip}$$

Proof: [of Proposition 4.3.9]

PROOF OF 4.3.9 - 1

$$\begin{aligned}(f * \mathsf{skip})\sigma &= \bigsqcap\{f\sigma_1 * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} \\ &\sqsubseteq \bigsqcap\{g\sigma_1 * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} \quad (f \sqsubseteq g \text{ and monotonicity of } * \text{ on } P(\Sigma)^\top) \\ &= (g * \mathsf{skip})\sigma\end{aligned}$$

PROOF OF 4.3.9 - 2

$$\begin{aligned}((f * \mathsf{skip}) * \mathsf{skip})\sigma &= \bigsqcap\{(f * \mathsf{skip})\sigma_1 * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} \\ &= \bigsqcap\{\bigsqcap\{f\sigma_3 * \{\sigma_4\} \mid \sigma_3 \bullet \sigma_4 = \sigma_1\} * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} \\ &= \bigsqcap\{\bigsqcap\{f\sigma_3 * \{\sigma_4\} * \{\sigma_2\} \mid \sigma_3 \bullet \sigma_4 = \sigma_1\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} (\text{Lemma 4.3.4}) \\ &= \bigsqcap\{f\sigma_3 * \{\sigma_4\} * \{\sigma_2\} \mid \sigma_3 \bullet \sigma_4 \bullet \sigma_2 = \sigma\} \\ &= \bigsqcap\{f\sigma_3 * \{\sigma_4\} * \{\sigma_2\} \mid \sigma_3 \bullet \sigma' = \sigma,\ \sigma_4 \bullet \sigma_2 = \sigma'\} \\ &= \bigsqcap\{f\sigma_3 * \{\sigma'\} \mid \sigma_3 \bullet \sigma' = \sigma\} \\ &= (f * \mathsf{skip})\sigma\end{aligned}$$

Lemma 4.3.4 states $\bigsqcap X * p = \bigsqcap\{x * p \mid x \in X\}$ if p is *precise*, and also that every singleton set is *precise*. For the step that uses this lemma we consider $X = \bigsqcap\{f\sigma_3 * \{\sigma_4\} \mid \sigma_3 \bullet \sigma_4\}$ and p to be the singleton $\{\sigma_2\}$.

PROOF OF 4.3.9 - 3

$$\begin{aligned}(\mathsf{skip} * \mathsf{skip})\sigma &= \bigsqcap\{\{\sigma_1\} * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} \\ &= \{\sigma\} \\ &= (\mathsf{skip})\sigma\end{aligned}$$

PROOF OF 4.3.9 - 4

There are two things to show:

(a) $f * \mathsf{skip} \sqsubseteq f$:

$$\begin{aligned}(f * \mathsf{skip})\sigma &= \bigsqcap\{f\sigma_1 * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} \\ &\sqsubseteq \bigsqcap\{f(\sigma \bullet u) \mid \sigma \bullet u = \sigma\} \qquad (\text{taking particular } \sigma_1, \sigma_2) \\ &= f\sigma\end{aligned}$$

(b) if $g \sqsubseteq f$ and g is local then $g \sqsubseteq f * \mathsf{skip}$:

$$\begin{array}{rcll} (f * \mathsf{skip})\sigma & = & \bigsqcap\{f\sigma_1 * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} & \\ & \sqsupseteq & \bigsqcap\{g\sigma_1 * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} & (g \sqsubseteq f \text{ and monotonicity of } *) \\ & \sqsupseteq & \bigsqcap\{g(\sigma \bullet u) \mid \sigma \bullet u = \sigma\} & (\text{by locality}) \\ & = & g\sigma & \end{array}$$

PROOF OF 4.3.9 - 5

The direction $g \sqsubseteq f * \mathsf{skip} \Rightarrow g \sqsubseteq f$ follows directly from the fact that $f * \mathsf{skip} \sqsubseteq f$ and $\sqsubseteq$ is transitive. The direction $g \sqsubseteq f \Rightarrow g \sqsubseteq f * \mathsf{skip}$ holds since $f * \mathsf{skip}$ is the largest Local Action smaller than f and g is a Local Action.

■

Before moving on it is worth remarking that keeping Action and Local Action separate, with a localiser between them, has a prettifying effect on the theory of Local Action. A good indication of this concerns one of the landmark developments in [9], where a definition was given of the largest or best local action corresponding to a Hoare triple. The relative simplicity of this (compared to what happened before faulting was put on the top, e.g., [39]) was one of the pleasant points. However, the definition came from thin air, as it were, and here we can derive it from an even simpler definition.

For $X, Y \in P(\Sigma)$ we define

$$great[X,Y] \; = \; (\lambda\sigma.\text{if } \sigma \in X \text{ then } Y \text{ else } \top) \; : \Sigma \to P(\Sigma)^{\top}.$$

It is easy to verify that $great[X,Y]$ is the the greatest state transformer satisfying the triple $\langle X \rangle - \langle Y \rangle$ (see Definition 4.4.1), according to the pointwise order on $[\Sigma \to P(\Sigma)^{\top}]$. Now, if we define best local actions (bla) as

$$bla[X,Y] \; = \; great[X,Y] * \mathsf{skip}$$

then we can calculate, using the definition of $*$,

$$bla[X,Y] \; = \; \lambda\sigma. \bigsqcap\{Y * \{\sigma_0\} \mid \sigma = \sigma_0 \bullet \sigma_1, \sigma_1 \in X\}$$

and this is just the definition of $bla[X,Y]$ from [9].

4.4 Relating Algebraic & Semantic Triples

We now present the connections between algebraic notion of triples and the interpretation of triples from [9]. In contrast to CKA where they rely on assertions to be from the same elements of the algebra as the programs, we give a translation from the assertions to the elements of the algebra and then connect the algebraic triples to semantic triples. Thus showing that the algebra is in agreement with the semantics, hence there is no cheat involved.

Definition 4.4.1 (Fault-avoiding triples [9])**.** . *If* $X, Y \in P(\Sigma)$ *and* $f : \Sigma \to P(\Sigma)^\top$ *then we define the triple* $\langle X \rangle\, f\, \langle Y \rangle$ *in the following way*

$$\langle X \rangle\, f\, \langle Y \rangle \Longleftrightarrow \forall \sigma \in X . f\sigma \sqsubseteq Y$$

This interpretation is 'fault avoiding', because the condition $f\sigma \sqsubseteq Y$ means that $f\sigma$ cannot be $\top$.

4.4.1 Hoare Triples

First, we relate the SL interpretation of the triple $\langle X \rangle\, f\, \langle Y \rangle$ to the one given in Section 2.5.3 where $\{p\}\, c\, \{q\} \Leftrightarrow p;c \sqsubseteq q$. To connect these two notions, we need to consider that the X and Y are different kinds of entities than the f in $\langle X \rangle\, f\, \langle Y \rangle$. A simple way to do so is with

$$\mathit{build}[Y] \ = \ \mathit{great}[\mathit{emp}, Y] \ : \Sigma \to P(\Sigma)^\top$$

So $\mathit{build}[Y](u) = Y$ and for $\sigma \neq u$, $\mathit{build}[Y](\sigma) = \top$.

Given *build*, it is very easy to connect the two kinds of triples in Action.

Proposition 4.4.2 (Agreement of Action and Algebraic Triples)**.** *For arbitrary* $f : \Sigma \to P(\Sigma)^\top$

$$\langle X \rangle\, f\, \langle Y \rangle \Longleftrightarrow \{\mathit{build}[X]\}\, f\, \{\mathit{build}[Y]\}$$

Note that here we are using the definition $\{p\}\, c\, \{q\} \Leftrightarrow p;c \sqsubseteq q$ which we introduced in the context of CKA. What we have is not quite a CKA, but the definition makes sense in any ordered monoid, without the need for the additional CKA structure.

Proof:

For the 'If' direction: $\{\mathit{build}[X]\}\, f\, \{\mathit{build}[Y]\} \Rightarrow \langle X \rangle\, f\, \langle Y \rangle$.

We know that $(\mathit{build}[X]; f)(u) \sqsubseteq \mathit{build}[Y](u)$ which immediately gives

$$\bigsqcup\{f(x) \mid x \in X\} \sqsubseteq Y, \quad \text{which implies} \quad \forall x \in X.\ f(x) \sqsubseteq Y$$

FOR THE 'ONLY-IF' DIRECTION: $\langle X\rangle\, f\, \langle Y\rangle \Rightarrow \{build[X]\}\, f\, \{build[Y]\}$.

We must show that for all σ, $(build[X]; f)(\sigma) \sqsubseteq build[Y](\sigma)$. If $\sigma \neq u$ then the right-hand side is top. If $\sigma = u$ then the inequation asks for the left-hand side of

$$\bigsqcup\{f(x) \mid x \in X\} \sqsubseteq Y \quad \Longleftrightarrow \quad \forall x \in X.\, f(x) \sqsubseteq Y$$

and the left-hand side is equivalent to the right, which we have assumed to hold (i.e., $\langle X\rangle\, f\, \langle Y\rangle$).

■

This only shows the agreement of algebraic Hoare triples and their state transformer cousins when locality is absent. $build[X]$ here is not local, so we cannot use it to characterise the fault-avoiding triples in the Local Action model. However, the localisation $(-) * \mathsf{skip}$ together with the Exchange Law gives all that we require.

Proposition 4.4.3 (Agreement of Local Action and Algebraic Triples)**.** *For arbitrary Local Action* $f : \Sigma \to P(\Sigma)^\top$

$$\langle X\rangle\, f\, \langle Y\rangle \Longleftrightarrow \{build[X] * \mathsf{skip}\}\, f\, \{build[Y] * \mathsf{skip}\}$$

Proof:

FOR THE 'IF' DIRECTION: $\{build[X] * \mathsf{skip}\}\, f\, \{build[Y] * \mathsf{skip}\} \Rightarrow \langle X\rangle\, f\, \langle Y\rangle$.

We evaluate the pre-condition of the right-hand side at u. Note that

$$\begin{aligned}(build[X] * \mathsf{skip})(\sigma) &= \bigsqcap\{build[X](\sigma_1) * \{\sigma_2\} \mid \sigma_1 \bullet \sigma_2 = \sigma\} \\ &= \bigsqcap\{X * \{\sigma_2\} \mid u \bullet \sigma_2 = \sigma\} \\ &= X * \{\sigma\}\end{aligned}$$

This is easy to calculate at u:

$$(build[X] * \mathsf{skip})(u) = X * \{u\} = X$$

So, evaluated in u, the right-hand side in-equation $(build[X] * \mathsf{skip}); f \sqsubseteq (build[Y] * \mathsf{skip})$ states that

$$\bigsqcup\{f(x) \mid x \in X\} \sqsubseteq Y, \quad \text{which implies that} \quad \forall x \in X.\, f(x) \sqsubseteq Y$$

and thus we obtain $\langle X\rangle\, f\, \langle Y\rangle$.

FOR THE 'ONLY-IF' DIRECTION: $\langle X\rangle\, f\, \langle Y\rangle \Longleftrightarrow \{build[X] * \mathsf{skip}\}\, f\, \{build[Y] * \mathsf{skip}\}$

Assuming $\langle X\rangle\, f\, \langle Y\rangle$, we have by Proposition 4.4.2 that $\{build[X]\}\, f\, \{build[Y]\}$. Using the *Frame* rule we then conclude $\{build[X] * \mathsf{skip}\}\, f\, \{build[Y] * \mathsf{skip}\}$.

■

4.4.2 Plotkin Triples

We now also connect the semantic triple to the Plotkin triple which was considered in section 2.5.3. We remind the reader of the definition:

$$P \rightarrow_C Q \text{ is defined to be } P \sqsupseteq C;Q$$

In terms of Local Action we can probe this idea using a forwards-running notion which we call $do-after$:

$$do-after[X] \; = \; bla[X, true]$$

Here $do-after[X]$ requires X to run, but nothing is known about the end state.

Proposition 4.4.4 (Connecting Plotkin to SL triples)**.**

$$\langle X \rangle \, f \, \langle Y * true \rangle \Longleftrightarrow do-after[X] \rightarrow_f do-after[Y]$$

Proof: Let us first calculate:

$$bla[X, true](\sigma) \quad = \quad \bigsqcap \{ true * \{\sigma_0\} \mid \sigma = \sigma_0 \bullet \sigma_1, \sigma_1 \in X \}$$

This is $\top$ if and only-if $\sigma \notin X * true$.

FOR THE 'IF' DIRECTION: Take $\sigma \in X$, then $(do-after\ X)(\sigma) \neq \top$. If $f(\sigma) \notin Y * true$ then $(f; do-after\ Y)(\sigma) = \top$.

FOR THE 'ONLY-IF' DIRECTION: If $\sigma \notin X * true$ then $(do-after\ X)(\sigma) = \top$. If $\sigma \in X * true$ then for any $\sigma_X \in X$ and $\sigma_H \in H$ such that $\sigma = \sigma_X \bullet \sigma_H$, $f(\sigma) \subseteq f(\sigma_X) * \{\sigma_H\}$. This follows from locality of f and the fact that f satisfy the fault-avoiding tripple, so $f(\sigma) \neq \top$.

Now $f(\sigma_X) \subseteq Y * true$ and for all $\sigma_Y \in Y * true$ we have

$$do-after[Y](\sigma_Y \bullet \sigma_H) \subseteq do-after[Y](\sigma_Y) * \{\sigma_H\}$$

Thus $(f; do-after[Y])(\sigma) \subseteq \bigcup \{ do-after[Y](\sigma_Y) * \{\sigma_H\} \mid \sigma_Y \in f(\sigma_X) \}$.

As $do-after[Y](\sigma_Y) \subseteq true$ we have $do-after[Y](\sigma_Y) * \{\sigma_H\} \subseteq true * \{\sigma_H\}$, and so $(f; do-after[Y])(\sigma) \subseteq true * \{\sigma_H\}$. Since this holds for any splitting of σ we conclude that $(f; do-after[Y])(\sigma) \subseteq do-after[X](\sigma)$.

