# LOCAL REASONING FOR STATEFUL PROGRAMS

BY

HONGSEOK YANG

B.S., Korea Advanced Institute of Science and Technology, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

LOCAL REASONING FOR STATEFUL PROGRAMS

Hongseok Yang, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2001
Uday S. Reddy, Advisor

Programs are called *stateful* when they manipulate the state of a computer explicitly, for example, by assignment. The main theme of the thesis is reasoning about stateful programs. In fact, for the past 30 years, significant efforts have been made to obtain good reasoning principles for such programs. However, reasoning techniques developed so far often result in significantly more complex verification than an informal argument; consequently, they don't attract attention from real programmers. In the thesis, we develop O'Hearn's idea of local reasoning, which was recently proposed to overcome such complexity problem in reasoning. The key observation of local reasoning is that most programs show conceptual locality in their use of storage: even though a program is able to access all global variables and all heap data structures in principle, it usually uses only a few of them. Local reasoning uses such an observation to simplify verification: one starts reasoning about a program considering only the accessed portion of storage so as to obtain a "local" fact; then, a "global" fact is inferred simply by conjoining as an invariant a property that only depends on the other portion of storage, which is not accessed by the program. In the thesis, we formulate a rule which allows one to derive a global fact from a local one. The rule is an extension of O'Hearn's Frame Rule for programs with procedures. The core of the rule is to specify requirements enough to ensure that a program phrase doesn't alter the portion of the store that a conjoined property depends on. We use a symmetric noninterference predicate to require that a conjoined property only involves variables not modified by a program, and Reynolds' spatial conjunction to say that the program doesn't access any heap cells which might affect the truth of the conjoined property. Then, the thesis studies the effectiveness of local reasoning formalized by the rule in both theoretical and practical perspectives. We consider completeness of the rule, which provides a sense that with the rule, all properties can be inferred. We also study reasoning principles for deciding whether a property doesn't involve the portion of storage accessed by a program. In particular, we show that the symmetric noninterference predicate and the spatial conjunction connective have good reasoning principles, which can co-exist with reasoning techniques for data abstraction and, in certain cases, can be automated. And the benefit of the formalism in the thesis is illustrated by the verification of the Schorr-Waite graph marking algorithm.

To my parents and MyoungShin

## ACKNOWLEDGMENTS

Firstly, I want to thank my advisor, Prof. Uday Reddy, for his guidance and encouragement for the past 5 years. I have numerous discussions with him about various issues, not only those in the thesis; they are critical in obtaining the current results in the thesis, and more importantly, they helped me to develop a right attitude toward research in computer science. I still remember the beginning of my graduate study, when I tried to spend more time on mathematics thinking that the main value of any computer science results (in theory) lied in deep mathematics; during formal/informal discussions with him, he explained to me various interesting aspects of computations, which exist prior to any mathematical formulations, and helped me to realize that obtaining such insights into computations is far more important than a mere mathematical formulation. I'm also grateful for him helping me to have chances to work with various other researchers in the US, Brazil and England; all such chances broadened my view and also gave great motivations. I'm also thankful to Prof. Peter O'Hearn in Queen Mary, University of London for his advice, kindness and encouragement; indeed, he has been my second advisor even though it is not official. He introduced me to the logic of Bunched Implications and gave me interesting but not too difficult questions, which eventually lead to the results in Part II of the thesis. And besides advice on research, he lets me see the joy of living as a scholar; I have never seen a person who enjoys doing research as much as he does. I feel lucky to get a guidance from such brilliant persons as Prof. Reddy and Prof. O'Hearn.

The thesis could not have been made without the help from many other persons. I like to thank Prof. David Naumann for kindness and discussions on predicate transformers. He has kindly invited me to Stevens Institute of Technology for a week, and also helped me to understand various issues regarding predicate transformers; in particular, he pointed out the work by Hesselink concerning the adequacy of the predicate transformer semantics, which is used in Chapter 6. I thank Prof. Paulo Borba, Prof. Ana Cavalcanti and Prof. Augusto

Sampaio for their hospitality during my visit to Brazil and interesting discussions on formal methods and predicate transformers. Prof. David Pym kindly explains to me about various issues in logic in general, and about the logic of Bunched Implications in particular. I want to thank the following persons for their encouragement and helpful comments on the work in the thesis: Prof. Gul Agha, Prof. Richard Bornat, Prof. Martin Escardo, Prof. John Gray, Prof. Achim Jung, Prof. Sam Kamin, Prof. John Power, Prof. John Reynolds, Prof. Eike Ritter, Prof. James Joyer, Prof. Paul Tayler, Prof. Hayo Thielecke, Prof. Steve Vickers and Prof. Kwangkeun Yi.

I thank all members in the programming language group at UIUC, in the LFP group at Queen Mary, University of London or in the programming language group at University of Birmingham for their friendship and encouraging discussions; especially, I'm thankful to Mattox Beckman, Lars Clausen, Brian Dunphy, Howard Huang, Joel Jones, Joel Klein, Cristiano Calcagno, Adam Eppendal, Paul Levy, Carsten Führmann, Paola Mannegia and Dan Ghica. I can't forget the immense help from Korean friends in UIUC; especially from JoonKyu Yoo and Jaejin Lee. The graduate study confirmed me about how great my friends from the undergraduate college, DongHoon Hyun, YoungChan Kim, JungHoon Lee and SeungHyun Son, are; they were always helpful whenever I had hard time during my graduate study, and they also have been a great source of motivations for studying; especially, SeungHyun and I had many discussions about being a right scholar and some technical problems, and he even answered my questions on topological groups while serving the army in Korea. I thank my parents and sisters for their constant love and understanding; they never stop loving and caring for me even when I made terrible mistakes. Finally, I thank MyungShin Kim for her patience, understanding and love: my final year in England would not have been so fruitful without her love.

# TABLE OF CONTENTS

# LIST OF TABLES

# CHAPTER 1

# Introduction

A programming language is called *stateful* if it has constructs for manipulating states explicitly, such as assignment and dynamic memory allocation. Examples of such languages include most of popular programming languages, such as C, C++, Java and even ML. This thesis is concerned with reasoning about programs written in stateful programming languages, which we call *stateful* programs. In fact, reasoning about stateful programs has been studied extensively in the area of programming language research. Starting from Hoare's axiomatic treatment of a simple imperative language in [20], researchers in the program verification community have explored various issues for formal verification of stateful programs: proof methods for procedures [21, 65], pointers [40, 6], objects [60] and parallel constructs [57] have been developed, and theoretical properties of Hoare's original programming logic and its extensions have been studied [11, 12, 10, 2]. The program analysis community also studies reasoning about stateful programs but with a different emphasis: automatic reasoning by a computer is of their prime concern. Tools for showing certain safety properties have been built. For instance, Leino *et al.* built Extended Static Checker which shows absence of typical run-time program errors, such as null-pointer dereference error and array bound error [15]. Various analysis techniques to infer useful informations for program optimizations have also been developed [46, 41]. For instance, a pointer analysis in [73] derives certain properties of heap storage that are maintained during the execution of a program; such information is used for various optimizations in a compilation process [73].

Even with such extensive research on reasoning for the past 30 years, program verification techniques are hardly used in practice. Such lack of popularity is not because there is a feature in programming languages for which no verification techniques are known. Instead, it is mainly because current verification techniques are too expensive to use in developing programs in practice. In fact, many existing program analysis techniques suffer from a similar efficiency problem. For instance, a shape analysis in [31], which gives useful information about how the structure of heap storage changes during the execution of a program, can not handle "real" programs because analyzing such programs requires too much time and space [69].

In this thesis, we develop the idea of *local reasoning*, which is recently proposed by O'Hearn as a solution for such an efficiency problem in program verification and possibly in a program analysis. The key observation of local reasoning is that most stateful programs show conceptual locality in their use of the store: even though each block of a program can access all stack variables and heap data structures in principle, it usually uses only a few of them. *Local reasoning* is a style of formal reasoning which exploits such locality of memory access: when each block of a program is verified or analyzed, one only considers the accessed portion of the store so as to obtain a "local" fact; then, a property of the other portion of the store is incorporated as an invariant so that a global fact is derived. For instance, when reasoning about a procedure call $sort(x)$ which sorts a linked list starting from a node pointed to by $x$, only the linked list $x$ is considered initially so as to conclude that if $x$ denotes a linked list, after calling the procedure *sort* with $x$, the linked list $x$ gets sorted; for another linked list $y$ which doesn't share any node with the linked list $x$, one can simply infer that any property, such as the linked list $y$ being sorted in a reverse order, is preserved by the procedure call $sort(x)$.

Local reasoning has one important assumption: it is significantly easier to reason about whether a portion of the store is not accessed by a program phrase than to verify a property of the phrase for the portion directly; then, verifying or analyzing a big program is reduced to handling a collection of small programs independently, which is much simpler than dealing with an entire program as a whole. For instance, suppose that it were more difficult to show that a linked list $y$ doesn't overlap with a linked list $x$ than to directly prove that calling $sort(x)$ preserves the linked list $y$ being sorted in a reverse order. Then, there would be no advantage in following local reasoning. Indeed, one of the main issues in the thesis is to find good reasoning principles for showing which portions of the store are not accessed by a program phrase.

The thesis is mainly concerned with local reasoning in program verification. Even though we address basic issues for automatic reasoning, such as decidability, we don't explicitly try to obtain a better program analysis based on local reasoning.

## 1.1 Frame Rule

Formalizing local reasoning in program verification amounts to having a rule which allows one to derive a "global" fact of a program phrase from a "local" one; such a derivation is made first by picking up a property for a portion of the store which is not accessed by the program phrase, and then by conjoining the property as an invariant to the local fact. The difficulty in devising such a rule mainly lies in formulating conditions which ensure that a conjoined property doesn't involve any stack variables or heap cells which might be modified by a program phrase. For instance, consider a simple imperative programming language without any constructs either for procedures or for heap storage. For such a programming language, the following admissible rule in Hoare logic formalizes local reasoning in the sense just described:

$$\frac{\{P\}\, C\, \{Q\}}{\{P \wedge R\}\, C\, \{Q \wedge R\}}\; C \text{ doesn't write to any identifiers in } R$$

The above rule states that if $R$ is a property of stack variables that are not changed by $C$, the truth of $R$ doesn't change before and after the execution of $C$. Note that the rule supports local reasoning: one can start to reason about $C$ only with the accessed stack variables in mind so as to obtain a local fact $\{P\}\, C\, \{Q\}$; and a global fact $\{P \wedge R\}\, C\, \{Q \wedge R\}$ is derived by the rule. The important insight in the rule is that in a simple programming language, just checking whether $C$ writes to any identifiers in $R$ is enough to ensure that $R$ doesn't depend on the portion of the store which $C$ modifies.

We formalize local reasoning for a programming language with procedures and pointers by *the Frame Rule*. For such a programming language, it is not straightforward to formulate conditions which ensure that a command $C$ doesn't change any stack variables and heap cells that might affect the truth of $R$. For instance, the command $x := x.1; x.1 := \mathsf{nil}$ might affect the truth of $y.1 \neq \mathsf{nil}$ even though they don't share any free identifiers.[1] It is possible for $x$ and $y$ to be aliased by call-by-reference parameter passing so that they denote the same stack variable; or the stack variables denoted by $x$ and $y$ might contain the address of the same heap cell. The Frame Rule uses a symmetric noninterference predicate, denoted by $\perp$, to state that a command $C$ doesn't write to any stack variables that $R$ depends on; and a spatial conjunction

[1] $x.1$ denotes the first field of the heap cell which is pointed to by the stack variable $x$.

connective, denoted by $*$, and a tight interpretation of Hoare triples to specify that $C$ and $R$ access disjoint portions of heap storage. The rule is:

FRAME RULE

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

The condition $C \perp R$ in the Frame Rule captures the requirement that $C$ doesn't change any stack variables which might affect the truth of $R$. Generally, we say that the phrases $M$ and $N$ do not interfere ($M \perp N$) if whenever a stack variable is used by both of them, both $M$ and $N$ only read yet not modify the stack variable. For instance, $(x := x.1; x.1 := \mathsf{nil}) \perp (y.1 \neq \mathsf{nil})$ implies that the expression $(y.1 \neq \mathsf{nil})$ and the command $(x := x.1; x.1 := \mathsf{nil})$ access disjoint stack variables, which is equivalent to saying that the two identifiers $x$ and $y$ are bound to different stack variables.

The other half of the requirement in the Frame Rule, which is that $C$ and $R$ access disjoint portions of heap storage, is ensured by the spatial conjunction $P*R$ and the tight interpretation of the Hoare triple $\{P\}\,C\,\{Q\}$. The spatial conjunction $P * R$ of two predicates holds in a state when both $P$ and $R$ hold, and at the same time, all allocated heap cells in the state are partitioned into two groups so that only heap cells in the first group are "essential" for the truth of $P$ and only those in the second "essential" for the truth of $R$: deallocating or modifying heap cells in the first group doesn't affect the truth of $R$, and deallocating or modifying those in the second group doesn't affect the truth of $P$.[2] The tight interpretation of a Hoare triple is the usual partial (or total) correctness interpretation with an additional requirement about which heap cells a command can access. When a Hoare triple $\{P\}\,C\,\{Q\}$ is interpreted tightly, $C$ is not allowed to access any heap cells which are not "essential" for the truth of $P$: for every state satisfying $P$, if deallocating an active heap cell in the state doesn't affect the truth of $P$, the heap cell should not be accessed by the command $C$. For example, in the tight interpretation of $\{x.1 \neq \mathsf{nil}\}\,C\,\{\mathsf{T}\}$, the only heap cell that the command $C$ is allowed to access is the one whose address is stored in the stack variable $x$ in the starting state. The command $x.1 := \mathsf{nil}$ satisfies the requirement, whereas $x := x.1; x.1 := \mathsf{nil}$ sometimes does (when $x.1 = x$ holds in a starting state) and sometimes doesn't (when $x.1 \neq x$ holds in a starting state). Therefore, only $x.1 := \mathsf{nil}$ satisfies the Hoare triple in the tight interpretation.

[2] The explanations of $*$ and the tight interpretation assume that $P$ and $Q$ are intuitionistic: that is, the truth of $P$ and $Q$ are not changed by allocating more heap cells. The precise definitions appear in Part II.

In the Frame Rule, $C$ and $R$ are required to access disjoint portions of heap storage: in every state satisfying the precondition $P*R$ in the global Hoare triple ($\{P*R\}\,C\,\{Q*R\}$), since the state satisfies the spatial conjunction of $P$ and $R$, all allocated heap cells in the state can be grouped into two disjoint sets such that manipulating heap cells in the first group doesn't affect the truth of $R$; and the command $C$ only accesses the first group because $\{P\}\,C\,\{Q\}$ holds in the tight sense. Such complication can not be avoided because unlike stack variables, heap cells accessed by $C$ depend on a state that $C$ starts with. For instance, the command $x := x.1; x.1 := \mathsf{nil}$ only accesses a stack variable denoted by $x$ regardless of what values stack variables or heap cells have before executing the command. On the other hand, the command accesses different heap cells depending on which addresses are initially stored in the stack variable $x$ and the heap cell pointed to by the stack variable $x$.

## 1.2 Contribution

The main contribution of the thesis is to develop local reasoning in program verification, especially by studying theoretical properties for all notions in the Frame Rule, including the rule itself. Especially, we focus on the symmetric noninterference predicate and an assertion language containing the spatial conjunction connective $*$ because the effectiveness of local reasoning with the Frame Rule depends on good reasoning principles for them. More concretely, the achievements of the thesis are as follows:

- **Symmetric Noninterference**:
  - A parametricity semantics of the symmetric noninterference predicate is presented in a category-free manner in Chapter 3 and categorically in Chapter 4. The model explains both the symmetric noninterference predicate and data abstraction; in particular, it provides a theoretical justification for using all important reasoning principles of the symmetric noninterference predicate in [65, 48] and, at the same time, the simulation-based proof method for showing that two different implementations of the same abstract data type are equivalent.
  - The notion of *parametric sheaves* is developed in Chapter 4, which combines parametricity theory and sheaf theory. This concept provides a mathematical framework

for explaining the symmetric noninterference predicate and data abstraction at the same time.

- **Spatial Conjunction**:
  - The algorithmic aspect of the **BI** assertion language, which is an enveloping assertion language for the spatial conjunction connective $*$, is studied in Chapter 8. Even though the whole assertion language is shown to be undecidable, we argue that quantifiers are the main source of the undecidability result; we show that the quantifier-free fragment of the **BI** assertion language, which includes the spatial conjunction connective $*$ and its adjoint $-\!\!*$, is decidable. Such a decidability result indicates that the spatial connectives $*$ and $-\!\!*$ don't increase the complexity of reasoning in a sense. It also becomes a starting point for a program analysis which exploits spatial conjunctions.

- **Local Reasoning and Frame Rule**:
  - The "completeness" of the Frame Rule is shown in Chapter 6. Note that the rule allows one to derive Hoare triples from a given one without looking at the actual implementation of a program phrase. The completeness result indicates that the rule together with other usual structural rules of Hoare logic indeed gives all such Hoare triples: if a new Hoare triple "follows" from a given one without referring to the implementation of a program, it is derivable from the given Hoare triple using the Frame Rule together with other usual structural rules of Hoare logic.
  - The Schorr-Waite graph marking algorithm is verified with local reasoning in mind in Chapter 7. The algorithm has been verified in several verification formalisms for pointer programs. Such a fact indicates that the algorithm is a good test case for the effectiveness of local reasoning. The resulting proof in the chapter is significantly simpler than the ones based on traditional techniques.
  - A semantics of a programming language with procedures and a pointer mechanism is given in Chapter 9 with a proper interpretation of the symmetric noninterference predicate and the **BI** assertion language. The semantics is shown to validate the Frame Rule.

## 1.3 Related Work

The underlying idea of local reasoning, which is to concentrate on the accessed portion of the store, has been recognized as a possible solution for the problem of verifying or analyzing "real" programs by various researchers. Their approaches in realizing the idea can be categorized into two groups depending on how they obtain reasoning principles for deciding whether a property is affected by a program phrase or not. The first group pursues restricting a programming language so that a simple syntactic criterion, such as checking the overlap of free identifiers, can determine that a property is not affected by a program phrase. Leino's modular reasoning in [29, 30] falls into this group. Modular soundness means that the verification of a property for a collection of modules remains valid when more modules are added to the collection. Note that it is similar to the idea of local reasoning in spirit. However, unlike the Frame Rule in the thesis, he restricts a programming language with modularity requirements (for dynamic dependencies) in order to ensure that a simple criterion guarantees that a program phrase doesn't alter heap cells which an assertion depends on [14, 30]. Other proposals for restricting a programming language, which are most relevant to the topic of the thesis, are Syntactic Control of Interference (in short, SCI) [62, 50] and Alias types [72, 76, 75]. Reynolds' SCI is a type system which ensures that two different identifiers in a typed-checked program do not interfere symmetrically: whenever a stack variable is accessed by the quantities denoted by both identifiers, it is only read by them. Therefore, for a program which typechecks in SCI, the symmetric noninterference between two phrases can be checked simply by looking at whether they share any free identifiers. Smith, Morriset and Walker's Alias type system checks whether heap storage has a certain shape at each point of a program. Especially, it uses the spatial conjunction connective to state that two stack variables point to different heap cells. So, Alias type can be seen as an automated verification scheme of certain spatial conjunctions.

The second group doesn't restrict a programming language feature. Instead, it uses additional constructs, such as the symmetric noninterference predicate and the spatial conjunction connective, in order to have sound reasoning principles for deciding whether a property is affected by a program phrase or not. The thesis builds upon research in this direction, especially Reynolds, O'Hearn, Tennent and Reddy's work on specification logic [65, 74, 52, 48, 60], work on modifies-clauses in refinement literature [43, 18, 3, 45, 36] and Reynolds, Ishtiaq and O'Hearn's work on the spatial conjunction connective and **BI** pointer logic [67, 24]. Reynolds' specification logic is an extension of Hoare logic for handling a procedure mechanism [65]. Even without pointers, the procedure mechanism makes it unsound to use a simple syntactic check, such as checking whether two phrases share any free identifiers, to infer that a certain property is not changed by a command. Specification logic uses the asymmetric noninterference predicate to infer that a property is not affected by a program phrase, and it has reasoning principles for the predicate, which meet the requirement of local reasoning: to show noninterference between an assertion and a command is much easier than to directly verify that the assertion is not changed by the command. O'Hearn later proposed the symmetric noninterference predicate because it simplifies reasoning [48], and the symmetric noninterference predicate is used for reasoning about object-oriented programs by Reddy [60]. In the thesis, we use the symmetric noninterference predicate and reasoning principles by Reynolds, O'Hearn and Reddy. The main achievement of the thesis regarding the symmetric noninterference predicate is to justify by a formal semantics that all such reasoning principles of the symmetric noninterference predicate hold even in object-oriented languages, which was an open question in [60].

Modifies-clauses have a different perspective on the use of storage by program phrases or assertions: each modifies-clause for a program specifies which stack variables are allowed to be modified. Modifies-clauses have been used as a part of specification statements in refinement calculus [36, 3], and reasoning principles similar to the Frame Rule have been considered [3, 43, 18]. The idea is extended to handle higher order procedures and procedure variables in [43]. The interpretation of the symmetric noninterference predicate in Part I uses the notion of *support*, which can be seen as a sematic counterpart of an "exact" modifies-clause.

The spatial conjunction of two assertions was implicitly used by Burstall in [7], and Reynolds used it explicitly to ensure that no heap cells are accessed by two assertions at the same time [67]. Ishtiaq and O'Hearn showed that the spatial conjunction connective is better explained in the context of the logic of Bunched Implications (in short, **BI**) because the **BI** framework naturally provides reasoning principles for spatial conjunctions and indicates that the spatial conjunction connective has an adjoint, which is useful to obtain weakest preconditions for language constructs. Moreover, O'Hearn went a step further and proposed the idea of local reasoning by formulating a version of the Frame Rule for programs manipulating heap storage. The treatment of heap storage in the thesis builds upon such developments on the spatial conjunction connective, and O'Hearn's formulation of local reasoning is a starting point of the thesis; the thesis provides theoretical and practical justification of O'Hearn's local reasoning. In particular, we show a decidability result of an assertion language with the spatial connectives, prove a completeness result of O'Hearn's Frame Rule, present an extensive example of local reasoning based on the Frame Rule, and extend O'Hearn's Frame Rule to handle a procedure mechanism.

## 1.4 Overview

The thesis is divided into three parts based on language features in consideration. In Part I, we considers a stateful programming language with a procedure mechanism, and Part II is concerned with a language with constructs for heap storage. A stateful language with both features is considered in Part III.

In Part I, we focus on whether important reasoning principles for the symmetric noninterference predicate are valid even in the presence of data abstraction, which is one of the main features in object-oriented languages. Chapter 2 explains and motivates such a question. It starts with describing Reynolds' Idealized Algol, which we use in Part I to model a programming language with procedures but without any constructs for heap storage. Then, the chapter explains reasoning principles of the symmetric noninterference predicate and illustrates why it is important to have reasoning principles both for the symmetric noninterference predicate and data abstraction at the same time. Chapter 3 presents a solution. In the chapter, we give a formal semantics of Idealized Algol and the symmetric noninterference predicate in a category-free style, which validates not only all reasoning principles of the symmetric noninterference predicate but also the simulation-based proof technique for showing that two different implementations of the same abstract data type are equivalent. In Chapter 4, we develop the novel mathematical framework, *parametric sheaves*, which combines parametricity theory and sheaf theory. The framework provides a setting for explaining the symmetric noninterference predicate and data abstraction at the same time. In fact, our semantics in Chapter 3 is an instance of the framework, and its well-definedness is a corollary of the fact that given an RG-site, the category of parametric sheaves on the RG-site forms a cartesian closed category. At the end of the chapter, we re-present the semantics of Idealized Algol and the symmetric noninterference predicate categorically, thereby making its relationship with parametric sheaves explicit.

Part II consists of four chapters. Chapter 5 contains a background material for Part II. It reviews a simple **WHILE** programming language extended with constructs for heap storage, which is used by Reynolds, Ishtiaq and O'Hearn. The **BI** assertion language, which is an enveloping assertion language containing spatial conjunctions, and **BI** pointer logic are reviewed in the same chapter. In Chapter 6, we prove that O'Hearn's Frame Rule for programs without procedures is complete. Intuitively, the completeness result indicates that given a Hoare triple, if a new Hoare triple "follows" from the given one without referring to the actual implementation of a command, the new Hoare triple is derivable using O'Hearn's Frame Rule and other usual structural rules in Hoare logic. Chapter 7 is concerned with an actual application of the idea of local reasoning; in the chapter, we prove the correctness of the Schorr-Waite graph marking algorithm in **BI** pointer logic with local reasoning in mind. The simplicity of the resulting verification illustrates the benefit of local reasoning in practice. Finally, in Chapter 8, we prove undecidability and decidability results of the **BI** assertion language. The whole language is shown to be undecidable. However, the undecidability proof, which is an encoding of a co-halting problem, uses quantifiers in an essential way. In fact, we also prove that the quantifier-free fragment of the **BI** assertion language is decidable.

In Part III, we wrap up results in the previous two parts. In Chapter 9, we consider a programming language with both procedure and pointer mechanisms and formulate the Frame Rule. The rule is proved to be a sound reasoning principle by a formal semantics. The semantics in the chapter also shows that all reasoning principles of the symmetric noninterference predicate and the **BI** assertion language still hold even when both procedures and heap storage are considered simultaneously.

In the rest of the thesis, we presume that the computer memory consists of stack storage and heap storage. We often call them simply *the stack* and *the heap*, respectively. And the status of the stack at a particular moment is called *a stack*. Similarly, the status of the heap is called *a heap*. When a programming language has constructs to manipulate the heap, *a state*, which is the status of the machine that a program in the language looks at, is a pair of a stack and a heap; in Part II and Part III, the word "state" has such a meaning. Otherwise, a state

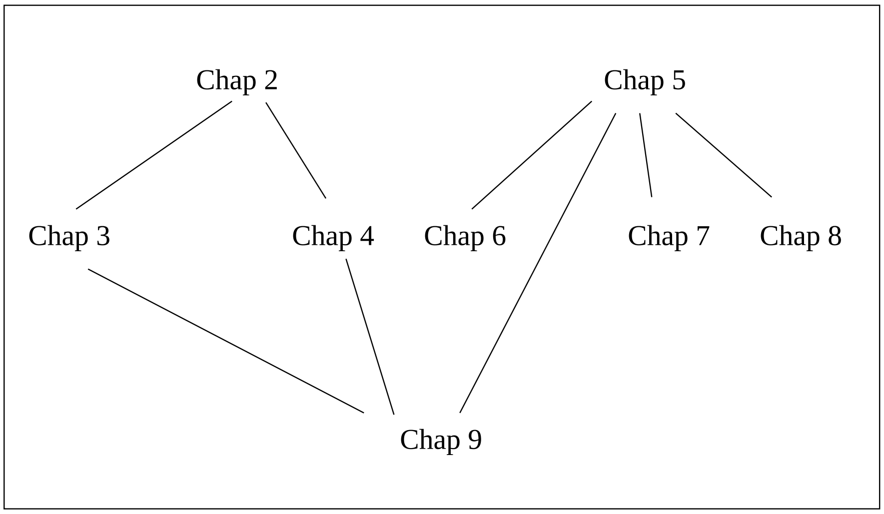



**Figure 1.1** Dependency of Chapters

only consists of a stack, which becomes the case in Part I. We also omit the qualifier "stack" in the word "stack variable" in order to avoid being verbose.

Part I of the thesis results from a joint work with Reddy [79]. Chapter 6 is a result of a joint effort with O'Hearn [47], and the results in Chapter 8 come from the discussion with Calcagno and O'Hearn [9]. The dependency of all chapters in the thesis is shown in Figure 1.1.

# PART I

# Local Reasoning for Programs with Procedures

# CHAPTER 2

## Introduction to Part I

Part I of the thesis concerns local reasoning for a programming language with a procedure mechanism but without constructs manipulating the heap. In particular, we focus on the symmetric noninterference predicate because for such a programming language, the symmetric noninterference predicate is all we need to formalize local reasoning. Intuitively, two program phrases $M$ and $N$ do not interfere symmetrically ($M \perp N$) when they don't use the same variable except in a read-only fashion. So, in a programming language without any mechanisms for the heap, when a command $C$ doesn't interfere with an assertion $R$ and $R$ holds initially, $R$ continues to hold during the execution of $C$. Reynolds' Constancy rule [65] captures such an argument:

$$\frac{\{P\}\,C\,\{Q\} \qquad C \perp R}{\{P \wedge R\}\,C\,\{Q \wedge R\}}$$

Note that the rule formalizes the idea of local reasoning in the thesis: it allows one to derive a global fact ($\{P \wedge R\}\,C\,\{Q \wedge R\}$) from a local one ($\{P\}\,C\,\{Q\}$) by incorporating a property ($R$) for the unaccessed portion of the store as an invariant.

The noninterference predicate either in an asymmetric form or in a symmetric form has already been studied by various researchers: the asymmetric noninterference predicate first appeared in [65]; its semantics has been studied in [74, 52]; the symmetric noninterference predicate was first proposed in [48] and used in formulating reasoning principles for object-oriented programs in [60].

Part I builds on such work on the noninterference predicate. The main contribution in Part I is to resolve an open question in [60] about obtaining a formal semantics which justifies that the symmetric noninterference predicate and data abstraction can live side-by-side harmoniously:

the semantics must validate important reasoning principles for the symmetric noninterference predicate and, at the same time, the simulation-based proof technique for showing that two different implementations of the same abstract data type are equivalent.

The main idea of obtaining such a semantics in Part I is to combine parametricity theory and sheaf theory, which have been used to explain data abstraction and symmetric noninterference, respectively. Reynolds' parametricity theory is originally developed to capture the notion of uniformity. But we are more interested in Reddy's use of parametricity theory in [60], where he showed that parametricity theory also provides a good explanation for data abstraction: in a parametricity model, two implementations of the same abstract data type have the same denotation as long as there is a simulation relation preserved by all corresponding operations in the two implementations; so, the notion of abstraction is explained in terms of an equivalence class induced by the existence of a simulation relation. However, defining the symmetric noninterference predicate properly in a parametricity framework turned out to be subtle. The main challenge is to validate the Noninterference Decomposition rule, which intuitively says that if $M$ doesn't interfere with $N_1$ and $N_2$ separately, $M$ doesn't interfere with $(N_1, N_2)$, either. For instance, Reddy failed to validate the Noninterference Decomposition rule in his parametricity semantics of Idealized Algol extended with classes. Our parametricity model in Part I solves such a problem by using sheaf theory following O'Hearn in [49], where he showed that state-independent resources, such as variables accessed by a command, can be explained with atomic sheaves: he used a well-known fact for atomic sheaves to precisely capture which variables each program phrase accesses; then, such a precise characterization of the accessed variables was used to give a semantics of the symmetric noninterference predicate, which can validate important reasoning principles for the predicate, including the Noninterference Decomposition rule. Our semantics in Part I is an instance of a parametricity framework with additional requirements from sheaf theory. However, it can be better seen as an instance of a novel framework of *parametric sheaves* which combines parametricity theory and sheaf theory in greater generality. Such a view provides a crucial insight for showing that all reasoning principles for the symmetric noninterference predicate are valid in our semantics because parametric sheaves have properties which are useful for analyzing procedures.

Part I consists of three chapters including this introduction. In this chapter, we briefly review Idealized Algol, which we use in Part I as a model programming language with a procedure mechanism, and reasoning principles for the symmetric noninterference predicate. We also give an example which illustrates the importance of understanding symmetric noninterference in connection with data abstraction. A semantics of symmetric noninterference is presented in a category-free style in Chapter 3 and categorically in Chapter 4. The notion of parametric sheaves is given in Chapter 4 with its mathematical properties.

In order to focus on the main issue, we approximate the notion of symmetric noninterference by ignoring the read-only use of a variable; so, the interpretation of the symmetric noninterference predicate in Part I says that two phrases do not interfere symmetrically when they don't access the same variable even in a read-only manner. However, the techniques in Part III can be used to incorporate the read-only use.

## 2.1 Background

### 2.1.1 Idealized Algol

In Part I, we use Reynolds' Idealized Algol (in short, IA) as a model programming language with procedures but without constructs for the heap. Reynolds' Idealized Algol is a framework for higher-order imperative languages just as PCF is so for functional languages. IA is a call-by-name typed lambda calculus with imperative base types, such as $\mathsf{com}$ and $\mathsf{exp}$, and imperative constant terms, such as assignment and sequencing. Types are stratified into two layers: *data types* for storable values and *phrase types* for program terms. Usually, only integer, real and boolean values are regarded as storable entities so that only $\mathsf{int}$, $\mathsf{real}$ and $\mathsf{bool}$ become data types. Phrase types are usual types in the lambda calculus sense, obtained from the command type ($\mathsf{com}$), expression types ($\mathsf{exp}[\mathsf{int}]$, $\mathsf{exp}[\mathsf{real}]$, $\mathsf{exp}[\mathsf{bool}]$) and variable types ($\mathsf{var}[\mathsf{int}]$, $\mathsf{var}[\mathsf{real}]$, $\mathsf{exp}[\mathsf{bool}]$) by applying function and product type constructors. Basic imperative actions, such as assignment and sequencing, are incorporated as constant terms. Even though there are different proposals for incorporating constant terms for the heap,[1] only the stack is available in the original Idealized Algol in [64], and IA in Part I also considers the stack only.

To simplify the presentation, we assume that only integers can be stored in variables, i.e., $\mathsf{int}$ is the only data type. We use $\mathsf{exp}$ and $\mathsf{var}$ to denote integer expressions and integer variables, respectively. Basic constants are chosen according to this simplification. The recursion operator

[1] One of them is, in fact, used in Part III of the thesis

**Types:**

$$\begin{array}{llll} \text{data type} & \delta & ::= & \mathsf{int} \\ \text{phrase type} & \theta & ::= & \mathsf{exp} \mid \mathsf{com} \mid \mathsf{var} \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 \end{array}$$

**Constant Terms:**

$$\begin{array}{rcl} 0 & : & \mathsf{exp} \\ \mathsf{succ} & : & \mathsf{exp} \to \mathsf{exp} \\ \mathsf{pred} & : & \mathsf{exp} \to \mathsf{exp} \\ \mathsf{skip} & : & \mathsf{com} \\ \_;\_ & : & \mathsf{com} \times \mathsf{com} \to \mathsf{com} \\ \mathsf{ifz} & : & \mathsf{exp} \times \beta \times \beta \to \beta \\ \mathsf{get} & : & \mathsf{var} \to \mathsf{exp} \\ \_ := \_ & : & \mathsf{var} \times \mathsf{exp} \to \mathsf{com} \\ \mathsf{new} & : & (\mathsf{var} \to \mathsf{com}) \to \mathsf{com} \end{array}$$

where $\beta = \mathsf{exp}$ or $\mathsf{com}$.

**Table 2.1** Syntax of Idealized Algol

Y is not considered for the sake of the simple presentation. However, the recursion operator can be easily incorporated following the treatment of recursion in Part III and [53, 55]. The whole syntax of IA that we are using is shown in Table 2.1, where the meaning of IA terms should be evident from their names. Note that the conditional statement can only be used with expressions or commands.[2]

*Example:*

- $y := 0; x := y + 1$ is represented by

$$(y := 0); (x := \mathsf{succ}\ (\mathsf{get}\ y))$$

- The function to swap two variable arguments is represented by

$$\begin{array}{l} \lambda x\colon \mathsf{var}.\, \lambda y\colon \mathsf{var}. \\ \quad \mathsf{new}\ \lambda t\colon \mathsf{var}. \\ \quad\quad (t := \mathsf{get}\ x); \\ \quad\quad (x := \mathsf{get}\ y); \\ \quad\quad (y := \mathsf{get}\ t) \end{array}$$

[2]Therefore, we don't allow bad variables.

- Even objects and classes can be encoded in IA [62, 60]. Consider a counter object which has an *increase* method of $\mathsf{com}$ type and a *read* method of $\mathsf{exp}$ type. Let $client\colon \mathsf{com}\times\mathsf{exp}\to\mathsf{com}$ be a client of this object. Then, a class "counter", i.e., a generator of counter objects, is:

$$
\begin{array}{l}
\lambda client\colon \mathsf{com}\times\mathsf{exp}\to\mathsf{com}.\\
\quad \mathsf{new}\ \lambda x\colon \mathsf{var}.\\
\quad\quad \mathsf{let}\ init =_{\mathsf{com}} (x := 0)\\
\quad\quad \mathsf{in}\ \mathsf{let}\ increase =_{\mathsf{com}} (x := (\mathsf{get}\ x) + 1)\\
\quad\quad \mathsf{in}\ \mathsf{let}\ read =_{\mathsf{exp}} (\mathsf{get}\ x)\\
\quad\quad \mathsf{in}\\
\quad\quad\quad\quad init; (client\ (increase, read))
\end{array}
$$

  where $\mathsf{let}\ x =_\theta M\ \mathsf{in}\ N$ is syntactic sugar for $(\lambda x\colon\theta.\,N)\,M$.

### 2.1.2 Symmetric Noninterference

The noninterference predicate first appeared in Reynolds' specification logic [63, 65], which is programming logic for IA. Specification logic is multi-sorted first-order intuitionistic logic with types of IA as its sorts. Its atomic formulas consist of Hoare triples, the noninterference predicate and the equality predicate. The logic contains rules in Hoare logic for simple imperative programming as well as rules for the simply typed lambda calculus with recursion such as $\beta$-equivalence and Scott's fixed point induction principle. In some sense, specification logic results from combining the logic, Scott LCF, for the typed lambda calculus with recursion and the logic, Hoare logic, for simple imperative programming just as IA is obtained by taking imperative base types in the simply typed lambda calculus.

Two terms do not interfere with each other symmetrically when they use disjoint portions of the stack.[3] We write $M\perp N$ to denote that $M$ and $N$ do not interfere symmetrically.[4] Important reasoning principles for the noninterference predicate, proposed by Reynolds, O'Hearn and Reddy in [65, 48, 60], are as follows:

1. if $P\perp M$ and $N\perp M$, then $P(N)\perp M$;

[3] In Part I, we don't consider in the definition of the symmetric noninterference predicate the issue of passivity, which is to allow a variable to be shared by two terms as long as the variable is not modified by either of them. Such an issue of passivity is handled in Part III.

[4] We use $\perp$ instead of Reynolds' # in order to contrast with the asymmetric noninterference predicate by Reynolds.

2. if $N \perp i$ for all free identifiers $i$ of $M$, then $N \perp M$;

3. a non-local entity $M$ and a local variable $x$ do not interfere symmetrically; and

4. if $M \perp N$, then $N \perp M$.

The second principle is also known as the Noninterference Decomposition rule, and it has been the most problematic rule in a parametricity semantics [60].

All four reasoning principles for the symmetric noninterference predicate do not depend on states: no reasoning principles deal with Hoare triples or implications between assertions. The reasoning principles only involve finding all free identifiers in a term and checking whether some of them are locally declared variables. Therefore, if it ever works, showing a Hoare triple mainly with the symmetric noninterference predicate, such as reasoning with the Constancy rule shown later, is generally easier than reasoning about the Hoare triple directly with axioms in Hoare logic. The state-independence of the reasoning principles mainly comes from the fact that all variables ever used by a term, which is what $\perp$ is all about, are determined in a state-independent manner.

One immediate use of the noninterference predicate and its principles is to obtain variants of Hoare's rules in the presence of a general procedure mechanism, that is, rules in specification logic. In particular, Hoare's assignment axiom is modified in specification logic to[5]

$$P \perp M \Longrightarrow \{P(E)\}\, M := E\, \{P(\mathsf{get}(M))\}$$

For instance, consider the Hoare triple $\{y = 2\}\, x := 1\, \{y = 2\}$. It holds only as long as $x$ and $y$ are not aliased, i.e., they denote different variables. This exactly corresponds to $x \perp y$. The new assignment rule with the second principle of noninterference captures such informal reasoning:

$$\begin{array}{rl} & (\lambda z.\, y = 2) \perp x \Longrightarrow \{(\lambda z.\, y = 2)1\}\, x := 1\, \{(\lambda z.\, y = 2)\ (\mathsf{get}\ x)\} \\ \Longrightarrow & \qquad \because \beta\text{-equivalence} \\ & (\lambda z.\, y = 2) \perp x \Longrightarrow \{y = 2\}\, x := 1\, \{y = 2\} \\ \Longrightarrow & \qquad \because \text{by the second principle,} \quad y \perp x \text{ implies } (\lambda z.\, y = 2) \perp x \\ & y \perp x \Longrightarrow \{y = 2\}\, x := 1\, \{y = 2\} \end{array}$$

[5] Usually, we also need a condition for bad variables. However, since the variant of IA that we are using doesn't allow bad variables, the noninterference condition in the rule suffices.

As noted in the beginning of this chapter, one important use of the noninterference predicate is to formalize a main step in local reasoning, which is to derive a global fact from a local one. In specification logic, the Constancy rule plays such a role:

$$\frac{\{P\}\, C\, \{Q\} \qquad C \perp R}{\{P \wedge R\}\, C\, \{Q \wedge R\}}\ \text{Constancy}$$

The rule says that if $R$ and $C$ do not interfere symmetrically, then $R$ becomes an invariant for the command $C$. Since the four reasoning principles for the symmetric noninterference predicate are easier to use than Hoare's usual axioms, deriving a Hoare triple with Constancy is more effective in practice than showing the Hoare triple directly. Conversely, the success of local reasoning with Constancy depends on having good reasoning principles for the symmetric noninterference predicate, such as the four reasoning principles in this section. For instance, if a new language feature significantly complicated reasoning about the symmetric noninterference predicate, the local reasoning would have to be formulated not with the symmetric noninterference predicate but with some other concepts. So, the main achievement in Part I can also be seen as a justification for using the notion of symmetric noninterference in local reasoning: by showing that the four reasoning principles for the symmetric noninterference predicate are valid even in the presence of data abstraction, we provide an evidence that reasoning about the symmetric noninterference predicate is intrinsically simple.