■

4.4.3 Counterexample to Associativity of $*$

Although the Action model give us some interesting results, we note here again that the Action model is somewhat unpleasant because the parallel operator is not associative. We give a counter example to illustrate this:

Example 4.4.5. *Suppose we have the following functions*

1. $f = \lambda s.if\ s = [1] \mapsto 10\ then\ \{[2] \mapsto 20\}\ else\{s\}$
2. $g = \lambda s.if\ s = [1] \mapsto 10\ then\ \{[3] \mapsto 30\}\ else\{s\}$
3. $h = \lambda s.if\ s = [4] \mapsto 40\ then\ \{[2] \mapsto 20, [3] \mapsto 30\}\ else \top$

Now

$$(f * g)([1] \mapsto 10) = glb\{\{[2] \mapsto 20\}, \{[3] \mapsto 30\}\} = \{\}$$

so

$$((f * g) * h)([1] \mapsto 10 \bullet [4] \mapsto 40) = \{\} * \{[2] \mapsto 20, [3] \mapsto 30\} = \{\}$$

On the other hand we have

$$(g * h)([1] \mapsto 10 \bullet [4] \mapsto 40) = \{[3] \mapsto 30\} * \{[2] \mapsto 20, [3] \mapsto 30\} = \{[2] \mapsto 20 \bullet [3] \mapsto 30\}$$

and

$$(g * h)(u \bullet [4] \mapsto 40) = \{u\} * \{[2] \mapsto 20, [3] \mapsto 30\} = \{[2] \mapsto 20, [3] \mapsto 30\}$$

Hence

$$\begin{aligned} & ((f * (g * h))([1] \mapsto 10 \bullet [4] \mapsto 40) \\ = \ & glb\{\{[3] \mapsto 30\} * \{s[2] \mapsto 20, [3] \mapsto 30\}, \{u\} * \{s[2] \mapsto 20 \bullet [3] \mapsto 30\}\} \\ = \ & glb\{\{[2] \mapsto 20 \bullet s3\}, \{[2] \mapsto 20 \bullet [3] \mapsto 30\}\} \\ = \ & \{[2] \mapsto 20 \bullet [3] \mapsto 30\} \end{aligned}$$

■

The reason why $*$ is not associative is because the *glb* does not distribute over the $*$ of predicates, for *imprecise* assertions. In order for $*$ to distribute over the *glb* we would need to define the $*$ operator in terms of *precise* predicates, as indicated by the counter example above. However, to do this we would need to interpret programs as functions delivering *precise* post-conditions, and this is not feasible as it would mean leaving out a number of programs such as the programs that use non-deterministic choice. So we do not follow this choice of programs as delivering *precise* post-conditions. In the next section we give an alternate model which calculates using joins rather than meets and as a result we do not run into this problem.

4.5 Predicate Transformers: The Resource Model

The Action model has many pleasant properties. Most importantly, there is a Galois connection between the local and non-local elements, and we obtain the Exchange Law. It is a model of CSL but, unfortunately, the model has the unpleasant property of $*$ not being associative. In this section we show how to remedy this situation, to have a model with these good properties of the Action model but not the unpleasant non-associativity.

We do this using predicate transformers rather than state transformers. In a technical sense, we do not run into the problems with associativity because the meaning of $*$ will be calculated using joins rather than meets, and joins distribute over separating conjunction (on predicates) nicely, while meets do not (unless a predicate is *precise*). We call it the Resource Model to be consistent with the terminology in our joint paper [21], where it was called this because it is over a separation algebra which models separation of resources. (Of course, this terminology might equally apply to the Action model, but we trust that no great confusion will arise in this terminology: the important point is just that the state transformer and predicate transformer models are given different names.)

Let $(\Sigma, \bullet, u)$ be a partial, commutative monoid and $(P(\Sigma), *, emp)$ be an ordered total commutative monoid where $*$ and emp are as defined in definition 4.1.2. We are interested in the monotone function space $[P(\Sigma) \to P(\Sigma)]$ and we let $PredTran$ denote the set of monotone predicate transformers. These functions represent (backwards) predicate transformers and have the following operators given in Figure 4.2.

$$
\begin{aligned}
(F_1 * F_2)Y &= \bigcup\{F_1 Y_1 * F_2 Y_2 \mid Y_1 * Y_2 \subseteq Y\} \\
\mathsf{nothing}\, Y &= Y \cap emp \\
(F_1 ; F_2)Y &= F1(F2(Y)) \\
\mathsf{skip}\, Y &= Y
\end{aligned}
$$

Figure 4.2: Resource Model Operators

For the definition of $F_1 * F_2$ we are relying on the context to disambiguate the use of the $*$ operator, where on the left of the definition, it refers to an operator on predicate transformers and on the right of the definition, it refers to an operator on predicates. The definition of the predicate transformer $F_1 * F_2$ follows from the concurrency proof rule of CSL, where starting with a post-condition Y, which is split into separate portions Y_1 and Y_2, the *Concurrency* rule is applied backwards to get a pre-condition $F_1Y_1 * F_2Y_2$ for the parallel composition of F_1 and F_2. Then the union of all such pre-conditions, obtained from all the possible way of splitting Y, is taken to get the weakest possible pre-condition. The intuition here is contrary to that of the Action model, where the conjunction of post-conditions was taken as apposed to the disjunction of pre-conditions. As a result this definition will solve the associative problem, encountered earlier, because of the distribution property

$$p * \bigcup\{A \mid A \in Y\} \;=\; \bigcup\{p * A \mid A \in Y\}$$

This model uses the reverse pointwise ordering $\sqsubseteq$ on Predicate Transformers: $F \sqsubseteq G$ iff $\forall X. FX \supseteq GX$. The function $\lambda X.\Sigma$ serves as the least element, according to the definition, corresponding to the weakest liberal pre-condition transformer for divergence. Note that the Resource Model also has distinct units: `nothing` is the unit of $*$ and `skip` of ;.

Our first task is to verify that the defect in the Action model is not present in the Resource Model.

Lemma 4.5.1. $*$ *is associative.*

Proof:

$$\begin{aligned}
((F*G)*H)X &= \bigcup\{(F*G)X_1 * HX_2 \mid X_1 * X_2 \subseteq X\} \\
&= \bigcup\{\bigcup\{FX_3 * GX_4 \mid X_3 * X_4 \subseteq X_1\} * HX_2 \mid X_1 * X_2 \subseteq X\} \\
&= \bigcup\{FX_3 * GX_4 * HX_2 \mid X_3 * X_4 \subseteq X_1 \wedge X_1 * X_2 \subseteq X\} \\
&= \bigcup\{FX_3 * GX_4 * HX_2 \mid X_3 * X_4 * X_2 \subseteq X\} \\
&= \bigcup\{FX_3 * GX_4 * HX_2 \mid X_4 * X_2 \subseteq X' \wedge X' * X_3 \subseteq X\} \\
&= \bigcup\{FX_3 * \bigcup\{GX_4 * HX_2 \mid X_4 * X_2 \subseteq X'\} \mid X_3 * X' \subseteq X\} \\
&= \bigcup\{FX_3 * (G*H)X' \mid X_3 * X' \subseteq X\} \\
&= (F*(G*H))X
\end{aligned}$$

■

We revisit the Exchange Law example given in Example 4.2.2 in light of the resource model. Our aim in doing this is to show the predicate transformer model at work by using a similar example as before, and also to throw some light on our choice of using the reverse pointwise order for predicate transformers.

Example 4.5.2 (Exchange in the Resource Model)**.**
We reuse the heap model as defined earlier in Example 4.1.9 and define the predicate transformer $([n] := m)X = \bigcup_z \{h[n \mapsto z] \mid h \in X,\ h(n) = m\}$*. We define heap* $h = \{10 \mapsto 88,\ 20 \mapsto 77\}$ *and use the singleton set* $\{h\}$ *as the post-condition for the example.*

$$([10] := 55 * [20] := 66)\,;([20] := 77 * [10] := 88)\{h\}$$
$$\sqsubseteq$$
$$([10] := 55\,;[20] := 77) * ([20] := 66\,;[10] := 88)\{h\}$$

The pre-condition for the lower term is the empty set $\emptyset$*, since there is no way of splitting the heap such that both operands of* $*$ *return a non-empty set. For the upper term we have*

$$([10] := 55 * [20] := 66)\big(([20] := 77 * [10] := 88)\{h\}\big),$$

therefore we can split the heap initially such that $\{10 \mapsto 88\}$ *is sent to* $[10] := 88$ *and* $\{20 \mapsto 77\}$ *is sent to* $[20] := 77$*. This yields* $\bigcup_y\{10 \mapsto y\}$ *and* $\bigcup_z\{20 \mapsto z\}$ *as the pre-condtions respectively. It is easy to see that applying the predicate transformer* $[10] := 55$ *to* $\bigcup_y\{10 \mapsto y\}$ *directly is the same as applying it to* $\{10 \mapsto 55\}$ *and similarly applying* $\bigcup_z\{20 \mapsto z\}$ *to* $[20] := 66$ *directly is the same as applying* $\{20 \mapsto 66\}$ *to it. Therefore, we have* $\bigcup_w\{10 \mapsto w\}$ *and* $\bigcup_x\{20 \mapsto x\}$ *as the pre-conditions for* $[10] := 55$ *and* $[20] := 66$ *respectively. The union of the two pre-conditions is non-empty, hence the pre-condition of the upper term is non-empty. Since the order is reverse inclusion, the treatment of this example is consistent with the exchange law.* ■

Notice how, in the above example, the racing program corresponds to having pre-condition $false$ (the empty set). By using the reverse inclusion order we thus send racing to the top of the lattice; the example shows that we cannot use the pointwise order and get the Exchange Law in this model: the reverse order is in some sense forced by the Exchange Law (in this model).

Proposition 4.5.3. *The Resource Model Satisfies the Exchange Law.*

Proof: Let F_1, F_2, G_2 and $G_2 \in PredTran$ and $X \subseteq \Sigma$. Since the order on predicate transformers is reverse pointwise, we aim to show

$$((F_1\,;F_2) * (G_1\,;G_2))X \subseteq ((F_1 * G_1)\,;(F_2 * G_2))X$$

The left hand side expands to

$$\bigcup\{F_1(F_2X_1) * G_1(G_2X_2) \mid X_1 * X_2 \subseteq X\}$$

Now for every sub-splitting $X_1 * X_2$ of X, we see that $F_2X_1 * G_2X_2$ is a sub-splitting of $(F_2 * G_2)X = \bigcup\{F_2A * G_2B \mid A * B \subseteq X\}$ simply because it is one of the elements of the union. So we can over-approximate the left hand side:

$$\bigcup\{F_1(F_2X_1) * G_1(G_2X_2) \mid X_1 * X_2 \subseteq X\} \subseteq \bigcup\{F_1A * G_1B \mid A * B \subseteq (F_2 * G_2)X\}$$

and this quickly rewrites to the right hand side:

$$\bigcup\{F_1A * G_1B \mid A * B \subseteq (F_2 * G_2)X\} = (F_1 * G_1)((F_2 * G_2)X) = (F_1 * G_1);(F_2 * G_2)X$$

■

The Resource Model is not an example of a CKA, as defined in Section 2.5.1. As mentioned previously, having two different units causes a difference. Additionally, the model does not have a right zero for ;, an element 0 where $p;0 = 0$ for all p (while all constant transformers are left zeros). Furthermore, we do not have that the structure is a *quantale*: a *quantale* requires that $p * (-)$ preserves all *lubs*, for arbitrary p. We give the counter example to this for the model in the next chapter (Example 5.3.5), which is a generalised version of the Resource Model. The Resource Model has the same structure as the Action model with the added property of parallel being associative. In summary the Resource Model provides two ordered monoids $(PredTran, \sqsubseteq, *, \mathsf{nothing})$ and $(PredTran, \sqsubseteq, ;, \mathsf{skip})$ representing parallel and sequential composition linked by the Exchange Law, where $*$, ; are monotone and $*$ is commutative. As before, we have a situation where the full exchange law is valid but the small exchange laws are not. Counterexamples to the small laws similar to the ones in the Action model (Example 4.2.3) can easily be constructed in the Resource model.

4.5.1 Local Elements

We can characterise locality in the Resource Model, just as we did in the Action model.

In the Resource Model, a transformer is called local if it can be described in terms of its action on parts of the state. This can be expressed as follows:

$$FP = \bigcup\{(FX) * R \mid X * R = P\} \tag{4.1}$$

We remark that 4.1 is easily seen to be equivalent to validity of the *Frame* rule, i.e.

$$\forall\, P, Q, R.\ \{P\}\, F\, \{Q\} \Longrightarrow \{P * R\}\, F\, \{Q * R\} \tag{4.2}$$

Before we give the proof for this equivalence, we remark that the previous result of the locality characterisation coincides with the locality characterisation described in 4.1.

Lemma 4.5.4. *A predicate transformer F is local in the sense of (4.1) if and only-if $F = F * \mathsf{skip}$. That is*

$$FP = \bigcup\{(FX) * R \mid X * R = P\} \Longleftrightarrow F = F * \mathsf{skip}$$

Proof:

FOR THE 'IF' DIRECTION: $F = F * \mathsf{skip} \Rightarrow FP = \bigcup\{(FX) * R \mid X * R = P\}$

$$\begin{aligned} FP &= (FP) * emp \\ &\subseteq \bigcup\{(FX) * R \mid X * R = P\} \end{aligned}$$

Choosing $X = P$ and $R = emp$

Conversely,

$$\begin{aligned} FP &= (F * \mathsf{skip})P \\ &= \bigcup\{(FX) * R \mid X * R \subseteq P\} \\ &\supseteq \bigcup\{(FX) * R \mid X * R = P\} \end{aligned}$$

FOR THE 'ONLY-IF' DIRECTION: $FP = \bigcup\{(FX) * R \mid X * R = P\} \Rightarrow F = F * \mathsf{skip}$

$$\begin{aligned} FP &= \bigcup\{(FX) * R \mid X * R = P\} \\ &\subseteq \bigcup\{(FX) * R \mid X * R \subseteq P\} \\ &= (F * \mathsf{skip})P \end{aligned}$$

Conversely, let $P = X * R$ for arbitrary $X,\ R$.

$$\begin{aligned} FP &= \bigcup\{F(X * R) \mid X * R \subseteq P\} \\ &\supseteq \bigcup\{(FX) * R \mid X * R \subseteq P\} \\ &= (F * \mathsf{skip})P \end{aligned}$$

We get the second step since $(FX) * R \subseteq \bigcup\{(FX) * R \mid X * R = P\} = FP = F(X * R)$

■

Lemma 4.5.5. *For either Hoare or Plotking triples, the validity of the Frame rule is equivalent to the locality characterisation* $F = F * \mathsf{skip}$. *That is*

$$F = F * \mathsf{skip} \iff (\forall\, P,Q,R.\ \{P\}\,F\,\{Q\} \Rightarrow \{P * R\}\,F\,\{Q * R\})$$

Proof:

Hoare Triple: $\{P\}\,F\,\{Q\} \Leftrightarrow P;F \sqsubseteq Q$

FOR THE 'IF' DIRECTION:

Since skip is the unit of ; we know that $\{\mathsf{skip}\}\,F\,\{F\}$ holds. By the *Frame* rule we know that $\{\mathsf{skip} * \mathsf{skip}\}\,F\,\{F * \mathsf{skip}\}$ holds. Since $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$ we get $\{\mathsf{skip}\}\,F\,\{F * \mathsf{skip}\}$ and this gives us $F \sqsubseteq F * \mathsf{skip}$. We also know that $F \sqsupseteq F * \mathsf{skip}$ holds for all elements.

FOR THE 'ONLY-IF' DIRECTION:

$$\begin{array}{rcll} P * R;F & = & (P * R);(F * \mathsf{skip}) & \text{locality characterisation} \\ & \sqsubseteq & (P;F) * (R;\mathsf{skip}) & \text{exchange law} \\ & = & (P;F) * R & \mathsf{skip}\text{ unit of ;} \\ & \sqsubseteq & Q * R & \end{array}$$

We get the last step from $\{P\}\,F\,\{Q\}$ and the monotonicity of $*$.

Plotkin Triple: $P \rightarrow_F Q \Leftrightarrow P \sqsupseteq F;Q$

FOR THE 'IF' DIRECTION:

Since skip is the unit of ; we know that $F \rightarrow_F \mathsf{skip}$ holds. By the *Frame* rule we know that $F * \mathsf{skip} \rightarrow_F \mathsf{skip} * \mathsf{skip}$ holds. Since $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$ and skip is the unit of ;, we get $F * \mathsf{skip} \rightarrow_F \mathsf{skip}$ and this gives us $F * \mathsf{skip} \sqsupseteq F$. We also know that $F * \mathsf{skip} \sqsubseteq F$ holds for all elements.

FOR THE 'ONLY-IF' DIRECTION:

$$\begin{array}{rcll} F;Q * R & = & (F * \mathsf{skip});(Q * R) & \text{locality characterisation} \\ & \sqsubseteq & (F;Q) * (\mathsf{skip};R) & \text{Exchange Law} \\ & = & (F;Q) * R & \mathsf{skip}\text{ unit of ;} \\ & \sqsubseteq & P * R & \end{array}$$

We get the last step from $P \rightarrow_F Q$ and the monotonicity of $*$.

■

The proofs for $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$ and $F * \mathsf{skip} \sqsubseteq F$ are given in Chapter 5 (Lemma 5.1.9) where we also relate the semantic and algebraic triples for the Resource model. Both operators also preserve locality and the proof for those are given (Lemma 5.1.8).

4.5.2 Summary

In this chapter we looked at the connection between Locality and the Exchange Law where we first find that the meaning of the *Frame* and the *Concurrency* rules are the same as the standard concrete model of Separation Logic. Secondly, in doing this we found some structure but not all of CKA. In particular we have two monoids on the same structure linked by the Exchange Law. However, in contrast to CKA, these monoids do not have the same unit. Rather the equation $f = f * \mathsf{skip}$ gives us a way to pick out the special local elements. This notion of locality coincides with that in the classic work on SL [9].

We find that the Action model has all the properties mentioned above but has a defect in that the $*$ operator is found not to be associative. To overcome this, we construct an alternative model based on predicate transformers which exhibits all the aforementioned structure including associativity of the $*$ operator.

Chapter 5

A Generalised Model of BCSL

In this chapter we build a model for Basic Concurrent Separation Logic (BCSL), using predicate transformers as in the last chapter but of a more generalised form. We use this generalised form of predicate transformers as it allows us to construct a model of the proof theory of Concurrent Separation Logic (CSL) by starting from an ordered monoid and secondly the generalisation allows us to consider an instantiation of Baby Session Types (BST) by starting with an ordered monoid of typing context.

The purpose of the construction in this chapter is one of generality. In the previous chapter we used the Action and Resource models to connect the algebra to well known and pre-existing specific models of programs, based on state transformers and predicate transformers. The point here is that the construction of the predicate transformer model can be carried out in a much more general setting which allows us to start not from a model of programs, but from the proof theory of program logic, in order to construct the appropriate algebra. Thus, the predicate transformer model we study is a technical device to provide a very general connection between the proof theory of a program logic (BCSL) and the algebraic notions stemming from Concurrent Kleene Algebra. The results suggest a greater range of applicability of the algebra than was previously known, as well as a tighter connection between the algebra and program logic. (Previously, it was known that from the algebra one could derive program logic, but not the converse.)

The first idea is to construct the model such that we get the algebraic structure discovered in Chapter 4. That is that we get two ordered monoids representing the parallel and sequential operators which are linked by the Exchange Law. The second idea is to interpret Hoare triples in

terms of Plotkin triple $F \sqsupseteq C;G$. This fits very well with intuitions concerning Session Types. In a triple $\{\Delta\}\,P\,\{\Delta'\}$, we think of

$$[\![\Delta]\!] \sqsupseteq [\![P]\!];[\![\Delta']\!]$$

as saying that Δ over-approximates what P followed by Δ' might do in the future.

The only point we wish to make is that the alternate triple makes good sense when thinking temporally, about over-approximating the future, as is the case with Session Types. Indeed, the Plotkin triple satisfies all of the rules of Hoare logic, and even BCSL with a further stipulation concerning locality; see Definition 5.1.7 and the discussion thereafter.

In order to verify that the Plotkin triples behave as they should we link them to what we call the Dijkstra triple, which is the standard definition of pre-condition and post-condition specification in terms of predicate transformers. This allows us to then give a soundness and completeness result linking the Dijkstra triple to provability in BCSL. These results give a complete picture linking the model, proof theory and algebra.

We also show that the Resource Model presented in the previous chapter is an instance of the Generalised Predicate Transformer Model here, by describing an isomorphism between the two models. Hence the properties that we conclude for the Generalised Predicate Transformer Model are also applicable to the Resource Model.

Finally, we observe that the set of monotone predicate transformers form a complete lattice, which is sufficient to show that fixed points exist should we wish to consider recursion.

5.1 Generalised Predicate Transformer Model

We now move on to the technical details.

Propositions or Worlds. We suppose first that we are given an ordered total commutative monoid $(Props, \vdash, *, emp)$ in which the order has a least element $\bot$, where $p * \bot = \bot * p = \bot$ for $p \in Props$ and $*$ is monotone with respect to $\vdash$. As the order is a pre-order, there can be many elements equivalent to $\bot$. We define

$$false \;\; = \;\; \{p \mid p \vdash \bot\}$$

In connecting to BCSL the elements of *Props* will be the propositions, but in the model construction they will use the kind of structure found in the possible worlds semantics of Bunched Logic [41, 46].

Predicates. The model is built from predicate transformers on subsets of $Props$. We consider only the down-closed subsets, which corresponds to the idea that pre-conditions are closed under the rule of consequence on the left in Hoare logic. That is, if $F(post) = pre$ then we think of pre as a disjunction, and it does no harm to throw in all the elements that imply anything in pre. We define the downwards closure of a proposition $p \in Props$ as:

$$q\Downarrow = \{p \mid p \vdash q\}$$

This notation extends to sets of propositions in the evident way: $A\Downarrow = \bigcup\{q\Downarrow \mid q \in A\}$. In understanding our use of down-closed sets it can be helpful to consider that, for $A, B \subseteq Props$, then

$$A\Downarrow \subseteq B\Downarrow \iff \forall a \in A. \exists b \in B. a \vdash b$$

where $(-)\Downarrow$ is the downwards closure. Downwards closure allows us to model a natural entailment as subset inclusion.

The downwards-closed sets have logical structure familiar from Kripke's semantics of intuitionistic logic: they form a complete Heyting algebra. In particular, they are closed under arbitrary unions, and this will be used below in the semantics of the parallel operator $\|$. The downwards-closed sets also simultaneously have a monoidal structure as used in (intuitionistic) Bunched Logic, using a construction dating at least back to [31]. (It is worth remarking that monotonicity of $*$ with respect to $\vdash$ is needed to conclude that $\otimes$ is monotone in the subset order and that I is the unit for $\otimes$ in the construction to follow.)

Definition 5.1.1. *Let Preds be the set of non-empty down-closed subsets of Props. That is, a predicate X is a non-empty subset of Props such that $p \in X$ and $q \vdash p$ implies $q \in X$. Preds, ordered by subset inclusion, has an ordered total commutative monoid structure $\otimes$, I (proposition 5.1.3), where*

$$\begin{aligned} X \otimes Y &= \{p \mid p \vdash x * y \wedge x \in X,\ y \in Y\} \\ I &= \{p \mid p \vdash emp\} \end{aligned}$$

Notice that, when instantiated to Session Types, the unit I consists of exactly the *completed* typing contexts plus the *inconsistent* ones. We are requiring predicates to be non-empty because of the presence of the least element $\perp$ amongst the propositions, internalising falsity as the set of *inconsistent* propositions. This is just a representation point. It would be isomorphic to say that a predicate is a (possibly empty) set of consistent propositions, down-closed amongst the *consistent* ones. Below we do a sanity check by proving that $X \otimes false = false$.

Proposition 5.1.2. $X \otimes \mathit{false} = \mathit{false}$.

Proof:

$X \otimes \mathit{false} \subseteq \mathit{false}$**:**

Let $p \in X \otimes \mathit{false}$. Then $p \vdash x * f$ for some $x \in X$, $f \in \mathit{false}$ by the definition of $\otimes$. We also know that $f \vdash \bot$ by the definition of *false*. From the monotonicity of $*$ we then get that $x * f \vdash x * \bot$, and since $x * \bot = \bot$ by our beginning assumptions concerning $\bot$, we obtain $p \vdash \bot$. Therefore $p \in \mathit{false}$ by the definition of *false* as the set $\{p \mid p \vdash \bot\}$.

$\mathit{false} \subseteq X \otimes \mathit{false}$**:**

Let $f \in \mathit{false}$. We know that $\forall X \in \mathit{Preds}.\ \bot \in X$ since $\bot \vdash x$ where $x \in X$. Therefore $f \in X \otimes I$ since $f \vdash \bot$.

■

Proposition 5.1.3 ([31])**.** *Preds, ordered by subset inclusion, has an ordered total commutative monoid structure.*

Proof:

Case `Monotonicity`**:** $X \subseteq X'$ and $Y \subseteq Y' \Rightarrow X \otimes Y \subseteq X' \otimes Y'$.

Let $p \in X \otimes Y$. Then $p \vdash x * y$ for some $x \in X$, $y \in Y$ by the definition of $\otimes$. Since $X \subseteq X'$ and $Y \subseteq Y'$ we know $x \in X'$ and $y \in Y'$. Therefore $x * y \in X' \otimes Y'$. Since $p \vdash x * y$ by downwards closure we get $p \in X' \otimes Y'$.

Case `Commutativity`**:** $\forall X, Y \in \mathit{Preds}.\ X \otimes Y = Y \otimes X$.

$$
\begin{aligned}
X \otimes Y &= \{p \mid p \vdash x * y \wedge x \in X,\ y \in Y\} \\
&= \{p \mid p \vdash y * x \wedge x \in X,\ y \in Y\} \quad \text{commutativity of } * \\
&= Y \otimes X
\end{aligned}
$$

Case `Closure`**:** $\forall X, Y \in \mathit{Preds}.\ X \otimes Y \in \mathit{Preds}$.

$X \otimes Y$ is non-empty since X and Y are non-empty: e.g., given $x \in X$ and $y \in Y$, we know that at least $x * y \in X \otimes Y$. To show downwards closure, suppose $p \vdash x * y$ and that $q \vdash p$. Then, $q \vdash x * y$ by transitivity of $\vdash$, and this establishes that $X \otimes Y$ is downwards closed.

Case `Associativity`**:** $\forall X,Y,Z \in Preds.\ (X \otimes Y) \otimes Z = X \otimes (Y \otimes Z)$.

Let $p \in (X \otimes Y) \otimes Z$ then $p \vdash (x * y) * z$ where $x \in X,\ y \in Y$ and $z \in Z$. By the associativity of $*$ we know that $p \vdash x * (y * z)$ and this gives our desired result.

Case `Identity Element`**:** $I \in Preds.\ \forall X \in Preds.\ X \otimes I = X$.

$X \otimes I \subseteq X$**:**

Let $p \in X \otimes I$. Then $p \vdash x * i$ for some $x \in X$, $i \in I$ by the definition of $\otimes$. We also know that $i \vdash emp$ by the definition of I. From the monotonicity of $*$ we then get that $x * i \vdash x * emp$, and since $x * emp = x$ we obtain $p \vdash x$. Therefore $p \in X$ by downwards closure.

$X \subseteq X \otimes I$**:**

Let $x \in X$ then we know $x * emp \in X$ we also know that $x * emp \vdash x * emp$ and $emp \in I$ therefore $x * emp \in X \otimes I$.

■

Predicate Transformers. We are going to define several operations on the monotone function space $Preds \to Preds$. In each case the input parameter X is an element of $Preds$, i.e., a non-empty down-closed subset of $Props$. The full set of commands are given in Figure 5.1. Parallel composition is the union of all ways of splitting predicate X and handing one part to F and the other to G. It is the same definition as the one given to the parallel composition of the Resource Model. Sequential composition and skip are defined in the usual way for predicate transformers and nothing is the unit of parallel. We will later discuss the role of the function $do - after[Y]$ but for now we note that $Y \in Preds$.

$$
\begin{aligned}
(F_1 \parallel F_2)X &= \bigcup\{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \\
\mathsf{nothing}X &= \texttt{if } X \supseteq I \texttt{ then } I \texttt{ else } \mathit{false} \\
(F_1;F_2)X &= F_1(F_2(X)) \\
\mathsf{skip}X &= X \\
do - after[Y]X &= \texttt{if } X = true(= Props) \texttt{ then } Y \texttt{ else } \mathit{false}
\end{aligned}
$$

Figure 5.1: Predicate Transformer Commands

The ordering $\sqsubseteq$ we use on predicate transformers is the inverse pointwise order:

$$F \sqsubseteq G \iff \forall X. FX \supseteq GX.$$

Defining the order of the predicate transformers in this way allows us to characterise the order in terms of fault avoiding triples for partial correctness in the standard way, where if $\{P\}C\{Q\}$ and $C' \sqsubseteq C$ then $\{P\}C'\{Q\}$. That is we interpret a triple as $P \subseteq wlp(C,Q)$, where $wlp(C,-)$ is a weakest liberal pre-condition predicate transformer, with the additional expectation that the pre-condition P ensures avoidance of memory faults. If the order was subset inclusion then we would get the opposite effect which is not in line with partial correctness. The top element of this predicate transformer lattice is the function $\lambda X.false$ which validates the triple $\{false\}C\{Q\}$ and the bottom element of the lattice is the function $\lambda X.Props$ which validates all Hoare triples. We think of the top and bottom elements as universal faulting and diverging programs, respectively.

We remind the reader about why we are using predicate transformers here: they are a convenient technical device that allows us to give instances of our algebra, which then allows us to link the algebraic triples to triples derived from BCSL proof theory, as will be shown later in the chapter.