## 2.2 Symmetric Noninterference and Data Abstraction

Before closing this chapter, we provide an example which illustrates why it is important to justify that the symmetric noninterference predicate and data abstraction can co-exist while keeping their own reasoning principles. When a proof system includes all the four reasoning principles for the symmetric noninterference predicate as well as the simulation-based proof technique, the system supports one interesting proof strategy: when reasoning about a program which uses a class, we pick up another class which is equivalent but structured better in terms of noninterference; then, we show the intended property with respect to the new class, instead of a given one. Two classes can be proved to be equivalent by finding a simulation relation between "internal" variables of two classes such that all corresponding operations in the two classes "preserve" the simulation relation. And with the four reasoning principles for the symmetric noninterference predicate, we can exploit the fact that the newly picked-up class is organized better in terms of noninterference. In this section, we give an example which illustrates the proof strategy and its benefit. The IA in this section is slightly general than one in Section 2.1.1 in the sense that it has the $\mathsf{real}$ data type and several constant terms on $\mathsf{real}$ expressions such as $\mathsf{sin}$,$*$, etc. And we also often omit the $\mathsf{get}$ operation on a variable when it is used inside an expression.

Suppose that we attempt to implement a class for a point object in a 3-dimensional space. The object is supposed to have two methods *move* and *yCoord*: *move* method moves a point by $+1$ along the $x$-coordinate and by $+1$ along the $z$-coordinate, and *yCoord* method returns the $y$-coordinate in the cartesian coordinate representation. The following "class" *SPoint* implements this specification using the cylindrical coordinate representation:

$$
\begin{aligned}
&\mathit{SPoint} \;=\\
&\quad \lambda \mathit{client}\colon \mathsf{com} \times \mathsf{exp}[\mathsf{real}] \rightarrow \mathsf{com}.\\
&\qquad \mathsf{new}_{\mathsf{real}}\ \lambda r\colon \mathsf{var}[\mathsf{real}].\\
&\qquad \mathsf{new}_{\mathsf{real}}\ \lambda \theta\colon \mathsf{var}[\mathsf{real}].\\
&\qquad \mathsf{new}_{\mathsf{real}}\ \lambda z\colon \mathsf{var}[\mathsf{real}].\\
&\qquad\ r := 0.0; \theta := 0.0; z := 0.0;\\
&\qquad\ \mathit{client} \left( \left( \begin{array}{l}
\mathsf{new}_{\mathsf{real}}\ \lambda \theta'\colon \mathsf{var}[\mathsf{real}].\\
\quad \theta' := \theta;\\
\quad \theta := \mathsf{tan}^{-1}(\frac{r*\mathsf{cos}\,\theta'+1.0}{r*\mathsf{sin}\,\theta'});\\
\quad r := \sqrt{(r*\mathsf{cos}\,\theta'+1.0)^2+(r*\mathsf{sin}\,\theta')^2};\\
\quad z := z+1.0
\end{array} \right), r*\mathsf{sin}\,\theta \right)
\end{aligned}
$$

Consider the following client of the *SPoint* class. It caches the y-coordinate of an object ($y' := \mathsf{snd}(\mathit{point})$), runs the *move* method ($\mathsf{fst}(\mathit{point})$) and assigns 1 or 0 to $k$ depending on whether the y-coordinate ($\mathsf{snd}(\mathit{point})$) has been changed or not.

$$
\begin{aligned}
\mathit{Client} \;=\; & \lambda \mathit{point}\colon \mathsf{com} \times \mathsf{exp}[\mathsf{real}].\\
& \quad \mathsf{new}_{\mathsf{real}}\ \lambda y'\colon \mathsf{var}[\mathsf{real}].\\
& \qquad y' := \mathsf{snd}(\mathit{point});\\
& \qquad \mathsf{fst}(\mathit{point});\\
& \qquad \mathsf{ifz}_{\mathsf{real}}\ (\mathsf{snd}(\mathit{point}) - y')\ \mathsf{then}\ k := 1\ \mathsf{else}\ k := 0
\end{aligned}
$$

Since conceptually the *move* method only changes the $x$- and $z$-coordinates of a point object and *yCoord* is only about the $y$-coordinate of the point object, we expect that $k$ has 1 at the end of the execution. That is, the following Hoare triple should hold:

$$\{\mathsf{T}\}\,(SPoint\ \ Client)\{k = 1\}$$

The Hoare triple is expanded into the following one by applying $\beta$-reduction twice:

$$
\begin{array}{l}
\{\mathsf{T}\}\\
\quad \mathsf{new}_{\mathsf{real}}\ \lambda r\colon \mathsf{var}[\mathsf{real}].\\
\quad \mathsf{new}_{\mathsf{real}}\ \lambda \theta\colon \mathsf{var}[\mathsf{real}].\\
\quad \mathsf{new}_{\mathsf{real}}\ \lambda z\colon \mathsf{var}[\mathsf{real}].\\
\qquad r := 0.0; \theta := 0.0; z := 0.0;\\
\qquad \mathsf{new}_{\mathsf{real}}\ \lambda y'\colon \mathsf{var}[\mathsf{real}].\\
\qquad\quad y' := r * \mathsf{sin}\,\theta;\\
\qquad\quad \mathsf{new}_{\mathsf{real}}\ \lambda \theta'\colon \mathsf{var}[\mathsf{real}].\\
\qquad\qquad \theta' := \theta;\\
\qquad\qquad \theta := \mathsf{tan}^{-1}(\frac{r*\mathsf{cos}\,\theta'+1.0}{r*\mathsf{sin}\,\theta'});\\
\qquad\qquad r := \sqrt{(r * \mathsf{cos}\,\theta' + 1.0)^2 + (r * \mathsf{sin}\,\theta')^2};\\
\qquad\qquad z := z + 1.0;\\
\qquad\quad \mathsf{ifz}_{\mathsf{real}}\ ((r * \mathsf{sin}\,\theta) - y')\ \mathsf{then}\ k := 1\ \mathsf{else}\ k := 0\\
\{k = 1\}
\end{array}
$$

By just looking at the code, it is not obvious that the above Hoare triple is true; the crucial step is to show that $(r * \mathsf{sin}\,\theta) = y'$ holds after the series of assignments $\theta' := \theta; \theta := \ldots; r := \ldots; z := z + 1.0$, which requires nontrivial verification by the assignment axiom and Consequence.

The first step in the proof strategy is to find a class which is equivalent to the given one but structured better in terms of noninterference. For this purpose, consider an implementation of points with the cartesian coordinate representation, which we call $CPoint$:

$$
\begin{array}{lcl}
CPoint & = & \lambda client\colon \mathsf{com} \times \mathsf{exp}[\mathsf{real}] \to \mathsf{com}.\\
& & \qquad \mathsf{new}_{\mathsf{real}}\ \lambda x\colon \mathsf{var}[\mathsf{real}].\\
& & \qquad \mathsf{new}_{\mathsf{real}}\ \lambda y\colon \mathsf{var}[\mathsf{real}].\\
& & \qquad \mathsf{new}_{\mathsf{real}}\ \lambda z\colon \mathsf{var}[\mathsf{real}].\\
& & \qquad\quad x := 0.0; y := 0.0; z := 0.0;\\
& & \qquad\quad client\ (x := x + 1.0; z := z + 1.0,\ \mathsf{get}\ y)
\end{array}
$$

Informally, two classes *CPoint* and *SPoint* are equivalent because they are just two different representations of the same abstract data type for 3-dimensional points. Using the simulation-based proof technique, one can show such equivalence. The proof technique says that two classes are equivalent if there is a relation between "internal" variables preserved by all corresponding operations of the two classes. In the case of the two classes for 3-dimensional points, the relation between internal variables of the two classes is the usual $R$ given by:

$$\begin{aligned} x &= r * \mathsf{sin}\,\theta \\ y &= r * \mathsf{cos}\,\theta \\ z_S &= z_C \end{aligned}$$

where we annotate $z$ with subscripts $_S$ and $_C$ to indicate which class $z$ belongs to. All corresponding operations of *SPoint* and *CPoint* "preserve" $R$: the implementations of the *move* method in *SPoint* and *CPoint* map $R$-related states to $R$-related states, and those of the *yCoord* map $R$-related states to the same real number. Therefore, they are equivalent by the simulation-based proof technique.

The next step is to prove the intended Hoare triple for the newly chosen but equivalent class. In our case, it corresponds to verifying the following Hoare triple:

$$\{\mathsf{T}\}\,(CPoint\ \ Client)\{k = 1\}$$

We first unroll the above Hoare triple by applying $\beta$-reduction twice:

$$\begin{array}{l}
\{\mathsf{T}\} \\
\mathsf{new}_{\mathsf{real}}\ \lambda x \colon \mathsf{var}[\mathsf{real}]. \\
\mathsf{new}_{\mathsf{real}}\ \lambda y \colon \mathsf{var}[\mathsf{real}]. \\
\mathsf{new}_{\mathsf{real}}\ \lambda z \colon \mathsf{var}[\mathsf{real}]. \\
\quad x := 0.0; y := 0.0; z := 0.0; \\
\quad \mathsf{new}_{\mathsf{real}}\ \lambda y' \colon \mathsf{var}[\mathsf{real}]. \\
\quad\quad y' := y; \\
\quad\quad x := x + 1.0; \\
\quad\quad z := z + 1.0; \\
\quad\quad \mathsf{ifz}_{\mathsf{real}}\ (y - y')\ \mathsf{then}\ k := 1\ \mathsf{else}\ k := 0 \\
\{k = 1\}
\end{array}$$

Then, the most important step of the verification is to show

$$\{y' = y\}\, x := x + 1.0; z := z + 1.0\, \{y' = y\}$$

which can be derived using reasoning principles for the symmetric noninterference predicate and Constancy:

$$\dfrac{\dfrac{\dfrac{}{y' \perp x}\,3 \qquad \dfrac{}{y' \perp z}\,3 \qquad \dfrac{}{y \perp x}\,3 \qquad \dfrac{}{y \perp z}\,3}{(y' = y) \perp (x := x + 1.0; z := z + 1.0)}\,2 \qquad \{\mathsf{T}\}x := x + 1.0; z := z + 1.0\{\mathsf{T}\}}{\{y' = y\}\, x := x + 1.0; z := z + 1.0\, \{y' = y\}}$$

where each number indicates which principle in Section 2.1.2 is used and the last step of the derivation is by Constancy. Notice that the verification is done without ever considering the meaning of the assignment $x := x + 1.0; z := z + 1.0$ and the assertion $y' = y$; instead, it only uses "syntactic" checking.

In summary, instead of dealing with a given class directly, an equivalent but well-structured class in terms of noninterference is chosen, and the intended Hoare triple is verified with respect to the new class. Both the reasoning principles for the symmetric noninterference predicate and the simulation-based proof technique are crucial in such reasoning.

# CHAPTER 3

## Semantics of Symmetric Noninterference

In this chapter, we develop a semantics of the symmetric noninterference predicate in a parametricity model of IA, where all the principles of the symmetric noninterference predicate in Chapter 2 are sound. In order to simplify the presentation, the semantics is shown in two steps: a presheaf model and then, a parametricity model. We show that in both models, all the reasoning principles of the symmetric noninterference predicate hold. On the other hand, the additional requirement on the interpretation of types and terms in the parametricity model validates the simulation-based proof method for showing that two different implementations of the same abstract data type are equivalent, thereby explaining the data abstraction aspect. Both models are, in some sense, simplified versions of existing semantics [35, 49, 53, 60]. However, the minimal resource property in the second model (Theorem 9) is new in our work and becomes the basis to obtain a proper interpretation of the symmetric noninterference predicate. The property follows from a more general result on the novel mathematical framework of *parametric sheaves*, which is explained in more detail in Chapter 4. In this chapter, we follow the category-free style presentation in [53, 60] and focus on the second reasoning principle of the noninterference predicate: the Noninterference Decomposition rule. The parametricity model in the chapter is also presented categorically in Chapter 4.

### 3.1 Presheaf Model

One of the main features of a presheaf model is that each type is interpreted not by a set but by a parameterized set. Usually, this parameter represents the shape of the stack. For instance, when only three variables $\{x, y, z\}$ are allocated in the stack, the interpretation of a command at this shape can only include a transformation on states of $\{x, y, z\}$. A command which increases some other variable $w$ can't be included. The presheaf model in this section is a simplified version of [49, 35]. We mainly focus on the interpretation of types.

Suppose *Var* is a countably infinite set, which denotes all possible variables. A finite subset of *Var* is called a *world* and represents a set of variables currently allocated in the stack. The interpretation of each type and phrase is parameterized by these finite subsets of *Var*. Let $W$ be a world and $S(W)$ a set of states, which are just functions from $W$ to *Int*. The meanings of types other than function types are as follows:

- $[\![\mathsf{com}]\!](W) = S(W) \to S(W)$
- $[\![\mathsf{exp}]\!](W) = S(W) \to \mathit{Int}$
- $[\![\mathsf{var}]\!](W) = \{(\lambda n.\lambda s.\, s[x \mapsto n],\, \lambda s.\, s(x)) \in [\mathit{Int} \to [\![\mathsf{com}]\!](W)] \times [\![\mathsf{exp}]\!](W) \mid x \in W\}$
- $[\![\theta_1 \times \theta_2]\!](W) = [\![\theta_1]\!](W) \times [\![\theta_2]\!](W)$

where for sets $A$ and $B$, we use $A \times B$ and $A \to B$ to denote the cartesian product of $A$ and $B$ and the set of functions from $A$ to $B$, respectively. Notice that in the semantics, $\mathsf{var}$ is interpreted in an "object-oriented" fashion. Each variable is treated as an object with methods for updating and reading. The first component of the interpretation of a variable denotes a method for updating and the second component for reading.

The execution of a program goes through a series of changes in the shape of the stack; so, it is necessary to have a way to convert a meaning, for instance of a command, in one world to another. In the presheaf model, two kinds of changes of the stack are allowed: allocating more variables and renaming existing variables. We indicate such a change by an injection $f\colon W \to W'$; the set $W' - f(W)$ denotes newly allocated variables and the isomorphism $f\colon W \to f(W)$ renaming of existing variables. We use the notation $f\colon W \hookrightarrow W'$ to denote that $f$ is an *injection* from $W$ to $W'$. And the inclusion from $W$ to $W'$ is denoted by $\mathsf{inc}_{W,W'}$. For the change of worlds indicated by an injection $f\colon W \hookrightarrow W'$, the corresponding change in the meaning of a term, denoted by $\mathsf{expand}(f)(-)$, is described as follows:

- command: for any $c \in [\![\mathsf{com}]\!](W)$,

$$\begin{aligned}&\mathsf{expand}(f)_{\mathsf{com}}(c)\\ &= \lambda s'\colon S(W').\,\lambda x'\colon W'.\,\begin{cases} c(s' \circ f)(x) & \text{if } \exists x \in W.\, f(x) = x' \\ s'(x') & \text{otherwise}\end{cases}\end{aligned}$$

- expression: for any $e \in [\![\mathsf{exp}]\!](W)$,

$$\mathsf{expand}(f)_{\mathsf{exp}}(e) \;=\; \lambda s'\colon S(W').\, e(s' \circ f)$$

- variable: for any $v \in [\![\mathsf{var}]\!](W)$,

$$\begin{aligned}&\mathsf{expand}(f)_{\mathsf{var}}(v)\\ &= \Big(\mathsf{expand}(f)_{\mathsf{com}} \circ \mathsf{fst}(v),\ \mathsf{expand}(f)_{\mathsf{exp}}(\mathsf{snd}(v))\Big)\end{aligned}$$

- for any $(p_1, p_2) \in [\![\theta_1 \times \theta_2]\!](W)$,

$$\begin{aligned}&\mathsf{expand}(f)_{\theta_1 \times \theta_2}(p_1, p_2)\\ &= (\mathsf{expand}(f)_{\theta_1}(p_1),\, \mathsf{expand}(f)_{\theta_2}(p_2))\end{aligned}$$

The interpretation of $\theta_1 \to \theta_2$ at a world $W$ is more elaborate. It can not be given just as a set of functions from $[\![\theta_1]\!](W)$ to $[\![\theta_2]\!](W)$ because there is no canonical way to change a function at $W$ to one at $X$ for any world $X$ obtained by allocating new variables to $W$. Therefore, the function type is defined as a uniform family of functions, which contains all the necessary instances for possible changes of the world $W$:

- $[\![\theta_1 \to \theta_2]\!](W) \;=$

$\{\{m_g\colon [\![\theta_1]\!](X) \to [\![\theta_2]\!](X)\}_{g:W \hookrightarrow X} \mid$

for all injections $g\colon W \hookrightarrow X$ and $h\colon X \hookrightarrow Y$,

$$\begin{tikzcd} {[\![\theta_1]\!](X)} \arrow[r, "m_g"] \arrow[d, "\mathsf{expand}(h)_{\theta_1}"'] & {[\![\theta_2]\!](X)} \arrow[d, "\mathsf{expand}(h)_{\theta_2}"] \\ {[\![\theta_1]\!](Y)} \arrow[r, "m_{h \circ g}"'] & {[\![\theta_2]\!](Y)} \end{tikzcd}$$

i.e., $m_{h \circ g} \circ \mathsf{expand}(h)_{\theta_1} = \mathsf{expand}(h)_{\theta_2} \circ m_g\ \}$

- for $\{m_g\}_{g:W \hookrightarrow X} \in [\![\theta_1 \to \theta_2]\!](W)$,

$$\mathsf{expand}(f\colon W \hookrightarrow W')_{\theta_1 \to \theta_2}(\{m_g\}_{g:W \hookrightarrow X}) = \{m_{h \circ f}\}_{h:W' \hookrightarrow X}$$

The condition on the family of functions is called *naturality*, which intuitively states that all functions in the family essentially act "the same way." For instance, $\{\lambda c\colon [\![\mathsf{com}]\!](X).\, c \circ c\}_{g:\emptyset \hookrightarrow X}$ is an element of $[\![\mathsf{com} \to \mathsf{com}]\!](\emptyset)$.

Recall that intuitively, two program phrases do not interfere with each other symmetrically when they use disjoint sets of variables. Formalizing this requires us to define accurately what "used variables" means, which is done with the notion of *support*:

**Definition 1** *For any type $\theta$ and $a \in [\![\theta]\!](W)$, a subset $W'$ of $W$ is the* support *of $a$ iff*

1. *there exists $a' \in [\![\theta]\!](W')$ such that $\mathsf{expand}(\mathsf{inc}_{W',W})_\theta(a') = a$; and*

$$\begin{array}{c} a \in [\![\theta]\!](W) \\ \mathsf{expand}(\mathsf{inc}_{W',W})_\theta \Big\uparrow \\ a' \in [\![\theta]\!](W') \end{array}$$

2. *$W'$ is the least such set.*

For instance, the supports of the three commands, $\lambda s\colon S(\{x,y\}).\, s$, $\lambda s\colon S(\{x,y\}).\, s[x \mapsto 1]$ and $\lambda s\colon S(\{x,y\}).\, s[x \mapsto 1, y \mapsto 1]$, in a world $\{x,y\}$ are $\emptyset$, $\{x\}$ and $\{x,y\}$, respectively. One might worry that an element $a \in [\![\theta]\!]W$ might not have the support. The following theorem ensures that such a case never happens.

**Theorem 2 (minimal resource)** *For any world $W$, IA type $\theta$ and $a \in [\![\theta]\!](W)$, the support of $a$ always exists.*

We write $\mathsf{supp}(a)$ to denote the support of $a$. With the definition of support, the meaning of symmetric noninterference can easily be obtained by interpreting "disjoint" in the intuition literally.

**Definition 3** *For any $a \in [\![\theta_1]\!](W)$ and $b \in [\![\theta_2]\!](W)$, $a$ and $b$ don't interfere with each other ($a \perp b$) iff $\mathsf{supp}(a) \cap \mathsf{supp}(b) = \emptyset$.*

The remaining question is whether this definition of symmetric noninterference is satisfactory in the sense that all the principles in Chapter 2 are valid. This, in fact, becomes the case. The following lemma indicates why the second principle, the most problematic one, is valid in the model:

**Lemma 4** *For any $p \in [\![\theta]\!](W)$, $p_1 \in [\![\theta_1]\!](W)$ and $p_2 \in [\![\theta_2]\!](W)$, if $p \perp p_1$ and $p \perp p_2$, then $p \perp (p_1, p_2)$.*

*Proof:* Since $p \perp p_1$ and $p \perp p_2$, $\mathsf{supp}(p) \cap (\mathsf{supp}(p_1) \cup \mathsf{supp}(p_2)) = \emptyset$. Since $\mathsf{supp}((p_1, p_2)) = \mathsf{supp}(p_1) \cup \mathsf{supp}(p_2)$, the conclusion follows. $\square$

The model given in this section doesn't provide a good explanation for data abstraction because in the model, there exists a "semantic" client of an abstract data type which can "look under" the actual implementation of the abstract data type; so, reasoning based on simulation relations [66, 70, 53, 60] to show the equivalence of two different implementations of an abstract data type is not valid in the model. For instance, consider the following two different implementations of a counter class, which has *increase* and *read* methods. In one implementation, the internal variable $x$ is increased whenever *increase* is called, and *read* returns the value of $x$. In the other implementation, *increase* method decreases the value of $x$, but *read* takes the negation before returning the value. In IA, they can be encoded as follows:

$$\begin{array}{ll}
\textit{counter0} = & \textit{counter1} = \\
\lambda p \colon \mathsf{com} \times \mathsf{exp} \to \mathsf{com}. & \lambda p \colon \mathsf{com} \times \mathsf{exp} \to \mathsf{com}. \\
\quad \mathsf{new}\ \lambda x \colon \mathsf{var}. & \quad \mathsf{new}\ \lambda x \colon \mathsf{var}. \\
\qquad x := 0; & \qquad x := 0; \\
\qquad p\Big((x := (\mathsf{get}\ x) + 1),\ (\mathsf{get}\ x)\Big) & \qquad p\Big((x := (\mathsf{get}\ x) - 1),\ (-(\mathsf{get}\ x))\Big)
\end{array}$$

Objects generated by these two counter classes behave the same way, which can be formally described using simulation relations. Let $R$ be a relation on $S(\{x\})$ such that $s\,R\,s' \iff s(x) = -s'(x)$. The first parts (*increase*) in both counter objects map $R$-related states to $R$-related states. And, if the input states are related by $R$, the second parts (*read*) of the counter objects produce the same value. Moreover, the internal variable $x$ in each counter object is initialized properly (as 0) to set up the relation $R$ initially. Since a client can not access the internal variable $x$ directly, the two counter classes are behaviorally equivalent. However, these two classes are not equivalent in this presheaf model because there exists an element in the interpretation of a function type which acts differently on these two implementations. For instance, on one case, the element can assign 1 to another variable $y$ and on the other case, 2 to $y$. That is, the difference of two implementations is reflected on an "external" variable $y$. More precisely,

**Lemma 5** *There exists $m \in [\![\mathsf{com} \times \mathsf{exp} \to \mathsf{com}]\!](\{y\})$ such that when $m_{\mathsf{inc}_{\{y\},\{x,y\}}}$ is applied to counters from the two implementations,*

$\Big(\lambda s{:}\, S(\{x,y\}).\, s[x \mapsto s(x)+1],\ \ \lambda s{:}\, S(\{x,y\}).\, s(x)\Big)$ *and*

$\Big(\lambda s{:}\, S(\{x,y\}).\, s[x \mapsto s(x)-1],\ \ \lambda s{:}\, S(\{x,y\}).\, -s(x)\Big)$,

*the first and the second map a state* $[y \mapsto 0, x \mapsto 0]$ *to* $[y \mapsto 1, x \mapsto 1]$ *and* $[y \mapsto -1, x \mapsto -1]$, *respectively.*

## 3.2 Parametricity Model

The basic idea of a parametricity model is to impose a stronger uniformity condition on the interpretation of a function type using relations. In this model, each type denotes not only a parameterized set but also a parameterized relation. And this relation part is used to define the meaning of a function type which requires stronger uniformity on a family of functions than naturality. Since all the definitions of the previous section are carried over here except the meaning of function types, we only describe their action on relations when dealing with types other than function types.

Each $\theta$ maps a relation $R$ between $S(W_0)$ and $S(W_1)$ to a relation $[\![\theta]\!](R)$ between $[\![\theta]\!](W_0)$ and $[\![\theta]\!](W_1)$. We use $S{:}\, A \leftrightarrow B$ to denote the fact that $S$ is a relation between $A$ and $B$. Given $R{:}\, S(W_0) \leftrightarrow S(W_1)$, the relation part in the interpretation of each base type is given as follows:

- for $c_0 \in [\![\mathsf{com}]\!](W_0)$ and $c_1 \in [\![\mathsf{com}]\!](W_1)$, $c_0\, [\![\mathsf{com}]\!](R)\, c_1$ iff

  $R$-related states $s_0$ and $s_1$ are mapped to $R$-related states by $c_0$ and $c_1$, i.e., $s_0\, R\, s_1 \Rightarrow c_0(s_0)\, R\, c_1(s_1)$.

- for $e_0 \in [\![\mathsf{exp}]\!](W_0)$ and $e_1 \in [\![\mathsf{exp}]\!](W_1)$, $e_0\, [\![\mathsf{exp}]\!](R)\, e_1$ iff

  $R$-related states $s_0$ and $s_1$ are mapped to the same value by $e_0$ and $e_1$, i.e., $s_0\, R\, s_1 \Rightarrow e_0(s_0) = e_1(s_1)$.

- for $v_0 \in [\![\mathsf{var}]\!](W_0)$ and $v_1 \in [\![\mathsf{var}]\!](W_1)$, $v_0\, [\![\mathsf{var}]\!](R)\, v_1$ iff

  for any integer $n$ in *Int*, $\mathsf{fst}(v_0)(n)\ [\![\mathsf{com}]\!](R)\ \mathsf{fst}(v_1)(n)$
  and $\mathsf{snd}(v_0)\ [\![\mathsf{exp}]\!](R)\ \mathsf{snd}(v_1)$

- for $(p_0, q_0) \in [\![\theta \times \theta']\!](W_0)$ and $(p_1, q_1) \in [\![\theta \times \theta']\!](W_1)$,
  $(p_0, q_0)\, [\![\theta \times \theta']\!](R)\, (p_1, q_1)$ iff

$$p_0 \, [\![\theta]\!](R) \, p_1 \text{ and } q_0 \, [\![\theta']\!](R) \, q_1.$$

*Notation:* We use a notation familiar from work on logical relations [58]. For relations $R: A \leftrightarrow B$ and $S: C \leftrightarrow D$, $[R \to S]$ is a relation between function spaces $(A \to C)$ and $(B \to D)$ such that two functions are related iff they map $R$-related arguments to $S$-related arguments. Formally, $f \, [R \to S] \, g$ iff for all $a \in A$ and $b \in B$, $a \, R \, b \Rightarrow f(a) \, S \, g(b)$. And $R \times S$ is a pointwise relation between $A \times C$ and $B \times D$: $(a, c) \, (R \times S) \, (b, d)$ iff $a \, R \, b$ and $c \, S \, d$. We often denote that $f \, [R \to S] \, g$ by the following diagram:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, leftrightarrow, "R"'] & C \arrow[d, leftrightarrow, "S"] \\
B \arrow[r, "g"'] & D
\end{tikzcd}$$

The interpretation of a function type is also given by a family of functions with uniformity conditions as in the presheaf model. The only but significant difference is that the parametricity model requires a family of function to be more uniform: the family is supposed to satisfy the parametricity condition in addition to the usual naturality condition. We use an example from a programming language to illustrate the parametricity condition. Consider a procedure $p$ of type $\theta \to \theta'$, which is declared when no variables are allocated in the stack. Suppose that the procedure is first applied to an argument $a$ just after allocating a variable $x$, which we denote by $p_{\{x\}}(a)$, and then, it is applied again to an argument $b$ when one more variable $y$ is allocated, which we denote by $p_{\{x,y\}}(b)$. The parametricity condition says that for every relation $R$ between states of $x$ and states of $x, y$, if the arguments $a$ and $b$ are related by $[\![\theta]\!]R$, $p_{\{x\}}(a)$ and $p_{\{x,y\}}(b)$ are also related by $[\![\theta']\!]R$; the reason is that $p$ can access $x$ or $x, y$ only though its arguments $a$ or $b$.

For $R: S(W_0) \leftrightarrow S(W_1)$ and $P: S(L_0) \leftrightarrow S(L_1)$ such that $W_0 \cap L_0 = W_1 \cap L_1 = \emptyset$, let $R \times P$ be a relation between $S(W_0 \cup L_0)$ and $S(W_0 \cup L_1)$ such that $s_0 \, (R \times P) \, s_1$ iff $s_0|_{W_0} \, R \, s_1|_{W_1}$ and $s_0|_{L_0} \, P \, s_1|_{L_1}$. And for any set $A$, $\mathbf{\Delta}_A$ denotes a diagonal relation on $A$, i.e., $a_0 \, \mathbf{\Delta}_A \, a_1 \Leftrightarrow a_0 = a_1$. Using these notations, a function type is interpreted as follows:

- $[\![\theta_1 \to \theta_2]\!](W) \ =$

  $\{\{m_g\colon [\![\theta_1]\!](X) \to [\![\theta_2]\!](X)\}_{g:W\hookrightarrow X} \mid$

  for all injections $g\colon W \hookrightarrow X$ and $h\colon X \hookrightarrow Y$,

$$\begin{tikzcd}
[\![\theta_1]\!](X) \arrow[r, "m_g"] \arrow[d, "\mathsf{expand}(h)_{\theta_1}"'] & [\![\theta_2]\!](X) \arrow[d, "\mathsf{expand}(h)_{\theta_2}"] \\
[\![\theta_1]\!](Y) \arrow[r, "m_{h\circ g}"'] & [\![\theta_2]\!](Y)
\end{tikzcd}$$

  i.e., $m_{h\circ g} \circ \mathsf{expand}(h)_{\theta_1} = \mathsf{expand}(h)_{\theta_2} \circ m_g$

  for all nonempty relations $R\colon S(L_0) \leftrightarrow S(L_1)$ where $W \cap L_0 = W \cap L_1 = \emptyset$,

$$\begin{tikzcd}
[\![\theta_1]\!](W \cup L_0) \arrow[r, "m_{\mathsf{inc}_{W,W\cup L_0}}"] \arrow[d, leftrightarrow, "{[\![\theta_1]\!](\mathbf{\Delta}_{S(W)} \times R)}"'] & [\![\theta_2]\!](W \cup L_0) \arrow[d, leftrightarrow, "{[\![\theta_2]\!](\mathbf{\Delta}_{S(W)} \times R)}"] \\
[\![\theta_1]\!](W \cup L_1) \arrow[r, "m_{\mathsf{inc}_{W,W\cup L_1}}"'] & [\![\theta_2]\!](W \cup L_1)
\end{tikzcd}$$

  i.e., $m_{\mathsf{inc}_{W,W\cup L_0}}\ [[\![\theta_1]\!](\mathbf{\Delta}_{S(W)} \times R) \to [\![\theta_2]\!](\mathbf{\Delta}_{S(W)} \times R)]\, m_{\mathsf{inc}_{W,W\cup L_1}}\ \}$

- for a nonempty relation $R\colon S(W_0) \leftrightarrow S(W_1)$, $\{m_g\}_{g:W_0\hookrightarrow X}\ [\![\theta_1 \to \theta_2]\!](R)\ \{m'_h\}_{h:W_1\hookrightarrow X}$ iff

  for all nonempty relations $P\colon S(L_0) \leftrightarrow S(L_1)$, where $L_0 \cap W_0 = \emptyset$ and $L_1 \cap W_1 = \emptyset$,

$$\begin{tikzcd}
[\![\theta_1]\!](W_0 \cup L_0) \arrow[r, "m_{\mathsf{inc}_{W_0,W_0\cup L_0}}"] \arrow[d, leftrightarrow, "{[\![\theta_1]\!](R \times P)}"'] & [\![\theta_2]\!](W_0 \cup L_0) \arrow[d, leftrightarrow, "{[\![\theta_2]\!](R \times P)}"] \\
[\![\theta_1]\!](W_1 \cup L_1) \arrow[r, "m'_{\mathsf{inc}_{W_1,W_1\cup L_1}}"'] & [\![\theta_2]\!](W_1 \cup L_1)
\end{tikzcd}$$

  i.e., $m_{\mathsf{inc}_{W_0\cup L_0}}\ [[\![\theta_1]\!](R \times P) \to [\![\theta_2]\!](R \times P)]\, m'_{\mathsf{inc}_{W_1\cup L_1}}$.

- for $\{m_g\}_{g:W\hookrightarrow X} \in [\![\theta_1 \to \theta_2]\!](W)$,

  $\mathsf{expand}(f\colon W \hookrightarrow W')_{\theta_1\to\theta_2}(\{m_g\}_{g:W\hookrightarrow X}) = \{m_{h\circ f}\}_{h:W'\hookrightarrow X}$

With the interpretation of types, all expansions can be shown to satisfy a uniformity condition given by relations (parametricity).

**Lemma 6 (Expansion Parametricity)** *For all IA types $\theta$ and relations $R: S(W_0) \leftrightarrow S(W_1)$ and $P: S(L_0) \to S(L_1)$ such that $W_0 \cap L_0 = W_1 \cap L_1 = \emptyset$,*

$$\mathsf{expand}(\mathsf{inc}_{W_0, W_0 \cup L_0})_\theta \; [[\![\theta]\!](R) \to [\![\theta]\!](R \times P)] \; \mathsf{expand}(\mathsf{inc}_{W_1, W_1 \cup L_1})_\theta$$

And each IA type preserves an identity relation in the following sense:

**Lemma 7 (Identity Extension Lemma)** *For all IA types $\theta$ and worlds $W$,*

$$[\![\theta]\!](\mathbf{\Delta}_{S(W)}) = \mathbf{\Delta}_{[\![\theta]\!](W)}.$$

In the parametricity model, the additional uniformity condition in the interpretation of a function type ensures that no "semantic" clients can look under the actual implementation of an abstract data type: when all corresponding operations in two implementations of an abstract data type preserve a simulation relation on internal variables of the two, no clients can tell apart the two. For instance, consider again the example of two implementations of a counter class in the previous section. Let's assume for simplicity that only a variable $y$ is allocated initially. And let a variable $x$ be a newly allocated variable which is used as an internal variable of a counter object in each implementation. Let $R$ be the relation on $S(\{x\})$ as in the previous section: $s \, R \, s' \Leftrightarrow s(x) = -s'(x)$. We show that unlike in the presheaf model, no clients in $[\![\mathsf{com} \times \mathsf{exp} \to \mathsf{com}]\!](\{y\})$ act differently on counter objects from the two implementations. Let $m$ be a client in $[\![\mathsf{com} \times \mathsf{exp} \to \mathsf{com}]\!](\{y\})$. From the parametricity condition, the instance at $\{x, y\}$ of an element $m \in [\![\mathsf{com} \times \mathsf{exp} \to \mathsf{com}]\!](\{y\})$ is related to itself by $[[\![\mathsf{com} \times \mathsf{exp}]\!](\mathbf{\Delta}_{S(\{y\})} \times R) \to [\![\mathsf{com}]\!](\mathbf{\Delta}_{S(\{y\})} \times R)]$. Moreover, by the time that objects from the implementations are provided as arguments to $m_{\mathsf{inc}_{\{y\},\{x,y\}}}$, they are related by $[\![\mathsf{com} \times \mathsf{exp}]\!](\mathbf{\Delta}_{S(\{y\})} \times R)$. Therefore, $m$ can't reflect the difference of two objects on the "external" variable $y$ when the initial states are related by $\mathbf{\Delta}_{S(\{y\})} \times R$. More precisely,

**Lemma 8** *For all $m \in [\![\mathsf{com} \times \mathsf{exp} \to \mathsf{com}]\!](\{y\})$ and $n \in Int$, when $m_{\mathsf{inc}_{\{y\},\{x,y\}}}$ is applied to counter objects from the two implementations which are*

$\Big(\lambda s: S(\{x, y\}).\, s[x \mapsto s(x) + 1],\;\; \lambda s: S(\{x, y\}).\, s(x)\Big)$*; and*

$\Big(\lambda s: S(\{x, y\}).\, s[x \mapsto s(x) - 1],\;\; \lambda s: S(\{x, y\}).\, -s(x)\Big)$,

*the first and the second map a state $[y \mapsto n, x \mapsto 0]$ to $[y \mapsto n', x \mapsto n'']$ and $[y \mapsto n', x \mapsto -n'']$ for some $n', n'' \in Int$, respectively.*

Since the final phase of using counter objects is to deallocate the variable $x$, the lemma shows that $m$ acts the same way for counter objects from the two implementations.

The notion of support is defined the same way as in the previous section. However, it is not immediate that the minimal resource property in the previous section still holds because the interpretation of a function type is changed. Using parametric sheaves explained in the next section and in Chapter 4, we can show that the property holds.

**Theorem 9 (minimal resource)** *For every IA type $\theta$, $W$ and $a \in [\![\theta]\!](W)$, the support of $a$ always exists.*

Therefore, the definition of symmetric noninterference as in the previous section is still well-defined in the parametricity model. With this definition, all four principles of symmetric noninterference in Chapter 2 can be verified. In particular, the main lemma for the soundness of the Noninterference Decomposition rule holds by the same reason as for Lemma 4.

**Lemma 10** *For all $p \in [\![\theta]\!](W)$, $p_1 \in [\![\theta_1]\!](W)$ and $p_2 \in [\![\theta_2]\!](W)$, if $p \perp p_1$ and $p \perp p_2$, then $p \perp (p_1, p_2)$.*

## 3.3 Atomic Sheaves and Minimal Resources

In the previous section, Theorem 2 and Theorem 9 play an important role in defining symmetric noninterference. In this section, we explain why those theorems hold. The main observation is that the interpretation of each imperative type is not just parameterized set and relation; it satisfies an atomic-sheaf condition. In Section 3.3.1, we informally derive such a sheaf condition using an intuitive understanding on sequential imperative computation and state its connection with Theorem 2. In the next section, we extend the sheaf condition to the parametricity model.

### 3.3.1 The Atomic-Sheaf Condition in the Presheaf Model

Imperative types, such as $\mathsf{com}$ and $\mathsf{exp}$, have a property which is not shared by all parameterized sets. We explain the property by an example. Let's consider two commands $c_1$ and $c_2$ acting on worlds $\{x, y\}$ and $\{y, z\}$, respectively. That is, $c_1 \in [\![\mathsf{com}]\!](\{x, y\})$ and $c_2 \in [\![\mathsf{com}]\!](\{y, z\})$. Suppose our goal is to find a sufficient condition which guarantees that $c_1$ and $c_2$ only access the variable $y$ in the same way, i.e., they are expansions of the same command from the world $\{y\}$.

Let's start by considering a necessary condition: what would hold if $c_1$ and $c_2$ were expansions of the same command $c$ in the world $\{y\}$? Since all expanded commands from $c$ only access the variable $y$ in the same way as $c$, the three commands $c_1$, $c_2$ and $c$ must essentially do the same thing. Therefore, if both $c_1$ and $c_2$ are expanded again to the same bigger world $\{x,y,z\}$, the expanded commands should coincide: $\mathsf{expand}(\mathsf{inc}_{\{x,y\},\{x,y,z\}})_{\mathsf{com}}(c_1) = \mathsf{expand}(\mathsf{inc}_{\{y,z\},\{x,y,z\}})_{\mathsf{com}}(c_2)$. Diagrammatically,

$$\begin{tikzcd}
& c' \in [\![\mathsf{com}]\!](\{x,y,z\}) & \\
c_1 \in [\![\mathsf{com}]\!](\{x,y\}) \arrow[ur, "\mathsf{expand}(\mathsf{inc}_{\{x,y\},\{x,y,z\}})_{\mathsf{com}}"] & & c_2 \in [\![\mathsf{com}]\!](\{y,z\}) \arrow[ul, "\mathsf{expand}(\mathsf{inc}_{\{y,z\},\{x,y,z\}})_{\mathsf{com}}"']
\end{tikzcd}$$

Surprisingly, the converse still holds in this case: if the expansions of $c_1$ and $c_2$ to $\{x,y,z\}$ are the same, they should originate from the same command $c$ in the world $\{y\}$. The reason is as follows. The expansions of $c_1$ and $c_2$ to $\{x,y,z\}$ don't access $z$ and $x$, respectively. Since the expansions of $c_1$ and $c_2$ are the same, the same expanded command doesn't access either $x$ or $z$. Therefore, the same expansion must be an expansion of the command $c$ from the world $\{y\}$, which indicates that the commands $c_1$ and $c_2$ must be so, too. This converse gives rise to the answer of the question raised in the beginning.