5.1.1 Properties of the Model

With this definition of the predicate transformers and the order and in light of the work in the previous chapter we investigate the properties this model has. We check if the model has properties that we expect an adequate model to posses such as the properties listed in the preservation lemma 5.1.8. We start by proving the Exchange Law.

Lemma 5.1.4 (Exchange Law)**.** *The predicate transformers satisfy*

$$(F_1 \parallel F_2);(G_1 \parallel G_2) \sqsubseteq (F_1;G_1) \parallel (F_2;G_2)$$

Proof:

$$\begin{aligned}
& ((F_1 \parallel F_2);(G_1 \parallel G_2))X \\
= \; & \bigcup\{F_1Y_1 \otimes F_2Y_2 \mid Y_1 \otimes Y_2 \subseteq (G_1 \parallel G_2)X\} \\
= \; & \bigcup\{F_1Y_1 \otimes F_2Y_2 \mid Y_1 \otimes Y_2 \subseteq \bigcup\{G_1X_1 \otimes G_2X_2 \mid X_1 \otimes X_2 \subseteq X\}\} \\
\supseteq \; & \bigcup\{F_1(G_1X_1) \otimes F_2(G_2X_2) \mid X_1 \otimes X_2 \subseteq X\} \quad \texttt{Where } Y_1 = G_1X_1 \texttt{and } Y_2 = G_2X_2 \\
= \; & \bigcup\{(F_1;G_1)X_1 \otimes (F_2;G_2)X_2 \mid X_1 \otimes X_2 \subseteq X\} \\
= \; & ((F_1;G_1) \parallel (F_2;G_2))X
\end{aligned}$$

We are allowed to make the $\supseteq$ step because

$$X_1 \otimes X_2 \subseteq X \Rightarrow G_1X_1 \otimes G_2X_2 \subseteq \bigcup\{G_1X_1 \otimes G_2X_2 \mid X_1 \otimes X_2 \subseteq X\}$$

■

In light of our experience in the Action model of the previous chapter, it is important to check that associativity holds in this model.

Lemma 5.1.5. $\|$ *is associative.*

Proof:

$$
\begin{aligned}
((F \parallel G) \parallel H)X &= \bigcup\{(F \parallel G)X_1 \otimes HX_2 \mid X_1 \otimes X_2 \subseteq X\} \\
&= \bigcup\{\bigcup\{FX_3 \otimes GX_4 \mid X_3 \otimes X_4 \subseteq X_1\} \otimes HX_2 \mid X_1 \otimes X_2 \subseteq X\} \\
&= \bigcup\{FX_3 \otimes GX_4 \otimes HX_2 \mid X_3 \otimes X_4 \subseteq X_1 \wedge X_1 \otimes X_2 \subseteq X\} \\
&= \bigcup\{FX_3 \otimes GX_4 \otimes HX_2 \mid X_3 \otimes X_4 \otimes X_2 \subseteq X\} \\
&= \bigcup\{FX_3 \otimes GX_4 \otimes HX_2 \mid X_4 \otimes X_2 \subseteq X' \wedge X' \otimes X_3 \subseteq X\} \\
&= \bigcup\{FX_3 \otimes \bigcup\{GX_4 \otimes HX_2 \mid X_4 \otimes X_2 \subseteq X'\} \mid X_3 \otimes X' \subseteq X\} \\
&= \bigcup\{FX_3 \otimes (G \parallel H)X' \mid X_3 \otimes X' \subseteq X\} \\
&= (F \parallel (G \parallel H))X
\end{aligned}
$$

■

We also find in this model that skip is not the unit of $\|$ for all monotone predicate transformers: rather, the unit is nothing.

Proposition 5.1.6. nothing *is the unit of* $\|$ *for monotone predicate transformers.*

Proof: We simply calculate:

$$
\begin{aligned}
(F \parallel \mathsf{nothing})X &= \bigcup\{FX_1 \otimes \mathsf{nothing}X_2 \mid X_1 \otimes X_2 \subseteq X\} \\
&= \bigcup\{FX_1 \otimes I \mid X_1 \otimes X_2 \subseteq X \wedge X_2 \supseteq I\} \\
&= \bigcup\{FX_1 \mid X_1 \otimes X_2 \subseteq X \wedge X_2 \supseteq I\} \\
&= \bigcup\{FX_1 \mid X_1 \subseteq X\} \\
&= FX
\end{aligned}
$$

The second last step follows from the fact that $X_1 \supseteq X_1$ and by the monotonicity of $\otimes$ we know that $X_1 \otimes X_2 \supseteq X_1 \otimes I = X_1$ since $X_2 \supseteq I$. Since $X_1 \otimes X_2 \subseteq X$ we know that $X_1 \subseteq X_1 \otimes X_2 \subseteq X$. Conversely, if $X_1 \subseteq X$, we can always pick $X_2 = I$ to obtain $X_1 \otimes X_2 \subseteq X \wedge X_2 \supseteq I$.

For the last step we know from the monotonicity of F that $\forall X_1 \subseteq X.\, FX_1 \subseteq FX$ hence $\bigcup\{FX_1 \mid X_1 \subseteq X\} \subseteq FX$ furthermore we know that $FX \in \bigcup\{FX_1 \mid X_1 \subseteq X\}$.

■

While skip is not the unit of $\|$ in general, those elements that do have skip as a unit are also special in that they provide a neat characterisation of locality.

Definition 5.1.7. *A predicate transformer F is local if $F = F \parallel \mathsf{skip}$.*

The idea to define locality in this way comes from our work in Chapter 4. Previously, before the algebraic structure was considered, locality had been defined in a more elaborate manner [9].

We will not restrict attention to local transformers only. In particular, the transformer $do-after[Y]$ used to interpret pre-conditions and post-conditions below is not local. It is not local since $(do-after[Y] \parallel \mathsf{skip})true$ will be $\bigcup\{Y \otimes X_2 \mid true \otimes X_2 \subseteq true\}$, whereas if it was local we would require the result to be Y and it evidently is not.

However, while our model contains non-local elements, it will be important that all programs are local, and for this, the following lemma is essential. From this lemma we can conclude that if a program is built using ; and $\|$ from primitive commands that are monotone and local, then the program itself is monotone and local as well.

Lemma 5.1.8 (Preservation Lemma)**.**

1. skip *is local, and if F and G are local and monotone then $F;G$ and $F \parallel G$ are local.*

2. skip *and* nothing *are monotone, and if F and G are monotone (with respect to $\subseteq$) then so are $F;G$ and $F \parallel G$. (That is, if $X \subseteq Y$ then $(F;G)X \subseteq (F;G)Y$ and $(F \parallel G)X \subseteq (F \parallel G)Y$)*

3. $\|$ *and* ; *are monotone (with respect to $\sqsubseteq$): if $F_1 \sqsubseteq G_1$ and $F_2 \sqsubseteq G_2$ then $F_1 \parallel F_2 \sqsubseteq G_1 \parallel G_2$ and $F_1;F_2 \sqsubseteq G_1;G_2$.*

Proof: PROOF OF 5.1.8 - 1

Case skip `is local`**:**

$$\begin{aligned}(\mathsf{skip} \parallel \mathsf{skip})X &= \bigcup\{\mathsf{skip}X_1 \otimes \mathsf{skip}X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &= \bigcup\{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}\end{aligned}$$

Since X is an upper bound of $\{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$ then by definition $\bigcup\{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \subseteq X$. For the other direction we pick $X_1 = X$ and $X_2 = I$. Then we get that $X \otimes I \in \{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$, hence $X \subseteq \bigcup\{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$. So we conclude that $(\mathsf{skip} \parallel \mathsf{skip})X = X = \mathsf{skip}X$.

Case: $\|$ `preserves locality`**:**

$$
\begin{array}{rcll}
(F \parallel G) & = & (F \parallel \mathsf{skip}) \parallel (G \parallel \mathsf{skip}) & \\
 & = & ((F \parallel \mathsf{skip}) \parallel G) \parallel \mathsf{skip} & \texttt{associativity of } \parallel \\
 & = & (F \parallel (\mathsf{skip} \parallel G)) \parallel \mathsf{skip} & \texttt{associativity of } \parallel \\
 & = & (F \parallel (G \parallel \mathsf{skip})) \parallel \mathsf{skip} & \texttt{commutativity of } \parallel \\
 & = & ((F \parallel G) \parallel \mathsf{skip}) \parallel \mathsf{skip} & \texttt{associativity of } \parallel \\
 & = & (F \parallel G) \parallel (\mathsf{skip} \parallel \mathsf{skip}) & \texttt{associativity of } \parallel \\
 & = & (F \parallel G) \parallel \mathsf{skip} & \texttt{locality of } \mathsf{skip}
\end{array}
$$

Case: ; `preserves locality`**:** We first show $(F;G) \parallel \mathsf{skip} \sqsubseteq F;G$ as follows.

$$
\begin{array}{rcl}
((F;G) \parallel \mathsf{skip})X & = & \bigcup\{(F;G)X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \\
 & \supseteq & \bigcup\{(F;G)X \otimes I \mid X \otimes I \subseteq X\} \\
 & = & (F;G)X
\end{array}
$$

In this proof, we pick $X_1 = X$ and $X_2 = I$ at the appropriate point to give us $\{(F;G)X \otimes I \mid X \otimes I \subseteq X\} \subseteq \{(F;G)X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$, and so $\bigcup\{(F;G)X \otimes I \mid X \otimes I \subseteq X\} \subseteq \bigcup\{(F;G)X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\}$. Since $(F;G)X \otimes I = (F;G)X$ we know that $(F;G)X = \bigcup\{(F;G)X \otimes I \mid X \otimes I \subseteq X\}$.

Conversely, we show $(F;G) \parallel \mathsf{skip} \sqsupseteq F;G$, with the following calculation.

$$
\begin{array}{rcll}
(F;G) \parallel \mathsf{skip} & = & (F;G) \parallel (\mathsf{skip};\mathsf{skip}) & \mathsf{skip}\ \texttt{unit of ;} \\
 & \sqsupseteq & (F \parallel \mathsf{skip});(G \parallel \mathsf{skip}) & \texttt{Exchange Law} \\
 & = & F;G & \texttt{locality of } F,\ G
\end{array}
$$

PROOF OF 5.1.8 - 2

Suppose $X \subseteq Y$.

Case: skip `is monotone`**:** By definition of skip, skip $X = X \subseteq Y =$ skip Y.

Case: nothing `is monotone`**:** If nothing$X = \mathit{false}$ then we know $\mathit{false} \subseteq$ nothing$Y$. If nothing$X = I$ then we know that $X \supseteq I$ and since $X \subseteq Y$ we know that $Y \supseteq I$ and therefore by the definition of nothing we know that nothing$Y = I$.

Case: $\|$ `preserves monotonicity`**:**

$$\begin{aligned}(F \parallel G)X &= \bigcup\{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &\subseteq \bigcup\{FY_1 \otimes GY_2 \mid Y_1 \otimes Y_2 \subseteq Y\} \\ &= (F \parallel G)Y\end{aligned}$$

For any X_1, X_2. if $X_1 \otimes X_2 \subseteq X$ then $X_1 \otimes X_2 \subseteq Y$ because $X \subseteq Y$ and because $\subseteq$ is transitive. It follows that $\{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \subseteq \{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq Y\}$ and so by definition $\bigcup\{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} \subseteq \bigcup\{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq Y\}$.

Case: ; `preserves monotonicity`**:**

$$(F;G)X = F(GX)$$

Since G is monotone and $X \subseteq Y$ then we know that $GX \subseteq GY$. Since F is monotone then we know that $F(GX) \subseteq F(GY)$, thus we get our desired result $(F;G)X \subseteq (F;G)Y$.

PROOF OF 5.1.8 - 3

Suppose $F_1 \sqsubseteq G_1$ and $F_2 \sqsubseteq G_2$

Case: $\|$ `is monotone`**:**

$$\begin{aligned}(F_1 \parallel F_2)X &= \bigcup\{F_1X_1 \otimes F_2X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ (G_1 \parallel G_2)X &= \bigcup\{G_1X_1 \otimes G_2X_2 \mid X_1 \otimes X_2 \subseteq X\}\end{aligned}$$

By the monotonicity of $\otimes$ and the assumptions, we know that $F_1X_1 \otimes F_2X_2 \sqsubseteq G_1X_1 \otimes G_2X_2$. By definition we know that

$$\bigcup\{F_1X_1 \otimes F_2X_2 \mid X_1 \otimes X_2 \subseteq X\} \sqsubseteq \bigcup\{G_1X_1 \otimes G_2X_2 \mid X_1 \otimes X_2 \subseteq X\}$$

Case: ; `is monotone`**:**

$$\begin{aligned}(F_1;F_2)X &= F_1(F_2X) \\ (G_1;G_2)X &= G_1(G_2X)\end{aligned}$$

Since $F_2 \sqsubseteq G_2$ we know that $F_2X \supseteq G_2X$ and since $F_1 \sqsubseteq G_1$ we know that $F_1(F_2X) \supseteq G_1(G_2X)$ which by definition is $(F_1;F_2)X \sqsubseteq (G_1;G_2)X$.

■

If one has the Exchange Law and F is local and monotone, then the $Frame$ rule

$$\frac{P \sqsupseteq F;Q}{(P \parallel R) \sqsupseteq F;(Q \parallel R)}$$

holds for all P, Q and R. Note that the *Frame* rule needs only F to be local, not P, Q or R.

Next, we have an analogue of the result from the previous chapter on the structure of localisation.

Proposition 5.1.9 (Structure of Localisation)**.**

1. $(\cdot) \parallel \mathsf{skip}$ *is monotone and idempotent, with unit* skip*. In more detail, for arbitrary* $F, G : Preds \to Preds$
   (a) $F \sqsubseteq G \Rightarrow F \parallel \mathsf{skip} \sqsubseteq G \parallel \mathsf{skip}$,
   (b) $\mathsf{skip} \parallel \mathsf{skip} = \mathsf{skip}$.
2. $(F \parallel \mathsf{skip}) \parallel \mathsf{skip} = F \parallel \mathsf{skip}$,
3. $F \parallel \mathsf{skip}$ *is the largest monotone local predicate transformer smaller than* F*. In more detail, for arbitrary* $F, G : Preds \to Preds$
   (a) $F \parallel \mathsf{skip} \sqsubseteq F$
   (b) *if* $G \sqsubseteq F$ *and* G *is local then* $G \sqsubseteq F \parallel \mathsf{skip}$.
4. $(\cdot) \parallel \mathsf{skip}$ *is the right adjoint part of a Galois connection between monotone predicate transformers and local monotone predicate transformers satisfying the locality condition* $F = F \parallel \mathsf{skip}$*. In more detail, if* $F, G : Preds \to Preds$ *and* G *is local, then*

$$G \sqsubseteq F \qquad \Longleftrightarrow \qquad G \sqsubseteq F \parallel \mathsf{skip}$$

Proof: [of Proposition 5.1.9]

PROOF OF 5.1.9 - 1

(a) $F \sqsubseteq G \Rightarrow F \parallel \mathsf{skip} \sqsubseteq G \parallel \mathsf{skip}$

$$\begin{array}{ll} & (F \parallel \mathsf{skip})X \\ = & \bigcup\{FX_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ \sqsubseteq & \bigcup\{GX_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \quad (F \sqsubseteq G \text{ and monotonicity of } \otimes \text{ on } Preds) \\ = & (G \parallel \mathsf{skip})X \end{array}$$

(b) $\mathsf{skip} \parallel \mathsf{skip} = \mathsf{skip}$

$$\begin{aligned}(\mathsf{skip} \parallel \mathsf{skip})X &= \bigcup\{X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &= X \\ &= \mathsf{skip}X\end{aligned}$$

PROOF OF 5.1.9 - 2

$$\begin{aligned}((F \parallel \mathsf{skip}) \parallel \mathsf{skip})X &= \bigcup\{(F \parallel \mathsf{skip})X_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &= \bigcup\{\bigcup\{FX_3 \otimes X_4 \mid X_3 \otimes X_4 \subseteq X_1\} \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &= \bigcup\{\bigcup\{FX_3 \otimes X_4 \otimes X_2 \mid X_3 \otimes X_4 \subseteq X_1\} \mid X_1 \otimes X_2 \subseteq X\} \\ &= \bigcup\{FX_3 \otimes X_4 \otimes X_2 \mid X_3 \otimes X_4 \otimes X_2 \subseteq X\} \\ &= \bigcup\{FX_3 \otimes X_4 \otimes X_2 \mid X_3 \otimes X' \subseteq X \wedge X_4 \otimes X_2 \subseteq X'\} \\ &= \bigcup\{FX_3 \otimes X' \mid X_3 \otimes X' \subseteq X\} \\ &= (F \parallel \mathsf{skip})X\end{aligned}$$

PROOF OF 5.1.9 - 3

There are two things to show:

(a) $F \parallel \mathsf{skip} \sqsubseteq F$:

$$\begin{aligned}(F \parallel \mathsf{skip})X &= \bigcup\{FX_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &\sqsubseteq \bigcup\{FX \otimes I \mid X \otimes I \subseteq X\} && \text{(taking particular } X_1, X_2\text{)} \\ &= FX\end{aligned}$$

(b) if $G \sqsubseteq F$ and G is local then $G \sqsubseteq F \parallel \mathsf{skip}$:

$$\begin{aligned}(F \parallel \mathsf{skip})X &= \bigcup\{FX_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} \\ &\sqsupseteq \bigcup\{GX_1 \otimes X_2 \mid X_1 \otimes X_2 \subseteq X\} && (G \sqsubseteq F \text{ and monotonicity of } \otimes) \\ &= (G \parallel \mathsf{skip})X \\ &= GX && \text{(by locality of } G\text{)}\end{aligned}$$

PROOF OF 5.1.9 - 4

The direction $G \sqsubseteq F \parallel \mathsf{skip} \Rightarrow G \sqsubseteq F$ follows directly from the fact that $F \parallel \mathsf{skip} \sqsubseteq F$ and $\sqsubseteq$ is transitive. The direction $G \sqsubseteq F \Rightarrow G \sqsubseteq F \parallel \mathsf{skip}$ holds since $F \parallel \mathsf{skip}$ is the largest local predicate transformer smaller than F and G is local predicate transformer.

■

5.1.2 Dijkstra's Healthiness Condition

Historically, when Dijkstra introduced predicate transformers, Dijkstra described certain healthiness (well-formedness) conditions on them [11], intending to rule out certain transformers with properties that did not match his computational intuition. Although they are not needed for our technical results, it is worth remarking on these historical properties in our model. (This subsection can be skipped by the reader without loss of continuity.)

The healthiness conditions that Dijkstra imposes on predicate transformers are: *monotonicity*, *feasibility*, *disjunctivity* and *conjunctivity*. We have already given the proof for it monotonicity and as for *feasibility* i.e the Law of Excluded Miracle ($F(\mathit{false}) = \mathit{false}$), then this does not apply to our model as we are using a partial correctness (weakest liberal pre-conditions) rather than total correctness (which is what Dijkstra was considering when insisting on these laws).

Since we are working with partial correctness we know that for a diverging program F the triple $\{-\}\,F\,\{\mathit{false}\}$ holds. To remind the reader this is because, in traditional partial correctness, $\{P\}\,C\,\{Q\}$ means that for all states s satisfying P, if program C terminates starting in state s and delivers output state s', then s' satisfies Q. A program that delivers no output states satisfies partial correctness triples trivially. In terms of predicate transformers, this means that, if F is a diverging program, we would expect that $F(\mathit{false}) = true$: anything is a good pre-condition for the universally diverging program (i.e.,'while *true* do skip'). Hence, we do not expect $F(\mathit{false}) = \mathit{false}$, and so we do not require the Law of Excluded Miracle.

We find that ; preserves $conjunctivity$ and $disjunctivity$ but $\|$ only preserves $disjunctivity$ not $conjunctivity$ as illustrated by the counter example given in Example 5.1.16.

Definition 5.1.10 (Disjunctive Predicate Transformer)**.** *If F is a disjunctive predicate transformer and X,Y are predicates then $F(X \cup Y) = FX \cup FY$*

Definition 5.1.11 (Conjunctive Predicate Transformer)**.** *If F is a conjunctive predicate transformer and X,Y are predicates then $F(X \cap Y) = FX \cap FY$*

We will use the following lemma to prove lemma 5.1.13 which in turn will be used in the proof of the parallel case for $disjunctivity$, lemma 5.1.14.

Lemma 5.1.12. $\forall\, Y \in Preds, \forall\, P \subseteq Preds.\ \bigcup\{X \mid X \in P\} \otimes Y = \bigcup\{X \otimes Y \mid X \in P\}$.

Proof:

Case $\forall\, Y \in Preds, \forall\, P \subseteq Preds.\ \bigcup\{X \mid X \in P\} \otimes Y \subseteq \bigcup\{X \otimes Y \mid X \in P\}$**:**

Suppose $p \in \bigcup\{X \mid X \in P\} \otimes Y$. Then $p \vdash x * y$ for some $x \in \bigcup\{X \mid X \in P\}$, $y \in Y$ by the definition of $\otimes$. Since $x \in \bigcup\{X \mid X \in P\}$ we know that $x \in X$. Therefore we know that $p \in X \otimes Y$ for some $X \in P$. Fixing such an X we know that $X \otimes Y \subseteq \bigcup\{X \otimes Y \mid X \in P\}$, and therefore that $p \in \bigcup\{X \otimes Y \mid X \in P\}$.

Case $\forall\, Y \in Preds, \forall\, P \subseteq Preds.\ \bigcup\{X \mid X \in P\} \otimes Y \supseteq \bigcup\{X \otimes Y \mid X \in P\}$**:**

Suppose $p \in \bigcup\{X \otimes Y \mid X \in P\}$. Then we know that $p \in X \otimes Y$ for some $X \in P$. Fixing such an X we know that $p \vdash x * y$ for some $x \in X$, $y \in Y$ by the definition of $\otimes$. Since $X \subseteq \bigcup\{X \mid X \in P\}$ we know that $x \in \bigcup\{X \mid X \in P\}$. Therefore $p \in \bigcup\{X \mid X \in P\} \otimes Y$.

■

Lemma 5.1.13. *Let F, G be disjunctive predicate transformers and $X \in Preds$ then* $(F \parallel G)X = \bigcup\{F(\{x_1\}) \otimes G(\{x_2\}) \mid x_1 * x_2 \in X\}$

Proof:

We will use the fact that a predicate X can be written as $\bigcup\{\{x\} \mid x \in X\}$ and therefore when F is disjunctive, we have $F(X) = F(\bigcup\{\{x\} \mid x \in X\}) = \bigcup\{F(\{x\}) \mid x \in X\}$.

$$
\begin{array}{lll}
 & (F \parallel G)X & \\
= & \bigcup\{FX_1 \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} & \\
= & \bigcup\{\bigcup\{F(\{x_1\}) \mid x_1 \in X_1\} \otimes GX_2 \mid X_1 \otimes X_2 \subseteq X\} & \\
= & \bigcup\{\bigcup\{F(\{x_1\}) \otimes GX_2 \mid x_1 \in X_1\} \mid X_1 \otimes X_2 \subseteq X\} & \text{lemma 5.1.12} \\
= & \bigcup\{\bigcup\{F(\{x_1\}) \otimes \bigcup\{G(\{x_2\}) \mid x_2 \in X_2\} \mid x_1 \in X_1\} \mid X_1 \otimes X_2 \subseteq X\} & \\
= & \bigcup\{\bigcup\{\bigcup\{F(\{x_1\}) \otimes G(\{x_2\}) \mid x_2 \in X_2\} \mid x_1 \in X_1\} \mid X_1 \otimes X_2 \subseteq X\} & \text{lemma 5.1.12} \\
= & \bigcup\{\bigcup\{F(\{x_1\}) \otimes G(\{x_2\}) \mid x_1 \in X_1, x_2 \in X_2\} \mid X_1 \otimes X_2 \subseteq X\} & \\
= & \bigcup\{F(\{x_1\}) \otimes G(\{x_2\}) \mid x_1 * x_2 \in X\} &
\end{array}
$$

■

Lemma 5.1.14. *If F and G are disjunctive then so are $F;G$ and $F \parallel G$.*

Proof:

Case ; `is disjunctive`**:** We need to show $(F;G)(X \cup Y) = (F;G)X \cup (F;G)Y$

$$\begin{array}{rcll} (F;G)(X \cup Y) & = & F(G(X \cup Y)) & \\ & = & F(GX \cup GY) & \text{disjunctivity of G} \\ & = & F(GX) \cup F(GY) & \text{disjunctivity of F} \\ & = & (F;G)X \cup (F;G)Y & \end{array}$$

Case $\parallel$ `is disjunctive`**:** We need to show $(F \parallel G)(X \cup Y) = (F \parallel G)X \cup (F \parallel G)Y$.

$$\begin{array}{cll} & (F \parallel G)(X \cup Y) & \\ = & \bigcup\{F(\{z_1\}) \otimes G(\{z_2\}) \mid z_1 * z_2 \in X \cup Y\} & \text{lemma 5.1.13} \\ = & \bigcup\{F(\{z_1\}) \otimes G(\{z_2\}) \mid z_1 * z_2 \in X\} & \\ & \cup & \\ & \bigcup\{F(\{z_1\}) \otimes G(\{z_2\}) \mid z_1 * z_2 \in Y\} & \\ = & (F \parallel G)X \cup (F \parallel G)Y & \text{lemma 5.1.13} \end{array}$$

■

Lemma 5.1.15. *If F and G are conjunctive then so is $F;G$.*

Proof: We need to show $(F;G)(X \cap Y) = (F;G)X \cap (F;G)Y$

$$\begin{array}{rcll} (F;G)(X \cap Y) & = & F(G(X \cap Y)) & \\ & = & F(GX \cap GY) & \text{conjunctivity of G} \\ & = & F(GX) \cap F(GY) & \text{conjunctivity of F} \\ & = & (F;G)X \cap (F;G)Y & \end{array}$$

■

Example 5.1.16. $\|$ *does not preserve conjunctivity: Consider the natural numbers* $\mathbb{N}$ *with the following ordering:*

$$x \sqsubseteq y \Leftrightarrow x = 0$$

then 0 *is the least element and* $+$ *is monotone. We then take as the separation algebra* $(\mathbb{N}, \sqsubseteq, +, 0)$*. Then the predicates are just subsets of natural numbers that contain* 0*. Now consider* $f,\ g : \mathbb{N} \mapsto \mathbb{N}$ *given by*

$$\begin{aligned} fx &= x \\ gx &= 2x \end{aligned}$$

These can be lifted to $F, G : Preds \mapsto Preds$ *and since* f *and* g *are injective, their lifting,* F *and* G *are conjunctive and disjunctive. Now choose predicates* X *and* Y *in the following way:*

$$\begin{aligned} X &= \{0, 1\} \\ Y &= \{0, 2\} \end{aligned}$$

Then

$$\begin{aligned} LHS &= (F \parallel G)(X \cap Y) = \{0\} \\ RHS &= (F \parallel G)(X) \cap (F \parallel G)(Y) = \{0, 2\} \end{aligned}$$

■

The fact that the parallel operator does not preserve *conjunctivity* means that the *Conjunction* rule is not sound. Since BCSL does not have a *Conjunction* rule

$$\frac{\{P_1\}C\{Q_1\} \quad \{P_2\}C\{Q_2\}}{\{P_1 \wedge P_2\}C\{Q_1 \wedge Q_2\}}$$

this result does not really affect our model. However, it is worth noting that it is not a serious technical problem if the *Conjunction* rule is unsound because in [40] it has been shown that soundness of the logic can be achieved through the concept of precision of certain formulae (resource invariants) which we do not consider here because BCSL does not have locks or critical regions. In fact more recently soundness has been shown without the *Conjunction* rule [14].

5.1.3 Completeness Results: Connecting Various Triples

Next we prove our core completeness results concerning various triples. Our first result, Theorem 5.1.17, connects the algebraic Plotkin triple to the standard interpretation of triples stemming from Dijkstra's view of predicate transformers. Our second result, Theorem 5.1.18, connects the proof theory of BCSL with either of these semantic triples. This, then, ties the proof theory and the model together.

The standard predicate transformer view of triples is that $\{Y\}F\{Z\}$ means $Y \subseteq FZ$ (let us call the latter the 'Dijkstra triple'). We can connect this to Plotkin triples, providing a measure of justification for the latter, by using the construct $do-after[\cdot]$ which converts a predicate into a predicate transformer. $do-after[Y]$ is the greatest predicate transformer corresponding to the pre-condition and post-condition specification $\{Y\} - \{true\}$, and the Plotkin triple:

$$do-after[X] \sqsupseteq F; do-after[Y]$$

is intended to provide a model of the pre-condition and post-condition specification $\{X\}F\{Y\}$ which says that the future of X over-approximates F followed by the future described by the post-condition.

Theorem 5.1.17 (Connecting Dijkstra and Plotkin Triples)**.** *For all $Y,Z \in Preds$ and monotone $F : Preds \to Preds$,*

$$Y \subseteq FZ \Leftrightarrow do-after[Y] \sqsupseteq F; do-after[Z]$$

Proof:

FOR THE 'IF' DIRECTION: $do-after[Y] \sqsupseteq F; do-after[Z] \Rightarrow Y \subseteq FZ$

We take X to be $true$, then we show $Y \subseteq FZ$. By the definition of $do-after$ and ; we know that $Y \subseteq F(Z)$ therefore we have our desired result.