The converse implication is not just a property of a command. All other imperative types ($\mathsf{exp}$ and $\mathsf{var}$) satisfy it, too. This feature of imperative types is what we want to characterize by a sheaf condition in this section.[1]More precisely,

**Definition 11 (Atomic-Sheaf Condition)** *For a type $\theta$, $[\![\theta]\!]$ satisfies* the atomic-sheaf condition *iff*

> *For a world $W$, its two subworlds $W_1, W_2 \subseteq W$ and $p_i \in [\![\theta]\!](W_i)$ for $i = 1,2$, if $\mathsf{expand}(\mathsf{inc}_{W_1,W})_\theta(p_1) = \mathsf{expand}(\mathsf{inc}_{W_2,W})_\theta(p_2)$, then $p_1$ and $p_2$ originate from a unique $p$ in $[\![\theta]\!](W_1 \cap W_2)$: for $i = 1,2$, $p_i = \mathsf{expand}(\mathsf{inc}_{W_1 \cap W_2, W_i})_\theta(p)$.*

[1]This is a sheaf condition for an atomic Grothendieck topology shown in Chapter 4. Therefore, we call it the atomic-sheaf condition.

IA is a higher-order imperative programming language. Often, the intuitions on base imperative types aren't generalized to function types. Characterizing a property of imperative types as a sheaf condition solves such a problem. A well-known result of sheaf theory guarantees that if the interpretation of base types satisfies a sheaf condition, so does the interpretation of all types constructed from base types using function and product type constructors. Since all base IA types satisfy the atomic-sheaf condition, all IA types also satisfy the condition.

**Theorem 12** *For every IA type $\theta$, $[\![\theta]\!]$ satisfies the atomic-sheaf condition.*

Proof of Theorem 2 is essentially based on this property. For any $p \in [\![\theta]\!](W)$, the family $\{x_{W'} \mid W' \subseteq W \text{ and } \mathsf{expand}(\mathsf{inc}_{W',W})_\theta(x_{W'}) = x\}$ is finite. Therefore, by applying Theorem 12 the finite number of times, it is possible to obtain a unique $x_{W''}$ in $W''$ which can be expanded to every $x_{W'}$ including $x$. $W''$ becomes the minimal resource.

### 3.3.2 The Atomic-Sheaf Condition in the Parametricity Model

In the parametricity model, the interpretation of a function type has a stronger uniformity condition, which makes the very advantage of characterizing a property by a sheaf condition be lost: the interpretation of a function type does not necessarily satisfy a sheaf condition even if its two constituent types do. One of the goals of considering a more general framework, where sheave theory and parametricity theory are combined, is to overcome such a problem: a function type satisfies an "extended" sheaf condition when its two constituent types do. The precise formulation of an "extended" sheaf condition in the framework of parametric sheaves is parameterized by a notion of "topology," and it is presented in Chapter 4. In this section, we present in more accessible form a particular "extended" sheaf condition, which captures certain computational aspect of expansions. In the remainder of the section, we drop the adjective "extended" from "extended sheaf condition" and simply call it "sheaf condition."

In the parametricity model, each type has two components: a parameterized set and a parameterized relation. Therefore, a sheaf condition should deal with the relation part as well as the set part. We use a particular sheaf condition in this section, which we call again the atomic-sheaf condition. The atomic-sheaf condition has the same requirement for sets as in the previous section. By following the same style in the previous section, we derive a property on relations which is captured by the atomic-sheaf condition.

Consider two commands $c_0$ and $c_1$ in a world $\{x\}$. Let $R$ be a relation on $S(\{x\})$. Suppose our goal is to show that $c_0$ and $c_1$ map $R$-related states to $R$-related states ($c_0 \, [\![\mathsf{com}]\!](R) \, c_1$) for a nonempty relation $R$. Let's start from observing what properties are going to hold when the goal is true. Suppose that $c_0$ and $c_1$ are $[\![\mathsf{com}]\!](R)$-related. Since an expanded command acts as an identity for newly allocated variables and just the same for the other already-existing variables as the original command, $\mathsf{expand}(\mathsf{inc}_{\{x\},\{x,y\}})c_0$ and $\mathsf{expand}(\mathsf{inc}_{\{x\},\{x,y\}})c_1$ map $R \times P$-related states to $R \times P$-related states for all nonempty relations $P$ on $S(\{y\})$.

Achieving the goal is based on a surprising fact: the converse to the above[2] also becomes true. Namely, if $\mathsf{expand}(\mathsf{inc}_{\{x\},\{x,y\}})c_0$ and $\mathsf{expand}(\mathsf{inc}_{\{x\},\{x,y\}})c_1$ map $R \times P$-related states to $R \times P$-related states for *some* nonempty relation $P \colon S(\{y\}) \leftrightarrow S(\{y\})$, then $c_0 [\![\mathsf{com}]\!](R) c_1$ holds, too. The reason is that the expansion of a command to a bigger stack doesn't change its behavior on already existing variables. And this converse is what the atomic-sheaf condition requires on relations.[3] More precisely,

**Definition 13 (Atomic-Sheaf Condition on Relations)** *For an IA type $\theta$, $[\![\theta]\!]$ satisfies* the atomic-sheaf condition on relations *iff*

> *for any non-identity nonempty relation $R \colon S(W_0) \leftrightarrow S(W_1)$ and a nonempty $P \colon S(L_0) \leftrightarrow S(L_1)$ such that $W_0 \cap L_0 = W_1 \cap L_1 = \emptyset$, if $\mathsf{expand}(\mathsf{inc}_{W_0, W_0 \cup L_0})_\theta(p_0) \;\; [\![\theta]\!](R \times P) \;\; \mathsf{expand}(\mathsf{inc}_{W_1, W_1 \cup L_1})_\theta(p_1)$, then $p_0 \, [\![\theta]\!](R) \, p_1$.*

All base imperative types can easily be shown to satisfy the above condition. The main advantage of using a sheaf condition in the parametricity model to capture a property of computations is the same as in the presheaf model: if all base types satisfy a sheaf condition, then all types constructed using function and product type constructors also satisfy the condition. And since only the requirement on sets of the atomic-sheaf condition is enough to show the existence of minimal resources, Theorem 9 follows easily.

[2] in fact stronger than the converse

[3] Strictly speaking, the atomic-sheaf condition on relations doesn't entirely capture the converse because it requires $R$ to be a non-identity relation. Informally, such a mismatch comes from the fact that the identity relation is treated specially in a parametricity framework; for instance, the interpretation of every IA type is required to map it to another identity relation.

# CHAPTER 4

## Parametric Sheaves

In this chapter, we develop the notion of *parametric sheaves*. It is obtained by combining parametricity theory and sheaf theory, which have been used for modelling data abstraction and symmetric noninterference, respectively. As a consequence, parametric sheaves provide a framework for explaining data abstraction and symmetric noninterference at the same time, which is illustrated by the semantics of symmetric noninterference and IA in this chapter.

Mathematically, parametric sheaves are Lawvere-Tierney sheaves internal to a certain topos, which we call *parametric-presheaf topos*. The interesting point is that they have an external characterization just as Lawvere-Tierney sheaves internal to a presheaf topos. In fact, the external characterization of parametric sheaves generalizes that of sheaves in a presheaf topos because a presheaf topos can be regarded as a particular parametric-presheaf topos.

This chapter starts with reviews of sheaf theory and parametricity theory. Then, we present a class of topoi, parametric-presheaf topoi, and study their topos structure. After that, the notion of parametric sheaves is developed, and we show that the notion coincides with the internal characterization. Finally, the semantics of symmetric noninterference and IA is given as an example.

## 4.1 Background

In this section, we briefly review sheaf theory and parametricity theory. The familiarity with the basic notions from category theory, such as category, functor and natural transformation, is assumed. The detailed exposition of parametricity theory can be found in [53, 59]. And the standard texts on sheaf theory are [32, 25].

### 4.1.1 Sheaf Theory

Originally, a sheaf condition is meant to capture locality. This can be best illustrated by continuous functions from an open subset of $\mathbf{R}$ to $\mathbf{R}$. All continuous functions satisfy two properties:

- *restriction*: If $f: O \to \mathbf{R}$ is a continuous function, its restriction to an open subset $O'$ of $O$ is also continuous.
- *collation*: Let $\{O_i\}_{i \in I}$ be an open cover of $O$, i.e., $\bigcup_{i \in I} O_i = O$. And let $f_i: O_i \to \mathbf{R}$ be continuous for each $i \in I$. If all of $f_i$ are "consistent," i.e., for all $i, j \in I$, $f_i|_{O_i \cap O_j} = f_j|_{O_i \cap O_j}$, then they can be patched together to form one continuous function $f$ from $O$ to $R$: there exists a unique $f: O \to \mathbf{R}$ such that $f|_{O_i} = f_i$.

Essentially, it is these two properties that characterize sheaves. A "consistent" family $\{f_i\}_{i \in I}$ in the collation property is called a *matching* for $\{O_i\}_{i \in I}$, and the patched function $f$ of $\{f_i\}_{i \in I}$ is called an *amalgamation*. The collation property can be restated as: for every open cover and every matching for it, there exists a unique amalgamation for the matching.

The definition of sheaves in a category is obtained by generalizing the meaning of open covers. However, the object of our generalization is not just open covers but down-closed ones. A collection of open sets $\{U_i\}_{i \in I}$ is down-closed iff for any open subset $U'$ of $U_i$, there exists $j \in I$ such that $U' = U_j$. A down-closed collection of open subsets of $O$ gets generalized to a *sieve*. A sieve on an object $C$ is a set of morphisms into $C$ which are closed under right composition. Formally,

**Definition 14 (sieve)** *For an object $C$ in a category $\mathbf{C}$, a set $S$ of morphisms is a* sieve *on $C$ iff*

1. *for all $f \in S$, $cod(f) = C$; and*
2. *for all $f: D \to C \in S$ and $g: E \to D$, $f \circ g \in S$.*

Note that for a sieve $S$ on $C$ and $f: D \to C$, $\{g \mid f \circ g \in S\}$ is also a sieve. We denote this sieve by $f^*(S)$. Also, notice that $t_C = \{f \mid cod(f) = C\}$ is a sieve on $C$. We call this sieve the *maximal sieve* and use the letter $t$ to denote it.

A Grothendieck topology specifies which sieves "cover" an object in a category and is a generalization of down-closed open covers in a topological space. For a topological space, down-closed open covers have the following properties:

1. *maximality*: for every open set $U$, all open subsets of $U$ cover $U$.
2. *stability*: if $\{U_i\}_{i \in I}$ covers $U$ and $V$ is an open subset of $U$, then $\{U_i \mid i \in I \wedge U_i \subseteq V\}$ covers $V$.
3. *transitivity*: if $\{U_i\}_{i \in I}$ covers $U$ and for all $U_i$, $\{V_j \mid j \in K \wedge V_j \subseteq U_i\}$ covers $U_i$, then $\{V_j\}_{j \in K}$ covers U.

The requirements for a collection of sieves to be a Grothendieck topology are obtained from the above.

**Definition 15 (Grothendieck topology, site)** *A* Grothendieck topology *on a category* $\mathbf{C}$ *is a function $J$ which assigns to each object $C$ of* $\mathbf{C}$ *a collection $J(C)$ of sieves on $C$ in such a way that*

1. *(maximality) the maximal sieve $t_C = \{f \mid cod(f) = C\}$ is in $J(C)$;*
2. *(stability) if $S \in J(C)$, then $h^*(S) \in J(D)$ for any arrow $h: D \to C$; and*
3. *(transitivity) if $S \in J(C)$ and $S'$ is a sieve on $C$ such that $h^*(S') \in J(D)$ for all $h: D \to C$ in $S$, then $S' \in J(C)$.*

*Each sieve in $J(C)$ is called a* covering sieve *or* cover*, and a category with a Grothendieck topology, $(\mathbf{C}, J)$, is called a* site*.*

A presheaf $F$ on $\mathbf{C}$ is a functor from $\mathbf{C}^{op}$ to $\mathbf{Set}$. For any $f: D \to C$ in $\mathbf{C}$ and $x \in F(C)$, we write $x \cdot f$ to denote $F(f)(x)$. Notice that the very existence of a function $F(f)$ corresponds to the restriction property of continuous functions. In order to be a sheaf on a site $(\mathbf{C}, J)$, $F$ should satisfy the sheaf condition given by the site, which corresponds to the collation property. The natural generalization of matching and amalgamation from the case of continuous functions gives rise to this sheaf condition:

**Definition 16 (matching, amalgamation, sheaf)** *For a site* $(\mathbf{C}, J)$ *and a covering sieve* $S \in J(C)$ *on an object* $C$*, a* matching *for* $S$ *of elements of* $F: \mathbf{C}^{op} \to \mathbf{Set}$ *is a family* $\{x_f \in F(dom(f))\}_{f \in S}$ *such that*

$$x_f \cdot g = x_{f \circ g} \quad \text{for all } f \in S \text{ and } g \text{ such that } cod(g) = dom(f)$$

*And an* amalgamation *of a matching* $\{x_f \in F(dom(f))\}_{f \in S}$ *is* $x \in F(C)$ *such that* $x \cdot f = x_f$ *for all* $f \in S$*. A presheaf* $F: \mathbf{C}^{op} \to \mathbf{Set}$ *is a* sheaf *on* $(\mathbf{C}, J)$ *iff for every covering sieve and every matching for the covering sieve, there exists a unique amalgamation for the matching.*

For any site $(\mathbf{C}, J)$, sheaves on the site have a more than enough structure to interpret the typed lambda calculus:

**Proposition 17** *For any site* $(\mathbf{C}, J)$*, sheaves on the site form a topos.*

### 4.1.2 Parametricity and Reflexive Graphs

Parametricity theory is a study on the notion of uniformity. The theory starts from Reynolds' influential paper on relational parametricity [66], where he used relations to characterize a *uniform* family. For instance, a family $\{f_A: A \to A\}_{A \in Set}$ of endo-functions indexed by a set is uniform according to Reynolds' relational parametricity iff for all relations $R: A \leftrightarrow B$,

$$x_A \, R \, x_B \Longrightarrow f_A(x_A) \, R \, f_B(x_B) \text{ for all } x_A \in A \text{ and } x_B \in B.$$

Intuitively, the condition formalizes that a uniform family doesn't depend on a particular indexing set because otherwise, there is a relation which can't be preserved by elements in the family. Indeed, when a family $\{f_A\}_{A \in Set}$ is uniform in the above sense, all members in the family behave the same way: for all indexing sets $A$, $f_A$ must be the identity function. The reason is that if $f_A$ is not an identity function so that $f_A(a) \neq a$, then $f_A$ is not related to itself by $[\{(a, a)\} \to \{(a, a)\}]$.

In the following developments, Reynolds' idea is generalized from sets to arbitrary categories especially using reflexive graphs of categories [71, 53]. A *reflexive graph* $\mathbf{G}$ consists of two categories:$\mathbf{G}_v$ and $\mathbf{G}_e$; and three functors in-between:two projections $\partial_0, \partial_1: \mathbf{G}_e \to \mathbf{G}_v$ and identity $\mathbf{I}: \mathbf{G}_v \to \mathbf{G}_e$ such that $\partial_0 \circ \mathbf{I} = \partial_1 \circ \mathbf{I} = \mathsf{Id}_{\mathbf{G}_v}$. Diagrammatically,

$$\begin{tikzcd} \mathbf{G}_e \arrow[r, shift left=2, "\partial_0"] \arrow[r, shift right=2, "\partial_1"'] & \mathbf{G}_v \arrow[l, "\mathbf{I}" description] \end{tikzcd}$$

$\mathbf{G}_v$ and $\mathbf{G}_e$ are called *vertex* and *edge* categories, respectively. An object and a morphism in $\mathbf{G}_v$ are generalizations of a set and a function, and they are called *vertex* and *vertex morphism*. An object in $\mathbf{G}_e$ is an generalization of a binary relation and called *edge*. Informally, a morphism $\tau: P \to R$ in $\mathbf{G}_e$, which we call *edge morphism*, is a generalization of a statement in sets which says that two functions ($\partial_0(\tau)$ and $\partial_1(\tau)$) map related elements (by $P$) to related elements (by $R$). We only consider reflexive graphs whose identity functor satisfies the *identity condition* in [27], which is that the identity functor $\mathbf{I}$ is full and faithful. And a reflexive graph is called *small* when its vertex and edge categories are small.

For an edge $R$, we write $R: V_0 \leftrightarrow V_1$ to denote that $\partial_0(R) = V_0$ and $\partial_1(R) = V_1$. And a morphism in the edge category is usually denoted by the *parametricity square*:

$$\begin{tikzcd} U_0 \arrow[r, "f_0"] \arrow[d, leftrightarrow, "P"'] \arrow[dr, phantom, "\tau"] & V_0 \arrow[d, leftrightarrow, "R"] \\ U_1 \arrow[r, "f_1"] & V_1 \end{tikzcd}$$

which denotes an edge morphism $\tau$ from $P: U_0 \leftrightarrow U_1$ to $R: V_0 \leftrightarrow V_1$ such that $\partial_i(\tau) = f_i$ for $i = 0, 1$.

**Example 18 (Rel)** A canonical reflexive graph is **Rel**. Its vertex and edge categories are

- vertex category $\mathbf{Rel}_v$: the category of sets (**Set**);
- edge category $\mathbf{Rel}_e$:

  objects — $(A_0, R, A_1)$ where $R$ is a relation between $A_0$ and $A_1$,

  morphisms — $(f_0, f_1): (A_0, R, A_1) \to (B_0, S, B_1)$ such that $f_0: A_0 \to B_0$ and $f_1: A_1 \to B_1$ and $a_0 \, R \, a_1 \Rightarrow f_0(a_0) \, S \, f_1(a_1)$ for all $a_0 \in A_0$ and $a_1 \in A_1$ ($f_0 \, [R \to S] \, f_1$).

We normally write $R$ for $(A_0, R, A_1)$ when there is no confusion. Projection functors as well as the identity functor are given as follows:

- $\partial_i(A_0, R, A_1) = A_i$ and $\partial_i(f_0, f_1) = f_i$ for $i = 0, 1$;
- $\mathbf{I}(A) = (A, \mathbf{\Delta}_A, A)$ and $\mathbf{I}(f) = (f, f)$ where $\mathbf{\Delta}_A$ is the diagonal relation on $A$, i.e., $a \mathbf{\Delta}_A a'$ iff $a = a'$.

Notice that in **Rel**, $\langle \partial_0, \partial_1 \rangle \colon \mathbf{G}_e \to \mathbf{G}_v \times \mathbf{G}_v$ is faithful. Because of this, edge morphisms in **Rel** are usually denoted just mentioning its projected parts, and in most cases, the parametricity square is used to state the existence of an edge morphism. For instance, we say the following parametricity square holds

$$\begin{tikzcd} U_0 \arrow[r, "f_0"] \arrow[d, leftrightarrow, "P"'] & V_0 \arrow[d, leftrightarrow, "R"] \\ U_1 \arrow[r, "f_1"] & V_1 \end{tikzcd}$$

when there exists an edge morphism in **Rel** with $f_0$ and $f_1$ as its two projections, equivalently, for any $u_0 \in U_0$ and $u_1 \in U_1$, $u_0 \, P \, u_1 \implies f_0(u_0) \, R \, f_1(u_1)$.

**Example 19 (Span)** A second example is **Span**, which will play an important role in this chapter.

- vertex category $\mathbf{Span}_v$: the category of sets (**Set**);
- edge category $\mathbf{Span}_e$:

  objects — a pair of functions $A_0 \overset{\rho_0}{\leftarrow} R \overset{\rho_1}{\rightarrow} A_1$ where $\rho_i$ is a function from a set $R$ to a set $A_i$ for $i = 0, 1$,

  morphisms — a span morphism, i.e., a triple of functions $(f_0, f_w, f_1)$ making the following diagram commute:

$$\begin{tikzcd} & A_0 \arrow[rr, "f_0"] & & B_0 \\ R \arrow[ur, "\rho_0"] \arrow[rr, "f_w"] \arrow[dr, "\rho_1"'] & & S \arrow[ur, "\varrho_0"] \arrow[dr, "\varrho_1"] & \\ & A_1 \arrow[rr, "f_1"'] & & B_1 \end{tikzcd}$$

- $\partial_i(A_0 \overset{\rho_0}{\leftarrow} R \overset{\rho_1}{\rightarrow} A_1) = A_i$ and $\partial_i(f_0, f_w, f_1) = f_i$;
- $\mathbf{I}(A) = A \overset{\mathsf{id}_A}{\leftarrow} A \overset{\mathsf{id}_A}{\rightarrow} A$ and $\mathbf{I}(f) = (f, f, f)$.

Spans are rather similar to relations, with the main difference being that we also get to encode "witnesses" for relating elements. Note that every relation can be regarded as a span $A_0 \leftarrow$

$R \to A_1$ by using projections of $R \subseteq A_0 \times A_1$ as arrows. And, there is a functor which picks up the "witness" set for each span:

$$\mathsf{wit}\colon \mathbf{Span}_e \to \mathbf{Set}$$

that sends a span $A_0 \overset{\rho_0}{\leftarrow} R \overset{\rho_1}{\rightarrow} A_1$ to the witness set $R$ and a span morphism $(f_0, f_w, f_1)$ to $f_w$.

**Example 20 ($\mathbf{G}^{op}$)** For a reflexive graph $\mathbf{G}$, the componentwise opposite of $\mathbf{G}$ is also a reflexive graph; the vertex and edge categories of $\mathbf{G}^{op}$ are $\mathbf{G}_v^{op}$ and $\mathbf{G}_e^{op}$, and the opposites of constituent functors, $\partial_0^{op}$, $\partial_1^{op}$ and $\mathbf{I}^{op}$, become projection and identity functors.

Notice that all example reflexive graphs satisfy the identity condition: reflexive graphs **Rel** and **Span** satisfy the identity condition, and if $\mathbf{G}$ satisfies the identity condition, so does $\mathbf{G}^{op}$.

A mapping between two reflexive graphs with the identity condition is called an RG-functor. An RG-functor consists of two functors for vertex and edge categories which commute with two projection and identity functors upto an isomorphism. Formally,

**Definition 21 (RG-functor)** *For reflexive graphs $\mathbf{G}$ and $\mathbf{E}$ with the identity condition, an* RG-functor *$F$ from $\mathbf{G}$ to $\mathbf{E}$ is a pair of functors $(F_v\colon \mathbf{G}_v \to \mathbf{E}_v, F_e\colon \mathbf{G}_e \to \mathbf{E}_e)$, such that for $i = 0, 1$, $\partial_i \circ F_e = F_v \circ \partial_i$;*

$$\begin{tikzcd} \mathbf{G}_e \arrow[r, "F_e"] \arrow[d, "\partial_i"'] & \mathbf{E}_e \arrow[d, "\partial_i"] \\ \mathbf{G}_v \arrow[r, "F_v"'] & \mathbf{E}_v \end{tikzcd}$$

*and there exists a natural isomorphism $\eta\colon F_e \circ \mathbf{I} \to \mathbf{I} \circ F_v$ such that $\partial_i \eta = \mathsf{id}_{F_v}$ for $i = 0, 1$ :*

$$\begin{tikzcd} \mathbf{G}_e \arrow[r, "F_e"] & \mathbf{E}_e \arrow[r, "\partial_i"] & \mathbf{E}_v \\ \mathbf{G}_v \arrow[u, "\mathbf{I}"] \arrow[r, "F_v"'] \arrow[ur, phantom, "\Downarrow \eta"] & \mathbf{E}_v \arrow[u, "\mathbf{I}"'] & \end{tikzcd} \quad = \quad \begin{tikzcd} \mathbf{G}_v \arrow[r, bend left, "F_v"] \arrow[r, bend right, "F_v"'] \arrow[r, phantom, "\Downarrow \mathsf{id}_{F_v}"] & \mathbf{E}_v \end{tikzcd}$$

As in the case of functors, we write $F\colon \mathbf{G} \to \mathbf{E}$ to indicate that $F$ is an RG-functor from $\mathbf{G}$ to $\mathbf{E}$.

The definition of RG-functors can be simplified by using the fact that $\mathbf{G}, \mathbf{E}$ satisfy the identity condition. We first note that the naturality of a family of isomorphisms $\{\eta_V\}_{V \in \mathbf{G}_v}$ comes for free:

**Lemma 22** *For reflexive graphs* $\mathbf{G}$ *and* $\mathbf{E}$ *with the identity condition, and a pair of functors* $(F_v \colon \mathbf{G}_v \to \mathbf{E}_v, F_e \colon \mathbf{G}_e \to \mathbf{E}_e)$ *such that* $\partial_i \circ F_e = F_v \circ \partial_i$ *for* $i = 0, 1$*, let* $\{\eta_V \colon F_e(\mathbf{I}(V)) \to \mathbf{I}(F_v(V))\}_{V \in \mathbf{G}_v}$ *be a family of isomorphisms in* $\mathbf{E}_e$*. If* $\partial_i(\eta_V) = \mathsf{id}_{F_v(V)}$ *for each vertex* $V$ *in* $\mathbf{G}$ *and* $i = 0, 1$*, the family* $\{\eta_V\}$ *satisfies the naturality condition.*

*Proof:* Let $f \colon V \to W$ be a vertex morphism in $\mathbf{G}$. We show that $\eta_W \circ F_e(\mathbf{I}(f)) \circ \eta_V^{-1} = \mathbf{I}(F_v(f))$:

$$\begin{tikzcd}
F_e(\mathbf{I}(V)) \arrow[r, "F_e(\mathbf{I}(f))"] & F_e(\mathbf{I}(W)) \arrow[d, "\eta_W"] \\
\mathbf{I}(F_v(V)) \arrow[u, "\eta_V^{-1}"] \arrow[r, "\mathbf{I}(F_v(f))"] & \mathbf{I}(F_v(W))
\end{tikzcd}$$

Since $\mathbf{I}$ is full by the identity condition, there is $k \colon F_v(V) \to F_v(W)$ such that $\eta_W \circ F_e(\mathbf{I}(f)) \circ \eta_V^{-1} = \mathbf{I}(k)$. Then, the conclusion follows because

$$\begin{aligned}
& k \\
= \quad & \\
& \partial_0(\mathbf{I}(k)) \\
= \quad & \qquad \because \mathbf{I}(k) = \eta_W \circ F_e(\mathbf{I}(f)) \circ \eta_V^{-1} \\
& \partial_0(\eta_W \circ F_e(\mathbf{I}(f)) \circ \eta_V^{-1}) \\
= \quad & \\
& \partial_0(\eta_W) \circ \partial_0(F_e(\mathbf{I}(f))) \circ \partial_0(\eta_V^{-1}) \\
= \quad & \\
& \mathsf{id}_{F_v(W)} \circ F_v(\partial_0(\mathbf{I}(f))) \circ (\partial_0(\eta_V))^{-1} \\
= \quad & \\
& F_v(\partial_0(\mathbf{I}(f))) \circ \mathsf{id}_{F_v(V)}^{-1} \\
= \quad & \\
& F_v(f)
\end{aligned}$$

□

Because of the lemma, RG-functors can be characterized in a simpler way:

> a pair of functors $(F_v, F_e)$ is an RG-functor iff $F_v \circ \partial_i = \partial_i \circ F_e$ and for all vertex $V$ in $\mathbf{G}$, $F_e(\mathbf{I}(V))$ is isomorphic to $\mathbf{I}(F_v(V))$ by an isomorphism which is mapped to $\mathsf{id}_{F_v(V)}$ by $\partial_i$ for $i = 0, 1$.

In the rest of this chapter, we use the above characterization without further notice.

A natural transformation can be lifted to reflexive graph setting, too.

**Definition 23 (parametric natural transformation)** *For RG-functors $F, G\colon \mathbf{G} \to \mathbf{E}$, a* parametric natural transformation *$\mu\colon F \to G$ consists of two natural transformations $\mu_v\colon F_v \to G_v$ and $\mu_e\colon F_e \to G_e$ such that $\partial_i \mu_e = \mu_v \partial_i$ for $i = 0, 1$.*

Notice that the definition of a parametric natural transformation doesn't involve identity functors in reflexive graphs $\mathbf{G}$ and $\mathbf{E}$. However, the following lemma shows that identity functors are preserved automatically in a certain sense.

**Lemma 24** *For RG-functors $F, G\colon \mathbf{G} \to \mathbf{E}$ and a parametric natural transformation $\mu\colon F \to G$, let $\eta\colon F_e \circ \mathbf{I} \to \mathbf{I} \circ F_v$ and $\eta'\colon G_e \circ \mathbf{I} \to \mathbf{I} \circ G_v$ be natural isomorphisms such that $\partial_i \eta = \mathsf{id}_{F_v}$ and $\partial_i \eta' = \mathsf{id}_{G_v}$ for $i = 0, 1$. Then, $\eta' \circ (\mu_e \mathbf{I}) = (\mathbf{I}\mu_v) \circ \eta$:*

$$
\begin{tikzcd}
\mathbf{G}_e \arrow[r, bend left, "F_e"] \arrow[r, bend right, "G_e"'] \arrow[r, phantom, "\Downarrow \mu_e"] & \mathbf{E}_e \\
\mathbf{G}_v \arrow[u, "\mathbf{I}"] \arrow[r, "G_v"'] \arrow[ur, phantom, "\Downarrow \eta'"] & \mathbf{E}_v \arrow[u, "\mathbf{I}"']
\end{tikzcd}
\quad = \quad
\begin{tikzcd}
\mathbf{G}_e \arrow[r, "F_e"] & \mathbf{E}_e \\
\mathbf{G}_v \arrow[u, "\mathbf{I}"] \arrow[r, bend left, "F_v"] \arrow[r, bend right, "G_v"'] \arrow[r, phantom, "\Downarrow \mu_v"] \arrow[ur, phantom, "\Downarrow \eta"] & \mathbf{E}_v \arrow[u, "\mathbf{I}"']
\end{tikzcd}
$$

*Proof:* Let $V$ be a vertex in $\mathbf{G}_v$. It suffices to show that the following diagram commutes:

$$
\begin{tikzcd}
F_e(\mathbf{I}(V)) \arrow[r, "(\mu_e)_{\mathbf{I}(V)}"] & G_e(\mathbf{I}(V)) \arrow[d, "\eta'_V"] \\
\mathbf{I}(F_v(V)) \arrow[u, "\eta_V^{-1}"] \arrow[r, "\mathbf{I}((\mu_v)_V)"] & \mathbf{I}(G_v(V))
\end{tikzcd}
$$

Since $\mathbf{I}$ is full, there is a morphism $k\colon F_v(V) \to G_v(V)$ in $\mathbf{E}_v$ such that $\mathbf{I}(k) = \eta'_V \circ (\mu_e)_{\mathbf{I}(V)} \circ \eta_V^{-1}$. Then, the conclusion follows because

$$
\begin{aligned}
& k \\
= \quad & \\
& \partial_0(\mathbf{I}(k)) \\
= \quad & \qquad \because \mathbf{I}(k) = \eta'_V \circ (\mu_e)_{\mathbf{I}(V)} \circ \eta_V^{-1} \\
& \partial_0(\eta'_V \circ (\mu_e)_{\mathbf{I}(V)} \circ \eta_V^{-1}) \\
= \quad & \\
& \partial_0(\eta'_V) \circ \partial_0((\mu_e)_{\mathbf{I}(V)}) \circ \partial_0(\eta_V^{-1}) \\
= \quad & \\
& \mathsf{id}_{G_v(V)} \circ (\mu_v)_{\partial_0(\mathbf{I}(V))} \circ (\partial_0(\eta_V))^{-1} \\
= \quad & \\
& (\mu_v)_{\partial_0(\mathbf{I}(V))} \circ (\mathsf{id}_{F_v(V)})^{-1} \\
= \quad & \\
& (\mu_v)_V
\end{aligned}
$$

$\square$

## 4.2 Parametric Presheaves

In the world of categories a presheaf is taken to be a functor of type $\mathbf{C}^{op} \to \mathbf{Set}$. The category of presheaves $\mathbf{Set}^{\mathbf{C}^{op}}$ forms a topos. Sheaves are then defined to be presheaves with special properties.

To duplicate these constructions in the world of reflexive graphs, we first need a canonical reflexive graph that plays the same role as $\mathbf{Set}$ does in the case of categories. One immediate candidate for the canonical reflexive graph is $\mathbf{Rel}$, whose edges are binary relations between sets. Unfortunately, $\mathbf{Rel}^{\mathbf{G}^{op}}$ is not in general a topos. So, this immediate choice has a shortcoming. In particular, this means that the Lawvere-Tierney notion of sheaves is not available to $\mathbf{Rel}^{\mathbf{G}^{op}}$. A second candidate for the canonical reflexive graph is $\mathbf{Span}$, whose edges are spans over $\mathbf{Set}$. It turns out that $\mathbf{Span}^{\mathbf{G}^{op}}$ is always a topos and, therefore, serves our purpose. In this section, we examine the topos structure of this category.

**Definition 25 (parametric presheaf)** *Given a reflexive graph with the identity condition* $\mathbf{G}$*, a* parametric presheaf *over* $\mathbf{G}$ *is an RG-functor* $\mathbf{G}^{op} \to \mathbf{Span}$*. The category of parametric presheaves with parametric natural transformations as morphisms is denoted* $\mathbf{Span}^{\mathbf{G}^{op}}$.

Since each parametric presheaf $F$ on $\mathbf{G}$ preserves projections ($\partial_i \circ F_e = F_v \circ \partial_i$), $F$ can be completely determined by $F_v$, $\mathsf{wit} \circ F_e$ and two mappings in $F_e(R)$ from the set of witnesses $\mathsf{wit}(F_e(R))$ to $F_v(\partial_0(R))$ and $F_v(\partial_1(R))$ for each edge $R$ in $\mathbf{G}$. In the rest of this chapter, whenever we construct a parametric presheaf $F$, we just specify those three: $F_v$, $\mathsf{wit} \circ F_e$, which we denote by $F_w$, and two such mappings in $F_e(R)$ for each edge $R$, which we call *projections in* $F_e(R)$. Similarly, by the same reason, each parametric natural transformation $\mu$ in $\mathbf{Span}^{\mathbf{G}^{op}}$ is presented by $\mu_v$ and $\mathsf{wit}\mu_e$, which we denote by $\mu_w$.

From now on, we assume that the reflexive graph $\mathbf{G}$ is *small*, i.e., both $\mathbf{G}_v$ and $\mathbf{G}_e$ are small categories. When $\mathbf{G}$ is a small reflexive graph satisfying the identity condition, we define two functors:

$$\begin{aligned} \mathcal{Y}_v &: \mathbf{G}_v \to \mathbf{Span}^{\mathbf{G}^{op}} \\ \mathcal{Y}_e &: \mathbf{G}_e \to \mathbf{Span}^{\mathbf{G}^{op}} \end{aligned}$$

which behave like an Yoneda embedding in category theory. We explain more on this later. $\mathcal{Y}_v$ is a functor from $\mathbf{G}_v$ to $\mathbf{Span}^{\mathbf{G}^{op}}$ given by:

- for each object $V$ in $\mathbf{G}_v$, an RG-functor $\mathcal{Y}_v(V)\colon \mathbf{G}^{op} \to \mathbf{Span}$ is defined as follows:
  - when $U$ and $f$ are a vertex and a vertex morphism, respectively, in $\mathbf{G}$,

$$\begin{aligned} \mathcal{Y}_v(V)_v(U) &= \mathsf{Hom}_{\mathbf{G}_v}[U, V] \\ \mathcal{Y}_v(V)_v(f) &= \mathsf{Hom}_{\mathbf{G}_v}[f, V] \end{aligned}$$

    and

  - when $R\colon U_0 \leftrightarrow U_1$ is an edge and $\varphi$ is an edge morphism in $\mathbf{G}$,

$$\begin{aligned} \mathcal{Y}_v(V)_w(R) &= \mathsf{Hom}_{\mathbf{G}_e}[R, \mathbf{I}(V)] \\ \mathcal{Y}_v(V)_w(\varphi) &= \mathsf{Hom}_{\mathbf{G}_e}[\varphi, \mathbf{I}(V)] \end{aligned}$$

    and two projections in $\mathcal{Y}_v(V)_e(R)$ are restrictions of projection functors $\partial_0, \partial_1$ in $\mathbf{G}$

$$\begin{aligned} \partial_0|_{\mathsf{Hom}_{\mathbf{G}_e}[R,\mathbf{I}(V)]} &: \mathsf{Hom}_{\mathbf{G}_e}[R, \mathbf{I}(V)] \to \mathsf{Hom}_{\mathbf{G}_v}[U_0, V] \\ \partial_1|_{\mathsf{Hom}_{\mathbf{G}_e}[R,\mathbf{I}(V)]} &: \mathsf{Hom}_{\mathbf{G}_e}[R, \mathbf{I}(V)] \to \mathsf{Hom}_{\mathbf{G}_v}[U_1, V] \end{aligned}$$

- for each morphism $f\colon V \to W$, $\mathcal{Y}_v(f)$ is a parametric natural transformation from $\mathcal{Y}_v(V)$ to $\mathcal{Y}_v(W)$ given by:
  - for each vertex $U$ in $\mathbf{G}_v$, $(\mathcal{Y}_v(f)_v)_U$ is a function from $\mathsf{Hom}_{\mathbf{G}_v}[U,V](=\mathcal{Y}_v(V)_v(U))$ to $\mathsf{Hom}_{\mathbf{G}_v}[U,W](=\mathcal{Y}_v(W)_v(U))$ given by $f\circ -$; and
  - for each edge $R$, $(\mathcal{Y}_v(f)_w)_R$ is a function from $\mathsf{Hom}_{\mathbf{G}_e}[R,\mathbf{I}(V)](=\mathcal{Y}_v(V)_w(R))$ to $\mathsf{Hom}_{\mathbf{G}_e}[R,\mathbf{I}(W)](=\mathcal{Y}_v(W)_w(R))$ given by $\mathbf{I}(f)\circ -$.

The construction of $\mathcal{Y}_e$ is more complicated. In particular, $\mathcal{Y}_e(R)$ involves taking disjoint unions and quotients by certain equivalence relations in order to make $\mathcal{Y}_e(R)$ have the identity extension property, i.e., for every vertex $V$ in $\mathbf{G}^{op}$, $\mathcal{Y}_e(R)_e(\mathbf{I}(V))$ is isomorphic to $\mathbf{I}(\mathcal{Y}_e(R)_v(V))$ by an isomorphism which is projected into an identity function by both $\partial_0$ and $\partial_1$. For each $R\colon V_0 \leftrightarrow V_1$, an RG-functor $\mathcal{Y}_e(R)\colon \mathbf{G}^{op} \to \mathbf{Span}$ is defined as follows:

- $\mathcal{Y}_e(R)_v(U) = (\mathsf{Hom}_{\mathbf{G}_v}[U,V_0] + \mathsf{Hom}_{\mathbf{G}_v}[U,V_1]) \,/\, \approx_U$ where $\approx_U$ is the least equivalence relation containing $\sim_U$ which is defined as follows:

  $f_0 \sim_U f_1$ iff there exists $\varphi\colon \mathbf{I}(U) \to R$ such that $\partial_i(\varphi) = f_i$ for $i = 0,1$

- $\mathcal{Y}_e(R)_v(g)[f] = [f \circ g]$

- $\mathcal{Y}_e(R)_w(P) = (\mathsf{Hom}_{\mathbf{G}_e}[P,\mathbf{I}(V_0)] + \mathsf{Hom}_{\mathbf{G}_e}[P,\mathbf{I}(V_1)] + \mathsf{Hom}_{\mathbf{G}_e}[P,R]) \,/\, \approx_P$ where $\approx_P$ is the least equivalence relation containing $\sim_P$ which is defined as follows:

  $\varphi \sim_P \psi$ iff $\psi$ is an edge morphism of the form $P \xrightarrow{\tau} \mathbf{I}(W) \xrightarrow{\kappa} R$ for some vertex $W$ such that $\varphi = \mathbf{I}(\partial_0(\kappa)) \circ \tau$ or $\varphi = \mathbf{I}(\partial_1(\kappa)) \circ \tau$

- $\mathcal{Y}_e(R)_w(\varphi)[\psi] = [\psi \circ \varphi]$

- two projections in $\mathcal{Y}_e(R)_e(P)$ send $[\varphi]$ to $[\partial_0(\varphi)]$ and $[\partial_1(\varphi)]$, respectively.

The action on a morphism $\varphi\colon R \to R'$ by $\mathcal{Y}_e$ with $R\colon V_0 \leftrightarrow V_1$ and $R'\colon V_0' \leftrightarrow V_1'$ is defined by picking up an element in an equivalence class and composing it with one of $\partial_i(\varphi)$, $\mathbf{I}(\partial_i(\varphi))$ and $\varphi$:

- $(\mathcal{Y}_e(\varphi)_v)_U[f] = \begin{cases} [\partial_0(\varphi)\circ f] & \text{if } f \in \mathsf{Hom}_{\mathbf{G}_v}[U,V_0] \\ [\partial_1(\varphi)\circ f] & \text{if } f \in \mathsf{Hom}_{\mathbf{G}_v}[U,V_1] \end{cases}$

- $(\mathcal{Y}_e(\varphi)_w)_P[\psi] = \begin{cases} [\mathbf{I}(\partial_0(\varphi)) \circ \psi] & \text{if } \psi \in \mathsf{Hom}_{\mathbf{G}_e}[P, \mathbf{I}(V_0)] \\ [\mathbf{I}(\partial_1(\varphi)) \circ \psi] & \text{if } \psi \in \mathsf{Hom}_{\mathbf{G}_e}[P, \mathbf{I}(V_1)] \\ [\varphi \circ \psi] & \text{if } \psi \in \mathsf{Hom}_{\mathbf{G}_e}[P, R] \end{cases}$

These two functors are well-defined and satisfy a property similar to Yoneda's lemma in a category of presheaves.