FOR THE 'ONLY-IF' DIRECTION: $Y \subseteq FZ \Rightarrow do-after[Y] \sqsupseteq F; do-after[Z]$

We do this in two cases:

Case $X \neq true$**:** We must show $do-after[Y]X \subseteq (F; do-after[Z])X$. By the definition of $do-after$ we know that $do-after[Y]X = false \subseteq (F; do-after[Z])X$.

Case $X = true$**:** We must show $do-after[Y]X \subseteq (F; do-after[Z])X$. By the definition of $do-after$ we know that $do-after[Y]X = Y$ and $do-after[Z]X = Z$. By the definition of ; we know that $(F; do-after[Z])X = F(Z)$. We have our desired result since $Y \subseteq FZ$.

■

For this result we did not mention locality at all. Suppose we were interested in an algebra containing only local transformers, including the pre-condition and post-condition specifications. Could we similarly characterise pre-condition and post-condition specifications using Plotkin triples?

We can almost get there using the following calculation, using $do-after[-] \parallel \texttt{skip}$ to create a local predicate transformer from a predicate.

$$\begin{array}{lll} & (do-after[Y] \parallel \mathsf{skip}) \sqsupseteq F; (do-after[Z] \parallel \mathsf{skip}) & \\ \text{iff} & (do-after[Y] \parallel \mathsf{skip}) \sqsupseteq (F \parallel \mathsf{skip}); (do-after[Z] \parallel \mathsf{skip}) & \text{by Locality} \\ \text{iff} & (do-after[Y] \parallel \mathsf{skip}) \sqsupseteq (F; do-after[Z]) \parallel (\mathsf{skip};\mathsf{skip}) & \text{by Exchange} \\ \text{iff} & (do-after[Y] \parallel \mathsf{skip}) \sqsupseteq (F; do-after[Z]) \parallel \mathsf{skip} & \text{by Unity} \end{array}$$

Then, if $Y \subseteq FZ$ we obtain $(do-after[Y] \parallel \mathsf{skip}) \sqsupseteq F; (do-after[Z] \parallel \mathsf{skip})$ from the monotonicity of $(\cdot) \parallel \mathsf{skip}$ and Theorem 5.1.17.

For the reverse direction, though, we cannot use this same sort of reasoning, because while $(-) \parallel \mathsf{skip}$ preserves order, it does not reflect it. That is, $F \parallel \mathsf{skip} \sqsupseteq G \parallel \mathsf{skip}$ does not imply $F \sqsupseteq G$ (for at least one of F, G non-local). Currently, we are unsure whether we can characterise 'Dijkstra triples' $Y \subseteq FZ$ in terms of Plotkin triples, when all transformers are required to be local. But, a characterisation is easily at hand if we drop the assumption of locality (for the pre-conditions and post-conditions, at any rate).

Thus, our use of the non-local $do-after[Y]$ in the characterisation of Dijkstra in terms of Plotkin might be read as a (weak) suggestion that locality $F = F \parallel \mathsf{skip}$ should perhaps not be taken as an absolute requirement in the development of algebraic structure linking $\parallel$ and ;.

The previous result, Theorem 5.1.17, on Plotkin and Dijkstra triples provides a kind of sanity check on the former, showing that they connect to the standard interpretation of pre-condition and post-condition specifications for predicate transformers. Our final result completes the picture, by linking the truth of the Dijkstra (hence, Plotkin) triple back to provability in BCSL.

Theorem 5.1.18 (Soundness and Completeness of Model)**.** *Let $[\![c]\!]$ be the predicate transformer defined by c according to the above. For primitive commands c_{prim} we assume that the* local *predicate transformer $[\![c_{prim}]\!]$ is given, and that the property "$\exists q \in X.\{p\}\, c_{prim}\, \{q\}$ is derivable in BCSL if and only-if $p \in [\![c_{prim}]\!]X$" holds for these primitive statements:*

$$p \in [\![c]\!]X \iff \exists q \in X.\ \{p\}\, c\, \{q\}$$

For the completeness direction we will use the following lemma in the proofs of the *Par* and *Frame* cases.

Lemma 5.1.19. *If X is a predicate and $q_1 * q_2 \in X$ and $X_1 = q_1 \Downarrow$, $X_2 = q_2 \Downarrow$ then $X_1 \otimes X_2 \subseteq X$.*

Proof:

Suppose $t \in X_1 \otimes X_2$; then by definition of $\otimes$ we know that $t \vdash x_1 * x_2$ for some $x_1 \in X_1$ and $x_2 \in X_2$. We also know that $x_1 \vdash q_1$ and $x_2 \vdash q_2$. By the monotonicity of $*$ it follows that $x_1 * x_2 \vdash q_1 * q_2$, and since $q_1 * q_2 \in X$ we know that $x_1 * x_2 \in X$ because X is downwards closed, hence we also know that $t \in X$; thus we conclude that $X_1 \otimes X_2 \subseteq X$.

■

Proof:

PROOF OF SOUNDNESS: $p \in [\![c]\!]X \Rightarrow \exists q \in X.\ \{p\}\, c\, \{q\}$. The proof is by induction on the structure of c, using the clauses in the definition of $[\![c]\!]$.

Case $\|$:

$$p \in [\![c_1 \parallel c_2]\!]X$$

From the definition of $\|$ we know that $\exists X_1, X_2.\ X_1 \otimes X_2 \subseteq X \wedge p \in [\![c_1]\!]X_1 \otimes [\![c_2]\!]X_2$. From this we know that there are p_1, p_2 where $p \vdash p_1 * p_2$ where $p_1 \in [\![c_1]\!]X_1$ and $p_2 \in [\![c_2]\!]X_2$. By induction hypothesis we get $\exists q_1 \in X_1.\ \{p_1\}\, c_1\, \{q_1\}$ and $\exists q_2 \in X_2.\ \{p_2\}\, c_2\, \{q_2\}$ and we know that $q_1 * q_2 \in X_1 \otimes X_2$. By the *Par* rule we get $\{p_1 * p_2\}\, c_1 \parallel c_2\, \{q_1 * q_2\}$ and then by the rule of consequence $\{p\}\, c_1 \parallel c_2\, \{q_1 * q_2\}$. Now, since $X_1 \otimes X_2 \subseteq X$ we know that $q_1 * q_2 \in X$.

Case ;:

$$p \in [\![c_1; c_2]\!]X$$

From the definition of ; we know that $p \in [\![c_1]\!]([\![c_2]\!]X)$. From induction hypothesis we get that $\exists r \in [\![c_2]\!]X.\ \{p\}\, c_1\, \{r\}$ and, fixing such an r, that $\exists q \in X.\ \{r\}\, c_2\, \{q\}$. We then apply the *Seq* rule to get our desired result choosing r as the linking assertion between c_1 and c_2.

Case `skip`:

$$p \in [\![\mathsf{skip}]\!]X$$

By definition of skip we know that $p \in X$ and the result follows from the *Skip* rule in BCSL $\{p\}\, \mathsf{skip}\, \{p\}$ where $q = p$.

Case c_{prim}:

$$p \in [\![c_{prim}]\!]X$$

This holds by the presumptions above.

PROOF OF COMPLETENESS: $(\exists q \in X.\ \{p\}\,c\,\{q\}) \Rightarrow p \in [\![c]\!]X$. The proof is by induction on the derivation of $\{p\}\,c\,\{q\}$, and goes by a case analysis according to the last rule in the derivation.

Case `Skip`:

$$\frac{}{\{p\}\,\mathsf{skip}\,\{q\}} \quad \text{where } q \in X$$

because the *Skip* rule has both pre-condition and post-condition equal, we know that $q = p$ and $p \in X$. The desired result follows since $[\![\mathsf{skip}]\!]X = X$.

Case `Par`:

$$\frac{\{p_1\}\,c_1\,\{q_1\} \quad \{p_2\}\,c_2\,\{q_2\}}{\{p_1 * p_2\}\,c_1 \parallel c_2\,\{q_1 * q_2\}} \quad \text{where } q_1 * q_2 \in X$$

Let $X_1 = q_1 \Downarrow, X_2 = q_2 \Downarrow$.

By lemma 5.1.19 we know that $X_1 \otimes X_2 \subseteq X$. By induction hypothesis we know that $p_1 \in [\![c_1]\!]X_1$ and $p_2 \in [\![c_2]\!]X_2$. Since $p_1 * p_2 \vdash p_1 * p_2$ we know that $p_1 * p_2 \in \{p \mid p \vdash p_1 * p_2 \wedge p_1 \in [\![c_1]\!]X_1 \wedge p_2 \in [\![c_2]\!]X_2\}$ where we take $p = p_1 * p_2$. Therefore we can conclude that $p_1 * p_2 \in [\![c_1]\!]X_1 \otimes [\![c_2]\!]X_2$.

From the definition of $\parallel$ we know that $[\![c_1 \parallel c_2]\!]X = \bigcup\{[\![c_1]\!]Y_1 \otimes [\![c_2]\!]Y_2 \mid Y_1 \otimes Y_2 \subseteq X\}$. For the particular X_1 and X_2 defined above we have already shown that $X_1 \otimes X_2 \subseteq X$, and it is therefore evident that $[\![c_1]\!]X_1 \otimes [\![c_2]\!]X_2 \subseteq \bigcup\{[\![c_1]\!]Y_1 \otimes [\![c_2]\!]Y_2 \mid Y_1 \otimes Y_2 \subseteq X\}$. Therefore, $[\![c_1]\!]X_1 \otimes [\![c_2]\!]X_2 \subseteq [\![c_1 \parallel c_2]\!]X$, and since we have already remarked $p_1 * p_2 \in [\![c_1]\!]X_1 \otimes [\![c_2]\!]X_2$, this proves $p_1 * p_2 \in [\![c_1 \parallel c_2]\!]X$ as required.

Case `Seq`:

$$\frac{\{p\}\,c_1\,\{r\} \quad \{r\}\,c_2\,\{q\}}{\{p\}\,c_1;c_2\,\{q\}} \quad \text{where } q \in X$$

Let $X_r = r\Downarrow$. From induction hypothesis we know that $p \in [\![c_1]\!]X_r$ and $r \in [\![c_2]\!]X$. By downwards closure we know that $r\Downarrow \subseteq [\![c_2]\!]X$, therefore $p \in [\![c_1]\!]([\![c_2]\!]X)$ by monotonicity of $[\![c_1]\!]$ and the result follows from the definition $[\![c_1;c_2]\!]X = [\![c_1]\!]([\![c_2]\!]X)$.

Case `Frame`**:**

$$\frac{\{p\}\,c\,\{q'\}}{\{p*r\}\,c\,\{q'*r\}} \quad \text{where } q = q' * r \in X$$

We are required to show that $p * r \in [\![c]\!]X$.

Let $X_1 = q' \Downarrow$, $X_2 = r \Downarrow$.

By lemma 5.1.19 we know that $X_1 \otimes X_2 \subseteq X$. By induction hypothesis we know that $p \in [\![c]\!]X_1$. Since $p * r \vdash p * r$ we know that $p * r \in \{p' \mid p' \vdash p_1 * p_2 \wedge p_1 \in [\![c]\!]X_1 \wedge p_2 \in X_2\}$ where we take $p' = p * r$. Therefore, we can conclude that $p * r \in [\![c]\!]X_1 \otimes X_2$. We can assume that $[\![c]\!]$ is local because the theorem assumes $[\![c_{prim}]\!]$ local, and Lemma 5.1.8 shows that locality is preserved by parallel and sequential composition. By locality, we know that $[\![c]\!](X) = [\![c \parallel \mathsf{skip}]\!](X) = \bigcup\{[\![c]\!]Y_1 \otimes \mathsf{skip}Y_2 \mid Y_1 \otimes Y_2 \subseteq X\} = \bigcup\{([\![c]\!]Y_1) \otimes Y_2 \mid Y_1 \otimes Y_2 \subseteq X\}$. For the particular X_1 and X_2 defined above we have already shown that $X_1 \otimes X_2 \subseteq X$, and it is therefore evident that $([\![c]\!]X_1) \otimes X_2 \subseteq \bigcup\{([\![c]\!]Y_1) \otimes Y_2 \mid Y_1 \otimes Y_2 \subseteq X\}$. Therefore, $([\![c]\!]X_1) \otimes X_2 \subseteq [\![c \parallel \mathsf{skip}]\!](X) = [\![c]\!](X)$, and since we have already remarked $p * r \in ([\![c]\!]X_1) \otimes X_2$, this proves $p * r \in [\![c]\!]X$ as required.

Case `Consequence`**:**

$$\frac{p' \vdash p \quad \{p\}\,c\,\{q\} \quad q \vdash q'}{\{p'\}\,c\,\{q'\}} \quad \text{where } q' \in X$$

Since X is downwards closed and we know that $q' \in X$ with the fact that $q \vdash q'$ then we know that $q \in X$ therefore $q \Downarrow \subseteq X$. Now we claim that $p' \in [\![c]\!](q \Downarrow) \Rightarrow p' \in [\![c]\!]X$; this is true since $[\![c]\!](q \Downarrow) \subseteq [\![c]\!]X$ because $q \Downarrow \subseteq X$ and $[\![c]\!]$ is monotone. We know by induction hypothesis that $p \in [\![c]\!](q \Downarrow)$ and since $p' \vdash p$ we know that $p' \in [\![c]\!](q \Downarrow)$ by downwards closure of $[\![c]\!](q \Downarrow)$.

■

The net effect of Theorems 5.1.17 and 5.1.18 is that, starting from BCSL, we have constructed an algebra in which the relationship $do-after[p \Downarrow] \sqsupseteq [\![c]\!]; do-after[q \Downarrow]$ is equivalent to provability of the Hoare triple $\{p\}\,c\,\{q\}$.

5.1.4 Connection to the Resource Model

In this section we show the connection between the Generalised Predicate Transformer Model (GPTM) discussed in this chapter and the Resource Model (RM) from Chapter 4. We do this by constructing the structure of a Resource Model which fits the structure of the Generalise Predicate Transformer Model. Then we show an isomorphism between the Resource Model and the Generalised Predicate Transformer Model. (This section does not affect the other technical work in this chapter, and can be skipped without loss of continuity.)

We begin by mapping $(\Sigma, \bullet, u)$ to $(\Sigma_\perp, \vdash, \bullet_\perp, \perp, u)$ where we include a least element $\perp$ and make the $\bullet$ operator total $\bullet_\perp$ by mapping undefined compositions to $\perp$ in the standard way. We define the order $\leq$ as $\sigma \vdash \delta \Leftrightarrow \sigma = \perp \vee \sigma = \delta$. Both these structures form the input structures for the Resource Model and the Generalised Predicate Transformer Model, using these we show homomorphisms between:

$$(P(\Sigma), \subseteq, *, \{\}, \{u\}) \quad \text{and} \quad (P_\downarrow(\Sigma_\perp), \subseteq, \otimes, false, I)$$

which are construction of the Resource Model and the Generalised Predicate Transformer Model respectively. We get the homomorphisms by adding the least element $\perp$ in one direction and removing it in the other. Figure 5.2 provides a diagrammatical view of the connections.

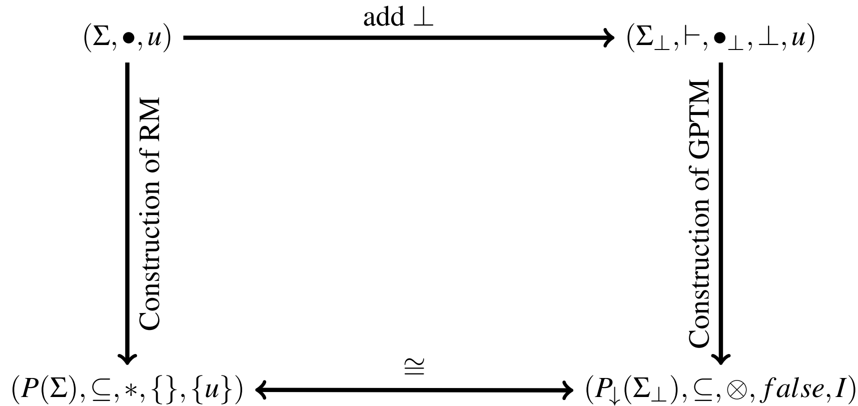


Figure 5.2: Connection Between RM and GPTM

Theorem 5.1.20. *The Resource Model is isomorphic to an instance of a Generalised Predicate Transformer Model*

Proof: We define the maps on the carrier sets and show that they are a bijection, then it is sufficient to show a homomorphism on the other cases of the structure.

$$f = \lambda X.X \cup \{\perp\} : \quad P(\Sigma) \mapsto P_{\downarrow}(\Sigma_{\perp})$$
$$g = \lambda X.X/\{\perp\} : \quad P_{\downarrow}(\Sigma_{\perp}) \mapsto P(\Sigma)$$

Here $\downarrow$ means that the set is a non-empty downwards closed. From the defintion of $\vdash$ we know that for a set to be a non-empty downwards closed it must have the $\perp$ element in it.

$$A \in P_{\downarrow}(\Sigma_{\perp}) \Leftrightarrow A \in P(\Sigma_{\perp}) \wedge (b \in A \wedge c \vdash b) \Rightarrow c \in A$$

Suppose $A \in P(\Sigma)$ then $g(f(A)) = g(A \cup \{\perp\}) = (A \cup \{\perp\})/\{\perp\} = A$. For the other direction suppose $B \in P_{\downarrow}(\Sigma_{\perp})$ then $f(g(B)) = f(B/\{\perp\}) = (B/\{\perp\}) \cup \{\perp\} = B$. We know that $\perp \in B$ from downwards closure.

Case $*$ *maps to* $\otimes$**:** Since

$$A \otimes B = \{p \mid p \vdash a * b \wedge a \in A \wedge b \in B\}$$

Hence

$$A \otimes B = \{p \mid p \vdash a \bullet_{\perp} b \wedge a \in A \wedge b \in B\} = (a * b) \cup \{\perp\} = f(A \otimes B)$$

We get this result from the definition of $*$ since all defined compositions are given by $(-) \bullet (-)$ and undefined composition will map to $\perp$.

Case $\{\}$ *maps to* $false$**:**

$$false = \{p \mid p \vdash \perp\} = \{\perp\} = f(\{\})$$

Case $\{u\}$ *maps to* I**:**

$$I = \{p \vdash emp\} = \{p \vdash u\} = \{u, \perp\} = f(\{u\})$$

■

5.2 On the Session Types Instantiation

We define the predicate transformers $[\![k!j]\!]$ and $[\![k?j.P]\!]$ by reference to provability in BCSL/ST.

$$\begin{array}{lcl} [\![k!j]\!]X & = & \{p \mid \exists q \in X.\{p\}\, k!j\, \{q\} \text{ is provable}\} \\ [\![k?j.P]\!]X & = & \{p \mid \exists q \in X.\{p\}\, k?j.P\, \{q\} \text{ is provable}\} \end{array}$$

That each of these predicates transformers is monotone follows from their definition. $[\![k!j]\!]X$ and $[\![k?j.P]\!]X$ can easily be seen to be local. The reason is that their definitions refer to the proof theory of BCSL and BCSL has the *Frame* rule: as we have seen (Lemma 4.5.5), locality is equivalent to the *Frame* rule. We remark that the technical results on completeness, etc, earlier in this section do not literally apply to the binding form $k?j.P$, because the results above refer to programs built from primitive commands using $\|$ and ;, and $k?j.P$ is not of this form. However, the extension of these results is straightforward, and omitted.

A Session Types version of the Exchange Law is

$$(k_1!a \parallel k_2!b);(k_2!c \parallel k_1!d) \quad \sqsubseteq \quad (k_1!a;k_2!c) \parallel (k_2!b;k_1!d)$$

The first command satisfies the triple

$$\{k_1 :![\alpha];![\alpha];\mathtt{end}, k_2 :![\alpha];![\alpha];\mathtt{end}, a,b,c,d : \alpha\}\,(k_1!a \parallel k_2!b);(k_2!c \parallel k_1!d)\,\{\mathtt{end}\}$$

The only pre-conditions that lead to a provable triple for the second program are *inconsistent* ones. As a consequence, the second program denotes $\top$ in the predicate transformer model. (The reader might enjoy working through the calculation that this program denotes $\top$.) The second program can be rendered in BST, and we can ask a typing question as follows:

$$(k_1!a.k_2!c.\mathtt{inact}) \parallel (k_2!b.k_1!d.\mathtt{inact}) \triangleright \Delta \qquad ?$$

In fact, there is no consistent typing context Δ that can be found as a Baby Session Types program making this judgement true, because it has channel *race*. This is the case for any session types program that (after translation to BCSL) denotes $\top$ in the predicate transformer model.

5.3 Lattice Structure

In this section we do some checks on the lattice structure of the set of predicate transformers to make sure that fix points exist in order to be able to deal with recursion. We will not be going through program logic here; this can be found in my joint work with Hoare, O'Hearn, Petersen, Möller and Struth [21]. We show that the collection of all predicate transformers form a complete lattice and since all primitive commands are monotone with the operators also preserving monotonicity, we know that fixed points exist by the Tarski's fixed point theorem. This section is apart from the main contribution of the thesis and is included mainly for context; it can be skipped by the reader without loss of continuity.

Definition 5.3.1 (Limits)**.** *Let A be a set of predicate transformers and X a predicate.*

Least Upper Bound:

$$(\bigsqcup A)X = \bigcap \{FX \mid F \in A\}$$

Greatest Lower Bound:

$$(\bigsqcap A)X = \bigcup \{FX \mid F \in A\}$$

Lemma 5.3.2. *The collection of all predicate transformers form a complete lattice.*

Proof:

1 Show $(\bigsqcup A)X$ is an upper bound: $\forall F \in A.\ FX \sqsubseteq (\bigsqcup A)X$

By the definition of $\sqsubseteq$ and $\bigsqcup$ we know that we need to show $\forall F \in A.\ FX \supseteq \bigcap \{FX \mid F \in A\}$ and this follows from the definition of $\bigcap$.

2 Show $(\bigsqcup A)X$ is the least upper bound: $\forall F \in A.\ F \sqsubseteq U \Rightarrow (\bigsqcup A)X \sqsubseteq U$

This follows from the definition of $(\bigsqcup A)X$ and subsequently the definition of $\bigcap$.

3 Show $(\bigsqcap A)X$ is a lower bound: $\forall F \in A.\ (\bigsqcap A)X \sqsubseteq FX$

By the definition of $\sqsubseteq$ and $\bigsqcap$ we know that we need to show $\forall F \in A.\ \bigcup \{FX \mid F \in A\} \supseteq FX$ and this follows from the definition of $\bigcup$.

4 Show $(\bigsqcap A)X$ is the greatest lower bound: $\forall F \in A.\ L \sqsubseteq F \Rightarrow L \sqsubseteq (\bigsqcap A)X$

This follows from the definition of $(\bigsqcap A)X$ and subsequently the definition of $\bigcup$.

■

Furthermore, local elements also have fix points and this is an essential property since we have previously mentioned that we intend for commands to be modelled by local elements.

Lemma 5.3.3. *Lubs preserve locality and monotonicity*

1. *If all elements of A are monotone then $\bigsqcup A$ is monotone.*

2. *If all elements of A are local then $\bigsqcup A$ is local.*

Proof:

PROOF OF 5.3.3 - 1

Suppose $X \subseteq X'$. Since all the elements of A are monotone then we know that $\forall F.\ FX \subseteq FX'$ therefore $\bigcap\{FX \mid F \in A\} \subseteq \bigcap\{FX' \mid F \in A\}$ which gives us our desired result.

PROOF OF 5.3.3 - 2

We know that F is local if and only-if $F = F \parallel \mathsf{skip}$. Therefore, an element is local if and only-if it is a fixed-point of the function $(-) \parallel \mathsf{skip}$. As $(-) \parallel \mathsf{skip}$ is monotone, Tarski's fixed-point theorem says that a monotone function on a complete lattice has a complete lattice of fixed-points. Therefore, the local elements form a complete lattice.

■

We also check for continuity and find that ; is not continuous. This is illustrated by the following counter example.

Example 5.3.4. *There exists a set A of predicate transformers, a predicate transformer G and a predicate X, such that $\bigsqcup(G;A) \neq G;\bigsqcup A$. Let*

$$G = \lambda P.\mathit{if}\ P = \emptyset\ \mathit{then}\ \emptyset\ \mathit{else}\ \{10 \mapsto 10\}, \quad F_i = \lambda P.\ \{i \mapsto i\}$$

then

$$\begin{aligned}(\textstyle\bigsqcup(G;A))X &= \textstyle\bigcap\{G(F_iX) \mid F_i \in A\} \\ &= \{10 \mapsto 10\}\end{aligned}$$

We are left with $\{10 \mapsto 10\}$ since each $F_iX \neq \emptyset$, on the other hand:

$$\begin{aligned}(G;\textstyle\bigsqcup A)X &= G((\textstyle\bigsqcup A)X) \\ &= G(\textstyle\bigcap\{F_iX) \mid F_i \in A\}) \\ &= \emptyset\end{aligned}$$

■

Finally, although we have noted that the *lub* of local transformers is local, curiously, we do not have that $(-) \parallel \mathsf{skip}$ distributes through $\bigsqcup$.

Example 5.3.5 (Localisation does not preserve binary $\bigsqcup$ [21]). *We first define a number of heaps:*

$$h = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}$$

$$h_{12} = \{1 \mapsto 1, 2 \mapsto 2\} \qquad h_{23} = \{2 \mapsto 2, 3 \mapsto 3\}$$

$$h_1 = \{1 \mapsto 1\} \qquad h_2 = \{2 \mapsto 2\} \qquad h_3 = \{3 \mapsto 3\}$$

This allows us to define the following (constant) transformers:

$$F = \lambda P.\ \{h_1\} \qquad G = \lambda P.\ \{h_3\}$$

And now we obtain a counterexample to distribution through lubs as

$$((F \sqcup G) \parallel \mathsf{skip})\{h_{12}, h_{23}\} \neq ((F \parallel \mathsf{skip}) \sqcup (G \parallel \mathsf{skip}))\{h_{12}, h_{23}\}$$

For the left hand side, we observe that $F \sqcup G = \lambda P.\ \{h_1\} \cap \{h_3\} = \lambda P.\ \emptyset$ *and so* $(F \sqcup G) \parallel \mathsf{skip} = \lambda P.\ \emptyset$. *For the right hand side, we observe that* $\{h\} = \{h_1\} \otimes \{h_{23}\} = (F\ emp) \otimes \{h_{23}\}$ *and that* $emp \otimes \{h_{23}\} \subseteq \{h_{12}, h_{23}\}$, *so* $\{h\} \subseteq (F \parallel \mathsf{skip})\{h_{12}, h_{23}\}$. *Similarly,* $\{h\} = \{h_3\} \otimes \{h_{12}\} = (G\ emp) \otimes \{h_{12}\}$ *and* $emp \otimes \{h_{12}\} \subseteq \{h_{12}, h_{23}\}$ *so* $\{h\} \subseteq (G \parallel \mathsf{skip})\{h_{12}, h_{23}\}$. *Thus the right hand side is not* $\emptyset$. ■

We have thus seen that the structure in our models has some of the structure of a Concurrent Kleene Algebra, but not all. It has two monoids linked by the Exchange Law, but some of the other structure is not present. In particular, strong properties of preservation of joins by * or ; are not present, and not required by program logic.

5.4 Summary of the Algebraic Structure

The model we have found exhibits the following structure.

1. Two ordered monoids $(M, \sqsubseteq, \parallel, \mathsf{nothing})$ and $(M, \sqsubseteq, ;, \mathsf{skip})$ satisfying the Exchange Law, in which $\parallel$ is commutative and $\parallel$ and skip are monotone.
2. A unit skip of ; which is idempotent with respect to $*$ (i.e., $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$).
3. The collection of all predicate transformers form a complete lattice.
4. Local and non-local predicate transformers form a Galois connection.

In this chapter we have identified that we do not need all of the structure of a CKA and this led us to study the algebraic structure further. In the next section we summarise properties that we can conclude from the structure we have identified.

Chapter 6

Locality Bimonoid

In this chapter we summarise the results from the previous work. Up until now our work has been very technical where we have identified algebraic structures for various specific models. As a result of the commonalities that have surfaced from the various models it is natural to see whether we can generalise these concepts. We will use this chapter to discuss how the algebraic notions can be generalised and what can be concluded in terms of the properties that can be derived from them. In contrast to the previous chapters, this chapter will not be of a technical nature and the technical proofs of all propositions can be found in my joint work with Hoare, O'Hearn, Petersen, Möller and Struth [21].

We began by working through a model of SL based on Action and recorded the algebraic structures identified in the model where the model provides two ordered sets each having a monotone operator representing parallel and sequential composition linked by the Exchange Law. Furthermore, the parallel operator is commutative and each operator has a distinct unit. We also found a more simplistic characterisation of locality than that of [9] in the form of the equation $f = f * \mathsf{skip}$. The set of Local Action gives us the same structure as Action except that both operators share the same unit skip, thus giving us the small exchange laws and in turn validating the *Frame* rule along with the *Concurrency* rule.