**Lemma 26** *Let $\mathbf{G}$ be a small reflexive graph with the identity condition.*

1. *$\mathcal{Y}_v$ and $\mathcal{Y}_e$ are well defined functors.*

2. *For every vertex $V$ in $\mathbf{G}$, there is a bijection $\alpha_V \colon \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(V), F] \cong F_v(V)$, which is natural in $V$.*

3. *For every edge $R$ in $\mathbf{G}$, there is a bijection $\beta_R \colon \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_e(R), F] \cong F_w(R)$, which is natural in $R$.*

4. *For all edges $R \colon V_0 \leftrightarrow V_1$ in $\mathbf{G}$, the above isomorphisms make the following diagram commute*

$$\begin{tikzcd}
& \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(V_0), F] \arrow[rr, "\alpha_{V_0}"] & & F_v(V_0) \\
\mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_e(R), F] \arrow[ur, "{\mathbf{Span}^{\mathbf{G}^{op}}[\iota_0, F]}"] \arrow[dr, "{\mathbf{Span}^{\mathbf{G}^{op}}[\iota_1, F]}"'] \arrow[rr, "\beta_R"] & & F_w(R) \arrow[ur, "\rho_0"] \arrow[dr, "\rho_1"'] & \\
& \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(V_1), F] \arrow[rr, "\alpha_{V_1}"'] & & F_v(V_1)
\end{tikzcd}$$

*where $\rho_0, \rho_1$ are two projections in $F_e(R)$ and $\iota_j \colon \mathcal{Y}_v(V_j) \to \mathcal{Y}_e(R)$ maps an element to an equivalence class containing it. The precise definition of $\iota_j$ is given by:*

- $((\iota_j)_v)_U(f) = [f]$
- $((\iota_j)_w)_P(\varphi) = [\varphi]$

**Lemma 27** *If $\mathbf{G}$ is a small reflexive graph with the identity condition, the corresponding parametric presheaf category $\mathbf{Span}^{\mathbf{G}^{op}}$ is cartesian closed.*

*Proof:* The details are as in [53] but generalized to nonrelational reflexive graphs. Products and a terminal object are given pointwise:

$$(F \times G)(-) = F(-) \times G(-) \text{ and } 1(-) = 1$$

(for spans and span-morphisms, the products $F_e(R) \times G_e(R)$ and $F_e(\varphi) \times G_e(\varphi)$ are given pointwise.) Suppose exponents exist in $\mathbf{Span}^{\mathbf{G}^{op}}$. Then, the following isomorphisms hold:

$$\begin{array}{rcll} (F \Rightarrow G)_v(V) & \cong & \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(V), F \Rightarrow G] & \because \text{Lemma 26} \\ & \cong & \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(V) \times F, G] & \because \text{Adjunction} \end{array}$$

$$\begin{array}{rcll} (F \Rightarrow G)_w(R) & \cong & \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_e(R), F \Rightarrow G] & \because \text{Lemma 26} \\ & \cong & \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_e(R) \times F, G] & \because \text{Adjunction} \end{array}$$

In fact, these give rise to an exponent of $F$ and $G$:

- $(F \Rightarrow G)_v(V) = \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(V) \times F, G]$;
- $(F \Rightarrow G)_v(f) = \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(f) \times F, G]$;
- $(F \Rightarrow G)_w(R) = \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_e(R) \times F, G]$;
- $(F \Rightarrow G)_w(\varphi) = \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_e(\varphi) \times F, G]$;
- two projections in $(F \Rightarrow G)_e(R)$ are $\mathbf{Span}^{\mathbf{G}^{op}}[\iota_0 \times F, G]$ and $\mathbf{Span}^{\mathbf{G}^{op}}[\iota_1 \times F, G]$ where $\iota_j$ is defined in Lemma 26.

The application $\mathsf{ap}\colon (F \Rightarrow G) \times F \to G$ is given pointwise:

$$\begin{array}{rcl} (\mathsf{ap}_v)_V & : & ((F \Rightarrow G)_v)(V) \times F_v(V) \to G_v(V) \\ (\mathsf{ap}_v)_V(\mu, x) & = & (\mu_v)_V(\mathsf{id}_V, x) \end{array}$$

$$\begin{array}{rcl} (\mathsf{ap}_w)_R & : & ((F \Rightarrow G)_w)(R) \times F_w(R) \to G_w(R) \\ (\mathsf{ap}_w)_R(\delta, w) & = & (\delta_w)_R([\mathsf{id}_R], w) \end{array}$$

□

**Theorem 28** *If* $\mathbf{G}$ *is a small reflexive graph with the identity condition, the corresponding parametric presheaf category* $\mathbf{Span}^{\mathbf{G}^{op}}$ *is a topos.*

*Proof:* Since we have already argued that $\mathbf{Span}^{\mathbf{G}^{op}}$ is a CCC, all that is left is to define pullbacks and a subobject classifier $\Omega' \colon \mathbf{G}^{op} \to \mathbf{Span}$ (which must be an RG-functor). Pullbacks are given pointwise:

$$(F \times_H G)(-) = F(-) \times_{H(-)} G(-)$$

(for spans, the pullback $F_e(R) \times_{H_e(R)} G_e(R)$ is given pointwise.) Because pullbacks exist, it is possible to define the functor $\mathsf{Sub} \colon (\mathbf{Span}^{\mathbf{G}^{op}})^{op} \to \mathbf{Set}$ which maps an RG-functor to the set of its subobjects and a parametric natural transformation to a function defined using pullbacks.

- $\mathsf{Sub}(F)$ is a set of subobjects of $F$; and
- $\mathsf{Sub}(\eta \colon F \to G)(G') = F'$ is given by the following pullback diagram:

$$\begin{tikzcd} F' \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & G' \arrow[d] \\ F \arrow[r] & G \end{tikzcd}$$

Suppose that a subobject classifier exists. Then, the following two isomorphisms have to hold:

$$\begin{array}{rcll} \Omega'_v(V) & \cong & \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_v(V), \Omega'] & \because \text{Lemma 26} \\ & \cong & \mathsf{Sub}(\mathcal{Y}_v(V)) & \because \text{Subobject classifier} \end{array}$$

$$\begin{array}{rcll} \Omega'_w(R) & \cong & \mathbf{Span}^{\mathbf{G}^{op}}[\mathcal{Y}_e(R), \Omega'] & \because \text{Lemma 26} \\ & \cong & \mathsf{Sub}(\mathcal{Y}_e(R)) & \because \text{Subobject classifier} \end{array}$$

In fact, a subobject classifier is given from the above isomorphisms:

- $\Omega'_v(U) = \mathsf{Sub}(\mathcal{Y}_v(U))$;
- $\Omega'_v(f) = \mathsf{Sub}(\mathcal{Y}_v(f))$;
- $\Omega'_w(R) = \mathsf{Sub}(\mathcal{Y}_e(R))$;
- $\Omega'_w(\varphi) = \mathsf{Sub}(\mathcal{Y}_e(\varphi))$;
- two projections in $\Omega'_e(R)$ are $\mathsf{Sub}(\iota_0)$ and $\mathsf{Sub}(\iota_1)$ where $\iota_j$ is a parametric natural transformation from $\mathcal{Y}_v(\partial_j(R))$ to $\mathcal{Y}_e(R)$ defined in Lemma 26.

And the true map $T'$ is given by picking up the greatest elements $\mathcal{Y}_v(U)$ and $\mathcal{Y}_e(R)$ in $\mathsf{Sub}(\mathcal{Y}_v(U))$ and $\mathsf{Sub}(\mathcal{Y}_e(R))$, respectively.

- $(T'_v)_U(*) = \mathcal{Y}_v(U)$; and
- $(T'_w)_R(*) = \mathcal{Y}_e(R)$.

$\square$

In the rest of this section, we present a subobject classifier in $\mathbf{Span}^{\mathbf{G}^{op}}$ more concretely using two kinds of sieves, called vertex-sieves and edge-sieves.

**Definition 29 (vertex-sieve)** *If $\mathbf{G}$ is a small reflexive graph with the identity condition and $V$ is a vertex in $\mathbf{G}$, a* vertex-sieve *on $V$ is a pair $S = (S_v, S_w)$ where*

- *$S_v$ is a sieve on $V$ in $\mathbf{G}_v$ and*
- *$S_w$ is a sieve on $\mathbf{I}(V)$ in $\mathbf{G}_e$*

*such that they are closed under the structure of the reflexive graph:*

$$\forall \varphi \in S_w.\, \partial_0(\varphi), \partial_1(\varphi) \in S_v \quad \textit{and} \quad \forall f \in S_v.\, \mathbf{I}(f) \in S_w.$$

**Definition 30 (edge-sieve)** *If $\mathbf{G}$ is a small reflexive graph with the identity condition and $R\colon V_0 \leftrightarrow V_1$ is an edge in $\mathbf{G}$, an* edge-sieve *on $R$ is a triple $\Sigma = (S_0, \Sigma_w, S_1)$ where*

- *$S_0$ and $S_1$ are vertex-sieves on $V_0$ and $V_1$ respectively, and*
- *$\Sigma_w$ is a sieve on $R$ in $\mathbf{G}_e$*

*such that*

1. *for every $\varphi \in \Sigma_w$, $\partial_i(\varphi) \in (S_i)_v$ for $i = 0, 1$ and*
2. *for every morphism of the form $\tau\colon \mathbf{I}(U) \to R$ with $\partial_i(\tau) = f_i$ for $i = 0, 1$, we have*

$$\mathbf{I}(f_0)^*((S_0)_w) = \mathbf{I}(f_1)^*((S_1)_w) = \tau^*(\Sigma_w).$$

With these definitions, a subobject classifier in $\mathbf{Span}^{\mathbf{G}^{op}}$ is represented more concretely:

**Lemma 31** *Let $\Omega$ be an RG-functor from $\mathbf{G}^{op}$ to $\mathbf{Span}$ defined by*

- $\Omega_v(V) =$ *the set of vertex-sieves on* $V$*;*
- $\Omega_v(f) = \lambda(S_v, S_w).\, (f^*(S_v), \mathbf{I}(f)^*(S_w))$*;*
- $\Omega_w(R) =$ *the set of edge-sieves on* $R$*;*
- $\Omega_w(\varphi) = \lambda(S_0, \Sigma_w, S_1).\, (\Omega_v(\partial_0(\varphi))(S_0), \varphi^*(\Sigma_w), \Omega_v(\partial_1(\varphi))(S_1))$*; and*
- *two projections in* $\Omega_e(R)$ *are* $\lambda(S_0, \Sigma_w, S_1).\, S_0$ *and* $\lambda(S_0, \Sigma_w, S_1).\, S_1$*.*

*Then,* $\Omega$ *is isomorphic to* $\Omega'$*.*

*Proof:* We define a parametric natural transformation $\mu$ from $\Omega$ to $\Omega'$. $\mu_v$ should map a vertex-sieve on $V$ to an RG-subfunctor of $\mathcal{Y}_v(V)$ and $\mu_w$ an edge-sieve on $R$ to an RG-subfunctor of $\mathcal{Y}_e(R)$.[1] Let $V, U$ be vertices in $\mathbf{G}$ and $R: V_0 \leftrightarrow V_1, P: U_0 \leftrightarrow U_1$ edges in $\mathbf{G}$.

- $(\mu_v)_V(S)_v(U) = \{f \mid f: U \to V \in S_v\}$;
- $(\mu_v)_V(S)_w(R) = \{\varphi \mid \varphi: R \to \mathbf{I}(V) \in S_w\}$;
- $(\mu_w)_R(S_0, \Sigma_w, S_1)_v(U) = \{[f] \mid f: U \to V_0 \in (S_0)_v \ \vee \ f: U \to V_1 \in (S_1)_v\}$;
- $$(\mu_w)_R(S_0, \Sigma_w, S_1)_w(P) = \{[\varphi] \mid \varphi: P \to \mathbf{I}(V_0) \in (S_0)_w \ \vee \ \varphi: P \to \mathbf{I}(V_1) \in (S_1)_w \\ \vee \ \varphi: P \to R \in \Sigma_w\}.$$

Since $\mathsf{Sub}(\mathcal{Y}_v(V))$ and $\mathsf{Sub}(\mathcal{Y}_e(R))$ are isomorphic to the set of RG-subfunctors of $\mathcal{Y}_v(V)$ and the set of RG-subfunctors of $\mathcal{Y}_e(R)$ respectively, $\mu$ can be easily shown to be an isomorphism. Therefore, the conclusion follows. $\square$

The truth predicate for the new representation of the subobject classifier $\Omega$ is:

$$T_V = (t_V, t_{\mathbf{I}(V)})$$
$$T_R = (T_{V_0}, t_R, T_{V_1})$$

where $t_X$ denotes the maximal sieve on $X$ (in a category).

To get some intuition for these definitions, we consider the notion of a *matching family* (which will be used to define sheaves in the next section).

[1] For parametric presheaves $F$ and $G$ in $\mathbf{Span}^{\mathbf{G}^{op}}$, we say that $F$ is an *RG-subfunctor* of $G$ iff $F_v$ and $F_w$ are subfunctors of $G_v$ and $G_w$, respectively. Notice that given $G$, such an RG-subfunctor $F$ is completely specified by $F_v(U)$ and $F_w(R)$.

**Definition 32 (matching family for a vertex-sieve)** *Let* $\mathbf{G}$ *be a small reflexive graph with the identity condition and* $F: \mathbf{G}^{op} \to \mathbf{Span}$ *an RG-functor. If* $S = (S_v, S_w)$ *is a vertex-sieve on a vertex* $V$*, a* matching family *for* $S$ *of elements of* $F$ *is a pair of families*

$$(\{x_f \in F_v(dom(f))\}_{f \in S_v}, \{s_\varphi \in F_w(dom(\varphi))\}_{\varphi \in S_w})$$

*such that*[2]

- $x_{f \circ g} = x_f \cdot g$ *for all* $f \in S_v$ *and* $g$ *such that* $dom(f) = cod(g)$*;*
- $s_{\varphi \circ \psi} = s_\varphi \cdot \psi$ *for all* $\varphi \in S_w$ *and* $\psi$ *such that* $dom(\varphi) = cod(\psi)$*; and*
- *for all* $\varphi: R \to \mathbf{I}(V)$ *in* $S_w$ *with* $F_e(R) = (F_v(U_0) \overset{\rho_0}{\leftarrow} F_w(R) \overset{\rho_1}{\rightarrow} F_v(U_1))$*,* $\rho_0(s_\varphi) = x_{\partial_0(\varphi)}$ *and* $\rho_1(s_\varphi) = x_{\partial_1(\varphi)}$*.*

This definition shows the role played by edge-parts of sieves. The matching families must be consistent in that they must respect all the relationships denoted by the edges included in the sieve.

Similarly, there is a notion of matching families for edge-sieves.

**Definition 33 (matching family for an edge-sieve)** *Let* $\mathbf{G}$ *be a small reflexive graph with the identity condition and* $F: \mathbf{G}^{op} \to \mathbf{Span}$ *an RG-functor. If* $\Sigma = (S_0, \Sigma_w, S_1)$ *is an edge-sieve on an edge* $R: V_0 \leftrightarrow V_1$ *of* $\mathbf{G}$*, a* matching family *for* $\Sigma$ *of elements of* $F$ *is a triple* $((\mathbf{x}, \mathbf{s}), \mathbf{w}, (\mathbf{y}, \mathbf{t}))$ *where*

- $(\mathbf{x}, \mathbf{s})$ *and* $(\mathbf{y}, \mathbf{t})$ *are matching families for* $S_0$ *and* $S_1$ *respectively, and*
- $\mathbf{w} = \{w_\varphi \in F_w(dom(\varphi))\}_{\varphi \in \Sigma_w}$

*satisfying the following conditions:*

1. $w_{\varphi \circ \varphi'} = w_\varphi \cdot \varphi'$ *for all* $\varphi \in \Sigma_w$ *and* $\varphi'$ *such that* $dom(\varphi) = cod(\varphi')$*;*
2. *for all* $\varphi \in P \to R$ *in* $\Sigma_w$*, with* $F_e(P) = (F_v(V_0) \overset{\rho_0}{\leftarrow} F_w(P) \overset{\rho_1}{\rightarrow} F_v(V_1))$*,* $\rho_0(w_\varphi) = x_{\partial_0(\varphi)}$ *and* $\rho_1(w_\varphi) = y_{\partial_1(\varphi)}$*; and*
3. *for every edge morphism of the form* $P \overset{\psi}{\rightarrow} \mathbf{I}(U) \overset{\varphi}{\rightarrow} R$ *in* $\Sigma_w$

$$w_{\varphi \circ \psi} = s_{\mathbf{I}(\partial_0(\varphi)) \circ \psi} = t_{\mathbf{I}(\partial_1(\varphi)) \circ \psi}.$$

[2] *For an RG-functor* $F: \mathbf{G}^{op} \to \mathbf{Span}$*, we write* $x \cdot f$ *for* $F_v(f)(x)$ *and* $w \cdot \varphi$ *for* $F_w(\varphi)(w)$*.*

### 4.2.1 Relational Parametric Presheaves

Having given the definitions for the general case, we consider a special case of parametric presheaves which are encountered in Algol semantics. Note that we have a reflexive graph embedding $E\colon \mathbf{Rel} \hookrightarrow \mathbf{Span}$ which sends a relation $R\colon A_0 \leftrightarrow A_1$ to the span $A_0 \leftarrow R \rightarrow A_1$ whose projections are usual ones from the product $A_0 \times A_1$ to $A_0$ or $A_1$. We call a parametric presheaf $F\colon \mathbf{G}^{op} \rightarrow \mathbf{Span}$ *relational* if it factors through this embedding upto an isomorphism, i.e., $F \cong EF'$ for an RG-functor $F'\colon \mathbf{G}^{op} \rightarrow \mathbf{Rel}$. So, we can regard $F_w(R)$, for every edge $R\colon V_0 \leftrightarrow V_1$ of $\mathbf{G}$, simply as a relation $F_v(V_0) \leftrightarrow F_v(V_1)$. The subcategory of relational parametric presheaves in $\mathbf{Span}^{\mathbf{G}^{op}}$ is a sub-CCC of $\mathbf{Span}^{\mathbf{G}^{op}}$. However, it does not have a subobject classifier.

For relational parametric presheaves, RG-functors, parametric natural transformations and matching families can be represented in a considerably simpler way than the general case. A relational parametric presheaf $F$ can be completely determined upto an isomorphism by a functor $F_v\colon \mathbf{G}_v^{op} \rightarrow \mathbf{Set}$ and a map from objects (edges) in $\mathbf{G}_e$ on a binary relation between $F_v(\partial_0(R))$ and $F_v(\partial_1(R))$ induced by $F_e(R)$. The reason is that the only missing data, which is the action of $F$ to edge morphisms, is uniquely determined by $\partial_0 \circ F_e (= F_v \circ \partial_0)$ and $\partial_1 \circ F_e (= F_v \circ \partial_1)$. In fact, it is possible to characterize the precise condition when such $F_v$ and an edge-to-relation map give rise to a relational parametric presheaf $F$:

**Lemma 34** *Let $F_v$ be a functor of type $\mathbf{G}_v^{op} \rightarrow \mathbf{Set}$ and $F_w^o$ a map from an edge $R$ in $\mathbf{G}_e$ to a binary relation $F_w^o(R)$ between $F_v(\partial_0(R))$ and $F_v(\partial_1(R))$. Then, $(F_v, F_w^o)$ uniquely extends to a relational parametric presheaf $F$ upto an isomorphism if*

1. *for each vertex $V$ in $\mathbf{G}_v$, $F_w^o(\mathbf{I}(V)) = \mathbf{\Delta}_{F_v(V)}$; and*
2. *for each edge morphism $\tau$ in $\mathbf{G}_e$ of the form*

$$\begin{tikzcd} V_0' \arrow[r, "f_0"] \arrow[d, leftrightarrow, "R'"'] \arrow[dr, phantom, "\tau"] & V_0 \arrow[d, leftrightarrow, "R"] \\ V_1' \arrow[r, "f_1"] & V_1 \end{tikzcd}$$

*we have*

$$x_0 \, [F_w^o(R)] \, x_1 \Longrightarrow x_0 \cdot f_0 \, [F_w^o(R')] \, x_1 \cdot f_1$$

*for all $x_0 \in F_v(V_0)$ and $x_1 \in F_v(V_1)$.*

*Proof:* It remains to define two projections in $F_e(R\colon V_0 \leftrightarrow V_1)$ and $F_w(\varphi)$ for all edges $R$ and edge morphisms $\varphi$ in $\mathbf{G}$. Two projections are given by usual projections from $F^o_w(R) \subseteq F_v(V_0) \times F_v(V_1)$. And $F_w(\varphi)$ is defined by $F_v(\partial_0(\varphi)) \times F_v(\partial_1(\varphi))$. $\square$

Notice that each relational parametric presheaf $F$ induces a pair $(F_v, F^o_w)$ satisfying the conditions in the lemma.

Similar simplification can be made for parametric natural transformations.

**Lemma 35** *For relational parametric presheaves $F, G$ on $\mathbf{G}$, let $\mu_v$ be a natural transformation from $F_v$ to $G_v$. $\mu_v$ can uniquely be extended to a parametric natural transformation from $F$ to $G$ if for every edge $R\colon V_0 \leftrightarrow V_1$ in $\mathbf{G}_e$, we have that for all $x_0 \in F_v(V_0)$ and $x_1 \in F_v(V_1)$,*

$$x_0\ [F_w(R)]\ x_1 \Longrightarrow (\mu_v)_{V_0}(x_0)\ [G_w(R)]\ (\mu_v)_{V_1}(x_1)$$

*where we regard $F_w(R)$ and $G_w(R)$ as binary relations.*

*Proof:* We need to reconstruct $\mu_e$. We only deal with the case that $F_w(R)$ and $G_w(R)$ are, in fact, relations and projections in $F_e(R)$ and $G_e(R)$ are those from products. In that case, $(\mu_e)_R = (\mu_v)_{\partial_0(R)} \times (\mu_v)_{\partial_1(R)}$. $\square$

The representation of matching families for relational parametric presheaves can also be simplified. Notice that for a relational parametric presheaf $F$, $x_0 \in F_v(V_0)$ and $x_1 \in F_v(V_1)$, there exists at most one $w \in F_w(R\colon V_0 \leftrightarrow V_1)$ which is mapped to $x_0$ and $x_1$ by two projections in $F_e(R)$. Therefore, a matching family for a vertex-sieve $S$ of elements of $F$ is equivalently specified by a family $\{x_f \in F_v(dom(f))\}_{f \in S_v}$ such that

1. for all $f\colon U \to V$ in $S_v$ and all vertex morphisms $g\colon U' \to U$, $x_{f \circ g} = x_f \cdot g$
2. for all $\varphi\colon R \to \mathbf{I}(V)$ in $S_w$, $x_{\partial_0(\varphi)}\,[F_w(R)]\,x_{\partial_1(\varphi)}$.

That is, we do not need a separate family of witnesses because the witnesses are unique whenever they exist. For relational parametric presheaves, we use such a simpler presentation. And by the same reason, for an edge-sieve $\Sigma = (S_0, \Sigma_w, S_1)$, a matching family is equivalently specified merely by a pair of matching families $(\mathbf{x}, \mathbf{y})$ for $S_0$ and $S_1$ such that,

$$\text{for all } \varphi\colon P \to R \text{ in } \Sigma_w,\ x_{\partial_0(\varphi)}\,[F_w(P)]\,y_{\partial_1(\varphi)}.$$

Again, we use the simpler presentation for relational parametric presheaves.

## 4.3 Parametric Sheaves

Sheaves can be defined internal to any topos using a Lawvere-Tierney topology [32]. Since $\mathbf{Span}^{\mathbf{G}^{op}}$ is a topos, we automatically have sheaves internal to it. We call them parametric sheaves.

Such an internal characterization is very abstract and hard to use in applications. In this section, we give an external characterization of parametric sheaves similar to the definition of sheaves in Section 4.1.1. We also show that it coincides with the internal characterization.

As in Section 4.1.1, we first need a Grothendieck topology $J$ that assigns a collection of sieves to each vertex and edge of a reflexive graph $\mathbf{G}$.

**Definition 36 (Grothendieck topology, RG-site)** *Let* $\mathbf{G}$ *be a small reflexive graph with the identity condition. A* Grothendieck topology *on* $\mathbf{G}$ *is a pair* $J = (J_v, J_w)$ *where*

- $J_v$ *assigns to each vertex* $V$ *of* $\mathbf{G}$ *a collection of vertex-sieves on* $V$ *(called the "covers" of* $V$*); and*
- $J_w$ *assigns to each edge* $R: V_0 \leftrightarrow V_1$ *of* $\mathbf{G}$ *a collection of edge-sieves on* $R$ *(called the "covers" of* $R$*) such that the following conditions are satisfied:*
  1. Closed under reflexive graph structure*:*
     - *For each* $(S_0, \Sigma_w, S_1) \in J_w(R)$*,* $S_0 \in J_v(V_0)$ *and* $S_1 \in J_v(V_1)$*.*
     - *For each* $S \in J_v(V)$*,* $(S, S_w, S) \in J_w(\mathbf{I}(V))$*.*
  2. Maximality*:* $T_V \in J_v(V)$ *and* $T_R \in J_w(R)$*.*
  3. Stability*:*
     - *If* $S \in J_v(V)$*, then* $\Omega_v(f)(S) \in J_v(U)$ *for any* $f: U \to V$*.*
     - *If* $\Sigma \in J_w(R)$*, then* $\Omega_w(\varphi)(\Sigma) \in J_w(P)$ *for any* $\varphi: P \to R$*.*
  4. Transitivity*:*
     - *If* $S \in J_v(V)$ *and* $S'$ *is a vertex-sieve on* $V$ *such that* $\Omega_v(f)(S') \in J_v(U)$ *for all* $f: U \to V$ *in* $S_v$ *and* $\Omega_w(\varphi)(S', S'_w, S') \in J_w(P)$ *for all* $\varphi: P \to \mathbf{I}(V)$ *in* $S_w$*, then* $S' \in J_v(V)$*.*

– *If* $\Sigma \in J_w(R)$ *and* $\Sigma' = (S'_0, \Sigma'_w, S'_1)$ *is an edge-sieve on* $R: V_0 \leftrightarrow V_1$ *such that* $S'_0 \in J_v(V_0)$, $S'_1 \in J_v(V_1)$ *and* $\Omega_w(\varphi)(\Sigma') \in J_w(P)$ *for all* $\varphi: P \to R$ *in* $\Sigma_w$, *then* $\Sigma' \in J_w(R)$.

*We call the pair* $(\mathbf{G}, J)$ *an* RG-site.

The idea of covers, as mentioned in Section 4.1.1, is to identify certain sieves that are intended to force unique amalgamations. A Grothendieck topology is meant to pick out such covers. A simple example of a Grothendieck topology is called the *atomic topology.* (This will play a role in Algol semantics of Section 4.4.) Denoting it by the symbol $\mathcal{A}$, we have:

$\mathcal{A}_v(V) =$ the set of all vertex-sieves $(S_v, S_w)$ such that $S_v$ and $S_w$ are nonempty.

$\mathcal{A}_w(R) =$ the set of all edge-sieves $(S_0, \Sigma_w, S_1)$ such that $S_0 \in \mathcal{A}_v(V_0)$, $S_1 \in \mathcal{A}_v(V_1)$ and $\Sigma_w$ is nonempty.

When for any two edge morphisms $\varphi: R' \to R$ and $\psi: R'' \to R$ in $\mathbf{G}$, there exist edge morphisms $\varphi': P \to R'$ and $\psi': P \to R''$ such that $\varphi \circ \varphi' = \psi \circ \psi'$, $\mathcal{A}$ satisfies all the axioms to be a Grothendieck topology on $\mathbf{G}$.

Matching families for vertex- and edge-sieves have already been defined in Section 4.2. So, we have the main definition:

**Definition 37 (parametric sheaf)** *A* parametric sheaf *on an RG-site* $(\mathbf{G}, J)$ *is a parametric presheaf* $F: \mathbf{G}^{op} \to \mathbf{Span}$ *such that*

- *for every vertex-sieve* $S \in J_v(V)$ *and matching family* $(\mathbf{x}, \mathbf{s})$ *for* $S$, *there is a unique amalgamation: an element* $x \in F_v(V)$ *such that* $x_f = x \cdot f$ *for all* $f \in S_v$ *and* $s_\varphi = w_x \cdot \varphi$ *for all* $\varphi \in S_w$, *where* $w_x$ *is a witness for* $(x, x)$*: it is mapped to* $x$ *by projections in* $F_e(\mathbf{I}(V))$*; and*

- *for every covering edge-sieve* $\Sigma \in J_w(R: V_0 \leftrightarrow V_1)$ *and matching family* $((\mathbf{x}, \mathbf{s}), \mathbf{w}, (\mathbf{y}, \mathbf{t}))$ *for* $\Sigma$, *there is a unique amalgamation: an element* $w \in F_w(R)$, *where* $F_e(R) = F_v(V_0) \xleftarrow{\rho_0} F_w(R) \xrightarrow{\rho_1} F_v(V_1)$, *such that* $\rho_0(w)$ *and* $\rho_1(w)$ *are the amalgamations of* $(\mathbf{x}, \mathbf{s})$ *and* $(\mathbf{y}, \mathbf{t})$ *respectively and* $w_\varphi = w \cdot \varphi$ *for all* $\varphi \in \Sigma_w$.

It is instructive to consider the special case of relational parametric sheaves.

**Lemma 38** *Let $F: \mathbf{G}^{op} \to \mathbf{Span}$ be a relational parametric presheaf. Then, it is a parametric sheaf iff*

- *for every $S \in J_v(V)$ and matching family $\mathbf{x}$ for $S$, there is a unique amalgamation: an element $x \in F_v(V)$ such that $x_f = x \cdot f$ for all $f \in S_v$.*
- *for every $\Sigma \in J_w(R: V_0 \leftrightarrow V_1)$ and matching family $(\mathbf{x}, \mathbf{y})$ for $\Sigma$, the amalgamations $x$ of $\mathbf{x}$ and $y$ of $\mathbf{y}$ satisfy $x\,[F_w(R)]\,y$, where $F_w(R)$ is regarded as a relation.*

In the next section, we will find several examples of relational parametric sheaves on an atomic RG-site.

It follows from the general results discussed in Section 4.3.1 that:

**Theorem 39** *A parametric sheaf category $\mathbf{Sh}(\mathbf{G}, J)$ is a topos (and, hence, cartesian closed.).*

### 4.3.1 Parametric Sheaves are Sheaves

The definition of parametric sheaves above involves excessive detail and it is hard to convince oneself that the definitions are right. In this section, we show that parametric sheaves are precisely the sheaves in the topos $\mathbf{Span}^{\mathbf{G}^{op}}$ using Lawvere-Tierney topologies.

A Grothendieck topology $J = (J_v, J_w)$ can be easily turned into an RG-functor. By equipping each $J_w(R)$ with projections that send $(S_0, \Sigma_w, S_1)$ to $S_0$ and $S_1$ respectively, we obtain a functor $J_e: \mathbf{G}_e^{op} \to \mathbf{Span}_e$. (The morphism part comes from the stability condition.) The pair $J = (J_v, J_e)$ does form an RG-functor.

Since the sets involved in $J$ are all sets of sieves, it is clear that $J \hookrightarrow \Omega$ is an RG-subfunctor.[3] The characteristic morphism of the RG-subfunctor, $j: \Omega \to \Omega$, is a Lawvere-Tierney topology on $\mathbf{Span}^{\mathbf{G}^{op}}$. Conversely, the kernel of every Lawvere-Tierney topology on $\mathbf{Span}^{\mathbf{G}^{op}}$ is a Grothendieck topology (regarded as an RG-functor).

**Lemma 40** *There is one-to-one correspondence between Grothendieck topologies on a small reflexive graph $\mathbf{G}$ with the identity condition and Lawvere-Tierney topologies on $\mathbf{Span}^{\mathbf{G}^{op}}$.*

**Theorem 41 (Externalization)** *For an RG-site $(\mathbf{G}, J)$, an RG-functor $F: \mathbf{G}^{op} \to \mathbf{Span}$ is a parametric sheaf iff it is a sheaf for $j = \mathsf{char}(J)$.*

[3] Given two parametric presheaves $F$ and $G$, $F$ is an RG-subfunctor of $\mathbf{G}$ iff $F_v$ and $F_w$ are subfunctors of $G_v$ and $G_w$ respectively.

We show the theorem following the proof of the corresponding theorem in a presheaf category case in [32]. We first deal with the only-if direction. Recall that $F$ is a Lawvere-Tierney sheaf for $j$ iff for every dense monomorphism $\tau: G \to H$ and every morphism $\eta: G \to F$, $\eta$ can be uniquely extended along $\tau$, i.e., there exists a unique $\eta': H \to F$ such that $\eta = \eta' \circ \tau$. The key observation in showing the only-if direction is that given a dense monomorphism $\tau: G \to H$, each element $x$ in $H_v(V)$ generates a matching family of elements of $G$ for a covering vertex-sieve on $V$ and each element $w$ in $H_w(R)$ a matching family of elements of $G$ for a covering edge-sieve on $R$; since a matching family either for a vertex-sieve or for an edge-sieve is mapped to a matching family by a parametric natural transformation, for all $\eta: G \to F$, $x \in H_v(V)$ and $w \in H_w(R)$, we end up having two matching families of elements of $F$, one for a covering vertex-sieve on $V$ and the other for a covering edge-sieve for $R$; then, $x$ and $w$ are mapped to the unique amalgamations of such matching families, which always exist because $F$ is a parametric sheaf. In the following, we give a characterization of dense monomorphisms in the topos $\mathbf{Span}^{\mathbf{G}^{op}}$, which clearly shows the observation. We first note that a monomorphism in $\mathbf{Span}^{\mathbf{G}^{op}}$ is characterized pointwise:

**Lemma 42** *A parametric natural transformation $\eta: F \to G$ is monic in $\mathbf{Span}^{\mathbf{G}^{op}}$ iff for any vertex $V$ and edge $R$, $(\eta_v)_V$ and $(\eta_w)_R$ are injections.*

Therefore, it suffices to consider that case that $(\eta_v)_V$ and $(\eta_w)_R$ are indeed inclusions. For such a case, a dense monomorphism has the following characterization:

**Lemma 43** *A monomorphism $\eta: F \hookrightarrow G$ in $\mathbf{Span}^{\mathbf{G}^{op}}$ is dense for a Lawvere-Tierney topology $j(= \mathsf{char}(J))$ iff*

1. *for any vertex $V$ in $\mathbf{G}$, $x \in G_v(V)$ and $w \in G_w(\mathbf{I}(V))$ such that $w$ is a witness for $(x, x)$, i.e., $\rho_0(w) = x$ and $\rho_1(w) = x$ for two projections $\rho_0, \rho_1$ in $G_e(R)$,*

   $$(\{f: W \to V \mid x \cdot f \in F_v(W)\}, \{\varphi: R \to \mathbf{I}(V) \mid w \cdot \varphi \in F_w(R)\}) \text{ is in } J_v(V)\text{; and}$$

2. *for any edge $R: V_0 \leftrightarrow V_1$ in $\mathbf{G}_e$, $w \in G_w(R)$, $x \in G_v(V_0)$ and $y \in G_v(V_1)$ such that $w, s, t$ are witnesses for $(x, y)$, $(x, x)$ and $(y, y)$, respectively,*

   $$\left.\begin{array}{l} ((\{f: U \to V_0 \mid x \cdot f \in F_v(U)\},\ \{\varphi: P \to \mathbf{I}(V_0) \mid s \cdot \varphi \in F_w(P)\}), \\ \quad \{\varphi: P \to R \mid w \cdot \varphi \in F_w(P)\}, \\ (\{f: U \to V_1 \mid y \cdot f \in F_v(U)\},\ \{\varphi: P \to \mathbf{I}(V_1) \mid t \cdot \varphi \in F_w(P)\})) \end{array}\right. \quad \text{is in } J_w(R)$$

The main step in proving the if direction is to translate a cover to a dense monomorphism and a matching family for the cover to a parametric natural transformation; then, the definition of a Lawvere-Tierney sheaf guarantees that the parametric natural transformation is expanded uniquely along the dense monomorphism, which is indeed equivalent to the existence of a unique amalgamation of the family. In fact, the way to translate a cover to a dense monomorphism is already given in Lemma 31; a vertex-sieve $S$ on $V$ and an edge-sieve $\Sigma$ on $R: V_0 \leftrightarrow V_1$ with $\Sigma = (S_0, \Sigma_w, S_1)$ are translated to the following RG-subfunctors $G, H$ of $\mathcal{Y}_v(V)$ and $\mathcal{Y}_e(R)$, respectively:

$$\begin{aligned} G_v(U) &= \{f \mid f: U \to V \in S_v\} \\ G_w(P) &= \{\varphi \mid \varphi: P \to \mathbf{I}(V) \in S_w\} \end{aligned}$$

$$\begin{aligned} H_v(U) &= \{[f] \mid f: U \to V_0 \in (S_0)_v \ \vee \ f: U \to V_1 \in (S_1)_v\} \\ H_w(P) &= \{[\varphi] \mid \varphi: P \to \mathbf{I}(V_0) \in (S_0)_w \ \vee \ \varphi: P \to \mathbf{I}(V_1) \in (S_1)_w \\ &\qquad \vee \ \varphi: P \to R \in \Sigma_w\} \end{aligned}$$

We note that there is a translation of a matching family for a covering vertex-sieve to a parametric natural transformation, which fulfills the goal:

**Lemma 44** *Let $(\mathbf{G}, J)$ be an RG-site, $S$ a covering on a vertex $V$ and $G$ an RG-subfunctor of $\mathcal{Y}_v(V)$ corresponding to $S$ by an isomorphism in Lemma 31.*

1. *A matching $(\mathbf{x}, \mathbf{s})$ for $S$ of elements of $F$ has one-to-one correspondence with the parametric natural transformation $\eta$ from $G$ to $F$ given by*

   $$(\eta_v)_U(f) = x_f \text{ for all } f \in G_v(U) \text{ and } (\eta_w)_R(\varphi) = w_\varphi \text{ for all } \varphi \in G_w(R).$$

2. *A matching family $(\mathbf{x}, \mathbf{s})$ of elements of $F$ has a unique amalgamation iff the corresponding parametric natural transformation $\eta$ can be uniquely extended along the inclusion from $G$ to $\mathcal{Y}_v(V)$.*

A matching family for a covering edge-sieve on an edge $R: V_0 \leftrightarrow V_1$ can be handled similarly:

**Lemma 45** *Let $(\mathbf{G}, J)$ be an RG-site, $\Sigma$ a covering on an edge $R$ and $H$ an RG-subfunctor of $\mathcal{Y}_v(R)$ corresponding to $\Sigma$ by an isomorphism in Lemma 31.*

1. *A matching* $((\mathbf{x},\mathbf{s}),\mathbf{w},(\mathbf{y},\mathbf{t}))$ *for* $\Sigma$ *of elements* $F$ *has one-to-one correspondence with the parametric natural transformation* $\eta$ *from* $H$ *to* $F$ *given by*

$$(\eta_v)_U([f]) = \begin{cases} x_f & \textit{if } f \in (S_0)_v \\ y_f & \textit{if } f \in (S_1)_v \end{cases} \quad \textit{and } (\eta_w)_R([\varphi]) = \begin{cases} s_\varphi & \textit{if } \varphi \in (S_0)_w \\ w_\varphi & \textit{if } \varphi \in \Sigma_w \\ t_\varphi & \textit{if } \varphi \in (S_1)_w \end{cases}$$

2. *A matching family* $((\mathbf{x},\mathbf{s}),\mathbf{w},(\mathbf{y},\mathbf{t}))$ *of elements of* $F$ *has a unique amalgamation iff the corresponding parametric natural transformation* $\eta$ *can be uniquely extended along the inclusion from* $H$ *to* $\mathcal{Y}_e(R)$.

## 4.4 Semantics of Symmetric Noninterference

In this section, we define symmetric noninterference in a parametricity semantics of Idealized Algol. The semantics is exactly the same as in Chapter 3; it is just a more thorough categorical presentation which explicitly shows the connection with parametric sheaves. We use the results in the previous section on parametric sheaves to show that the semantics is well-defined.

In this section, we interpret each IA type as a relational parametric sheaf on an atomic RG-site and each term as a parametric natural transformation. The semantics in the section is similar to [53, 51, 60] but has one significance difference, which is that each world denotes not an abstract set of states but a finite set of variables currently allocated in the stack. Because of such a concrete view on stack storage, it is possible to precisely define what stack variables are used by a "program phrase" using the idea of support in [49] even in a parametricity framework. As a consequence, the semantics of symmetric noninterference in this section validates all important principles in Chapter 2, including the most problematic Noninterference Decomposition rule.

We define a reflexive graph $\mathbf{W}$ which parameterizes the interpretation of types by requiring them to be relational RG-functors from $\mathbf{W}^{op}$ to $\mathbf{Span}$. The vertex and edge categories of $\mathbf{W}$ model two different aspects of computations. An object and a morphism in the vertex category represent the shape of the stack at a particular point of execution and its change. And an edge and an edge morphism represent a relation between states in two possibly different shapes of stack storage and preservation of relations by certain functions. Precisely, the vertex category $\mathbf{W}_v$ is defined as follows: let *Var* be the countably infinite set, whose elements are thought of "variables."

- objects in $\mathbf{W}_v$ are finite subsets of $Var$.
- morphisms $f: W \to V$ in $\mathbf{W}_v$: injections from $V$ to $W$.

Note the direction of a morphism is reversed; it is because we interpret each type as an RG-functor from $\mathbf{W}^{op}$ to **Span**. Each object in the vertex category is called a *world*. Let $S(W)$ be $[W \to Int]$, which denotes possible states at a world $W$. As shown in [51], each morphism $f: W \to V$ in $\mathbf{W}_v$ induces a projection map $\varphi_f: S(W) \to S(V)$ and a replacement map $\rho_f: S(W) \times S(V) \to S(W)$. They are defined as follows:

- $\varphi_f(w)(l) = w(f(l))$
- $\rho_f(w, v)(l) = \begin{cases} v(l') & \text{if } l = f(l') \text{ for some } l' \text{ in } V \\ w(l) & \text{otherwise} \end{cases}$

Intuitively, $\varphi_f$ projects a smaller state from a larger state, and $\rho_f(w, v)$ replaces the $V$ part of $w$ by $v$.