We acknowledged that the Action model has a defect in that the parallel operator is not associative. To resolve this we construct an alternative model based on predicate transformers (Resource Model) which exhibits the same algebraic structure as the Action model and additionally has an associative parallel operator. Furthermore, in this model the locality characterisation

was again of the form $f = f * \mathsf{skip}$ as identified in the Action model. With the inclusion of an associative parallel operator the final structure we noted was two ordered monoids representing parallel and sequential composition linked by the Exchange Law where ; and $*$ are monotone and $*$ is commutative.

Our final model was also based on predicate transformers, but of a more general nature. We use a generalised predicate transformer model so that we are able to model Session Types and give a model of BCSL. This model starts with an ordered commutative monoid (required to model BCSL) and the set up of which is influenced by the structures identified in the previous models. We find that the Generalised Predicate Transformer Model validates the Exchange Law and again has the same locality characterisation as the other models in discussion. This gives further evidence that some form of abstraction and generalisation of the algebra is lurking.

Our results for all the models show that they all share the same sort of algebraic structure. Moreover, we have shown that the algebra captures provability precisely and that the algebra agrees with the semantics. We did this by connecting various algebraic triples to semantic triples and the fact that we are able to derive the rules of CSL from the algebraic structure. This in turn illustrates that there is no trickery used, contrary to the thought of the authors of [22]. With the aforementioned results we find that we need not insist on all of the structure of CKA, rather we can use a weakened version of the structure. We now turn to formalising the abstraction of the identified algebraic structure and propose it as an alternate to CKA which we call Locality Bimonoid.

The algebraic notion of Locality Bimonoid retains all of the properties that we have found in the models and furthermore, the structures of the model are instances of Locality Bimonoid. In order to define Locality Bimonoid first, we use the following notions.

Definition 6.0.1 (Ordered Bisemigroup [21])**.** *An Ordered Bisemigroup $(M, \sqsubseteq, *, ;)$ is a paritially ordered set $(M, \sqsubseteq)$ with two monotone compostions $*$ and ;, such that $(M, *)$ is a commutative semigroup and $(M, ;)$ is a semigroup.*

We then consider units:

Definition 6.0.2 (Ordered Bimonoid [21])**.** *An Ordered Bimonoid is a structure $(M, \sqsubseteq, *, \mathsf{nothing}, ;, \mathsf{skip})$ such that $(M, \sqsubseteq, *, ;)$ is an Ordered Bisemigroup and $(M, *, \mathsf{nothing})$ and $(M, ;, \mathsf{skip})$ are monoids.*

Definition 6.0.3 (Local Elements [21])**.** *An element p in an Ordered Bimonoid is local if $p * \mathsf{skip} = p$.*

Definition 6.0.4 (Locality Bimonoid [21]). *A Locality Bimonoid is an Ordered Bimonoid with Exchange where* skip *is local.*

With this we will now summarise the properties that Locality Bimonoid have.

Proposition 6.0.5 (Provability of BCSL). *The rules of BCSL are sound for Hoare and Plotkin triples for any Locality Bimonoid when commands are local.*

In more detail:

1. If we interpret $\{P\}C\{Q\}$ as $P;C \sqsubseteq Q$ and entailment $\vdash$ as $\sqsubseteq$, then the rules of Figure 2.9 from Section 2.4 are sound as long as C is local.
2. If we interpret $\{P\}C\{Q\}$ as $P \sqsupseteq C;Q$ and entailment $\vdash$ as $\sqsupseteq$, then the rules of Figure 2.9 from Section 2.4 are sound as long as C is local.

Furthermore, our results in the previous chapter show how Locality Bimonoid give a sound and complete semantic of BCSL. However, we do not get the *Substitution* and *Existential* rules mentioned in [42] and the *Conjunction* and *Disjunction* rules from Hoare Logic.

Proposition 6.0.6 (Localisation [21]). *Let G be a Locality Bimonoid and $p \in G$. Then $p * \mathsf{skip}$ is local.*

Another way to say this is that the localiser $(-) * \mathsf{skip}$ is idempotent. This allows us to define a sub-bimonoid:

Definition 6.0.7 (Local Core [21]). *Let $G = (M, \sqsubseteq, *, \mathsf{nothing}, ;, \mathsf{skip})$ be a locality bimonoid. Then we define the* local core G_{loc} *of G by*

$$G_{loc} = (M_{loc}, \sqsubseteq, *, \mathsf{skip}, ;, \mathsf{skip})$$

where $M_{loc} = \{p \in M \mid p = p * \mathsf{skip}\}$, and $*$ and ; are restricted appropriately.

Since the localiser is idempotent, selecting the local elements is the same as selecting the image of the localiser. There is a simple connection between G and G_{loc}, given by the localising map $(-) * \mathsf{skip}$. We can even phrase it without needing skip to be local:

Proposition 6.0.8 ([21]). *In an Ordered Bimonoid with exchange, $p * \mathsf{skip}$ is the greatest local element smaller than p.*

When we have a Locality Bimonoid, the same reasoning gives us a Galois connection between the bimonoid and its core.

Proposition 6.0.9 (Galois Connection [21])**.** *Let G be a locality bimonoid. There is a Galois connection between G and G_{loc}, where localisation $(-) * \mathsf{skip} : G \to G_{loc}$ is right adjoint to the inclusion from G_{loc} into G.*

Proposition 6.0.10 (Locality Preservation)**.** *In a locality bimonoid $*$ and ; preserve locality.*

The preservation of locality is important to us since we intend our commands to be local, therefore the $*$ and ; composition should also be local.

Proposition 6.0.11. *If the Locality Bimonoid is a complete lattice then the set of local elements form a complete lattice.*

Propositions 6.0.10 and 6.0.11 together give semantics for recursion in any Locality Bimonoid that is a complete lattice. Furthermore, all programs including recursive programs will satisfy locality. This can be used to connect up to proof rules for recursive procedures. This has been shown in the above mentioned joint work [21].

Generally, a Locality Bimonoid may contain elements that are not local, and these would not be the denotations of programs. Some of these may contain incomputable concepts like quantification, which exist in assertions but not in programs. However, local elements are closed under sequential and concurrent composition, and we may consider these to be the programs (or to contain the programs). This is a crucial difference between Locality Bimonoid and Concurrent Kleene Algebra: in a Concurrent Kleene Algebra every element is local. There is no theoretical necessity in insisting that all elements be local, and so we suggest that this weaker structure is worth considering. On the other hand, we have also seen in Chapter 5, by the non-locality of $do-after$, that requiring locality of all elements does not lose anything in characterising pre-condition and post-condition specifications for state-based models.

Chapter 7

Conclusion

The general outlook in this work has been one of unification. The theory of concurrency is made up of a myriad of models based on many different concepts and this raises the question of whether we can find a more general theory that encompasses these diverse models. In this work we have been able to have a treatment of concurrency where *message-passing*, *shared-memory*, interleaving and independent models are seen as part of the same theory. Ideally we would want a unified theory to even connect up models, program logic and operational semantics. In concurrency theory, unification has long been a clear goal with many translations between models aiding understanding.

This dissertation can be seen as a contribution to a unification activity of a different sort, which is axiomatic in character [22, 21, 24, 25]. Our particular work is far from a grand unifying theory, but we have been investigating algebraic concepts by comparing them through concrete models. As a result we have altered current algebraic structures and hope that through this type of evolution a general theory will eventually emerge.

More specifically, in this dissertation we have investigated links between three formalisms for concurrency: Session Types, Concurrent Separation Logic, and algebraic models (exemplified by Concurrent Kleene Algebra).

The model we obtained for Session Types, as an instance of the Session instantiation of our Generalised Predicate Transformer Model (Chapter 5), is obtained from the syntax of types: it is thus partly syntactic in nature. It has some of the character of a term model in logic, built from proof theory, though the use of mathematical functions (predicate transformers) to interpret

some aspects of the language is not exclusively syntactic. However, the model cannot be taken as providing an independent meaning of the type system. Our original aim in this work was to provide a denotational, fully syntax free interpretation of Session Types using a model that simultaneously captures traces and ownership, as in [27]. However, we were unsuccessful in this attempt, and providing a convincing denotational model of Session Types, particularly a compositional interpretation of the contexts Δ, is a problem we do not see how to solve.

Rather than providing justification for proof rules, the value of the model lies in making links between and describing underlying principles of different formalisms. In particular, it interprets a Session Typing judgement $P \triangleright \Delta$ as a relationship $[\![P]\!] \sqsubseteq [\![\Delta]\!];[\![emp]\!]$, corresponding to the idea that Δ provides an over-approximation (in the sense used in abstract interpretation) of what P will do. When applied to Concurrent Separation Logic based on Session Types, it interprets a triple $\{\Delta\}\, P\, \{\Delta'\}$ as $[\![\Delta]\!] \sqsupseteq [\![P]\!];[\![\Delta']\!]$, thus providing a concrete version of pre-condition and post-condition specifications in which we think of the pre-condition as providing an over-approximation of what a program and its continuation might do in the future. However, obtaining a more genuinely denotational model of Session Types is a problem we leave for future work.

In this work we have described links between a standard model of Concurrent Separation Logic, the Resource Model (which uses predicate transformers), and algebraic models inspired by the recent Concurrent Kleene Algebra (Chapter 4). By looking at such a concrete, and previously-existing model, we hope to have shown that the notion of locality in the algebra generalises the understanding obtained from the concrete resource-based semantics of Separation Logic.

The algebraic structure used in this work admits both state-based and history-based models (the Resource and Traces models) and this exemplifies the unifying nature of the algebra, which allows principles to be stated independently of the syntax in specific models. Precursor works even used ‘true concurrency’ in addition to interleaving models, giving further evidence of the generality [12, 26] . In fact, we ended up with a weaker structure than Concurrent Kleene Algebra, and perhaps this work will also have some input into further developments into algebraic models for concurrency (Chapter 6). The most pressing issues going forward include axiomatisation of further primitives (e.g., distinguishing internal and external choice), exploring a wider range of concrete models, and determining the practical significance of the generalised program logic in any of the concrete models.

Bibliography

[1] S. Adve. Data races are evil with no exceptions: technical perspective. *Commun. ACM*, 53(11):84–84, November 2010.

[2] G. R. Andrews. *Concurrent programming: principles and practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[3] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[4] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *22nd OOPSLA*, pages 301–320, 2007.

[5] S. L. Bloom and Z. Ésik. Free shuffle algebras in language varieties. *Theor. Comput. Sci.*, 163(1&2):55–98, 1996.

[6] S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp16-34).

[7] L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008.

[8] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *7th APLAS*, pages 259–274, 2009.

[9] C. Calcagno, P.W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[10] H. H. Dang, P. Höfner, and B. Möller. Towards algebraic separation logic. In *RelMiCS, Springer LNCS 5827*, pages 59–72, 2009.

[11] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8), 1975.

[12] J. L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61:199–224, 1988.

[13] P. Godefroid. *Partial-order methods for the verification of concurrent systems—an approach to the state-explosion problem*. PhD thesis, University of Liege, 1995.

[14] A. Gotsman, J. Berdine, and B. Cook. Precision and the conjunction rule in concurrent separation logic. *to appear in MFPS*, 2011.

[15] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277, 2007.

[16] P. B. Hansen. *Operating System Principles*. Prentice-Hall, 1973.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580,583, 1969.

[18] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engler, editor, *Symposium on the Semantics of Algebraic Languages*, pages 102–116. Springer, 1971. Lecture Notes in Math. 188.

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

[20] C. A. R. Hoare. *Towards a theory of parallel programming*, pages 231–244. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[21] C. A. R. Hoare, A Hussain, B. Möller, P.W. O'Hearn, R.L. Petersen, and G. Struth. On locality and the exchange law for concurrent processes. *to appear in CONCUR*, 2011.

[22] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra. In *20th CONCUR, Springer LNCS 5710*, pages 399–414, 2009.

[23] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Foundation of concurrent kleene algebra. In *11th RelMiCS*, pages 166–186, 2009.

[24] T. Hoare. Process algebra: A unifying approach. In *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 655–663. Springer Berlin / Heidelberg, 2005.

[25] T. Hoare and J. He. Unifying theories for parallel programming. In *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 15–30. Springer Berlin / Heidelberg, 1997.

[26] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program*, 2011. (Preliminary verson in CONCUR'09).

[27] T. Hoare and P.W. O'Hearn. Separation logic semantics for communicating processes. *Proceedings of 1st FICS Conference, Electr. Notes Theor. Comput. Sci.*, 212:3–25, 2008.

[28] K Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th ESOP, Springer LNCS 1381*, pages 122–138, 1998.

[29] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[30] C. B. Jones. *Wanted: a compositional approach to concurrency*, pages 5–15. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[31] J. Lambek. Deductive systems and categories I: Syntactic calculus and residuated categories. *Mathematical Systems Theory*, 2, 1968.

[32] S. M. Lane. *Categories for the Working Mathematician (Graduate Texts in Mathematics)*. Springer, 2nd edition, September 1998.

[33] J. Larus and H. Sutter. Software and the concurrency revolution. *Queue. ACM*, 3(7):54–62, 2005.

[34] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In *FOSAD, Springer LNCS 5705*, pages 195–222, 2009.

[35] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[36] R. Milner. Functions as processes. In Michael Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer Berlin / Heidelberg, 1990. 10.1007/BFb0032030.

[37] R. Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

[38] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149 – 171, 1993.

[39] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL'04*, 2004.

[40] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007. Preliminary version appeared in CONCUR'04, LNCS 3170, 49–67.

[41] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[42] P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *15th CSL, LNCS 2142, pp1-19*, 2001.

[43] S. Owicki. *Axiomatic Proof Techniques For Parallel Programs*. PhD thesis, Cornell University., 1975.

[44] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976. 10.1007/BF00268134.

[45] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[46] D. Pym, P. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

[47] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[48] C. Stirling. A compositional reformulation of Owicki-Gries's partial correctness logic for a concurrent while language. In Laurent Kott, editor, *Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 407–415. Springer Berlin / Heidelberg, 1986.

[49] H. Sutter. The free lunch is over: A fundamental turn towards concurrency in software. *Dr. Dabb's Journal*, 30(3), 2005.

[50] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe*, PARLE '94, pages 398–413. Springer-Verlag, 1994.

[51] R. D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, 1991.

[52] C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, CADE-21, pages 35–50, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[54] V. T. Vasconcelos. Fundamentals of session types. In *9th SFM, Springer LNCS 5569*, pages 158–186, 2009.

[55] J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In *7th APLAS, Springer LNCS 5904*, pages 194–209, 2009.

[56] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, 2002. pages 402-416.

[57] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.