The edge category $\mathbf{W}_e$ is defined as follows:

- objects are triples $(W_0, R, W_1)$ where $W_0$ and $W_1$ are worlds (objects in the vertex category $\mathbf{W}_v$), and $R$ is a nonempty relation between $S(W_0)$ and $S(W_1)$, that is, a nonempty subset of $S(W_0) \times S(W_1)$.
- morphisms from $(W_0, R, W_1)$ to $(X_0, P, X_1)$ are pairs $(f: W_0 \to X_0, g: W_1 \to X_1)$ of morphisms in the $\mathbf{W}_v$ such that
  1. $\rho_f[R \times P \to R]\rho_g$; and
  2. $\varphi_f[R \to P]\varphi_g$.

The Grothendieck topology on $\mathbf{W}$, which we use to give the semantics of IA, is an atomic topology $\mathcal{A}$ explained in Section 4.3. Each type in IA is interpreted as a relational parametric sheaf on a site $(\mathbf{W}, \mathcal{A})$. Since relational parametric sheaves always form a sub-CCC of $\mathbf{Span}^{\mathbf{W}^{op}}$, it is enough to define base types as relational parametric sheaves. We use the fact that each type is relational to simplify the presentation: we regard $[\![\theta]\!]_w(R: V_0 \leftrightarrow V_1)$ as a relation $F_v(V_0) \leftrightarrow F_v(V_1)$ and omit the description of $[\![\theta]\!]_e$ other than presenting what relation $[\![\theta]\!]_w(R)$ is for all edges $R$ because $[\![\theta]\!]_e$ can be determined uniquely upto an isomorphism from all other existing data.

- command:
  - $[\![\mathsf{com}]\!]_v(W) = S(W) \to S(W)$
  - $[\![\mathsf{com}]\!]_v(f\colon W \to V) = \lambda c\colon [\![\mathsf{com}]\!]_v(V).\, \lambda s\colon S(W).\ \rho_f(s, c(\varphi_f(s)))$
  - $[\![\mathsf{com}]\!]_w(R\colon W_0 \leftrightarrow W_1) = [R \to R]$
- expression:
  - $[\![\mathsf{exp}]\!]_v(W) = S(W) \to \mathit{Int}$
  - $[\![\mathsf{exp}]\!]_v(f\colon W \to V) = \lambda e\colon [\![\mathsf{exp}]\!]_v(V).\, \lambda s\colon S(W).\ e(\varphi_f(s))$
  - $[\![\mathsf{exp}]\!]_w(R) = [R \to \mathbf{\Delta}_{\mathit{Int}}]$
- variable:
  - $[\![\mathsf{var}]\!]_v(W) =$
    $$\{(\lambda n.\lambda s.\, s[x \mapsto n],\ \lambda s.\, s(x)) \in (\mathit{Int} \to [\![\mathsf{com}]\!]_v(W)) \times [\![\mathsf{exp}]\!]_v(W) \mid x \in W\}$$
  - $[\![\mathsf{var}]\!]_v(f\colon W \to V) =$
    $$\lambda v\colon [\![\mathsf{var}]\!]_v(V).\ ([\![\mathsf{com}]\!]_v(f) \circ \mathsf{fst}(v),\ [\![\mathsf{exp}]\!]_v(f)(\mathsf{snd}(v)))$$
  - $[\![\mathsf{var}]\!]_w(R) = [\mathbf{\Delta}_{\mathit{Int}} \to [\![\mathsf{com}]\!](R)] \times [\![\mathsf{exp}]\!](R)$

In Section 4.3, the condition for being parametric sheaves is expressed by the existence of a unique amalgamation for every matching family. The following lemma presents another equivalent characterization for the RG-site $(\mathbf{W}, \mathcal{A})$:

**Lemma 46** *For a relational RG-functor $F$ in $\mathbf{Span}^{\mathbf{W}^{op}}$, $F$ is a parametric sheaf on $(\mathbf{W}, \mathcal{A})$ iff*

- *for every vertex morphism $f\colon V \to W$ in $\mathbf{W}$ and every $y \in F_v(V)$, if $y \cdot g = y \cdot h$ for all diagrams*
  $$U \underset{h}{\overset{g}{\rightrightarrows}} V \xrightarrow{f} W$$
  *with $f \circ g = f \circ h$, then $y = x \cdot f$ for a unique $x \in F_v(W)$ (this is equivalent to requiring that $F_v$ is an atomic sheaf [32]); and*
- *for every edge morphism $\varphi\colon P \to R$ to a non-identity edge $R$ in $\mathbf{W}$, if $x \cdot \partial_0(\varphi)\,[F_w(P)]\,y \cdot \partial_1(\varphi)$, then $x\,[F_w(R)]\,y$.*

The first condition is equivalent to saying that every matching family for any covering vertex-sieve in $\mathcal{A}_v(W)$ has a unique amalgamation. To see the equivalence, notice a fact: $f$ and $y$ in the hypothesis can generate a vertex-sieve in $\mathcal{A}_v(W)$ and a matching family for it, respectively. Intuitively, the condition says that if renaming variables in $V - f(W)$ in different ways doesn't change the meaning of a program, the program must access only the variables in $f(W)$. And under the first condition, the second condition is also equivalent to requiring that all matching families for all edge-sieves in $\mathcal{A}_w(R)$ have unique amalgamations. The equivalence follows by a similar reason: an edge-sieve in $\mathcal{A}_w(R)$ and a matching family for it can be generated by $\varphi$ and, $x \cdot \partial_0(\varphi)$ and $y \cdot \partial_1(\varphi)$, respectively. Since an RG-functor always satisfies the converse $(x\,[F_w(R)]\,y \Longrightarrow x \cdot \partial_0(\varphi)\,[F_w(P)]\,y \cdot \partial_1(\varphi))$, the second condition indicates that an expansion doesn't change being-relatedness.

Another important observation is that the vertex part of a parametric sheaf preserves pullbacks [26].

**Lemma 47** *For all parametric sheaves $F$ on $(\mathbf{W}, \mathcal{A})$, $F_v$ preserves pullbacks in ${\mathbf{W}_v}^{op}$.*

*Proof:* In [26], it is shown that the atomic sheaves on the category $\mathbf{W}_v$ preserve pullbacks in ${\mathbf{W}_v}^{op}$. The conclusion follows because $F_v$ satisfies the first condition in Lemma 46, which is equivalent to the condition for an atomic sheaf on the category $\mathbf{W}_v$. $\square$

In order to see the computational interpretation of pullback preservation, let $W$, $V$ and $X$ be worlds such that $V, X \subseteq W$. And consider the denotation $w$ of a program phrase of type $\theta$ at a world $W$ ($w \in [\![\theta]\!]_v(W)$), which only accesses $X$ and $V$: there exist $x \in [\![\theta]\!]_v(X)$ and $y \in [\![\theta]\!]_v(V)$ such that $x \cdot \mathsf{inc}_{X,W} = w$ and $y \cdot \mathsf{inc}_{V,W} = w$. The computational intuition says that $w$ only accesses $X \cap V$. The pullback preservation captures this: there exists a unique $z \in [\![\theta]\!]_v(X \cap V)$ such that $z \cdot \mathsf{inc}_{X \cap V,W} = w$.

**Proposition 48** $[\![\mathsf{com}]\!]$, $[\![\mathsf{exp}]\!]$ *and* $[\![\mathsf{var}]\!]$ *are parametric sheaves on the RG-site* $(\mathbf{W}, J)$*.*

*Proof:* We show the conditions in Lemma 46 hold for $[\![\mathsf{exp}]\!]$. The case of $[\![\mathsf{com}]\!]$ is similar, and $[\![\mathsf{var}]\!]$ can be handled using the fact that $[\![\mathsf{var}]\!]$ is an RG-subfunctor of a parametric sheaf $(\mathit{Int} \Rightarrow [\![\mathsf{com}]\!]) \times [\![\mathsf{exp}]\!]$.

For the first condition in Lemma 46, let $y \in [\![\mathsf{exp}]\!]_v(V)$ be such an expression. Given $s \in S(W)$, there exists $s' \in S(V)$ such that $\varphi_f(s') = s$. Define $x \in [\![\mathsf{exp}]\!]_v(W)$ by $x(s) = y(s')$

for such $s' \in S(V)$. We note that because of the assumption on $y$, $x(s)$ doesn't depend on the choice of $s'$. More precisely, consider $s', s'' \in S(V)$ such that $\varphi_f(s') = \varphi_f(s'')$. Then, there exist $g, h\colon U \to V$ and $r \in S(U)$ such that

$$(\varphi_g(r) = s') \wedge (\varphi_h(r) = s'') \wedge (f \circ g = f \circ h)$$

Then, by assumption, we have $y(s') = y(\varphi_g(r)) = y(\varphi_h(r)) = y(s'')$. It is now easy to show $x \cdot f = y$, which is the existence part. The uniqueness is straightforward.

For the second condition, let $\varphi$ be an edge morphism in $\mathbf{G}$ of the form:

$$\begin{tikzcd} U_0 \arrow[r, "f_0"] \arrow[d, leftrightarrow, "P"'] \arrow[dr, phantom, "\varphi"] & V_0 \arrow[d, leftrightarrow, "R"] \\ U_1 \arrow[r, "f_1"'] & V_1 \end{tikzcd}$$

where $R$ is a non-identity edge. We should show that for any $x \in [\![\mathsf{exp}]\!]_v(V_0)$ and $y$ in $[\![\mathsf{exp}]\!]_v(V_1)$ such that $x \cdot f_0\, [[\![\mathsf{exp}]\!]_w(P)]\, y \cdot f_1$, we have $x\, [[\![\mathsf{exp}]\!]_w(R)]\, y$. We note that for every $s_0 \in S(V_0)$ and $s_1 \in S(V_1)$ such that $s_0\, R\, s_1$, there exists $r_0 \in S(U_0)$ and $r_1 \in S(U_1)$ such that $\varphi_{f_0}(r_0) = s_0$ and $\varphi_{f_1}(r_1) = s_1$ and $r_0\, P\, r_1$ by the nonemptiness of $P$ and the definition of edge morphisms in $\mathbf{W}_e$. Since $x \cdot f_0\, [[\![\mathsf{exp}]\!]_w(P)]\, y \cdot f_1$, $(x \cdot f_0)(r_0) = (y \cdot f_1)(r_1)$. By unrolling the definition, $x(\varphi_{f_0}(r_0)) = y(\varphi_{f_1}(r_1))$, from which $x(s_0) = y(s_1)$ follows. $\square$

Recall that two program phrases do not interfere with each other symmetrically when they use disjoint sets of variables. In order to formalize such an intuition properly, we use the notion of *support* to capture precisely what variables each program accesses:

**Definition 49 (support)** *For any parametric sheaf $F$ on the site $(\mathbf{W}, \mathcal{A})$ and $a \in F_v(W)$, a subset $W'$ of $W$ is called the* support *of $a$ if it is the least set with an element $a' \in F_v(W')$ such that $a' \cdot \mathsf{inc}_{W',W} = a$.*

For instance, the supports of the following commands at a world $\{x_0, x_1\}$ are as follows:

| command | support |
|---|---|
| $\lambda s\colon S(\{x_0, x_1\}).\, s[x_0 \mapsto 1, x_1 \mapsto 1]$ | $\{x_0, x_1\}$ |
| $\lambda s\colon S(\{x_0, x_1\}).\, s[x_0 \mapsto 1]$ | $\{x_0\}$ |
| $\lambda s\colon S(\{x_0, x_1\}).\, s$ | $\emptyset$ |

One might wonder when the support of an element exists. This is the place where the additional requirement for each type being a parametric sheaf plays a role.

**Proposition 50** *For all parametric sheaves $F$ on the site $(\mathbf{W}, \mathcal{A})$, every $a \in F_v(W)$ has a support.*

*Proof:* For all parametric sheaves $F$ on the site $(\mathbf{W}, \mathcal{A})$, the vertex part $F_v$ preserves pullbacks in $\mathbf{W}^{op}$ by Lemma 47. The existence of the support of $a$ follows from this pullback preservation of $F_v$ [49]. $\square$

We write $\mathsf{supp}(a)$ to denote the support of $a$. With the notion of support, the meaning of symmetric noninterference is easily obtained by interpreting "disjoint" in the intuition literally.

**Definition 51** *For any $a \in [\![\theta_1]\!]_v(W)$ and $b \in [\![\theta_2]\!]_v(W)$, $a$ and $b$* do not interfere symmetrically *$(a \perp b)$ iff $\mathsf{supp}(a) \cap \mathsf{supp}(b) = \emptyset$.*

The remaining question is whether this definition of symmetric noninterference is satisfactory in the sense that all the principles in Chapter 2 are valid. We only show that the second and the third reasoning principles are valid; the fourth principle immediately follows because the definition of $\perp$ itself is symmetric; and we postpone the treatment of the first principle to Part III because the first principle doesn't impose any difficulty once we have a reasonable interpretation of $\mathsf{new}$ and handling it complicates the presentation. The second and third principles can be restated by the following single lemma because each term, including an application, is interpreted as a parametric natural transformation.

**Lemma 52** *For all $p \in [\![\theta]\!]_v(W)$, $p_1 \in [\![\theta_1]\!]_v(W)$, $p_2 \in [\![\theta_2]\!]_v(W)$, and all parametric natural transformations $\eta\colon [\![\theta_1]\!] \times [\![\theta_2]\!] \to [\![\theta_3]\!]$, if $p \perp p_1$ and $p \perp p_2$, then $p \perp (\eta_v)_W(p_1, p_2)$.*

To see why the lemma holds, we show properties of $\perp$ for a parametric natural transformations and for a paring operation as in [49].

**Lemma 53** *For all $p \in [\![\theta]\!]_v(W), p' \in [\![\theta']\!]_v(W)$, and all parametric natural transformations $\eta$ of type $[\![\theta']\!] \to [\![\theta'']\!]$, if $p \perp p'$, then $p \perp (\eta_v)_W(p')$.*

*Proof:* Let $p'_0$ be an element in $[\![\theta']\!]_v(\mathsf{supp}(p'))$ such that $[\![\theta']\!]_v(\mathsf{inc}_{\mathsf{supp}(p'),W})(p'_0) = p'$. Then,

$$\begin{array}{rcll} (\eta_v)_W(p') & = & (\eta_v)_W([\![\theta']\!]_v(\mathsf{inc}_{\mathsf{supp}(p'),W})(p'_0)) & \\ & = & [\![\theta'']\!]_v(\mathsf{inc}_{\mathsf{supp}(p'),W})((\eta_v)_{\mathsf{supp}(p')}(p'_0)) & \because \text{naturality of } \eta_v \end{array}$$

So, $\mathsf{supp}((\eta_v)_W(p')) \subseteq \mathsf{supp}(p')$. Since $\mathsf{supp}(p') \cap \mathsf{supp}(p) = \emptyset$, $\mathsf{supp}((\eta_v)_W(p')) \cap \mathsf{supp}(p) = \emptyset$. Therefore, the conclusion follows. $\square$

**Lemma 54** *For all $p \in [\![\theta]\!]_v(W)$, $p_1 \in [\![\theta_1]\!]_v(W)$ and $p_2 \in [\![\theta_2]\!]_v(W)$, if $p \bot p_1$ and $p \bot p_2$, then $p \bot (p_1, p_2)$.*

*Proof:* Since $p \bot p_1$ and $p \bot p_2$, $\mathsf{supp}(p) \cap (\mathsf{supp}(p_1) \cup \mathsf{supp}(p_2)) = \emptyset$. Since $\mathsf{supp}((p_1, p_2)) = \mathsf{supp}(p_1) \cup \mathsf{supp}(p_2)$, the conclusion follows. $\square$

# PART II

# Local Reasoning for Programs with Pointers

# CHAPTER 5

## Introduction to Part II

In popular programming languages such as C, C++ and Java, pointers have been considered as a main source of errors. In [19], Hoare and He gave a logical explanation for the difficulty of handling pointers: when the "link" field of a heap cell is changed to denote a different heap cell, which Hoare and He called *pointer swing*, it affects infinitely many "seemingly innocent" expressions. For instance, suppose that the variable $x$ points to a heap cell whose $next$ field points to the cell itself. When we assign an address $p$, which is different from the value of $x$, to the $next$ field of the heap cell ($x.next := p$), it affects not only $x.next$, but also seemingly unrelated expressions: all expressions containing $x.next$, such as $x.next.next$ and $x.next.next.key$, and expressions $y.next$ and $y.next.key$ when $x = y$. As a consequence, even though a pointer swing is a change of just a single heap cell, writing a correct program with pointer swings requires considerable effort.

Most of reasoning formalisms for pointers take a conservative approach for the pointer swing problem: whenever there is a pointer swing, check all expressions of interest and make sure that they are updated properly. Therefore, it is not surprising that reasoning within such formalisms often becomes very complicated.

Recently, Reynolds proposed the use of a *spatial conjunction connective* in reasoning about pointers [67]. With the spatial conjunction connective, one can state the conjunction of two properties and, at the same time, the disjointedness of heap cells which may affect the truth of two assertions. More formally, the spatial conjunction $P * R$ of assertions $P$ and $R$ holds in a heap when the heap can be partitioned into two subheaps so that $P$ and $R$ hold for each subheap. With this connective, Reynolds devised a simple rule for a pointer swing:

$$\{(E \mapsto -, E_2) * P\}\, E.1 := E_1\, \{(E \mapsto E_1, E_2) * P\}$$

Notice that since the assertion $P$ doesn't depend on the cell being changed, $P$ becomes an invariant. The "spatial" information in spatial conjunction makes it possible to reflect in the formalism the fact that only a single heap cell is changed by a pointer swing.

The idea was further developed by Ishtiaq and O'Hearn in [24] in two directions. First, they showed that Reynolds' assertion language with spatial conjunction is better explained in the context of the Logic of Bunched Implications (in short, **BI** [56]). Their new insight gives a way to obtain a weakest precondition for all programming constructs other than loops [67] and turns out to be also of practical importance [47, 77]. Second, O'Hearn proposed the idea of local reasoning; just as a single pointer swing has evident locality, a block of a program also has such locality, and more efficient reasoning can be obtained exploiting such locality of a program with spatial conjunction. He formulated his idea by giving a rule to derive a global property from a local one.

FRAME RULE

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}}\ C \text{ doesn't write to any identifiers in } R$$

Part II of the thesis builds on the work by Reynolds, Ishtiaq and O'Hearn. The main contribution in Part II is to provide theoretical and practical justification for the benefit of the spatial conjunction connective and local reasoning based on O'Hearn's Frame Rule (in short, FR). First, we present a "completeness" result of FR in Chapter 6. FR is attractive for reasoning because with FR, one can derive properties of a program from a given property without looking at the actual implementation of the program. The completeness results provide a sense in which with FR, one can derive "all" properties of a program which follow from a given property without referring to the actual implementation. Second, we provide evidence for the effectiveness of local reasoning in practice in Chapter 7: the Schorr-Waite graph marking algorithm is verified with local reasoning in mind. The algorithm has been verified a couple of times with different pointer formalisms and therefore becomes a good testbed for the effectiveness of local reasoning. The resulting verification turns out to be simpler than the ones in traditional formalisms. Finally, we study the algorithmic aspects of the **BI** assertion language in Chapter 8, which is an enveloping language for spatial conjunction. Besides the usual theoretical interest, the decidability result of such an assertion language is crucial to justify that a pointer logic based on spatial conjunction does not just shift the difficulty of reasoning from pointers to the spatial conjunction connective $*$ (and its adjoint $-\!*$ in the **BI** assertion language), but indeed reduces the complexity. We show that the whole assertion language is undecidable. The main reason for undecidability is quantifiers. We point out that without quantifiers, the assertion language, including $*$ and $-\!*$, becomes decidable. Such a result is also a starting point for applying the pointer logic based on the **BI** assertion language to an automatic program analysis.

Since we are not considering procedures in Part II, each identifier in a program denotes a stack variable, and distinct identifiers denote distinct stack variables. We use this fact to simplify the presentation in Part II: instead of having another name space for identifiers in a program other than the name space *Var* for stack variables, we use *Var* for both identifiers and stack variables; we also take the name of each identifier to be identical to that of the stack variable which the identifier denotes, thereby eliminating the need for a mapping (such as environment) from identifiers to stack variables. We call both identifiers and stack variables simply *variables*, but when it is necessary to emphasize that a variable is indeed an identifier which does not denote any stack variables, we call the variable *auxiliary variable*. And in Part II, we call O'Hearn's Frame Rule *the Frame Rule* or *FR*. This is not to be confused with the Frame Rule in Part III, which is a generalization of O'Hearn's FR for programs with procedures and pointers. However, the analysis of O'Hearn's FR in Part II, in particular the completeness result in Chapter 6, is applicable to the Frame Rule in Part III.

In the remainder of this chapter, we review the setting for Part II: the programming language **WHILE**h, the **BI** assertion language and **BI** pointer logic. More detailed explanations can be found in [67, 24, 47, 68].

## 5.1 Background

### 5.1.1 WHILEh

In [67], Reynolds used an extension of a simple imperative programming language **WHILE** with constructs for manipulating the heap via pointers, which we call **WHILE**h in the thesis. The main characteristic of the language is that all heap manipulations, such as dereferencing pointers and creating a new heap cell, are done in *commands* not expressions. As a consequence, expressions depend only on stack storage. For instance, the conditional statement $\mathsf{if}\ (x.1 = y.1)\ \mathsf{then\ skip\ else\ skip}$ can not be written in **WHILE**h because the expression $x.1 = y.1$ involves dereferencing two heap cells pointed to by $x$ and $y$. Instead, an equivalent program can be written in **WHILE**h with two additional variables. Namely, $z := x.1; w := y.1;\ \mathsf{if}\ (z = w)\ \mathsf{then\ skip\ else\ skip}$. This restriction does not result in the loss of expressive power because programs with heap-dependent expressions can be translated into those in **WHILE**h. Moreover, it simplifies theory and validates Hoare's simple assignment rule for variables. In Part II, we use **WHILE**h.

The full grammar for expressions and commands in **WHILE**h is as follows: for $i = 1, 2$,

$$\begin{array}{lcl} \text{commands} \quad C & ::= & x := E.i \mid E.i := E \mid x := \mathsf{cons}(E, E) \mid \mathsf{dispose}(E) \\ & & \mid x := E \mid \mathsf{skip} \mid C; C \mid \mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od} \\ \text{expression} \quad E & ::= & \mathsf{nil} \mid x, y, \ldots \mid 0 \mid 1 \mid E + E \mid E - E \mid E \times E \\ & & \mid \mathsf{false} \mid \mathsf{true} \mid E \Rightarrow E \mid E = E \mid E < E \end{array}$$

The language is untyped, and it also has Lisp-like constructs for manipulating cells in the heap: $x := E.i$ reads the $i$-th field of the cell $E$ and assigns the read value to $x$; $E.i := E'$ updates the $i$-th field of the heap cell $E$ by $E'$; and $x := \mathsf{cons}(E_1, E_2)$ creates a new heap cell with content $(E_1, E_2)$ and stores its address in the variable $x$.

Storage is modelled by a pair consisting of a stack and a heap. A stack is just a function from variables to values, and a heap is a finite function from locations to a pair of values, where the domain of the function denotes all heap cells currently allocated in memory. More formally, the semantic domains for storage are defined as follows: let *Var*, *Loc* and *Atom* be disjoint countably infinite sets such that any of them doesn't contain *nil*, and *Atom* includes the set of integers *Int* and the set of boolean values $\{true, false\}$.

$$\begin{array}{rcl} Val & \triangleq & Atom \cup Loc \cup \{nil\} \\ Stack & \triangleq & Var \rightarrow Val \\ Heap & \triangleq & Loc \rightharpoonup_{fin} Val \times Val \\ State & \triangleq & Stack \times Heap \end{array}$$

To state the semantics formally, we use the following notations:

1. for $h, h' \in \mathit{Heap}$, formula $h \# h'$ indicates that the domains of $h$ and $h'$ are disjoint, that is, $\mathit{dom}(h) \cap \mathit{dom}(h') = \emptyset$;

2. for $h, h' \in \mathit{heap}$, the heap $h * h'$ is defined only when $h \# h'$ holds and is a finite function obtained by taking the union of $h$ and $h'$;

3. $i \mapsto j$ is a singleton partial function which maps $i$ to $j$; and

4. for partial functions $f$ from $U$ to $W$ and $f'$ from $V$ to $W$, partial function $f[f']$ from $U \cup V$ to $W$ is defined by:

$$f[f'](i) \triangleq \begin{cases} f'(i) & \text{if } i \in \mathit{dom}(f') \\ f(i) & \text{if } i \notin \mathit{dom}(f') \wedge i \in \mathit{dom}(f) \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Each expression is interpreted as a function from *Stack* to *Val*. Note that an expression only depends on stacks because its domain is *Stack* not *State*. The interpretation is the usual one except that whenever there is a type error, the interpretation of an expression becomes *nil*.

We give an operational semantics for commands in Table 5.1. Each command is interpreted using a relation $\rightsquigarrow$ between configurations, which are triples consisting of a command, a stack and a heap. Intuitively, the relation represents computation steps.

We use some terminology to specify certain properties of computations.

- "$(C, s, h)$ is terminal" if $C \equiv \mathsf{skip}$.

- "$(C, s, h)$ is stuck" if it is not terminal and there is no configuration $K$ such that $C, s, h \rightsquigarrow K$.

- "$(C, s, h)$ is safe" if for all configurations $K$ such that $C, s, h \rightsquigarrow^* K$, configuration $K$ is not stuck.

- "$(C, s, h)$ must terminate normally" if all maximal $\rightsquigarrow$-sequences starting from $(C, s, h)$ are finite and end with terminal configurations.

In the interpretation of $x := E.i$, $E.i := E'$ and $\mathsf{dispose}(E)$, there is a next step if and only if $[\![E]\!]s$ is in the domain of the initial heap, that is, a command doesn't dereferences dangling pointers or non-addresses such as *nil* and natural numbers. In the thesis, we assume that a

$$\frac{}{x := E, s, h \rightsquigarrow \mathsf{skip}, s[x \mapsto [\![E]\!]s], h}$$

$$\frac{[\![E]\!]s = l \in dom(h) \quad h(l) = (v_1, v_2)}{x := E.1, s, h \rightsquigarrow \mathsf{skip}, s[x \mapsto v_1], h} \qquad \frac{[\![E]\!]s = l \in dom(h) \quad h(l) = (v_1, v_2)}{x := E.2, s, h \rightsquigarrow \mathsf{skip}, s[x \mapsto v_2], h}$$

$$\frac{[\![E]\!]s = l \in dom(h) \quad h(l) = (v_1, v_2)}{E.1 := E', s, h \rightsquigarrow \mathsf{skip}, s, h[l \mapsto ([\![E']\!]s, v_2)]}$$

$$\frac{[\![E]\!]s = l \in dom(h) \quad h(l) = (v_1, v_2)}{E.2 := E', s, h \rightsquigarrow \mathsf{skip}, s, h[l \mapsto (v_1, [\![E']\!]s)]}$$

$$\frac{l \in Loc \quad l \notin dom(h)}{x := \mathsf{cons}(E_1, E_2), s, h \rightsquigarrow \mathsf{skip}, s, h * [l \mapsto ([\![E_1]\!]s, [\![E_2]\!]s)]}$$

$$\frac{[\![E]\!]s \in Loc \quad [\![E]\!]s \notin dom(h)}{\mathsf{dispose}(E), s, h * [[\![E]\!]s \mapsto (v_1, v_2)] \rightsquigarrow \mathsf{skip}, s, h}$$

$$\frac{C_1, s, h \rightsquigarrow C_1', s', h'}{C_1; C_2, s, h \rightsquigarrow C_1'; C_2, s', h'} \qquad \frac{}{\mathsf{skip}; C_2, s, h \rightsquigarrow C_2, s, h}$$

$$\frac{[\![E]\!]s \neq true}{\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s, h \rightsquigarrow C_2, s, h} \qquad \frac{[\![E]\!]s = true}{\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s, h \rightsquigarrow C_1, s, h}$$

$$\frac{[\![E]\!]s \neq true}{\mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}, s, h \rightsquigarrow \mathsf{skip}, s, h}$$

$$\frac{[\![E]\!]s = true}{\mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}, s, h \rightsquigarrow (C; \mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}), s, h}$$

**Table 5.1** Operational Semantics of **WHILE**h machine immediately stops execution and reports an error when a dangling pointer or a non-address is dereferenced, and we call such an error *memory fault*. Since all atomic commands other than $x := E.i$, $E.i := E'$ and $\mathsf{dispose}(E)$ never get stuck, a configuration in general is safe if and only if while the configuration is run, it never generates a memory fault. Note that when a configuration must terminate normally, it is also safe.

Each command exhibits local behavior for stack storage and heap storage. Such local behavior is crucial to validate the Frame Rule, as we discuss in Section 5.1.3. For stack storage, locality says: when a variable is not assigned by a command, the value of the variable is not changed by the command; and when, in addition, the variable isn't even read, the variable doesn't affect the execution of the command. The lemma below formalizes this:

**Lemma 55 (Stack Locality)** *For a command $C$, let $\mathit{ModifyOnly}(C)$ be the set of variables assigned in the command $C$ and $FV(C)$ the set of variables occurring free in $C$.*

1. *Write Locality: if $C, s, h \leadsto^* \mathsf{skip}, s', h'$ and $x \notin \mathit{ModifyOnly}(C)$, then $s(x) = s'(x)$.*
2. *Access Locality: if $C, s, h \leadsto^* \mathsf{skip}, s', h'$, then for any function $s_0\colon (\mathit{Var} - FV(C)) \to \mathit{Val}$, we have that $C, s[s_0], h \leadsto^* \mathsf{skip}, s'[s_0], h'$.*

*Proof:* Both locality properties can be shown by induction over the length of computation $C, s, h \leadsto^* \mathsf{skip}, s', h'$. $\square$

The local behavior of commands for heap storage is more subtle. Informally, it describes that when we are given a safe configuration $(C, s, h)$ and add a new heap cell $l$ to $h$ with content $(v_1, v_2)$, no computations of $(C, s, h * (l \mapsto (v_1, v_2)))$ access $l$; so, for any finite/infinite computation sequence of $(C, s, h * (l \mapsto (0, 0)))$, deleting the cell $l$ from the sequence gives back a computation sequence of $(C, s, h)$. Notice that we don't say that adding a heap cell $l$ to a computation sequence of $(C, s, h)$ gives one for $(C, s, h * (l \mapsto (v_1, v_2)))$. It doesn't hold since the newly added heap cell $l$ might reduce the choice of a new location when $\mathsf{cons}$ is executed. For instance, $x := \mathsf{cons}(0, 0), s, \emptyset \leadsto \mathsf{skip}, s[x \mapsto l], (l \mapsto (0, 0))$ holds but $x := \mathsf{cons}(0, 0), s, (l \mapsto (0, 0)) \leadsto \mathsf{skip}, s[x \mapsto l], (l \mapsto (0, 0)) * (l \mapsto (0, 0))$ doesn't. The following lemmas indicate that all computations of $(C, s, h * (l \mapsto (v_1, v_2)))$ are from those of $(C, s, h)$. The first lemma on monotonicity deals with nonterminating computations, and the second "Heap Locality" deals with terminating computations.

**Lemma 56 (Safety and Termination Monotonicity)**

1. *If $C, s, h$ is safe and $h \# h'$, then $C, s, h * h'$ is safe.*

2. *If $C, s, h$ must terminate normally and $h \# h'$, then $C, s, h * h'$ must terminate normally.*

*Proof:* Suppose $C, s, h * h'$ is not safe. Then, there is a computation sequence from $C, s, h * h'$, which gets stuck. Taking out $h'$ from the computation sequence results in a computation sequence from $C, s, h$ which gets stuck (possibly earlier than the computation sequence from $C, s, h * h'$). So, $C, s, h$ is not safe, either.

For the termination monotonicity, it suffices to show that $C, s, h * h'$ always terminates because the safety of $C, s, h * h'$ follows from the safety monotonicity. Suppose that there is an infinite computation sequence from $C, s, h * h'$. Then, taking out $h'$ from the computation sequence results in a computation sequence from $C, s, h$ which either is infinite or gets stuck. Both cases amount to the fact that $C, s, h$ must terminate normally. $\Box$

In order to see the connection of the above lemma with nontermination, note that when $C, s, h$ is safe and the first claim holds, the second claim is equivalent to

> if $C, s, h * h'$ has an infinite computation, so does $C, s, h$.

**Lemma 57 (Heap Locality)** *When $C, s, h_0$ is safe and $C, s, h_0 * h_1 \leadsto^* \mathsf{skip}, s', h'$, there is $h_0'$ such that $C, s, h_0 \leadsto^* \mathsf{skip}, s', h_0'$ and $h' = h_0' * h_1$.*

*Proof:* We prove the following stronger property:

> When $C, s, h_0$ is safe and $C, s, h_0 * h_1 \leadsto^* C', s', h'$, there is $h_0'$ such that $C, s, h_0 \leadsto^* C', s', h_0'$ and $h' = h_0' * h_1$.

The property can be shown by induction over the length of computation $C, s, h_0 * h_1 \leadsto^* C', s', h'$. $\Box$

### 5.1.2 The BI Assertion Language

The **BI** assertion language is first-order classical logic with a points-to relation ($E \mapsto E_1, E_2$), equality ($E = E'$) and three more connectives: spatial conjunction $*$, spatial implication $-\!\!*$ and empty heap $\mathsf{emp}$. Formally, the context-free grammar for the assertion language can be given as follows:

$$
\begin{array}{rcll}
P & ::= & (E \mapsto E, E) & \text{Points-to Relation} \\
 & & \mid E = E & \text{Equality} \\
 & & \mid \mathsf{F} & \text{Falsity} \\
 & & \mid P \Rightarrow P & \text{Classical Implication} \\
 & & \mid \mathsf{emp} & \text{Empty Heap} \\
 & & \mid P * P & \text{Spatial Conjunction} \\
 & & \mid P \mathrel{-\!\!*} P & \text{Spatial Implication} \\
 & & \mid \forall x.\, P & \text{Universal Quantification}
\end{array}
$$

Other connectives in classical logic are definable from the ones above: $\neg P \triangleq (P \Rightarrow \mathsf{F})$, $\mathsf{T} \triangleq \neg\mathsf{F}$, $P \wedge Q \triangleq \neg(P \Rightarrow \neg Q)$, $P \vee Q \triangleq \neg(\neg P \wedge \neg Q)$ and $\exists x.P \triangleq \neg\forall x.\neg P$.

The formal semantics of an assertion is given by a forcing relation $\models$ between a state and an assertion; $(s, h) \models P$ means that the assertion $P$ holds in the state $(s, h)$. The relation is defined inductively in Table 5.2. The points-to relation $(E \mapsto E_1, E_2)$ denotes that only one heap cell $E$ is currently allocated in memory and its content is $(E_1, E_2)$. Three additional connectives provide ways to "manipulate" heaps with assertions, such as splitting a heap, adding a new heap and pointing out the empty heap: $P * Q$ holds in a heap if the heap can be split into two subheaps so that $P$ and $Q$ hold for each; $P \mathrel{-\!\!*} Q$ holds in a heap if whenever a new heap satisfying $P$ is added to the old heap, the combined heap satisfies $Q$; assertion $\mathsf{emp}$ holds in a heap if the heap is empty.

When $(s, h) \models P$ implies $(s, h) \models Q$ for all states $(s, h)$, we write $P \models Q$ to denote it. The relation $\models$ between assertions satisfies all valid implications in first-order classical logic. In addition, the interpretation of assertions validates the commutative monoid laws for $\mathsf{emp}$ and $*$, the "parallel rule" for $*$ and the "adjunction rules" for $\mathrel{-\!\!*}$:

$$
\frac{}{P * Q \models Q * P} \quad \frac{}{(P * Q) * R \models P * (Q * R)} \quad \frac{}{\mathsf{emp} * R \models R} \quad \frac{}{R \models \mathsf{emp} * R}
$$

$$
\frac{P \models Q \quad R \models S}{P * R \models Q * S}
$$

$$
\frac{P * R \models S}{P \models R \mathrel{-\!\!*} S} \quad \frac{P \models R \mathrel{-\!\!*} S \quad Q \models R}{P * Q \models S}
$$

| | | |
|---|---|---|
| $s, h \models (E \mapsto E_1, E_2)$ | iff | $dom(h) = [\![E]\!]s$ and $h([\![E]\!]s) = ([\![E_1]\!]s, [\![E_2]\!]s)$ |
| $s, h \models E = E'$ | iff | $[\![E]\!]s = [\![E']\!]s$ |
| $s, h \models \mathsf{F}$ | iff | False |
| $s, h \models P \Rightarrow P'$ | iff | if $s, h \models P$, then $s, h \models P'$ |
| $s, h \models \mathsf{emp}$ | iff | $dom(h) = \emptyset$ |
| $s, h \models P_1 * P_2$ | iff | there exist $h_1$ and $h_2$ such that $h_1 \# h_2, h_1 * h_2 = h, s, h_1 \models P_1, s, h_2 \models P_2$ |
| $s, h \models P_1 \mathrel{-\!\!*} P_2$ | iff | for all $h_1$ such that $h \# h_1$ and $(s, h_1) \models P_1$, $(s, h * h_1) \models P_2$ |
| $s, h \models \forall x. P$ | iff | for any $v$ in $Val, s[x \mapsto v], h \models P$ |

**Table 5.2** Semantics of the **BI** Assertion Language

### 5.1.3 BI Pointer Logic

Using three additional connectives $*, \mathrel{-\!\!*}, \mathsf{emp}$ for assertions, we can obtain proof rules for commands in **WHILE**h, which are simpler than those based on component substitution [40, 6]. In this section, we review **BI** pointer logic, which is an extension of Hoare logic for **WHILE**h and exploits the additional connectives in the **BI** assertion language.

Each Hoare triple can be interpreted for partial correctness and for total correctness:

- *Partial Correctness*: $\{P\}\, C\, \{Q\}$ holds iff for all states $(s, h)$ with $(s, h) \models P$,

  1. $(C, s, h)$ is safe; and
  2. for all $(s', h')$ such that $(C, s, h) \rightsquigarrow^* (\mathsf{skip}, s', h')$, we have that $(s', h') \models Q$.

- *Total Correctness*: $\{P\}\, C\, \{Q\}$ holds iff for all states $(s, h)$ with $(s, h) \models P$,

  1. $(C, s, h)$ must terminate normally; and
  2. for all $(s', h')$ such that $(C, s, h) \rightsquigarrow^* (\mathsf{skip}, s', h')$, we have that $(s', h') \models Q$.

Note that the interpretation of a Hoare triple for partial correctness is slightly different from the usual one: in the first condition, we require that all computations from a state satisfying the precondition should not "go wrong": they should not generate any memory faults. For instance, $\{\mathsf{T}\}\, x.1 := 2\, \{\mathsf{T}\}$ doesn't hold because in a state $(s, \emptyset)$, the assignment $x.1 := 2$ generates a memory fault. Such an interpretation was first proposed by O'Hearn in [24], where he calls it the *tight* interpretation and makes the slogan "a well-specified program doesn't go wrong."

The tight interpretation plays a crucial role in the soundness of the Frame Rule, which we explain at the end of this section. Note that the total correctness interpretation also has such a requirement because $(C, s, h)$ is safe if $(C, s, h)$ must terminate normally.

The proof rules in **BI** pointer logic are shown in Table 5.3 and Table 5.4. Table 5.3 contains rules for programming constructs, and Table 5.4 shows *structural* rules, i.e., rules which don't depend on particular programming constructs. All rules other than the one for loops handle both partial correctness and total correctness. For handling loops, we only show one for partial correctness because incorporating a proper rule for total correctness unnecessarily complicates the presentation.[1] All the rules are from [67, 24, 47].

The forward rule for component assignment illustrates the advantage of the formalism: a component-assignment statement changes only a single heap cell, and the rule exactly reflects this. One might worry that the rule might not be enough because it requires a precondition to have a particular shape. We show that for any postcondition, the weakest precondition for component assignment can be obtained with the forward rule and structural rules. We first note that the backward rule, on the other hand, provides a complete treatment of component assignment since it gives the weakest precondition for an arbitrary postcondition. In order to see the reason, consider the clause $(E \mapsto x, y) * ((E \mapsto E_1, y) \mathrel{-\!\!*} P)$. The interpretation of $(E \mapsto x, y) * ((E \mapsto E_1, y) \mathrel{-\!\!*} P)$ is that the current heap contains a heap cell $E$ whose content is $(x, y)$ and if we take out the cell $E$ from the heap, the remaining part satisfies $((E \mapsto E_1, y) \mathrel{-\!\!*} P)$. And $(E \mapsto E_1, y) \mathrel{-\!\!*} P$ means that if a cell $E$ is not allocated currently so that it is possible to allocate the cell $E$, then when the heap cell $E$ with its content $(E_1, y)$ is added, the resulting heap satisfies $P$. Therefore, $(E \mapsto x, y) * ((E \mapsto E_1, y) \mathrel{-\!\!*} P)$ expresses that a heap contains a heap cell $E$ whose content is $(x, y)$ and if we take out the heap cell $E$ and add the same cell $E$ but with possibly different content $(E_1, y)$, the resulting heap satisfies $P$. In this way, the use of $*$ with $\mathrel{-\!\!*}$ describes the update of a heap cell. Now, it becomes easy to see that the backward rule for component assignment does give the weakest precondition for an assertion $P$. And the following derivation shows that the backward rule is derivable from the forward one using Consequence and Auxiliary Variable Elimination: given a postcondition $P$, let $x, y$

[1] Since the language has countably nondeterministic constructs, we need countable ordinals to formulate a proper rule for loops in total correctness verification. The details can be found in [1].

SEQUENCING

$$\frac{\{P\}\,C\,\{Q\} \quad \{Q\}\,C'\,\{R\}}{\{P\}\,C;C'\,\{R\}}$$

LOOP

$$\frac{\{P \wedge (E = \mathsf{true})\}\,C\,\{P\}}{\{P\}\,\mathsf{while}\,E\,\mathsf{do}\,C\,\mathsf{od}\,\{P \wedge \neg(E = \mathsf{true})\}}$$

CONDITIONAL

$$\frac{\{P \wedge (E = \mathsf{true})\}\,C_1\,\{Q\} \quad \{P \wedge \neg(E = \mathsf{true})\}\,C_2\,\{Q\}}{\{P\}\,\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\,\{Q\}}$$

SKIP

$$\{P\}\,\mathsf{skip}\,\{P\}$$

SIMPLE ASSIGNMENT

$$\{P[E/x]\}\,x := E\,\{P\}$$

FORWARD COMPONENT ASSIGNMENT

$$\{(E \mapsto E_1, E_2) * P\}\,E.1 := E'\,\{(E \mapsto E', E_2) * P\}$$

BACKWARD COMPONENT ASSIGNMENT

($x, y$ are different and fresh)

$$\{\exists x, y.\,(E \mapsto x, y) * ((E \mapsto E', y) -\!\!* P)\}\,E.1 := E'\,\{P\}$$

FORWARD CONS

($x'$ is fresh)

$$\{P\}\,x := \mathsf{cons}(E_1, E_2)\,\{\exists x'.\,P[x'/x] * (x \mapsto E_1[x'/x], E_2[x'/x])\}$$

BACKWARD CONS

($x'$ is not free in $E_1, E_2, P$)

$$\{\forall x'.\,(x' \mapsto E_1, E_2) -\!\!* P[x'/x]\}\,x := \mathsf{cons}(E_1, E_2)\,\{P\}$$

COMPONENT LOOKUP

($x_1$,$x_2$ are different and not free in $P$ and $E$)

$$\{\exists x_1.\,P[x_1/x] \wedge (\exists x_2.\,(E \mapsto x_1, x_2) * \mathsf{T})\}\,x := E.1\,\{P\}$$

DISPOSE

($x_1, x_2$ are different and not free in $E$)

$$\{P * \exists x_1, x_2.\,(E \mapsto x_1, x_2)\}\,\mathsf{dispose}(E)\,\{P\}$$

**Table 5.3** Rules for Program Constructs in **BI** Pointer Logic

CONSEQUENCE

$$\frac{P' \models P \quad \{P\}\, C\, \{Q\} \quad Q \models Q'}{\{P'\}\, C\, \{Q'\}}$$

AUXILIARY VARIABLE RENAMING
($x$ is not free in $C$ and $y$ is fresh)

$$\frac{\{P\}\, C\, \{Q\}}{\{P[y/x]\}\, C\, \{Q[y/x]\}}$$

AUXILIARY VARIABLE ELIMINATION
($x$ is not free in $C$)

$$\frac{\{P\}\, C\, \{Q\}}{\{\exists x.\, P\}\, C\, \{\exists x.\, Q\}}$$

FRAME RULE
($\mathit{ModifyOnly}(C)$ and $FV(R)$ are disjoint)

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

where $\mathit{ModifyOnly}(C)$ denote the set of variables assigned to in $C$:

$$\begin{array}{rcl}
\mathit{ModifyOnly}(x := E.i) & = & \{x\} \\
\mathit{ModifyOnly}(E.i := E') & = & \{\ \} \\
\mathit{ModifyOnly}(x := \mathsf{cons}(E_1, E_2)) & = & \{x\} \\
\mathit{ModifyOnly}(\mathsf{dispose}(E)) & = & \{\ \} \\
\mathit{ModifyOnly}(x := E) & = & \{x\} \\
\mathit{ModifyOnly}(\mathsf{skip}) & = & \{\ \} \\
\mathit{ModifyOnly}(C_1; C_2) & = & \mathit{ModifyOnly}(C_1) \cup \mathit{ModifyOnly}(C_2) \\
\mathit{ModifyOnly}(\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) & = & \mathit{ModifyOnly}(C_1) \cup \mathit{ModifyOnly}(C_2) \\
\mathit{ModifyOnly}(\mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}) & = & \mathit{ModifyOnly}(C)
\end{array}$$

**Table 5.4** Structural Rules in **BI** Pointer Logic be fresh variables.

$$\dfrac{\dfrac{\{(E \mapsto x, y) * ((E \mapsto E', y) \mathrel{-\!\!*} P)\}\, E.1 := E' \,\{(E \mapsto E', y) * ((E \mapsto E', y) \mathrel{-\!\!*} P)\}}{\{(E \mapsto x, y) * ((E \mapsto E', y) \mathrel{-\!\!*} P)\}\, E.1 := E' \,\{P\}}}{\{\exists x, y.\, (E \mapsto x, y) * ((E \mapsto E', y) \mathrel{-\!\!*} P)\}\, E.1 := E' \,\{P\}}$$

We provide two rules for $\mathsf{cons}$. The forward rule calculates the strongest postcondition for a given precondition; for a precondition $P$, the postcondition simply says that there exists a new cell $x$ with content $(E_1, E_2)$ where $x$ in $E_i$'s denotes the old value of $x$. The backward rule gives the weakest precondition for a postcondition $P$; whenever a new cell with content $(E_1, E_2)$ is added to the current heap and its address is stored in $x$, the resulting heap satisfies $P$. The rule for component lookup is similar to Simple Assignment except the machinery to take the value of the first field of the heap cell $E$. So, the second conjunct in the precondition ensures that the value of the first field of the heap cell $E$ is substituted for $x$ in $P$. And the rule for $\mathsf{dispose}$ is straightforward; it says that for a postcondition $P$, the weakest precondition is that there is a heap cell pointed to by $E$, which is not "mentioned" in $P$ at all.

The proof rules in Table 5.3 are not in the simplist form; they are chosen because they are useful in practice [68]. For instance, the forward rule for component assignment and the rule for $\mathsf{dispose}$ can be derived from the following simpler ones using the Frame Rule:

$$\{(E \mapsto E_1, E_2)\}\, E.1 := E' \,\{(E \mapsto E', E_2)\}$$

$$\{\exists x_1, x_2.\, (E \mapsto x_1, x_2)\}\, \mathsf{dispose}(E) \,\{\mathsf{emp}\} \quad (x_1, x_2 \text{ are different and not free in } E)$$

However, in most cases of program verification, the simpler rules need to be changed into the ones in this chapter because a property of interest usually involves more than a single heap cell [68]. The issue of the simplist proof rules is considered in Chapter 6, where we present streamlined proof rules, called *small axioms*, which fully exploit the structural rules in Table 5.4.

The Frame Rule (in short, FR) in Table 5.4 provides a way to derive a new Hoare triple ($\{P * R\}\, C \,\{Q * R\}$) from a given one ($\{P\}\, C \,\{Q\}$) by incorporating an assertion ($R$) as an invariant; the side condition is that the command in the triple ($C$) doesn't modify any heap cells and variables which might affect the truth of the assertion ($R$). With FR, we can derive a fact for the bigger portion of the store from what is already known for the smaller portion. In this way, FR formalizes a crucial step in local reasoning.

The soundness of FR is subtle. It relies on correctly formulating a condition which guarantees that $C$ doesn't modify any portion of the store which might affect the truth of the assertion $R$; then, $R$ is preserved because of local behavior of the command $C$ in Section 5.1.1. The difficult part is to make such a condition for heap storage since for variables, it is enough just to ensure that any variables in $R$ are not assigned to in $C$. FR uses the tight interpretation of Hoare triples, the spatial conjunction $*$ and the local behavior of a command to impose a proper condition for heap storage. Recall that the interpretation of Hoare triple $\{P\}\,C\,\{Q\}$ in this section has a requirement for safety: for any state satisfying the precondition $P$, the command should not go wrong, in particular, doesn't dereference any dangling pointers. Therefore, when $\{P\}\,C\,\{Q\}$ holds and a state $(s,h)$ satisfies $P * R$, only the portion $h_1$ of the heap $h$ for the assertion $P$ ($(s,h_1) \models P$) is going to be accessed; otherwise, $C$ dereferences a dangling pointer when started in $(s,h_1)$, so $\{P\}\,C\,\{Q\}$ can't be true.

**Lemma 58** *The Frame Rule is sound.*

*Proof:* We consider partial correctness first. Suppose that the premise of the rule holds, $h_1 \# h_2$, $(s,h_1) \models P$ and $(s,h_2) \models R$. Since $\{P\}\,C\,\{Q\}$ holds, $(C,s,h_1)$ is safe. The safety of $(s,h_1 * h_2)$ follows from this by Safety Monotonicity. For all $(s',h')$ such that $(C,s,h_1 * h_2) \leadsto^* (\mathsf{skip},s',h')$, there is $h_1'$ such that $(C,s,h_1) \leadsto^* (\mathsf{skip},s',h_1')$ and $h_1' * h_2 = h$ by Heap Locality because $(C,s,h_1)$ is safe. So, $(s',h_1') \models Q$. In order to see $(s',h_2) \in R$, notice that $s(y) = s'(y)$ for all free variables in $R$ and $(s,h_2) \models R$.

For total correctness, suppose that $\{P\}\,C\,\{Q\}$ holds, $(s,h_1) \models P$ and $(s,h_2) \models R$. Note that $(C,s,h_1)$ is safe because $(C,s,h_1)$ must terminate normally. Therefore, the above argument for partial correctness equally applies in this case. Then, Termination Monotonicity gives the conclusion. $\square$

# CHAPTER 6

# Completeness

This chapter concerns "completeness" of the Frame Rule. FR allows one to derive a Hoare triple from a given one without ever looking at the actual implementation of a program. So, it is desirable that the rule is strong enough to give all Hoare triples which "follow" from a given one without referring to the implementation. Because, then, one has to look at the code of a program only when it is absolutely necessary. The completeness theorem in this chapter shows that FR is strong enough in that sense. More accurately, it says the following:

> Let $P, Q, P', Q'$ be assertions and $X, Y$ sets of variables with $Y \supseteq X$. Suppose that for all commands $C$ which only modify variables in $X$ and only access variables in $Y$, when $\{P\}\,C\,\{Q\}$ holds, so does $\{P'\}\,C\,\{Q'\}$. Then, for the same kind of commands $C$, one can derive $\{P'\}\,C\,\{Q'\}$ from $\{P\}\,C\,\{Q\}$ only using FR and the other three structural rules in Table 5.4.

Such a question about completeness is subtle. Suppose that all commands do not necessarily exhibit local behavior for storage in Chapter 5; consequently, the rule of Consequence becomes the only appropriate structural rule. Then, the obvious hypothesis, which is that the rule of Consequence is complete, fails. Consider the case that $P' \equiv \mathsf{F}$ and $Q \not\models Q'$. For a command $C$, Consequence is not enough to derive $\{P'\}\,C\,\{Q'\}$ from $\{P\}\,C\,\{Q\}$ because $Q \not\models Q'$. On the other hand, $\{P'\}\,C\,\{Q'\}$ always holds regardless of which one of partial or total correctness we take to interpret Hoare triples.

The question becomes more difficult when local behavior of commands in Chapter 5 is considered and each Hoare triple is interpreted tightly. Suppose we are given a specification which says that a command $C$ satisfies a Hoare triple $\{P\}\,C\,\{Q\}$ and moreover, it only modifies variables in $X$ and only accesses variables in $Y$. When a command $C$ satisfies such a specification, all variables only appearing in assertions $P$ and $Q$ (that is, $FV(P \wedge Q) - Y$), which are called *auxiliary variables*, are neither read nor altered by the command $C$. And if a variable is not in $X$, it is not modified by $C$. Moreover, since a command $C$ shows local behavior for heap storage and the Hoare triple $\{P\}\, C\, \{Q\}$ holds in the tight sense in Chapter 5, command $C$ can only access heap cells "mentioned" in $P$. Such properties, which are shared by all commands satisfying the specification, additionally make certain other Hoare triples hold for the commands.

When dealing with stack storage only, so only with stack locality, a complete set of structural rules have been studied under the name of "adaptation completeness" [28] or "sharpness" [42]. Our choice of structural rules in Chapter 5 for auxiliary variables, that is, Auxiliary Variable Renaming (in short, AVR) and Auxiliary Variable Elimination (in short, AVE), are inspired by work on sharpness in [42]. However, their approach ignores local behavior of commands for heap storage, which is the main concern of FR. The completeness result in this chapter incorporates all local behavior of commands in Chapter 5, including that for heap storage.

The main idea to establish the completeness result is to use the domain of *local* predicate transformers to interpret the command type. Such predicate transformers exhibit local behavior of commands in Chapter 5. So, all structural rules become valid because they are based on such local behavior. Moreover, given a specification, the domain always contains the worst local predicate transformer satisfying the specification, and weakest preconditions of the predicate transformer can be calculated by structural rules in **BI** pointer logic; the completeness theorem is a corollary of such a feature of the worst local predicate transformer.

This chapter starts with the definition of local predicate transformers and their properties. Then, we give a predicate transformer semantics of **WHILE**h and connect it to the operational semantics. Finally, we show that all structural rules are sound and complete in the domain of local predicate transformers.

## 6.1 Local Predicate Transformers

In this section, we study certain predicate transformers which exhibit local behavior of commands in Chapter 5.

Recall the definitions of semantic domains in Chapter 5: *Var*, *Loc* and *Atom* are disjoint countably infinite sets such that any of them doesn't contain *nil* and *Atom* includes the set of integers *Int* and the set of boolean values $\{true, false\}$; and *Val*, *Stack*, *Heap* and *State* are defined as follows:

$$\begin{array}{rcl} Val & \triangleq & Atom \cup Loc \cup \{nil\} \\ Stack & \triangleq & Var \rightarrow Val \\ Heap & \triangleq & Loc \rightharpoonup_{fin} Val \times Val \\ State & \triangleq & Stack \times Heap \end{array}$$

In order to handle auxiliary variables properly, we need a semantic explanation about free variables in assertions and commands. For such a purpose, we parameterize *Stack* and *State* by a set $Y$ of variables:

$$\begin{array}{rcl} Stack(Y) & \triangleq & Y \rightarrow Val \\ State(Y) & \triangleq & Stack(Y) \times Heap \end{array}$$

Intuitively, $Stack(Y)$ and $State(Y)$ denote stacks and states when all allocated variables in stack storage are precisely those in $Y$. With these, the domains for assertions and commands with free variables in $Y$ can be obtained:

$$\begin{array}{rcl} Pred(Y) & \triangleq & \mathcal{P}(State(Y)) \\ PT(Y) & \triangleq & Pred(Y) \rightarrow_{monotone} Pred(Y) \end{array}$$

When a variable doesn't appear in an assertion, it doesn't matter to the truth of the assertion which value the variable contains; so, an assertion with all its free variables in $Y$ can be regarded as a subset of $State(Y)$. We call such a subset *predicate*. Similarly, variables not occurring in a command neither affect nor are affected by the execution of the command. So, for each command with free variables in $Y$, we regard it as a command for $State(Y)$ instead of one for *State*. We view a command as a predicate transformer following [16]: a command $C$ maps an assertion $Q$ to the weakest assertion $P$ such that $\{P\}\,C\,\{Q\}$ holds.[1] So, a command with free variables in $Y$ is an element of the domain $PT(Y)$, which maps a "postcondition" predicate in $Pred(Y)$ to the weakest "precondition" predicate in $Pred(Y)$ and is monotone with respect to the subset ordering of $Pred(Y)$. We call elements in $PT(Y)$ *predicate transformers*.

The domain of predicate transformers, $PT(Y)$, is not yet a proper interpretation of commands with free variables in $Y$. The reason is that $PT(Y)$ contains certain predicate trans-

[1]Here we don't specify whether we take a partial correctness or total correctness interpretation because such a choice affects not an interpretation of the command type but an interpretation of commands.

formers which, when viewed operationally, don't show local behavior for heap storage. As a result, a semantic version of FR is not valid in $PT(Y)$. In order to obtain a proper domain, we refine $PT(Y)$ by imposing a locality condition. Let $X$ be a subset of $Y$. We first define semantic counterparts of $*$, $-\!\!*$, $\mathsf{emp}$, $\forall X$ and $\exists X$. When $p$ and $q$ are predicates in $\mathit{Pred}(Y)$,

$$
\begin{aligned}
p * q &\triangleq \{(s, h * h') \mid (s,h) \in p \wedge (s,h') \in q \wedge h \# h'\} \\
p -\!\!* q &\triangleq \{(s,h) \mid \forall h'.\, h' \# h \wedge (s,h') \in p \Longrightarrow (s, h * h') \in q\} \\
\mathsf{emp} &\triangleq \{(s, \emptyset)\} \\
\forall X.p &\triangleq \{(s,h) \mid \forall s' \in \mathit{Stack}(X).\, (s[s'], h) \in p\} \\
\exists X.p &\triangleq \{(s,h) \mid \exists s' \in \mathit{Stack}(X).\, (s[s'], h) \in p\}
\end{aligned}
$$

where $s[s']$ intuitively denotes the update of $s$ by $s'$:

$$
s[s'](y) \triangleq \begin{cases} s'(y) & \text{if } y \in \mathit{dom}(s') \\ s(y) & \text{otherwise} \end{cases}
$$

Notice that as in syntax, $(\exists X.p) = p$ and $(\forall X.p) = p$ are equivalent, and they intuitively express that any variable in $X$ doesn't "appear" in $p$, i.e., doesn't affect the truth of $p$. The locality condition can now be defined:

$$
\text{Locality for } X\text{: } \forall r, q \in \mathit{Pred}(Y).\, (r = \forall X.r) \Longrightarrow t(q) * r \subseteq t(q * r)
$$

One might feel that the locality condition is contrived because the condition is equivalent to say that $t$ satisfies a semantic version of FR, which is:

$$
\text{FR: } \forall p, r, q \in \mathit{Pred}(Y).\, (r = \forall X.r) \wedge p \subseteq t(q) \Longrightarrow p * r \subseteq t(q * r)
$$

When we describe the correspondence between predicate transformer semantics and operational semantics in Section 6.2, we provide an operational intuition which shows that the locality for $X$ is "almost" equivalent to locality conditions for heap storage and Write Locality for stack storage in Chapter 5. The domain $\mathit{LocalPT}(X,Y)$ of *local predicate transformer with free variables in $Y$ and modified variables in $X$* is defined as:

$$
\mathit{LocalPT}(X,Y) \triangleq \{t \in PT(Y) \mid t \text{ satisfies locality for } X\}
$$

The domain $\mathit{LocalPT}(X,Y)$ inherits the ordering from $PT(Y)$, which is just a pointwise ordering induced by the subset ordering of $\mathit{Pred}(Y)$. With such an ordering, $\mathit{LocalPT}(X,Y)$ forms a complete lattice, in fact, a complete sublattice of $PT(Y)$.

**Lemma 59** *$LocalPT(X, Y)$ is a complete sublattice of $PT(Y)$. Therefore, given a set $\{t_i\}_{i\in I}$ of elements in $LocalPT(X, Y)$, its least upper bound is given by $\lambda q.\ \bigcup_{i\in I} t_i(q)$, and its greatest lower bound is given by $\lambda q.\ \bigcap_{i\in I} t_i(q)$.*

*Proof:* Let $r$ be a predicate such that $r = (\forall X.r)$. It suffices to show that $\bigcup_{i\in I} t_i(q * r)$ and $\bigcap_{i\in I} t_i(q*r)$ include $(\bigcup_{i\in I} t_i(q)) * r$ and $(\bigcap_{i\in I} t_i(q)) * r$, respectively. The first inclusion follows because $\bigcup_{i\in I}(t_i(q) * r) = (\bigcup_{i\in I} t_i(q_i)) * r$, and the second inclusion holds since $*$ is monotone with respect to the subset ordering. □

When writing assertions or commands, we often compose assertions or commands whose sets of free variables are different. Similar situations arise when dealing with semantic counterparts. For instance, we often need to compose two predicates in $Pred(Y)$ and $Pred(Z)$, respectively, even when $Y \neq Z$. We use an expansion operator for predicates and predicate transformers from [44, 4, 33], which changes a meaning in a smaller set of variables to that in a bigger set. Let $Y, Z$ be sets of variables with $Y \subseteq Z$. And let $p$ be a predicate in $Pred(Y)$ and $t$ a predicate transformer in $PT(Y)$. The expansions of $p$ and $t$ to $Pred(Z)$ and $PT(Z)$, respectively, are defined as follows:

$$\begin{aligned}
\mathsf{expand}(Y,Z)(p) &\triangleq \{(s * s', h') \mid (s', h') \in p \wedge s \in Stack(Z - Y)\} \\
\mathsf{expand}(Y,Z)(t)(q) &\triangleq \textstyle\bigcup_{s:Stack(Z-Y)} \{(s * s', h') \mid (s', h') \in t(q_s)\}
\end{aligned}$$

where $s_1 * s_2$ is a combining operator for states with disjoint domains (otherwise undefined):

$$(s_1 * s_2)(x) \triangleq \begin{cases} s_1(x) & \text{if } x \in dom(s_1) \\ s_2(x) & \text{if } x \in dom(s_2) \end{cases}$$

and for $s \in Stack(Z - Y)$, operator $-_s$ projects a predicate in $Pred(Z)$ to $Pred(Y)$:

$$r_s \triangleq \{(s', h) \mid s' \in Stack(Y) \wedge (s * s', h) \in r\}.$$

Note that for all $s \in Stack(Z - Y)$, the operator $-_s$ is a left inverse of the expansion of predicates from $Pred(Y)$ to $Pred(Z)$. That is, $\mathsf{expand}(Y, Z)(p)_s = p$.

Expansions behave nicely with respect to the existing structures of predicates and predicate transformers. Let $X, Y, Z$ be sets of variables such that $X \subseteq Y \subseteq Z$. The expansion $\mathsf{expand}(Y, Z)$ of predicate transformers preserves the complete lattice structure of $PT(Y)$. Similarly, the expansion for predicates preserves the complete lattice structure of $Pred(Y)$. Moreover, it also preserves semantic counterparts of $*$, $-\!\!*$, $\mathsf{emp}$, $\forall X.-$ and $\exists X.-$.

**Lemma 60** *Let $Y, Z$ be sets of variables such that $Y \subseteq Z$. Then, for both predicates and predicate transformers,* $\mathsf{expand}(Y, Z)$ *preserves least upper bounds and greatest lower bounds. Moreover, the expansion* $\mathsf{expand}(Y, Z)$ *for predicates also preserves* $*$, $-\!\!*$, $\mathsf{emp}$, $\forall X.-$ *and* $\exists X.-$.

The following two lemmas state that applying a predicate transformer to a predicate commutes with expansions and that expansions are injective and compose.

**Lemma 61** *For sets $Y, Z$ of variables with $Y \subseteq Z$, and $t \in PT(Y)$ and $q \in Pred(Y)$,*

$$\mathsf{expand}(Y, Z)(t(q)) = \mathsf{expand}(Y, Z)(t)(\mathsf{expand}(Y, Z)(q))$$

*Proof:*

$$\begin{array}{ll}
 & \mathsf{expand}(Y, Z)(t)(\mathsf{expand}(Y, Z)(q)) \\
= & \\
 & \bigcup_{s \in Stack(Z-Y)} \{(s * s', h') \mid (s', h') \in t((\mathsf{expand}(Y, Z)(q))_s)\} \\
= & \qquad \because (\mathsf{expand}(Y, Z)(q))_s = q \\
 & \bigcup_{s \in Stack(Z-Y)} \{(s * s', h') \mid (s', h') \in t(q)\} \\
= & \\
 & \mathsf{expand}(Y, Z)(t(q))
\end{array}$$

$\square$

**Lemma 62** *Let $X, Y, Z$ be sets of variables such that $X \subseteq Y \subseteq Z$. Then, for both predicates and predicate transformers,*

1. $\mathsf{expand}(X, Y)$ *is injective; and*
2. $\mathsf{expand}(Y, Z) \circ \mathsf{expand}(X, Y) = \mathsf{expand}(X, Z)$.

*Proof:* We first deal with the predicate case: to see why $\mathsf{expand}$ is injective, note that the expansion $\mathsf{expand}(X, Y)$ for predicates has a left inverse, which is $-_s$ for a state $s \in Stack(Y - X)$. And the second claim is straightforward.

For predicate transformers, $\mathsf{expand}(X, Y)$ is injective because of Lemma 61 and the injectivity of $\mathsf{expand}(X, Y)$ for predicates. And the following calculation shows the second claim for predicate transformers: let $Y_0$ be $Y - X$ and let $Z_0$ be $Z - Y$. Then, for all predicate transformers $t \in PT(X)$ and all predicates $q$ in $Pred(Z)$,

$$
\begin{aligned}
& \mathsf{expand}(Y,Z)(\mathsf{expand}(X,Y)(t))(q) \\
= \\
& \textstyle\bigcup_{s0 \in Stack(Z_0)} \{(s_0 * s', h') \mid (s', h') \in \mathsf{expand}(X,Y)(t)(q_{s_0})\} \\
= \\
& \textstyle\bigcup_{s0 \in Stack(Z_0)} \{(s_0 * s', h') \mid \\
& \qquad\qquad (s', h') \in \textstyle\bigcup_{s1 \in Stack(Y_0)} \{(s_1 * s'', h'') \mid (s'', h'') \in t((q_{s_0})_{s_1})\}\} \\
= \\
& \textstyle\bigcup_{s0 \in Stack(Z_0), s1 \in Stack(Y_0)} \{(s_0 * s_1 * s', h') \mid (s', h') \in t((q_{s_0})_{s_1})\} \\
= & \qquad \because (q_{s0})_{s_1} = q_{s0 * s_1} \\
& \textstyle\bigcup_{s0 \in Stack(Z_0), s1 \in Stack(Y_0)} \{(s_0 * s_1 * s', h') \mid (s', h') \in t(q_{s0 * s_1})\} \\
= \\
& \textstyle\bigcup_{s \in Stack(Z_0 \cup Y_0)} \{(s * s', h') \mid (s', h') \in t(q_s)\} \\
= \\
& \mathsf{expand}(X,Z)(t)(q)
\end{aligned}
$$

$\square$

And the expansion of predicate transformers preserves and reflects locality for $X$.

**Lemma 63** *For sets $X, Y, Z$ of variables with $X \subseteq Y \subseteq Z$ and $t \in PT(Y)$, predicate transformer $t$ is in $LocalPT(X, Y)$ iff $\mathsf{expand}(Y, Z)(t)$ is in $LocalPT(X, Z)$.*

*Proof:* For the if direction, let $p, r$ be predicates in $Pred(Y)$ such that $r = \forall X.r$. Since $\mathsf{expand}(Y, Z)$ for predicates has a left inverse $-_s$ for a stack $s \in Stack(Z - Y)$ and such $-_s$ preserves the subset ordering $\subseteq$ on predicates, it suffices to show that

$$\mathsf{expand}(Y, Z)(t(p) * r) \subseteq \mathsf{expand}(Y, Z)(t(p * r)).$$

The left hand side becomes $\mathsf{expand}(Y, Z)(t)(\mathsf{expand}(Y, Z)(p)) * \mathsf{expand}(Y, Z)(r)$ because the expansion preserves $*$ (Lemma 60) and it commutes with the application of a predicate transformer to a predicate (Lemma 61). Similarly, $\mathsf{expand}(Y, Z)(t)(\mathsf{expand}(Y, Z)(p) * \mathsf{expand}(Y, Z)(r))$ is equal to the right hand side. And the conclusion follows since the expansion doesn't add any free variables: $r = \forall X.r$ implies $\mathsf{expand}(Y, Z)(r) = \forall X.\mathsf{expand}(Y, Z)(r)$.

For the only-if direction, let $r, q$ be predicates in $\mathit{Pred}(Z)$ such that $r = \forall X.r$. We have to show that

$$\mathsf{expand}(Y,Z)(t)(q * r) \supseteq \mathsf{expand}(Y,Z)(t)(q) * r$$

It holds because:

$$\begin{array}{ll}
& \bigcup_{s:\mathit{Stack}(Z-Y)}\{(s * s', h') \mid (s', h') \in t((q * r)_s)\} \\
= & \qquad \because (q * r)_s = q_s * r_s \\
& \bigcup_{s:\mathit{Stack}(Z-Y)}\{(s * s', h') \mid (s', h') \in t(q_s * r_s)\} \\
\supseteq & \qquad \because (r = \forall X.r) \wedge X \subseteq Y \wedge s \in \mathit{Stack}(Z-Y) \text{ implies } (r_s = \forall X.r_s) \\
& \bigcup_{s:\mathit{Stack}(Z-Y)}\{(s * s', h') \mid (s', h') \in t(q_s) * r_s\} \\
= & \\
& \bigcup_{s:\mathit{Stack}(Z-Y)}\Big(\{(s * s', h') \mid (s', h') \in t(q_s)\} * r\Big) \\
= & \\
& \Big(\bigcup_{s:\mathit{Stack}(Z-Y)}\{(s * s', h') \mid (s', h') \in t(q_s)\}\Big) * r
\end{array}$$

□

The definitions of $\mathit{Pred}(-)$, $PT(-)$ and $\mathit{LocalPT}(-,-)$ with expand can be considered as functors from the category, whose objects are pairs $(X, Y)$ of subsets of $\mathit{Var}$ with $X \subseteq Y$ and morphisms are inclusions of both smaller and bigger sets, to the category of complete lattices with structure-preserving functions as morphisms. For instance, $\mathit{Pred}(-)$ maps an object $(X, Y)$ to $\mathit{Pred}(Y)$ and a morphism $(X, Y) \subseteq (W, Z)$ to $\mathsf{expand}(Y, Z)$. Lemma 60 and 62 say that $PT(-)$ and $\mathit{Pred}(-)$ are well-defined injective functors. And Lemma 63 indicates that $\mathit{LocalPT}(-,-)$ is a subfunctor of $PT(-)$. Finally, Lemma 61 states that the pointwise application is a natural transformation from $PT(-) \times \mathit{Pred}(-)$ to $\mathit{Pred}(-)$.

## 6.2 Predicate Transformer Semantics

The interpretation of assertions and commands can be given with what has been developed so far. Essentially, the domain for assertions is obtained just by taking all predicates with a finite set of free variables, and that for commands is obtained similarly. The constraint of finiteness reflects that each command or assertion contains only finitely many free variables.

$$\begin{array}{rcl} [\![\mathsf{assert}]\!] & \triangleq & \bigcup_{X \subseteq_{fin} Var} \{\mathsf{expand}(X, Var)(p) \mid p \in Pred(X)\} \\ [\![\mathsf{com}]\!] & \triangleq & \bigcup_{Y \subseteq_{fin} Var} \{\mathsf{expand}(Y, Var)(t) \mid t \in LocalPT(Y,Y)\} \end{array}$$

Notice the second definition is the same as

$$\bigcup_{X,Y \subseteq_{fin} Var, X \subseteq Y} \{\mathsf{expand}(Y, Var)(t) \mid t \in LocalPT(X,Y)\}$$

because $LocalPT(X,Y) \subseteq LocalPT(Y,Y)$ for all subsets $X$ of $Y$. For both domains, when an element $m$ is from $Y$, we often denote it by $free(m,Y)$. For instance, for a local predicate transformer $t \in [\![\mathsf{com}]\!]$ and a set $Y$ of variables, we write $free(t,Y)$ to denote that $\mathsf{expand}(Y, Var)(t_0) = t$ for some $t_0 \in LocalPT(Y,Y)$.

Even though we define a local predicate transformer in $[\![\mathsf{com}]\!]$ to "contain" only finitely many variables, it is not immediate that such a transformer maps a predicate "with" finitely many variables to another "with" finitely many variables. The following lemma shows that it becomes the case:

**Lemma 64** *For each $t \in [\![\mathsf{com}]\!]$, local predicate transformer $t$ restricts to a map from $[\![\mathsf{assert}]\!]$ to $[\![\mathsf{assert}]\!]$. Moreover, the restriction is injective: if $t$ and $t'$ restrict to the same map from $[\![\mathsf{assert}]\!]$ to $[\![\mathsf{assert}]\!]$, they are, in fact, the same.*

*Proof:* Let $q$ be a predicate in $[\![\mathsf{assert}]\!]$ and let $t$ be a local predicate transformer in $[\![\mathsf{com}]\!]$. Then, there are finite subsets $X, Y$ of $Var$ and $q_0 \in Pred(X), t_0 \in LocalPT(Y,Y)$ such that $q = \mathsf{expand}(X, Var)(q_0)$ and $t = \mathsf{expand}(Y, Var)(t_0)$. By Lemma 61,

$$\begin{array}{l} t = \mathsf{expand}(X \cup Y, Var)(\mathsf{expand}(Y, X \cup Y)(t_0)); \text{ and} \\ q = \mathsf{expand}(X \cup Y, Var)(\mathsf{expand}(X, X \cup Y)(q_0)). \end{array}$$

Notice that because of Lemma 62, for all $t' \in PT(X \cup Y)$ and $q' \in Pred(X \cup Y)$,

$$\Big(\mathsf{expand}(X \cup Y, Var)(t')\Big)\Big(\mathsf{expand}(X \cup Y, Var)(q')\Big) = \mathsf{expand}(X \cup Y, Var)(t'(q')).$$

Therefore, we have $free(t(q), X \cup Y)$.

For showing that the restriction is injective, suppose $t$ and $t'$ are different local predicate transformers in $[\![\mathsf{com}]\!]$. Let $X, Y$ be finite subsets of $Var$ such that $t = \mathsf{expand}(X, Var)(t_0)$ and $t' = \mathsf{expand}(Y, Var)(t'_0)$ for some $t_0$ and $t'_0$. Let $t_1 = \mathsf{expand}(X, X \cup Y)(t_0)$ and $t'_1 =$

$\mathsf{expand}(Y, X \cup Y)(t'_0)$. Since $\mathsf{expand}$ composes as shown in Lemma 62, $t_1$ and $t'_1$ are different. That is, there is $q \in \mathit{Pred}(X \cup Y)$ such that $t_1(q)$ and $t'_1(q)$ are not the same. Then, $t(\mathsf{expand}(X \cup Y, \mathit{Var})(q)) = \mathsf{expand}(X \cup Y, \mathit{Var})(t_1(q))$ and $t'(\mathsf{expand}(X \cup Y, \mathit{Var})(q)) = \mathsf{expand}(X \cup Y, \mathit{Var})(t'_1(q))$ are different, too, because $\mathsf{expand}$ is injective (Lemma 62). $\square$

We regard each local predicate transformer in $[\![\mathsf{com}]\!]$ as a map from $[\![\mathsf{assert}]\!]$ to $[\![\mathsf{assert}]\!]$.

Interpreting assertions is straightforward. The interpretation $[\![P]\!]$ of an assertion $P$ is just given by $\{(s,h) \mid (s,h) \models P\}$ where $\models$ is a relation defined in Chapter 5. Table 6.1 presents the semantics algebraically with semantic operators $*$, $-\!\!*$, $\mathsf{emp}$ and $\forall x.$ for $[\![\mathsf{assert}]\!]$. Assuming that $[\![E]\!]$ is an appropriate interpretation of an expression $E$, the semantics of commands is given in Table 6.2. When it is necessary to distinguish whether the semantics is for partial correctness or total correctness, we use subscripts: $[\![-]\!]_{wp}$ is for total correctness and $[\![-]\!]_{wlp}$ for partial correctness. Each command is interpreted following the idea of "weakest precondition." The only tricky part is to interpret loops appropriately both for partial correctness and for total correctness. We use ideas from [5, 17]. When the semantics is for total correctness, loops are interpreted using least fixed points in $[\![\mathsf{com}]\!]$. For partial correctness, we take greatest fixed points in $[\![\mathsf{com}]\!]$ to interpret loops. One might wonder whether such fixed points exist because $[\![\mathsf{com}]\!]$ might not be a complete lattice.[2] In order to see the existence of such fixed points, consider a finite set $Y$ of variables and a monotone function $f$ of type $[\![\mathsf{com}]\!] \to [\![\mathsf{com}]\!]$ which "has free variables only in $Y$":

$$\forall Y' \supseteq Y. \forall t \in [\![\mathsf{com}]\!]. \mathit{free}(t, Y') \Longrightarrow \mathit{free}(f(t), Y')$$

We note that the subset $T = \{t \in [\![\mathsf{com}]\!] \mid \mathit{free}(t, Y)\}$ of $\mathsf{com}$ is a complete lattice because it is the image of the complete lattice $\mathit{LocalPT}(Y, Y)$ by an order embedding $\mathsf{expand}(Y, \mathit{Var})$. Moreover, since $T$ is closed under $f$, $T$ has both the least and the greatest fixed points of $f$. These fixed points, in fact, become least and greatest ones in $[\![\mathsf{com}]\!]$; the reason is that for any other

[2] $[\![\mathsf{com}]\!]$ is not a complete sublattice of $\mathit{LocalPT}(\mathit{Var}, \mathit{Var})$. Let $x_0, x_1, \ldots$ be an enumeration of variables. Then, the following set $\{t_n\}$ becomes a counter example:

$$t_n = \lambda q. \{(s,h) \mid (s[x_n \mapsto 0], h) \in q\}$$

Note that each $t_n$ is in $[\![\mathsf{com}]\!]$, but the least upper bound $u$ and the greatest lower bound $l$ in $\mathit{LocalPT}(\mathit{Var}, \mathit{Var})$ are not in $[\![\mathsf{com}]\!]$:

$$\begin{array}{rcl} u & = & \lambda q. \{(s,h) \mid \exists n. (s[x_n \mapsto 0], h) \in q\} \\ l & = & \lambda q. \{(s,h) \mid \forall n. (s[x_n \mapsto 0], h) \in q\} \end{array}$$

$$
\begin{array}{rcl}
[\![(E \mapsto E_1, E_2)]\!] & = & \{(s,h) \mid dom(h) = \{[\![E]\!]s\} \wedge h([\![E]\!](s)) = ([\![E_1]\!](s), [\![E_2]\!](s))\} \\
[\![E = E']\!] & = & \{(s,h) \mid [\![E]\!](s) = [\![E']\!](s)\} \\
[\![\mathsf{F}]\!] & = & \emptyset \\
[\![P \Rightarrow Q]\!] & = & (State(Var) - [\![P]\!]) \cup [\![Q]\!] \\
[\![\forall x.\, P]\!] & = & \forall x.\, [\![P]\!] \\
[\![\mathsf{emp}]\!] & = & \mathsf{emp} \\
[\![P * Q]\!] & = & [\![P]\!] * [\![Q]\!] \\
[\![P -\!\!* Q]\!] & = & [\![P]\!] -\!\!* [\![Q]\!]
\end{array}
$$

**Table 6.1** Semantics of Assertions

fixed point $t_0$ and a set $Z$ with $\mathit{free}(t_0, Z)$ and $Z \supseteq Y$, the set $T' = \{t \in [\![\mathsf{com}]\!] \mid \mathit{free}(t, Z)\}$ is closed under $f$ and has $T$ as a complete sublattice. Therefore, the two fixed points still become least and greatest even in $T'$, so they are smaller and bigger than $t_0$, respectively. The well-definedness of the semantics follows from the above argument since for any command $C$ with free variables in $Y$, all monotonic functions involved in the definitions of $[\![C]\!]_{wp}$ and $[\![C]\!]_{wlp}$ "have free variables only in $Y$."

The interpretation of loops in the predicate transformer semantics needs more explanations because it is not immediate that least fixed points give weakest preconditions for total correctness and greatest fixed points for partial correctness. In the remainder of the section, we provide such an explanation by connecting the semantics with the operational semantics in Chapter 5. We also explain locality condition for predicate transformers and the formula *free* operationally.

The operational semantics in Chapter 5 can be seen as a semantics based on relations. That is, each command is interpreted as a relation $R$ from *State* to $\mathit{State} \cup \{\mathit{fault}, \perp\}$, where *fault* represents a memory fault, and $\perp$ a divergence. For a command $C$, we write $[C]$ to denote a relation induced by the operational semantics for $C$. Following terminologies in Chapter 5, we say that $R, s, h$ is safe when $s, h$ is not related to *fault* by $R$ and that $R, s, h$ must terminate normally when $R, s, h$ is safe and $s, h$ is not related to $\perp$ by $R$. The notions of total correctness for a predicate $q \in [\![\mathsf{assert}]\!]$,

$$
\begin{array}{rcl}
[\![x := E]\!]q & = & \{(s,h) \mid (s[x \mapsto [\![E]\!]s], h) \in q\} \\
[\![x := E.1]\!]q & = & \{(s,h) \mid [\![E]\!]s \in dom(h) \\
 & & \qquad\qquad\quad \wedge\, (s[x \mapsto \mathsf{fst}(h([\![E]\!]s))], h) \in q\} \\
[\![x := E.2]\!]q & = & \{(s,h) \mid [\![E]\!]s \in dom(h) \\
 & & \qquad\qquad\quad \wedge\, (s[x \mapsto \mathsf{snd}(h([\![E]\!]s))], h) \in q\} \\
[\![E.1 := E']\!]q & = & \{(s,h) \mid [\![E]\!]s \in dom(h) \wedge h([\![E]\!]s) = (n,m) \\
 & & \qquad\qquad\quad \wedge\, (s, h[[\![E]\!]s \mapsto ([\![E']\!]s, m)]) \in q\} \\
[\![E.2 := E']\!]q & = & \{(s,h) \mid [\![E]\!]s \in dom(h) \wedge h([\![E]\!]s) = (n,m) \\
 & & \qquad\qquad\quad \wedge\, (s, h[[\![E]\!]s \mapsto (n, [\![E']\!]s)]) \in q\} \\
[\![x := \mathsf{cons}(E,E')]\!]q & = & \{(s,h) \mid \forall l \in Loc - dom(h). \\
 & & \qquad\qquad\quad (s[x \mapsto l], h * (l \mapsto ([\![E]\!]s, [\![E']\!]s))) \in q\} \\
[\![\mathsf{dispose}(E)]\!]q & = & \{(s, h * (l \mapsto v_1, v_2)) \mid v_1, v_2 \in Val \wedge (s,h) \in q \\
 & & \qquad\qquad\qquad\qquad \wedge\, [\![E]\!]s = l \in Loc - dom(h)\} \\
[\![\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]q & = & \{(s,h) \mid \text{if } [\![E]\!]s = true \text{ then } (s,h) \in [\![C_1]\!]q \\
 & & \qquad\qquad\qquad\qquad \text{else } (s,h) \in [\![C_2]\!]q\} \\
[\![\mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}]\!]q & = & \Big(\mathsf{fix}\ \lambda t.\lambda q'.\, \{(s,h) \mid \text{if } [\![E]\!]s = true \\
 & & \qquad\qquad\qquad\qquad \text{then } (s,h) \in [\![C]\!](t(q')) \\
 & & \qquad\qquad\qquad\qquad \text{else } (s,h) \in q'\}\Big)q
\end{array}
$$

where fix is the least fixed point in $[\![\mathsf{com}]\!]$ when the semantics is for total correctness, and it is the greatest fixed point when the semantics is for partial correctness.

**Table 6.2** Semantics of Commands

and partial correctness give rise to two maps from relations to predicate transformers:

$$
\begin{array}{rcl}
wp(R)(q) & \triangleq & \{(s,h) \mid (R,s,h) \text{ must terminate normally} \\
& & \qquad\qquad \wedge \forall (s',h').\, (s,h)\, R\, (s',h') \Longrightarrow (s',h') \in q\} \\
wlp(R)(q) & \triangleq & \{(s,h) \mid (R,s,h) \text{ is safe} \\
& & \qquad\qquad \wedge \forall (s',h').\, (s,h)\, R\, (s',h') \Longrightarrow (s',h') \in q\}
\end{array}
$$

The semantics in this section is intended to give the weakest precondition for partial correctness or for total correctness. Now that a command has another interpretation as a relation, such consistency of the semantics can be formally stated:

**Lemma 65** *For all commands $C$ and predicates $q$ in $[\![\mathsf{assert}]\!]$, predicates $[\![C]\!]_{wp}q$ and $[\![C]\!]_{wlp}q$ are the weakest preconditions for $q$ and $C$ for total correctness and for partial correctness, respectively: that is, $wp([C])(q) = [\![C]\!]_{wp}q$ and $wlp([C])(q) = [\![C]\!]_{wlp}q$.*

*Proof:* The proof is given by induction on the structure of $C$. We only deal with the case for loops, which is the most interesting case. All the important ideas of handling the loop case are from [17], where a similar result is shown. Let $C$ be $\mathsf{while}\ E\ \mathsf{do}\ C'\ \mathsf{od}$. The starting point is to note that the functional for which we take the least or the greatest fixed point to interpret $C$ can be rewritten with $[C']$.

$$
\begin{array}{rcl}
[\![C]\!]_{wp} & = & \mathsf{lfix}\ \lambda t.\lambda q.\, \{(s,h) \mid \text{if } [\![E]\!]s = \mathit{true} \\
& & \qquad\qquad\qquad \text{then } ([C'],s,h) \text{ must terminate normally and} \\
& & \qquad\qquad\qquad\quad \forall (s',h').\, (s,h)\, [C']\, (s',h') \Longrightarrow (s',h') \in t(q) \\
& & \qquad\qquad\qquad \text{else } (s,h) \in q\} \\
[\![C]\!]_{wlp} & = & \mathsf{gfix}\ \lambda t.\lambda q.\, \{(s,h) \mid \text{if } [\![E]\!]s = \mathit{true} \\
& & \qquad\qquad\qquad \text{then } ([C'],s,h) \text{ is safe and} \\
& & \qquad\qquad\qquad\quad \forall (s',h').\, (s,h)\, [C']\, (s',h') \Longrightarrow (s',h') \in t(q) \\
& & \qquad\qquad\qquad \text{else } (s,h) \in q\}
\end{array}
$$

where $\mathsf{lfix}$ and $\mathsf{gfix}$ denote least and greatest fixed points, respectively. Let $F_{wp}$ and $F_{wlp}$ be functionals in the above for which we take fixed points for $[\![C]\!]_{wp}$ and $[\![C]\!]_{wlp}$, respectively. Then, the easy calculation shows that $wp([C])$ and $wlp([C])$ are fixed points of $F_{wp}$ and $F_{wlp}$, respectively. So, we focus on showing that $wp([C])$ is the *least* fixed point and $wlp([C])$ the *greatest* fixed point.

We deal with the easy case first: $wlp([C])$ is the greatest fixed point. Let $t$ be a fixed point of $F_{wlp}$, and let $q$ be a predicate in $[\![\mathsf{assert}]\!]$ and $(s,h)$ a state in $t(q)$. Then, $([C],s,h)$ is safe. Otherwise, there is a sequence of states

$$(s_0,h_0)\,[C']\,(s_1,h_1)\,[C']\,(s_2,h_2)\,[C']\ldots[C']\,(s_n,h_n)\,[C']\,\mathit{fault}$$

where $[\![E]\!](s_i) = \mathit{true}$ for $i = 0\ldots n$ and $(s_0,h_0) = (s,h)$. Then, $(s_n,h_n) \in F_{wlp}(t)(q)$, so $([C'],s_n,h_n)$ is supposed to be safe, which contradicts to the assumption. We use similar reasoning to show that $(C,s,h)$ doesn't lead to a state which violates $q$. For any $(s',h')$ with $(s,h)\,[C]\,(s',h')$, there is a sequence of states

$$(s_0,h_0)\,[C']\,(s_1,h_1)\,[C']\,(s_2,h_2)\,[C']\ldots[C']\,(s_n,h_n)\,[C']\,(s',h')$$

where $[\![E]\!](s_i) = \mathit{true}$ for $i = 0\ldots n$, $(s_0,h_0) = (s,h)$ and $[\![E]\!](s') \neq \mathit{true}$. Since $(s',h') \in F_{wlp}(t)(q)$, we must have $(s',h') \in q$.

Showing that $wp([C])$ is the least fixed point is a little more tricky. Let $t$ be a fixed point of $F_{wp}$ and $q$ a predicate in $[\![\mathsf{assert}]\!]$. We show that if a state $(s,h)$ is not in $t(q)$ but for all $s',h'$ with $(s,h)\,[C]\,(s'h')$, the state $(s',h')$ belongs to $q$, then $(C,s,h)$ must not terminate normally. Define a predicate $g$ by:

$$(s_0,h_0) \in g \;\triangleq\; (s_0,h_0) \notin t(q) \wedge \Big(\forall s',h'.\,(s_0,h_0)\,[C]\,(s'h') \Longrightarrow (s',h') \in q\Big)$$

Notice that it suffices to show that

$$\begin{aligned}(s_0,h_0) \in g \;\Longrightarrow\;& (s_0,h_0)\,[C']\perp \vee (s_0,h_0)\,[C']\,\mathit{fault}\\ & \vee [\![E]\!](s_0) = \mathit{true} \wedge \Big(\exists (s',h').\,(s,h)\,[C']\,(s',h') \wedge (s',h') \in g\Big)\end{aligned}$$

which holds because

$$\begin{aligned}
& (s_0,h_0) \in g \wedge \neg((s_0,h_0)\,[C']\perp) \wedge \neg((s_0,h_0)\,[C']\,\mathit{fault})\\
\Longrightarrow\quad & \because (s_0,h_0) \in g \text{ implies } [\![E]\!](s_0) = \mathit{true}\\
& [\![E]\!](s_0) = \mathit{true} \wedge \Big(\exists (s',h').\,(s,h)\,[C']\,(s',h') \wedge (s',h') \notin t(q)\Big)\\
& \qquad \wedge \Big(\forall s',h'.\,(s_0,h_0)\,[C]\,(s'h') \Longrightarrow (s',h') \in q\Big)\\
\Longrightarrow\quad & \\
& [\![E]\!](s_0) = \mathit{true} \wedge \exists s',h'.\;\; (s,h)\,[C']\,(s',h') \wedge (s',h') \notin t(q)\\
& \qquad \wedge \Big(\forall s'',h''.\,(s',h')\,[C]\,(s'',h'') \Longrightarrow (s'',h'') \in q\Big)\\
\Longrightarrow\quad & \\
& [\![E]\!](s_0) = \mathit{true} \wedge \Big(\exists s',h'.\,(s,h)\,[C']\,(s',h') \wedge (s',h') \in g\Big)
\end{aligned}$$

$\square$

The locality condition for predicate transformers can be explained with two maps *wp* and *wlp*. The following lemma shows that the locality condition captures local behavior of commands:

**Lemma 66** *For a relation $R$: State $\leftrightarrow$ State $\cup\{\bot, fault\}$, both $wp(R)$ and $wlp(R)$ satisfy locality for $X$ iff the following four conditions hold for $R$:*

1. *Write Locality: if $R, s, h$ is safe and $(s, h)\, R\, (s', h')$, then $s(y) = s'(y)$ for all $y \in Var - X$;*
2. *Safety Monotonicity: if $R, s, h$ is safe and $h\#h'$, then $R, s, h * h'$ is safe, too;*
3. *Termination Monotonicity: if $R, s, h$ must terminate normally and $h\#h'$, then $R, s, h * h'$ must terminate normally; and*
4. *Heap Locality: if $R, s, h_1$ is safe, $h_1\#h_2$ and $(s, h_1 * h_2)\, R\, (s', h')$, then there exists $h_1'$ such that $(s, h_1)\, R\, (s', h_1')$, $h_1'\#h_2$ and $h' = h_1' * h_2$.*

Notice that Safety Monotonicity, Termination Monotonicity and Heap Locality in the lemma are exactly those in Chapter 5; Write Locality is the only exception. However, it is as much correspondence as we can get since both *wp* and *wlp* completely ignore all unsafe configurations, so we can not state any properties of unsafe configurations with predicate transformers.

*Proof:* For if direction, let $p, q$ be predicates in $[\![\mathsf{assert}]\!]$ such that $p = (\forall X.p)$. Pick $(s, h) \in wp(R)(q) * p$. Then, there exist $h_1, h_2$ such that $h_1\#h_2$, $h = h_1 * h_2$, $(s, h_1) \in wp(R)(q)$ and $(s, h_2) \in p$. By Termination Monotonicity, $(s, h_1 * h_2)$ must terminate normally. For all $(s, h_1 * h_2)\, R\, (s', h')$, Write Locality says that $s' = s[s_X]$ for some $s_X \in Stack(X)$, and Heap locality implies that $h' = h_1' * h_2$ for some $h_1'$ with $(s, h_1)\, R\, (s', h_1')$; therefore, $(s', h') \in q * p$ because $(s[s_X], h_2) \in p$ and $(s', h_1') \in q$. The case for *wlp* can be handled similarly; the only difference is to use Safety Monotonicity instead of Termination Monotonicity.

For the other direction, we handle each item separately:

1. *Write Locality*: suppose that $R, s, h$ is safe and $(s, h)\, R\, (s', h')$. Let $p = \{(s[s_X], \emptyset) \mid s_X \in Stack(X)\}$ and let $q$ be $\{(s'', h'') \mid (s, h)\, R\, (s'', h'')\}$. Since $p = \forall X.p$, we have that $wlp(R)(q * p) \supseteq wlp(R)(q) * p$. Note that $(s, h) \in wlp(R)(q) * p$. Therefore, it also belongs to $wlp(R)(q * p)$, so $(s', h') \in q * p$. The conclusion follows since $(s', \emptyset) \in p$.

2. *Safety Monotonicity*: suppose that $R, s, h$ is safe and $h\#h'$. Let $p = \{(s[s_X], h') \mid s_X \in \mathit{Stack}(X)\}$ and let $q$ be $\{(s'', h'') \mid (s, h)\, R\, (s'', h'')\}$. Since $p = \forall X.p$, we have that $\mathit{wlp}(R)(q * p) \supseteq \mathit{wlp}(R)(q) * p$. Note that $(s, h * h') \in \mathit{wlp}(R)(q) * p$. Therefore, $(s, h * h') \in \mathit{wlp}(R)(q * p)$, so $R, s, h * h'$ is safe.

3. *Termination Monotonicity*: the proof is essentially the same as the case of Safety Monotonicity. It can be obtained just replacing "is safe" by "must terminate normally" and $\mathit{wlp}$ by $\mathit{wp}$.

4. *Heap Locality*: suppose that $R, s, h_1$ is safe, $h_1\#h_2$ and $(s, h_1 * h_2)\, R\, (s', h')$. Let $p = \{(s[s_X], h_2) \mid s_X \in \mathit{Stack}(X)\}$ and let $q$ be $\{(s'', h'') \mid (s, h_1)\, R\, (s'', h'')\}$. Since $p = \forall X.p$, we have that $\mathit{wlp}(R)(q * p) \supseteq \mathit{wlp}(R)(q) * p$. Since $(s, h_1 * h_2) \in \mathit{wlp}(R)(q) * p$, the state $(s, h_1 * h_2)$ belongs to $\mathit{wlp}(R)(q * p)$. So, $(s', h') \in q * p$. Then, there are $h_1'$ and $s_X \in \mathit{Stack}(X)$ such that $(s, h_1)\, R\, (s[s_X], h_1')$ , $s' = s[s_X]$ and $h' = h_1' * h_2$.

□

We have a semantic explanation of which variables are "free" in a predicate and a predicate transformer. The following lemma indicates that the semantic notion of "free" variables is approximated by the syntactic counterpart.

**Lemma 67** *For a command $C$ and an assertion $P$, suppose that all free variables in $C$ and $P$ belong to $Y$ and $C$ writes to only those variables in $X$ ($\mathit{ModifyOnly}(C) \subseteq X$) for some sets $X, Y$ of variables with $X \subseteq Y$. Then, there exist $t_0, t_1 \in \mathit{LocalPT}(X, Y)$ and $p_0 \in \mathit{Pred}(Y)$ which can be expanded to $[\![C]\!]_{wp}$, $[\![C]\!]_{wlp}$ and $[\![P]\!]$, respectively.*

## 6.3 Soundness and Completeness

In this section, we show that all structural rules are sound and complete.

In order to state completeness, we need a proof system which formalizes what one can derive from a given Hoare triple in **BI** pointer logic without looking at the implementation of a program. We take structural rules in Chapter 5 and formulate such a proof system with them. The proof system is about a Hoare triple with a hole $\{P\} - \{Q\}$, which intuitively represents all commands $C$ satisfying $\{P\}\, C\, \{Q\}$. Following [38, 78], we call a Hoare triple with a hole

$$\textsc{Consequence}\quad \frac{P' \models P \quad \{P\} - \{Q\} \quad Q \models Q'}{\{P'\} - \{Q'\}}$$

$$\textsc{Auxiliary Variable Renaming}\quad \frac{\{P\} - \{Q\}}{\{P[y/x]\} - \{Q[y/x]\}}\ x \notin Y \wedge y \notin (Y \cup FV(P \wedge Q))$$

$$\textsc{Auxiliary Variable Elimination}\quad \frac{\{P\} - \{Q\}}{\{\exists x.\, P\} - \{\exists x.\, Q\}}\ x \notin Y$$

$$\textsc{Frame Rule}\quad \frac{\{P\} - \{Q\}}{\{P * R\} - \{Q * R\}}\ FV(R) \cap X = \emptyset$$

**Table 6.3** Proof System for Commands $C$ with $\mathit{ModifyOnly}(C) \subseteq X$ and with $FV(C) \subseteq Y$

*specification statement*. Let $X$ and $Y$ be finite sets of variables with $X \subseteq Y$. Table 6.3 shows a proof system for commands which only modifies variables in $X$ and only accesses variables in $Y$. Notice that the rules in the system are nothing but structural rules in Chapter 5; so, it faithfully models which Hoare triples are derivable from a given one in **BI** pointer logic without looking at the implementation of a program. When $\{P'\} - \{Q'\}$ is derivable from $\{P\} - \{Q\}$, we write $\{P\} - \{Q\} \vdash_{X,Y} \{P'\} - \{Q'\}$ to denote this. Informally, $\{P\} - \{Q\} \vdash_{X,Y} \{P'\} - \{Q'\}$ means that for every command $C$ which only modifies variables in $X$ and only accesses variables in $Y$, if $\{P\}\, C\, \{Q\}$ holds, so does $\{P'\}\, C\, \{Q'\}$.

The interpretation of each specification statement is given by a forcing relation between local predicate transformers in $[\![\mathsf{com}]\!]$, which are expanded from $\mathit{LocalPT}(X, Y)$, and specification statements. For a local predicate transformer $t$ expanded from $\mathit{LocalPT}(X, Y)$ and assertions $P$ and $Q$,

$$t \models_{X,Y} \{P\} - \{Q\} \iff [\![P]\!] \subseteq t([\![Q]\!])$$

We write $\{P\}-\{Q\} \models_{X,Y} \{P'\}-\{Q'\}$ when $t \models_{X,Y} \{P\}-\{Q\}$ implies $t \models_{X,Y} \{P'\}-\{Q'\}$ for all local predicate transformers $t$ expanded from $\mathit{LocalPT}(X, Y)$. The definition of $\{P\}-\{Q\} \models_{X,Y} \{P'\} - \{Q'\}$ is intended to indicate that $\{P'\} - \{Q'\}$ follows from $\{P\} - \{Q\}$ "without ever referring to the actual implementation of a program." To see a justification for such an intention, we note that if $\{P\} - \{Q\} \models_{X,Y} \{P'\} - \{Q'\}$, then $\{P\}\, C\, \{Q\}$ implies $\{P'\}\, C\, \{Q'\}$ for all commands $C$ with $\mathit{ModifyOnly}(C) \subseteq X$ and $FV(C) \subseteq Y$. The reason is that for all commands $C$ with $\mathit{ModifyOnly}(C) \subseteq X$ and $FV(C) \subseteq Y$, both $[\![C]\!]_{wp}$ and $[\![C]\!]_{wlp}$ are local predicate transformers expanded from $\mathit{LocalPT}(X, Y)$ (Lemma 67), and regardless of partial or total correctness, $\{P\}\, C\, \{Q\}$ holds if and only if $[\![C]\!] \models_{X,Y} \{P\} - \{Q\}$ (Lemma 65). The proof system is sound with respect to the interpretation.

**Proposition 68 (Soundness)** *If* $\{P\} - \{Q\} \vdash_{X,Y} \{P'\} - \{Q'\}$, *then* $\{P\} - \{Q\} \models_{X,Y} \{P'\} - \{Q'\}$.

*Proof:* We show that each rule in Table 6.3 is sound. It suffices to show the following four facts, each of which implies the soundness of the corresponding syntactic rule: suppose $p \subseteq t(q)$ for predicates $p, q$ in $[\![\mathsf{assert}]\!]$ and for a local predicate transformer $t$ in $[\![\mathsf{com}]\!]$ expanded from $\mathit{LocalPT}(X, Y)$.

1. *Consequence*: for all predicates $p', q' \in [\![\mathsf{assert}]\!]$, if $p' \subseteq p$ and $q \subseteq q'$, then $p' \subseteq t(q')$.

2. *Frame Rule*: for all $r \in [\![\mathsf{assert}]\!]$, if $\mathit{free}(r, \mathit{Var} - X)$, then $(p * r) \subseteq t(q * r)$.

3. *Auxiliary Variable Elimination*: for a set $W$ of variables such that $Y$ and $W$ are disjoint, $(\exists W.p) \subseteq t(\exists W.q)$ holds.

4. *Auxiliary Variable Renaming*: for all bijections $\eta$: $\mathit{Var} \to \mathit{Var}$ satisfying that $\eta(y) = y$ for all $y \in Y$, we have $p[\eta] \subseteq t(q[\eta])$, where $r[\eta]$ is the set $\{(s, h) \mid (s \circ \eta, h) \in r\}$.

Consequence follows from the fact that $t$ is monotone. We deal with the other three cases in the following:

1. *Frame Rule*: Since expansions preserve locality for $X$, local predicate transformer $t$ also has locality for $X$. Therefore, if we show $r = (\forall X.r)$, the locality for $X$ gives $t(q * r) \supseteq t(q) * r$, and the assumption on $t$ implies that $t(q) * r \supseteq p * r$. The conclusion follows since $\mathit{free}(r, \mathit{Var} - X)$ is equivalent to $r = (\forall X.r)$; both of them intuitively say that "$r$ doesn't contain any variables in $X$."

2. *Auxiliary Variable Elimination*: Let $t_0$ be a local predicate transformer in $\mathit{LocalPT}(X,Y)$ such that $t = \mathsf{expand}(Y, \mathit{Var})(t_0)$. We first note the following two facts: for all $s \in \mathit{Stack}(\mathit{Var} - Y)$,

   (a) $p_s \subseteq t_0(q_s)$; and

   (b) $t_0((\exists W.q)_s) \supseteq \bigcup_{s' \in \mathit{Stack}(W)} t_0(q_{s[s']})$.

   The first holds because

$$\begin{array}{rcl} (t(q))_s & = & \left(\bigcup_{s_0 \in \mathit{Stack}(\mathit{Var}-Y)} \{(s_0 * s', h') \mid (s', h') \in t_0(q_{s_0})\}\right)_s \\ & = & \{(s', h') \mid (s', h') \in t_0(q_s)\} \\ & = & t_0(q_s) \end{array}$$

   The following shows why the second holds:

$$\begin{array}{ll} & t_0((\exists W.q)_s) \\ = & \\ & t_0(\{(s'', h'') \mid \exists s' \in \mathit{Stack}(W).\, ((s * s'')[s'], h'') \in q\}) \\ = & \quad \because s \in \mathit{Stack}(\mathit{Var} - Y) \wedge W \subseteq \mathit{Var} - Y \\ & t_0(\{(s'', h'') \mid \exists s' \in \mathit{Stack}(W).\, ((s[s']) * s'', h'') \in q\}) \\ = & \\ & t_0(\bigcup_{s' \in \mathit{Stack}(W)} q_{s[s']}) \\ \supseteq & \quad \because t_0 \text{ is monotone} \\ & \bigcup_{s' \in \mathit{Stack}(W)} t_0(q_{s[s']}) \end{array}$$

With the above facts, the claim can be shown easily:

$$
\begin{array}{ll}
& t(\exists W.q) \\
= & \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s'', h'') \mid (s'', h'') \in t_0((\exists W.q)_s)\} \\
\supseteq & \quad \because \text{the second fact} \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s'', h'') \mid (s'', h'') \in \bigcup_{s' \in Stack(W)} t_0(q_{s[s']})\} \\
= & \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s'', h'') \mid \exists s' \in Stack(W).\, (s'', h'') \in t_0(q_{s[s']})\} \\
\supseteq & \quad \because \text{the first fact} \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s'', h'') \mid \exists s' \in Stack(W).\, (s'', h'') \in p_{s[s']}\} \\
= & \\
& \exists W.(\bigcup_{s \in Stack(Var-Y)} \{(s * s'', h'') \mid (s'', h'') \in p_s\}) \\
= & \\
& \exists W.p
\end{array}
$$

3. *Auxiliary Variable Renaming*: Let $t_0$ be a local predicate transformer in $LocalPT(X, Y)$ such that $t = \mathsf{expand}(Y, Var)(t_0)$. As shown in the previous case, for all $s \in Stack(Var - Y)$, we have that $p_s \subseteq t_0(q_s)$. The crucial fact for this case is that for all $s \in Stack(Var - Y)$ and all predicates $p'$ in $[\![\mathsf{assert}]\!]$,

$$(p'[\eta])_s \;=\; p'_{s \circ \eta}$$

because

$$
\begin{array}{ll}
& (p'[\eta])_s \\
= & \\
& \{(s', h') \mid (s * s', h') \in p'[\eta]\} \\
= & \\
& \{(s', h') \mid ((s * s') \circ \eta, h') \in p'\} \\
= & \quad \because s' \in Stack(Y) \text{ and } \eta(y) = y \text{ for all } y \in Y \\
& \{(s', h') \mid ((s \circ \eta) * s', h') \in p'\} \\
= & \\
& p'_{s \circ \eta}
\end{array}
$$

The following shows the claim:

$$
\begin{array}{cl}
& t(q[\eta]) \\
= & \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s', h') \mid (s', h') \in t_0(q[\eta]_s)\} \\
= & \quad \because q[\eta]_s = q_{s \circ \eta} \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s', h') \mid (s', h') \in t_0(q_{s \circ \eta})\} \\
\supseteq & \quad \because p \subseteq t(q) \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s', h') \mid (s', h') \in p_{s \circ \eta}\} \\
= & \quad \because p[\eta]_s = p_{s \circ \eta} \\
& \bigcup_{s \in Stack(Var-Y)} \{(s * s', h') \mid (s', h') \in p[\eta]_s\} \\
= & \\
& p[\eta]
\end{array}
$$

□

Intuitively, the completeness theorem indicates that given a Hoare triple, the structural rules in **BI** pointer logic are sufficient to derive all Hoare triples which can be derivable with all conceivable sound structural rules.

**Theorem 69 (Completeness)** *If* $\{P\} - \{Q\} \models_{X,Y} \{P'\} - \{Q'\}$, *then* $\{P\} - \{Q\} \vdash_{X,Y} \{P'\} - \{Q'\}$.

We prove the completeness theorem by finding the worst local predicate transformer $t$ in $[\![\mathsf{com}]\!]$ such that $t$ is expanded from $\mathit{LocalPT}(X, Y)$, and it satisfies a given specification ($t \models \{P\} - \{Q\}$). First, we show such a predicate transformer exists. Then, we prove that the weakest precondition of the predicate transformer can be calculated by the rules in Table 6.3.

**Lemma 70** *Given predicates* $p, q \in [\![\mathsf{assert}]\!]$, *the below* $t \in [\![\mathsf{com}]\!]$ *is expanded from* $\mathit{LocalPT}(X, Y)$ *and satisfies* $p \subseteq t(q)$. *Moreover, it is the worst such with respect to the pointwise ordering in* $[\![\mathsf{com}]\!]$*:*

$$
\lambda r. \bigcup_{s:Stack(Var-Y)} \{(s * s'', h'') \mid (s'', h'') \in \bigcup_{s' \in Stack(Var-Y)} p_{s'} * \forall X.(q_{s'} \mathrel{-\!\!*} r_s)\}
$$

*Proof:* We first show that the predicate transformer $t$ in the lemma is, in fact, a local predicate transformer satisfying $p \subseteq t(q)$ and expanded from $LocalPT(X,Y)$. Then, we prove that $t$ is the worst such. Let $t_0$ be a predicate transformer in $PT(Y)$ defined by:

$$t_0(r) = \bigcup_{s' \in Stack(Var-Y)} p_{s'} * \forall X.(q_{s'} \mathrel{-\!\!*} r)$$

Then, it is easy to see that $t$ is the expansion of $t_0$: $t = \mathsf{expand}(Y, Var)(t_0)$. So, in order to show that $t$ is expanded from $LocalPT(X,Y)$, it is sufficient to prove that $t_0$ is in $LocalPT(X,Y)$. Let $r, r'$ be predicates in $Pred(Y)$ such that $r = \forall X.r$. The following calculation shows that $t_0 \in LocalPT(X,Y)$:

$$\begin{array}{ll}
 & t_0(r' * r) \\
= & \\
 & \bigcup_{s' \in Stack(Var-Y)} p_{s'} * \forall X.(q_{s'} \mathrel{-\!\!*} (r' * r)) \\
\supseteq & \\
 & \bigcup_{s' \in Stack(Var-Y)} p_{s'} * \forall X.((q_{s'} \mathrel{-\!\!*} r') * r) \\
\supseteq & \quad \because r = \forall X.r \\
 & \bigcup_{s' \in Stack(Var-Y)} p_{s'} * (\forall X.(q_{s'} \mathrel{-\!\!*} r')) * r \\
= & \\
 & \left(\bigcup_{s' \in Stack(Var-Y)} p_{s'} * (\forall X.(q_{s'} \mathrel{-\!\!*} r'))\right) * r \\
= & \\
 & t_0(r') * r
\end{array}$$

For $p \subseteq t(q)$, we note that it is equivalent to show that $p_s \subseteq (t(q))_s$ for all $s \in Stack(Var - Y)$. As shown in the proof for the AVE case in Lemma 68, $(t(q))_s = t_0(q_s)$. The following calculation fills the only missing gap: $p_s \subseteq t_0(q_s)$.

$$
\begin{array}{ll}
 & t_0(q_s) \\
= & \\
 & \bigcup_{s' \in Stack(Var-Y)} p_{s'} * \forall X.(q_{s'} \mathrel{-\!\!*} q_s) \\
\supseteq & \\
 & p_s * \forall X.(q_s \mathrel{-\!\!*} q_s) \\
\supseteq & \\
 & p_s * \{(s'', \emptyset) \mid s'' \in Stack(Y)\} \\
= & \\
 & p_s
\end{array}
$$

In order to show that $t$ is worst, we use the fact that $\mathsf{expand}(Y, Var)$ is monotone. Let $t'$ be a local predicate transformer in $[\![\mathsf{com}]\!]$ such that $p \subseteq t'(q)$ and $t'$ is expanded from $t'_0$ in $LocalPT(X, Y)$. Since $\mathsf{expand}$ is monotone, it suffices to show that $t'_0$ is better than $t_0$. Let $r$ be a predicate in $Pred(Y)$. We note that $p_s \subseteq t'_0(q_s)$ for all $s \in Stack(Var - Y)$ since $p_s \subseteq t'(q)_s$ and $t'(q)_s = t'_0(q_s)$; for the latter, look at the proof of the AVE case in Lemma 68. Since $(\forall X.\forall X.r') = (\forall X.r')$ for all predicates $r'$ in $Pred(Y)$, by the locality for $X$ of $t'_0$, we have that

$$p_s * \forall X.(q_s \mathrel{-\!\!*} r) \;\subseteq\; t'_0(q_s * \forall X.(q_s \mathrel{-\!\!*} r))$$

Since $q_s * \forall X.(q_s \mathrel{-\!\!*} r) \subseteq r$, by the monotonicity of $t'_0$,

$$p_s * \forall X.(q_s \mathrel{-\!\!*} r) \;\subseteq\; t'_0(r)$$

Therefore,

$$\bigcup_{s' \in Stack(Var-Y)} p_{s'} * \forall X.(q_{s'} \mathrel{-\!\!*} r) \subseteq t'_0(r)$$

□

We denote the worst local predicate transformer by $worst(p, q, X, Y)$. The following lemma says that the weakest precondition of such a predicate transformer can be calculated with FR, AVE, AVR and Consequence.

**Lemma 71** *Given assertions P,Q,R and a finite set A of variables such that Y and A are mutually disjoint, $FV(P) \subseteq (Y \cup A)$ and $FV(Q) \subseteq (Y \cup A)$, let $\eta$ be a bijection from Var to Var such that*

1. *$\{\eta(a) \mid a \in A\}$ is disjoint from all of $Y$, $A$ and $FV(R)$; and*

2. *$\eta$ only renames variables in $A$ and $\{\eta(a) \mid a \in A\}$.*

*Then, when $B = \{\eta(a) \mid a \in A\}$,*

$$worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y)(\llbracket R \rrbracket) = \llbracket \exists B. P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R) \rrbracket$$

*where $P[\eta]$ denotes an application of simultaneous renaming $\eta$ to $P$ and $\forall X$ and $\exists B$ sequences of universal and existential quantifications by all variables in $X$ and $B$, respectively.*

When $\{P\} - \{Q\} \models_{X,Y} \{S\} - \{R\}$ holds, $\llbracket S \rrbracket$ is a subset of $worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y)(\llbracket R \rrbracket)$ because $worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y)(\llbracket R \rrbracket) \models_{X,Y} \{S\} - \{R\}$. Therefore, the completeness theorem follows from the above lemma and Consequence since using AVR, FR, Consequence and AVE, the equivalent characterization of $worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y)(\llbracket R \rrbracket)$ in the lemma can be obtained:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\{P\} - \{Q\}}{\{P[\eta]\} - \{Q[\eta]\}}}{\{P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R)\} \ - \ \{Q[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R)\}}}{\{P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R)\} \ - \ \{R\}}}{\{\exists B.\, P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R)\} \ - \ \{\exists B.\, R\}}}{\{\exists B.\, P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R)\} \ - \ \{R\}}$$

*Proof:* The above derivation shows $\{P\} - \{Q\} \vdash_{X,Y} \{\exists B.\, P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R)\} - \{R\}$. Since the proof system is sound and $worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y) \models_{X,Y} \{P\} - \{Q\}$, we have

$$\llbracket \exists B.\, P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R) \rrbracket \quad \subseteq \quad worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y)(\llbracket R \rrbracket)$$

So, it suffices to show the other inclusion. We write $r'[\eta]$ to denote $\{(s,h) \mid (s \circ \eta, h) \in r'\}$ for a predicate $r'$ in $\llbracket \mathsf{assert} \rrbracket$; then, $\llbracket Q[\eta] \rrbracket = \llbracket Q \rrbracket[\eta]$ and $\llbracket P[\eta] \rrbracket = \llbracket P \rrbracket[\eta]$. The inclusion which we are supposed to show can be rewritten as follows:

$$\exists B. \llbracket P \rrbracket[\eta] * \forall X.(\llbracket Q \rrbracket[\eta] \mathrel{-\!\!*} \llbracket R \rrbracket)$$
$$\supseteq$$
$$\bigcup_{s:Stack(Var-Y)} \{(s * s'', h'') \mid (s'', h'') \in \bigcup_{s' \in Stack(Var-Y)} \llbracket P \rrbracket_{s'} * \forall X.(\llbracket Q \rrbracket_{s'} \mathrel{-\!\!*} \llbracket R \rrbracket_s)\}$$

where we unrolled the definition of $worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y)(\llbracket R \rrbracket)$. Pick a state $(s_0, h_0)$ from the set $worst(\llbracket P \rrbracket, \llbracket Q \rrbracket, X, Y)(\llbracket R \rrbracket)$. Then, there exist stacks, $s_{Var-Y}$, $s'_{Var-Y}$, $s_Y$, and heaps, $h_1$, $h_2$, such that

1. $s_{Var-Y}, s'_{Var-Y} \in Stack(Var - Y), s_Y \in Stack(Y)$;

2. $s_{Var-Y} * s_Y = s_0$ and $h_1 * h_2 = h_0$;

3. $(s'_{Var-Y} * s_Y, h_1) \in [\![P]\!]$; and

4. $(s_Y, h_2) \in \forall X. [\![Q]\!]_{s'_{Var-Y}} \mathrel{-\!\!*} [\![R]\!]_{s_{Var-Y}}$.

Let $s'_A$ be the restriction of $s'_{Var-Y}$ to $Stack(A)$ and let $s'_B \in Stack(B)$ be $s'_A \circ \eta^{-1}$. We show that $(s_{Var-Y}[s'_B] * s_Y, h_1) \in [\![P]\!][\eta]$ and $(s_{Var-Y}[s'_B] * s_Y, h_2) \in \forall X.([\![Q]\!][\eta] \mathrel{-\!\!*} [\![R]\!])$, thereby proving that $(s_0, h_0)$ belongs to $[\![\exists B. P[\eta] * \forall X.(Q[\eta] \mathrel{-\!\!*} R)]\!]$. The first formula $(s_{Var-Y}[s'_B] * s_Y, h_1) \in [\![P]\!][\eta]$ is shown as follows:

$$
\begin{array}{ll}
 & (s'_{Var-Y} * s_Y, h_1) \in [\![P]\!] \\
\Longrightarrow & \quad \because \mathit{free}([\![P]\!], Y \cup A) \\
 & ((s_{Var-Y} \circ \eta)[s'_A] * s_Y, h_1) \in [\![P]\!] \\
\Longrightarrow & \quad \because s'_A \circ \eta^{-1} = s'_B \\
 & ((s_{Var-Y} \circ \eta)[s'_B \circ \eta] * s_Y, h_1) \in [\![P]\!] \\
\Longrightarrow & \\
 & ((s_{Var-Y}[s'_B] \circ \eta) * s_Y, h_1) \in [\![P]\!] \\
\Longrightarrow & \quad \because \eta(y) = y \text{ for all } y \in Y \\
 & (s_{Var-Y}[s'_B] * s_Y, h_1) \in [\![P]\!][\eta]
\end{array}
$$

The following shows the other formula:

$$
\begin{aligned}
& (s_Y, h_2) \in \forall X.\, (\llbracket Q \rrbracket_{s'_{Var-Y}} \mathrel{-\!\!*} \llbracket R \rrbracket_{s_{Var-Y}}) \\
\Longrightarrow \quad & \\
& \forall u \in \mathit{Stack}(X).\, (s_Y[u], h_2) \in (\llbracket Q \rrbracket_{s'_{Var-Y}} \mathrel{-\!\!*} \llbracket R \rrbracket_{s_{Var-Y}}) \\
\Longrightarrow \quad & \\
& \forall u \in \mathit{Stack}(X).\, \forall h. \\
& \quad h \# h_2 \wedge (s'_{Var-Y} * s_Y[u], h) \in \llbracket Q \rrbracket \Longrightarrow (s_{Var-Y} * s_Y[u], h * h_2) \in \llbracket R \rrbracket \\
\Longrightarrow \quad & \quad \because \mathit{free}(\llbracket Q \rrbracket, Y \cup A) \text{ and } \mathit{free}(\llbracket R \rrbracket, \mathit{Var} - B) \\
& \forall u \in \mathit{Stack}(X).\, \forall h. \\
& \quad h \# h_2 \wedge ((s_{Var-Y} \circ \eta)[s'_A] * s_Y[u], h) \in \llbracket Q \rrbracket \Longrightarrow (s_{Var-Y}[s'_B] * s_Y[u], h * h_2) \in \llbracket R \rrbracket \\
\Longrightarrow \quad & \quad \because s'_A \circ \eta^{-1} = s'_B \\
& \forall u \in \mathit{Stack}(X).\, \forall h. \\
& \quad h \# h_2 \wedge ((s_{Var-Y}[s'_B] \circ \eta) * s_Y[u], h) \in \llbracket Q \rrbracket \Longrightarrow (s_{Var-Y}[s'_B] * s_Y[u], h * h_2) \in \llbracket R \rrbracket \\
\Longrightarrow \quad & \quad \because \eta(y) = y \text{ for all } y \in Y \\
& \forall u \in \mathit{Stack}(X).\, \forall h. \\
& \quad h \# h_2 \wedge (s_{Var-Y}[s'_B] * s_Y[u], h) \in \llbracket Q \rrbracket[\eta] \Longrightarrow (s_{Var-Y}[s'_B] * s_Y[u], h * h_2) \in \llbracket R \rrbracket \\
\Longrightarrow \quad & \\
& \forall u \in \mathit{Stack}(X).\, (s_{Var-Y}[s'_B] * s_Y[u], h_2) \in (\llbracket Q \rrbracket[\eta] \mathrel{-\!\!*} \llbracket R \rrbracket) \\
\Longrightarrow \quad & \\
& (s_{Var-Y}[s'_B] * s_Y, h_2) \in \forall X.(\llbracket Q \rrbracket[\eta] \mathrel{-\!\!*} \llbracket R \rrbracket)
\end{aligned}
$$

$\square$

One interesting consequence of the completeness result is that when a command $C$ is the worst one satisfying a Hoare triple $\{P\}\,C\,\{Q\}$, the Hoare triple completely determines the behavior of $C$: for all other valid Hoare triples $\{P'\}\,C\,\{Q'\}$, the completeness result says that $\{P'\}\,C\,\{Q'\}$ is derivable from $\{P\}\,C\,\{Q\}$ only with structural rules. In the remainder of the section, we use such a property of a Hoare triple to formulate an alternative set of axioms for **WHILE**h, which fully exploit the power of structural rules: each axiom only mentions variables and heap cells actually accessed by a command. The axioms are as follows: when $n_1, n_2, m$ are distinct fresh variables, i.e., they are not in $FV(E) \cup FV(E') \cup \{x\}$ and distinct,

THE SMALL AXIOMS

$$\{(E \mapsto n_1, n_2)\}\, E.1 := E'\, \{(E \mapsto E', n_2)\}$$

$$\{(E \mapsto n_1, n_2)\}\, \mathsf{dispose}(E)\, \{\mathsf{emp}\}$$

$$\{\mathsf{emp} \wedge x = m\}\, x := \mathsf{cons}(E, E')\, \{(x \mapsto E[m/x], E'[m/x])\}$$

$$\{\mathsf{emp} \wedge x = m\}\, x := E\, \{\mathsf{emp} \wedge x = E[m/x]\}$$

$$\{(E \mapsto n_1, n_2) \wedge x = m\}\, x := E.1\, \{(E[m/x] \mapsto n_1, n_2) \wedge x = n_1\}$$

**Lemma 72** *For every small axiom, the command in the Hoare triple is the worst one satisfying the triple.*

*Proof:* For an assertion $R$ and a small axiom $\{P\}\, C\, \{Q\}$, the (backward) axiom for $C$ in **BI** pointer logic gives the weakest precondition for $R$ and $C$. We show that the weakest precondition is equvalent to $\mathit{worst}([\![P]\!], [\![Q]\!], X, Y)([\![R]\!])$ with $X = \mathit{ModifyOnly}(C)$ and $Y = FV(C)$. We use Lemma 71 to have a better representation of $\mathit{worst}([\![P]\!], [\![Q]\!], X, Y)([\![R]\!])$. We also use the fact that all of $m, n_1, n_2$ can be renamed to fresh names by Auxiliary Variable Renaming; so, it can be assumed that none of $m, n_1, n_2$ occurs free in $R$. Since the case that $C$ is $E.1 := E'$ is immediate, we only handle the other cases:

- $C \equiv \mathsf{dispose}(E)$:

$$\begin{array}{ll} & \exists n_1, n_2.\, (E \mapsto n_1, n_2) * (\mathsf{emp} -\!\!* R) \\ \iff & \quad \because \mathsf{emp} -\!\!* P' \text{ iff } P' \\ & \exists n_1, n_2.\, (E \mapsto n_1, n_2) * R \\ \iff & \quad \because n_1, n_2 \notin FV(R) \\ & (\exists n_1, n_2.\, E \mapsto n_1, n_2) * R \end{array}$$

- $C \equiv x := \mathsf{cons}(E)$:

$$
\begin{array}{ll}
 & \exists m.\, (\mathsf{emp} \wedge x = m) * (\forall x.\, (x \mapsto E[m/x], E'[m/x]) \mathrel{-\!\!*} R) \\
\Longleftrightarrow & \qquad \because (x = m \wedge P') * Q' \text{ iff } x = m \wedge P' * Q' \\
 & \exists m.\, x = m \wedge \mathsf{emp} * (\forall x.\, (x \mapsto E[m/x], E'[m/x]) \mathrel{-\!\!*} R) \\
\Longleftrightarrow & \qquad \text{Pick a fresh variable } x' \\
 & \exists m.\, x = m \wedge (\forall x'.\, (x' \mapsto E[m/x], E'[m/x]) \mathrel{-\!\!*} R[x'/x]) \\
\Longleftrightarrow & \qquad \because m \notin FV(E) \cup FV(E') \cup FV(R[x'/x]) \\
 & \forall x'.\, (x' \mapsto E, E') \mathrel{-\!\!*} R[x'/x]
\end{array}
$$

- $C \equiv x := E$:

$$
\begin{array}{ll}
 & \exists m.\, x = m \wedge (\forall x.\, (x = E[m/x] \wedge \mathsf{emp}) \mathrel{-\!\!*} R) \\
\Longleftrightarrow & \qquad \because (x = E' \wedge P') \mathrel{-\!\!*} Q' \text{ iff } x = E' \Rightarrow (P' \mathrel{-\!\!*} Q') \\
 & \exists m.\, x = m \wedge (\forall x.\, x = E[m/x] \Rightarrow (\mathsf{emp} \mathrel{-\!\!*} R)) \\
\Longleftrightarrow & \qquad \because \mathsf{emp} \mathrel{-\!\!*} P' \text{ iff } P' \\
 & \exists m.\, x = m \wedge (\forall x.\, x = E[m/x] \Rightarrow R) \\
\Longleftrightarrow & \\
 & \exists m.\, x = m \wedge R[E[m/x]/x] \\
\Longleftrightarrow & \\
 & R[E[m/x]/x][x/m] \\
\Longleftrightarrow & \qquad \because m \notin FV(R) \\
 & R[E[m/x][x/m]/x] \\
\Longleftrightarrow & \qquad \because m \notin FV(E) \\
 & R[E/x]
\end{array}
$$

- $C \equiv x := E.1$:

$$
\begin{array}{ll}
& \exists n_1, n_2, m.\,((E \mapsto n_1, n_2) \wedge x = m) * (\forall x.\,((E[m/x] \mapsto n_1, n_2) \wedge x = n_1) {-\!\!*}\, R) \\
\iff & \quad \because (x = n_1 \wedge P') {-\!\!*}\, Q' \text{ iff } x = n_1 \Rightarrow (P' {-\!\!*}\, Q') \\
& \exists n_1, n_2, m.\,((E \mapsto n_1, n_2) \wedge x = m) * (\forall x.\, x = n_1 \Rightarrow (E[m/x] \mapsto n_1, n_2) {-\!\!*}\, R) \\
\iff & \\
& \exists n_1, n_2, m.\,((E \mapsto n_1, n_2) \wedge x = m) * ((E[m/x] \mapsto n_1, n_2) {-\!\!*}\, R[n_1/x]) \\
\iff & \quad \because (x = m \wedge P') * Q' \text{ iff } x = m \wedge (P' * Q') \\
& \exists n_1, n_2, m.\, x = m \wedge (E \mapsto n_1, n_2) * ((E[m/x] \mapsto n_1, n_2) {-\!\!*}\, R[n_1/x]) \\
\iff & \quad \because m \notin FV(E) \cup FV(R) \\
& \exists n_1, n_2.\,(E \mapsto n_1, n_2) * (E[m/x][x/m] \mapsto n_1, n_2) {-\!\!*}\, R[n_1/x]) \\
\iff & \quad \because (E \mapsto n_1, n_2) * ((E \mapsto n_1, n_2) {-\!\!*}\, P') \text{ iff } (E \mapsto n_1, n_2) * \top \wedge P' \\
& \exists n_1, n_2.\, R[n_1/x] \wedge (E \mapsto n_1, n_2) * \top \\
\iff & \quad \because n_2 \notin FV(R[n_1/x]) \\
& \exists n_1.\, R[n_1/x] \wedge \exists n_2.\,(E \mapsto n_1, n_2) * \top
\end{array}
$$

$\square$

### 6.3.1 Positive Conjunctivity

The completeness result in Section 6.3 has a shortcoming because not all local predicate transformers can be defined by programs. In particular, certain local predicate transformers do not even correspond to any relations: they are not in the image of either *wlp* or *wp*; consequently, the operational meaning of such predicate transformers can not be given easily. In this section, we consider local predicate transformers which correspond to relations, and formulate a sufficient condition to guarantee that structural rules remain complete for such predicate transformers.

A predicate transformer $t \in PT(\mathit{Var})$ is *positively conjunctive* if and only if for a nonempty family of predicates $\{p_i \in \mathit{Pred}(\mathit{Var})\}_{i \in I}$,

$$t(\bigcap_{i \in I} p_i) \;=\; \bigcap_{i \in I} t(p_i)$$

It is well-known that positively conjunctively predicate transformers correspond to relations [39, 44]. In the framework of this thesis, such a correspondence can be phrased as follows:

**Lemma 73** *The image of the map wlp from relations of type State* $\leftrightarrow$ *State* $\cup \{fault, \bot\}$ *to predicate transformers* ($PT(Var)$) *is precisely the set of positively conjunctive predicate transformers. The image of wp is also the same.*

*Proof:* Since it is straightforward to show that the images of *wlp* and *wp* are included in the set of positively conjunctive predicate transformers, we concentrate on the surjectivity of both maps: we show that for every positively conjunctive predicate transformer $t \in PT(Var)$, there is a relation $R$: *State* $\leftrightarrow$ *State* $\cup \{fault, \bot\}$ which is mapped to $t$ by *both wlp* and *wp*. Such a relation $R$ is defined as follows:

$$
\begin{array}{lll}
(s,h)\ R\ (s',h') & \text{iff} & (s',h') \in (\bigcap_{(s,h)\in t(r)} r) \text{ and } (s,h) \in t(State) \\
(s,h)\ R\ fault & \text{iff} & (s,h) \notin t(State)
\end{array}
$$

□

A predicate transformer is not necessarily positively conjunctive. However, given a predicate transformer $t \in PT(Var)$, it is always possible to find the positively conjunctive one $\overline{t} \in PT(Var)$ such that for all positively conjunctive predicate transformer $t' \in PT(Var)$,

$$t \sqsubseteq t' \quad \text{iff} \quad \overline{t} \sqsubseteq t'.$$

We call $\overline{t}$ *positively conjunctive ceiling* of $t$. The ceiling $\overline{t}$ is obtained as follows [39, 44]:

$$\overline{t}(q) = \{(s,h) \in t(State) \mid (\bigcap_{(s,h)\in t(r)} r) \subseteq q\}$$

We note that the ceiling operator preserves the locality condition for $X$.

**Lemma 74** *For a local predicate transformer* $t \in [\![\mathsf{com}]\!]$ *expanded from* $LocalPT(X,Y)$*, the ceiling* $\overline{t}$ *is also expanded from* $LocalPT(X,Y)$*.*

*Proof:* Let $t$ be a local predicate transformer in $[\![\mathsf{com}]\!]$ expanded from $t_0$ in $LocalPT(X,Y)$. It is easy to show that $\overline{t}$ is expanded from a predicate transformer $t'_0$ in $PT(Y)$. So, we'll only show that $\overline{t}$ is indeed from one in $LocalPT(X,Y)$. Since the locality condition for $X$ is preserved and reflected by expansions (Lemma 63), it suffices to show that for all $r, q \in Pred(Var)$ such that $(\forall X.r) = r$,

$$\overline{t}(q) * r \subseteq \overline{t}(q * r).$$

assuming that the same locality condition for $X$ holds for $t$. Pick $(s,h) \in \overline{t}(q) * r$. Then, there exist $h_1, h_2$ such that

1. $h = h_1 * h_2$;

2. $(s, h_1) \in \bar{t}(q)$, that is, $(s, h_1) \in t(\mathit{State})$ and $\bigcap_{(s,h_1)\in t(p)} p \subseteq q$; and

3. $(s, h_2) \in r$.

Note that $(s, h) \in t(\mathit{State})$ because $t(\mathit{State} * r) \subseteq t(\mathit{State})$ and $t(\mathit{State}) * r \subseteq t(\mathit{State} * r)$. So, the conclusion follows once we show the following two inclusions:

1. $\left(\bigcap_{(s,h)\in t(p)} p\right) \subseteq \left(\bigcap_{(s,h_1)\in t(p)} p * \exists X.\{(s, h_2)\}\right)$; and
2. $\left(\bigcap_{(s,h_1)\in t(p)} p * \exists X.\{(s, h_2)\}\right) \subseteq q * r$.

The first inclusion holds because for all predicates $p \in \mathit{Pred}(\mathit{Var})$ with $(s, h_1) \in t(p)$,

$$\begin{array}{rcll} (s, h_1 * h_2) & \in & t(p) * \exists X.\{(s, h_2)\} & \\ & \subseteq & t(p * \exists X.\{(s, h_2)\}) & \because (\forall X.\exists X.\{(s, h_2)\}) = \exists X.\{(s, h_2)\} \end{array}$$

To see the second inclusion, we notice that

$$\begin{array}{rl} & \left(\bigcap_{(s,h_1)\in t(p)} p * \exists X.\{(s, h_2)\}\right) \\ = & \quad \because (s', h') \in \exists X.\{(s, h_2)\} \text{ implies } h' = h_2 \\ & \left(\bigcap_{(s,h_1)\in t(p)} p\right) * \exists X.\{(s, h_2)\} \\ \subseteq & \quad \because (\forall X.r) = r \text{ and } (s, h_2) \in r \\ & q * r \end{array}$$

□

The key step of the completeness proof in Section 6.3 is to find the worst local predicate transformer to satisfy a specification statement. The following proposition precisely characterizes when such a worst predicate transformer happens to be positively conjunctive. The characterization also provides a sufficient condition to tell for which specification statements one can restrict attention to positively conjunctive local predicate transformers without worrying about incompleteness of structural rules. The reason is that given a specification statement $\{P\} - \{Q\}$ and finite sets $X, Y$ of stack variables with $X \subseteq Y$, the first below statement implies the second:

1. $\mathit{worst}([\![P]\!], [\![Q]\!], X, Y)$ is positively conjunctive;

2. $\{P\}-\{Q\} \vdash_{X,Y} \{P'\}-\{Q'\}$ if and only if $t \models_{X,Y} \{P\}-\{Q\}$ implies $t \models_{X,Y} \{P'\}-\{Q'\}$ for all positively conjunctive local predicate transformers $t$ (which we denote by $\{P\}-\{Q\} \models^{\mathrm{pc}}_{X,Y} \{P'\}-\{Q'\}$).

To see the implication, we note that derivability ($\vdash_{X,Y}$) and validity ($\models^{\mathrm{pc}}_{X,Y}$) are expressible with worst local predicate transformers:

1. $\{P\}-\{Q\} \vdash_{X,Y} \{P'\}-\{Q'\}$ iff $[\![P']\!] \subseteq \mathit{worst}([\![P]\!],[\![Q]\!],X,Y)([\![Q']\!])$; and
2. $\{P\}-\{Q\} \models^{\mathrm{pc}}_{X,Y} \{P'\}-\{Q'\}$ iff $[\![P']\!] \subseteq \overline{\mathit{worst}([\![P]\!],[\![Q]\!],X,Y)}([\![Q']\!])$.

**Proposition 75** *Given predicates $p,q$ in $[\![\mathsf{assert}]\!]$, finite sets of variables $X,Y$ with $X \subseteq Y$ and a set of variables $A$ with $A = \mathit{Var} - Y$, the local predicate transformer $\mathit{worst}(p,q,X,Y)$ is positively conjunctive iff the following condition is met: for all states $(s,h)$,*

> *if there are heaps $h_1,h_2$ and a stack $u_A \in \mathit{Stack}(A)$ such that $(s[u_A],h_1) \in p$ and $h = h_1 * h_2$, then there exist worst such $k_1,k_2$ and $v_A$ in the sense that for all other such $h_1,h_2,u_A$, we have*
>
> $$\Big(\{(s',h') \mid (s'[v_A],h') \in q\} * \exists X.\{(s,k_2)\}\Big) \subseteq \Big(\{(s',h') \mid (s'[u_A],h') \in q\} * \exists X.\{(s,h_2)\}\Big)$$

*Proof:* Let $t$ be the local predicate transformer $\mathit{worst}(p,q,X,Y)$. We will show that $\overline{t} = t$ is equivalent to the latter condition in the proposition. The important observations are:

1. a state $(s,h)$ belongs to $t(\mathit{State})$ iff there are heaps $h_1,h_2$ and a stack $u_A \in \mathit{Stack}(A)$ such that $(s[u_A],h_1) \in p$ and $h = h_1 * h_2$; and
2. for all states $(s,h) \in t(\mathit{State})$,

$$\bigcap_{(s,h)\in t(r)} r = \bigcap_{u_A \in \mathit{Stack}(A),\,(s[u_A],h_1)\in p,\,h=h_1*h_2} \{(s',h') \mid (s'[u_A],h') \in q\} * \exists X.\{(s,h_2)\}.$$

We prove each direction in the proposition separately.

- *if direction*: let $r$ be a predicate in $\mathit{Pred}(\mathit{Var})$. Since $t \sqsubseteq \overline{t}$, it suffices to show that $\overline{t}(r) \subseteq t(r)$. Consider a state $(s,h)$ in $\overline{t}(r)$. Since $(s,h)$ is in $t(\mathit{State})$, the first observation and the assumption imply that there are worst $k_1, k_2$ and $v_A \in \mathit{Stack}(A)$ such that $(s[v_A], k_1) \in p$ and $h = k_1 * k_2$. Then,

$$\{(s', h') \mid (s'[v_A], h') \in q\} * \exists X.\{(s, k_2)\}$$
$$=$$
$$\bigcap_{u_A \in \mathit{Stack}(A),\, (s[u_A], h_1) \in p,\, h = h_1 * h_2} \{(s', h') \mid (s'[u_A], h') \in q\} * \exists X.\{(s, h_2)\}$$

Moreover, $\bigcap_{(s,h) \in t(r')} r' \subseteq r$ from the definition of $(s, h) \in \overline{t}(r)$. Therefore, the second observation says that

$$\Big(\{(s', h') \mid (s'[v_A], h') \in q\} * \exists X.\{(s, k_2)\}\Big) \subseteq r$$

When $s_Y, s_A$ are the restrictions of $s$ to $Y$ and $A$ respectively, the above inclusion can equivalently expressed as follows:

$$(s_Y, k_2) \in \forall X.\, q_{v_A} \mathrel{-\!\!*} r_{s_A}.$$

Since $(s_Y, k_1) \in p_{v_A}$, state $(s, h)$ belongs to $t(r)$.

- *only-if direction*: let $(s, h)$ be a state such that $(s[u_A], h_1) \in p$ and $h = h_1 * h_2$ for a stack $u_A$ in $\mathit{Stack}(A)$ and heaps $h_1, h_2$. Then, the first observation tells us that $(s, h) \in t(\mathit{State})$. Since $\overline{t} = t$ and $\overline{t}$ is positively conjunctive,

  $$t(\bigcap_{(s,h) \in t(r)} r) = \bigcap_{(s,h) \in t(r)} t(r)$$

  So, $(s, h) \in t(\bigcap_{(s,h) \in t(r)} r)$. That is, there exists $v_A \in \mathit{Stack}(A)$ and $k_1, k_2$ such that

  1. $h = k_1 * k_2$;
  2. $(s_Y, k_1) \in p_{v_A}$; and
  3. $(s_Y, k_2) \in \forall X. q_{v_A} \mathrel{-\!\!*} (\bigcap_{(s,h) \in t(r)} r)_{s_A}$.

  where $s_Y$ and $s_A$ are the restrictions of $s$ to $Y$ and $A$, respectively. These $v_A, k_1, k_2$ are what we are looking for: the third condition for $v_A, k_1, k_2$ is equivalent to

  $$\Big(\{(s', h') \mid (s'[v_A], h') \in q\} * \exists X.\{(s, k_2)\}\Big) \subseteq \bigcap_{(s,h) \in t(r)} r,$$

  and by the second observation, the right hand side is equal to

  $$\bigcap_{u_A \in \mathit{Stack}(A), (s[u_A], h_1) \in p, h = h_1 * h_2} \{(s', h') \mid (s'[u_A], h') \in q\} * \exists X.\{(s, h_2)\}.$$

$\square$

# CHAPTER 7

## Example: Schorr-Waite Graph Marking Algorithm

In this chapter, we provide an extensive example which illustrates the benefit of local reasoning: using **BI** pointer logic, we prove that the Schorr-Waite graph marking algorithm satisfies a given specification. The algorithm is a well-known test case for programming logic for pointers because complex pointer manipulations in the algorithm impose a challenge to the effectiveness of such logic. The algorithm mixes two different data structures by implementing one data structure inside the other. As a result, it becomes very difficult to manage the complexity of formal verification. Most of known simplification techniques in other literatures do not work: techniques based on reachability are not applicable since nodes in two data structures might be reachable from each other; those based on different field names don't help much, either, because two mixed data structures can not be separated merely by field names. In the chapter, we show that even for such an algorithm, one can still exploit locality of memory access with the Frame Rule and, as a result, obtain simpler formal verification.

The main observation in this chapter is that an informal correctness argument common in an algorithm textbook can be directly reflected with the Frame Rule. When an algorithm is presented in a textbook, the correctness of the algorithm is often argued in two steps: first, the algorithm implements a local surgery of a data structure correctly, and second, such a local surgery preserves an invariant of the algorithm. For instance, for a program which deletes a node in a doubly-linked list, the first step corresponds to showing that the program implements a change of links appropriately, which is often described by a picture; the second step is to verify that such a change doesn't violate an invariant of a doubly-linked list, for instance, chasing the *next* field followed by the *prev* field leads to the starting node. The verification in this chapter uses such two step reasoning: we verify a local fact, which mostly describes a local surgery of the

Schorr-Waite graph marking algorithm; then, the global property, which is initially intended to show, is derived from the local fact using the Frame Rule, Consequence and Auxiliary Variable Elimination.

We start by explaining the assertion language and the programming language in this chapter, which are slightly different from those in Chapter 5. An implementation and a specification of the Schorr-Waite graph marking algorithm are shown in Section 7.2; they are essentially ones by Morris and Bornat. In Section 7.3, we prove that the implementation satisfies the specification. Instead of showing all steps involved, we focus on showing the preservation of a loop invariant, which best illustrates the main benefit of local reasoning in **BI** pointer logic.

## 7.1 Assertion and Programming Languages

The assertion language and the programming language in this chapter are slightly different from those in Chapter 5: they are typed, we have types not only for locations and boolean values but also for finite trees and two kinds of lists, and each heap cell has four fields. We make such a difference clear with revised definitions of semantic domains for storage. Let *Var* and *Loc* be as in Chapter 5: they are disjoint countably infinite sets such that neither of them contains *nil*. Each variable in *Var* is supposed to have a type among $\mathsf{bool}$, $\mathsf{loc}$, $\mathsf{list}$, $\mathsf{blist}$ and $\mathsf{tree}$, which are interpreted as follows:

$$
\begin{array}{rclcl}
[\![\mathsf{bool}]\!] & = & \mathit{Bool} & \triangleq & \{\mathit{true}, \mathit{false}\} \\
[\![\mathsf{loc}]\!] & = & \mathit{Loc}^{-} & \triangleq & \mathit{Loc} \cup \{\mathit{nil}\} \\
[\![\mathsf{list}]\!] & = & \mathit{List} & \triangleq & (\mathit{Loc}^{-} \times \mathit{Loc}^{-} \times \mathit{Loc}^{-} \times \mathit{Bool})^{*} \\
[\![\mathsf{blist}]\!] & = & \mathit{BList} & \triangleq & \mathit{Bool}^{*} \\
[\![\mathsf{tree}]\!] & = & \mathit{Tree} & = & \mathit{Loc}^{-} \cup \{(n, t_0, t_1) \mid n \in \mathit{Loc}^{-} \wedge t_0, t_1 \in \mathit{Tree}\}
\end{array}
$$

where *Tree* is the least set satisfying the given equation. For specifying the type of a variable, we assume that a function $\eta$ from *Var* to $\{\mathsf{loc}, \mathsf{bool}, \mathsf{list}, \mathsf{blist}, \mathsf{tree}\}$ is given, which maps infinite many variables to each type. When it is required to specify the type of a variable explicitly, we use a superscript to denote it, as in $x^{\mathsf{loc}}$. However, we don't write the type of a variable explicitly whenever the type is clear from the context. Finally, storage is interpreted as follows:

$$\begin{aligned}
\mathit{Stack} &\triangleq \{s\colon \mathit{Var} \to (\mathit{Bool} \cup \mathit{Loc}^- \cup \mathit{List} \cup \mathit{BList} \cup \mathit{Tree}) \mid \text{for all } x \in \mathit{Var}.\ s(x) \in [\![\eta(x)]\!]\} \\
\mathit{Heap} &\triangleq \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Loc}^- \times \mathit{Loc}^- \times \mathit{Bool} \times \mathit{Bool} \\
\mathit{State} &\triangleq \mathit{Stack} \times \mathit{Heap}
\end{aligned}$$

A stack is a map from $\mathit{Var}$ to the union of $\mathit{Bool}$, $\mathit{Loc}^-$, $\mathit{List}$, $\mathit{BList}$ and $\mathit{Tree}$, which preserves the type of a variable given by $\eta$; for instance, it maps $x^{\mathsf{loc}}$ to a location value in $\mathit{Loc}^-$. And each heap cell is presumed to have four fields, the first two of which contain location values and the last two of which boolean values.

The programming language is obtained from one in Chapter 5 just by incorporating types. For $i = 1, 2$ and $j = 3, 4$,

$$\begin{array}{llll}
\text{commands} & C & ::= & x^{\mathsf{bool}} := E.j \mid E.j := B \mid x^{\mathsf{loc}} := E.i \mid E.i := E \\
 & & & \mid x^{\mathsf{loc}} := \mathsf{cons}(E, E, B, B) \mid \mathsf{dispose}(E) \\
 & & & \mid x^{\mathsf{bool}} := B \mid x^{\mathsf{loc}} := E \mid \mathsf{skip} \mid C; C \\
 & & & \mid \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od} \\
\mathsf{loc}\ \text{expression} & E & ::= & x^{\mathsf{loc}} \ldots \mid \mathsf{nil} \\
\mathsf{bool}\ \text{expression} & B & ::= & x^{\mathsf{bool}} \ldots \mid \mathsf{false} \mid \mathsf{true} \mid \neg B \\
 & & & \mid B = B \mid B \vee B \mid B \wedge B \mid E = E
\end{array}$$

Notice that a variable in a program has either the $\mathsf{loc}$ type or the $\mathsf{bool}$ type. Variables of the other three types, $\mathsf{list}$, $\mathsf{blist}$ and $\mathsf{tree}$, are only used in assertions as auxiliary variables. The operational semantics of the programming language is the same as one in Chapter 5 except that all rules involving the access of heap cells is modified to handle heap cells with four fields. For instance, the rule for component assignment $E.3 := B$ is

$$\frac{[\![E]\!]s = l \in \mathit{dom}(h) \quad h(l) = (l_1, l_2, b_1, b_2)}{E.3 := B, s, h \ \rightsquigarrow\ \mathsf{skip}, s, h[l \mapsto (l_1, l_2, [\![B]\!]s, b_2)]}$$

One might wonder what happens when a location value is assigned to the third component of heap cells. Such a case never happens since the context free grammar of the programming language allows only type-correct programs.

The assertion language is similar to one in Chapter 5. The key difference is that the assertion language is typed so that each quantifier involves a type which a variable ranges over; every boolean expression is incorporated as an assertion; and the points-to relation becomes 5-ary to handle the different structure of heap cells. The syntax of the assertion language is given by the following context free grammar:

$$
\begin{array}{rcll}
P & ::= & (E \mapsto E, E, B, B) & \text{Points-to Relation} \\
 & & \mid B & \text{Boolean Expression} \\
 & & \mid L = L & \text{Equality for } \mathsf{list} \\
 & & \mid N = N & \text{Equality for } \mathsf{blist} \\
 & & \mid T = T & \text{Equality for } \mathsf{tree} \\
 & & \mid \mathsf{F} & \text{Falsity} \\
 & & \mid P \Rightarrow P & \text{Classical Implication} \\
 & & \mid \mathsf{emp} & \text{Empty Heap} \\
 & & \mid P * P & \text{Spatial Conjunction} \\
 & & \mid P \mathrel{-\!\!*} P & \text{Spatial Implication} \\
 & & \mid \forall x{:}\, \delta.\, P & \text{Universal Quantification}
\end{array}
$$

where $\delta$ is one of loc,bool,list,blist and tree; and $L$,$N$ and $T$ denote expressions of type list, blist and tree, respectively:

$$
\begin{array}{lrcl}
\mathsf{list}\text{ expression} & L & ::= & x^{\mathsf{list}} \ldots \mid \mathsf{lnil} \mid (E, E, E, B) :: L \\
\mathsf{blist}\text{ expression} & N & ::= & x^{\mathsf{blist}} \ldots \mid \mathsf{blnil} \mid B :: N \\
\mathsf{tree}\text{ expression} & T & ::= & x^{\mathsf{tree}} \ldots \mid E \mid (E, T, T)
\end{array}
$$

The interpretation of assertions is the same as in Chapter 5 except for a points-to relation, a boolean expression and quantification. However, the interpretation of them is straightforward:

$$
\begin{array}{lll}
s, h \models (E \mapsto E_1, E_2, B_1, B_2) & \text{iff} & dom(h) = \{[\![E]\!]s\} \text{ and} \\
 & & h([\![E]\!]s) = ([\![E_1]\!]s, [\![E_2]\!]s, [\![B_1]\!]s, [\![B_2]\!]s) \\
s, h \models B & \text{iff} & [\![B]\!]s = \mathit{true} \\
s, h \models \forall x{:}\, \delta.\, P & \text{iff} & \text{for all } v \text{ in } [\![\delta]\!],\ s[x \mapsto v], h \models P
\end{array}
$$

All rules in **BI** pointer logic still hold for the setting in this section. Recall that the axioms for if and while in Chapter 5 are different from the standard ones because the language in Chapter 5 is untyped. Since the language in this chapter is typed, the standard rules for if and while become sound. We recall them here:

Loop

$$\frac{\{P \wedge B\}\, C\, \{P\}}{\{P\}\ \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{od}\ \{P \wedge \neg B\}}$$

If

$$\frac{\{P \wedge B\}\, C\, \{Q\} \quad \{P \wedge \neg B\}\, C'\, \{Q\}}{\{P\}\ \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ C'\{Q\}}$$

## 7.2 Marking Algorithm and Specification

The implementation of the Schorr-Waite graph marking algorithm in this section is from [6], which is again Morris' implementation in [40] with a minor variation. Since the implementation is verified several times by others, the "semantic" elements of the proof, such as what pre- and post- conditions and a loop invariant should mean semantically, are already known. In fact, in this section, these pre- and post- conditions are used to write a specification, and in the next section, the loop invariant is used in formal verification. However, the assertions in this chapter, which express such semantic elements, are considerably different from those in other work. Moreover, the proof in the chapter proceeds exploiting locality of memory access; such a use of locality in the proof is hardly found elsewhere. We show the first point in this section, while postponing the more important second one to the next section.

### 7.2.1 Implementation

The Schorr-Waite algorithm marks all nodes reachable from *root*, which are not marked initially. It is essentially the depth first traversal with a clever implementation of the stack data structure using pointer reversals.

The implementation appears in Table 7.1. For each allocated heap cell, the first two fields contain links to other nodes. We call the first field *left* field and the second *right* field. The third field of the cell, which is called *check* field in this chapter, indicates whether traversing the *left* link is finished or not. So, when a node is in the stack data structure so that either *left* or *right* field is reversed, the *check* field contains *true* if and only if the *right* field is reversed. And the fourth field denotes whether the cell has already been visited, and it is called *mark* field. In the implementation, we use *left*, *right*, *check* and *mark* to denote 1st, 2nd, 3rd and 4th fields of a heap cell, respectively, for the sake of readability. So, $E.left$ is macro-expanded to $E.1$.

The variable $r$ points to the node currently visited by the algorithm, and $p$ denotes the top node in the stack data structure. The boolean variable $rm$ is used to hold the $mark$ field of $r$ when $r$ denotes an allocated heap cell. So, the condition for the while loop is that the stack data structure is not empty or the current node is an allocated unvisited heap cell. And inside the loop, the three cases are considered by three different branches: when the top node in the stack data structure should be popped, when the *left* link of the top node has been completely traversed so that the *right* link should be visited, when the current node is to be visited and pushed into the stack data structure since it is an allocated unvisited cell.

### 7.2.2 Specification

Any reasonable specification for the Schorr-Waite graph marking algorithm should say that if all nodes reachable from *root* are not marked initially, after executing *GraphMarking(root)*, all nodes initially reachable from *root* get marked but only their *mark* and *check* fields are changed. The specification in this section follows this common pattern with a slight variation on two aspects: in the precondition, we require that all pointers stored in a cell or in the variable *root* are not dangling and that all allocated heap cells are reachable from *root*. Even if the second aspect may seem strange to some readers, it reflects the idea of a so-called "tight" specification: a specification which mentions only those heap cells in concern. In fact, the "full" specification which additionally specifies that all unreachable heap cells are not altered follows from the "tight" one by the Frame Rule: if the implementation for the algorithm satisfies the "tight" specification, any properties of unreachable heap cells don't change.

Before giving a specification for the program, we define atomic predicates which state certain properties about finite lists, finite trees and heap storage. When all allocated heap cells are relevant to the truth of a predicate, we add $R$ at the end of the name of the predicate to indicate this. The letter $R$ stands for "relevant", and more technically $R$ indicates that a predicate is not intuitionistic[1]. We use the predicate $\hookrightarrow$ and the abbreviation ?. The predicate $\hookrightarrow$ is an intuitionistic version of the $\mapsto$ predicate: $(x \hookrightarrow l, r, c, m)$ is equivalent to $(x \mapsto l, r, c, m) * \top$. And ? either in $\mapsto$ or in $\hookrightarrow$ stands for an existentially quantified variable. For instance, the

[1]A predicate $P$ is intuitionistic iff for any $s, h, h'$ such that $h \# h'$, $(s, h \models P)$ implies that $s, h * h' \models P$

```
GraphMarking(root^loc)
local
    r^loc, p^loc, pl^loc, q^loc, pc^bool, rm^bool
in
    r := root;
    p := nil;
    if (r ≠ nil) then (rm := r.mark) else (rm := true);
    while (p ≠ nil ∨ (r ≠ nil ∧ ¬(rm)))
      do
        if ¬(r ≠ nil ∧ ¬(rm))
        then pc := p.check;
            if (pc) then q := r;
                         r := p;
                         p := p.right;
                         r.right := q
            else q := r;
                 r := p.right;
                 pl := p.left;
                 p.right := pl;
                 p.left := q;
                 p.check := true
        else q := p;
            p := r;
            r := r.left;
            p.left := q;
            p.mark := true;
            p.check := false;
        if (r ≠ nil) then (rm := r.mark) else (rm := true)
      od
end
```

**Table 7.1** The Schorr-Waite Graph Marking Algorithm formulas $(x \mapsto ?, ?, c, m)$ and $(x \hookrightarrow l, r, ?, ?)$ are abbreviations of $\exists l, r \colon \mathsf{loc}.\, (x \mapsto l, r, c, m)$ and $\exists c, m \colon \mathsf{bool}.\, (x \hookrightarrow l, r, c, m)$, respectively.

The predicate $\mathit{markedListR}(L, E)$ means that $E$ denotes the head of a "linked" list of marked nodes which "implements" $L$: by following the reversed pointers from $E$ upto $\mathit{nil}$, a finite linked list of marked nodes is obtained, and $L$ is formed just by taking the address and *left*, *right* and *check* fields of every node in the linked list. As the postfix $R$ indicates, the predicate also requires that all allocated heap cells should participate in the implementation. In the specification below, we use the predicate to state that $E$ denotes the head of the stack data structure, whose content is $L$. Formally, given an expression $E$ of type $\mathsf{loc}$ and an expression $L$ of type $\mathsf{list}$, $s, h \models \mathit{markedListR}(L, E)$ if and only if $h \in f([\![L]\!]s, [\![E]\!]s)$ where $f$ is a unique function from $\mathit{List} \times \mathit{Loc}^{-}$ to $\mathcal{P}(\mathit{Heap})$ satisfying the following equations:

$$
\begin{aligned}
& f(\epsilon, n) \\
& \quad = \begin{cases} \{\emptyset\} & \text{if } n = \mathit{nil} \\ \emptyset & \text{otherwise} \end{cases} \\
& f((n_0, n_l, n_r, \mathit{true})tl, n) \\
& \quad = \begin{cases} \{h' * (n_0 \mapsto n_l, n_r, \mathit{true}, \mathit{true}) \mid h' \in f(tl, n_r)\} & \text{if } n_0 = n \\ \emptyset & \text{otherwise} \end{cases} \\
& f((n_0, n_l, n_r, \mathit{false})tl, n) \\
& \quad = \begin{cases} \{h' * (n_0 \mapsto n_l, n_r, \mathit{false}, \mathit{true}) \mid h' \in f(tl, n_l)\} & \text{if } n_0 = n \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

The following three equivalences between assertions hold for all states because they are just paraphrases of three defining equations above:

$$
\begin{aligned}
& \mathit{markedListR}(\mathsf{lnil}, E) \\
& \quad \Longleftrightarrow (E = \mathsf{nil}) \wedge \mathsf{emp} \\
& \mathit{markedListR}((E_0, E_l, E_r, \mathsf{true}) :: L, E) \\
& \quad \Longleftrightarrow (E_0 = E) \wedge (E_0 \mapsto E_l, E_r, \mathsf{true}, \mathsf{true}) * \mathit{markedListR}(L, E_r) \\
& \mathit{markedListR}((E_0, E_l, E_r, \mathsf{false}) :: L, E) \\
& \quad \Longleftrightarrow (E_0 = E) \wedge (E_0 \mapsto E_l, E_r, \mathsf{false}, \mathsf{true}) * \mathit{markedListR}(L, E_l)
\end{aligned}
$$

The predicate $\mathit{restoredListR}(L, E)$ is intended to denote that there is a linked list ending with $E$ in heap storage which represents the "restored" version of the list $L$: when either the second or the third value of each tuple in $L$ is "restored back appropriately" and the list itself is reversed, the first three values of each tuple in the resulting list are the address and *left* and *right* fields of the corresponding node in the linked list. And all allocated heap cells should be a part of the linked list. In the specification below, we use the predicate to denote a heap which is obtained by restoring back the reversed pointers in the stack data structure. Formally, given an expression $E$ of type $\mathsf{loc}$ and an expression $L$ of type $\mathsf{list}$, $s, h \models \mathit{restoredListR}(L, E)$ if and only if $h \in f([\![L]\!]s, [\![E]\!]s)$ where $f$ is a unique function from $\mathit{List} \times \mathit{Loc}^-$ to $\mathcal{P}(\mathit{Heap})$ satisfying the following equations:

$$\begin{array}{rcl} f(\epsilon, n) & = & \{\emptyset\} \\ f((n_0, n_l, n_r, \mathit{true})tl, n) & = & \{h * (n_0 \mapsto n_l, n, \mathit{true}, \mathit{true}) \mid h \in f(tl, n_0)\} \\ f((n_0, n_l, n_r, \mathit{false})tl, n) & = & \{h * (n_0 \mapsto n, n_r, \mathit{true}, \mathit{true}) \mid h \in f(tl, n_0)\} \end{array}$$

The above three defining equations give rise to the following three equivalences between assertions, which hold for all states:

$$\begin{array}{l} \mathit{restoredListR}(\mathsf{lnil}, E) \\ \qquad \iff \mathsf{emp} \\ \mathit{restoredListR}((E_0, E_l, E_r, \mathsf{true}) :: L, E) \\ \qquad \iff (E_0 \mapsto E_l, E, \mathsf{true}, \mathsf{true}) * \mathit{restoredListR}(L, E_0) \\ \mathit{restoredListR}((E_0, E_l, E_r, \mathsf{false}) :: L, E) \\ \qquad \iff (E_0 \mapsto E, E_r, \mathsf{true}, \mathsf{true}) * \mathit{restoredListR}(L, E_0) \end{array}$$

The predicate $\mathit{spansR}(T, E)$ denotes that a tree $T$ spans all reachable nodes and links from $E$ and, moreover, all currently allocated nodes are reachable from $E$. Formally, given an expression $T$ of type $\mathsf{tree}$ and an expression $E$ of type $\mathsf{loc}$, $s, h \models \mathit{spansR}(T, E)$ if and only if $h \in f([\![T]\!]s, [\![E]\!]s)$ where $f$ is a unique function from $\mathit{Tree} \times \mathit{Loc}^-$ to $\mathcal{P}(\mathit{Heap})$ satisfying the following equations:

$$\begin{array}{l} f(n_0, n) \\ = \begin{cases} \{\emptyset\} & \text{if } n = n_0 \\ \emptyset & \text{otherwise} \end{cases} \\ f((n_0, t_0, t_1), n) \\ = \begin{cases} \{(n_0 \mapsto n_l, n_r, b_c, b_m) * h_0 * h_1 \mid h_0 \in f(t_0, n_l) \wedge h_1 \in f(t_1, n_r)\} & \text{if } n = n_0 \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

We provide equivalences between assertions which correspond to the defining equations for $f$:

$$\begin{aligned}
& \mathit{spansR}(E_0, E) \\
& \quad \iff (E_0 = E) \wedge \mathsf{emp} \\
& \mathit{spansR}((E_0, T_l, T_r), E) \\
& \quad \iff (E_0 = E) \wedge (\exists l, r \colon \mathsf{loc}.\, (E_0 \mapsto l, r, ?, ?) * \mathit{spansR}(T_l, l) * \mathit{spansR}(T_r, r))
\end{aligned}$$

where $l, r$ don't occur free in $E_0, T_l, T_r$.

The predicate $\mathit{reach}(E_0, E)$ denotes that there is a path of nodes in heap storage from the node $E_0$ to the node $E$. And $\mathit{reachRightChildInList}(L, E)$ means that the node $E$ is reachable from the right child of a node in the list $L$, i.e., the third element of a tuple in $L$. For the precise definitions of $\mathit{reach}$ and $\mathit{reachRightChildInList}$, we define functions $f$ of type $\mathit{Loc}^- \times \mathit{Loc}^- \times \mathit{BList} \to \mathcal{P}(\mathit{Heap})$ and $g$ of type $\mathit{List} \times \mathit{Loc}^- \to \mathcal{P}(\mathit{Heap})$. They are the unique solutions for the following equations:

$$\begin{aligned}
f(n_0, n, \epsilon) &= \begin{cases} \mathit{Heap} & \text{if } n_0 = n \\ \emptyset & \text{otherwise} \end{cases} \\
f(n_0, n, (\mathit{true})\mathit{tl}) &= \{(n_0 \mapsto n_l, n_r, b_c, b_m) * h' \mid h' \in f(n_r, n, \mathit{tl})\} \\
f(n_0, n, (\mathit{false})\mathit{tl}) &= \{(n_0 \mapsto n_l, n_r, b_c, b_m) * h' \mid h' \in f(n_l, n, \mathit{tl})\}
\end{aligned}$$

$$\begin{aligned}
g(\epsilon, n) &= \emptyset \\
g((n_0, n_l, n_r, b_c)\mathit{tl}, n) &= \left(\bigcup\nolimits_{l \in \mathit{BList}} f(n_r, n, l)\right) \cup g(\mathit{tl}, n)
\end{aligned}$$

Then, $\mathit{reach}(E_0, E)$ and $\mathit{reachRightChildInList}(L, E)$ are defined by:

$$\begin{aligned}
s, h \models \mathit{reach}(E_0, E) \quad & \text{iff} \quad h \in \bigcup\nolimits_{l \in \mathit{BList}} f([\![E_0]\!]s, [\![E]\!]s, l) \\
s, h \models \mathit{reachRightChildInList}(L, E) \quad & \text{iff} \quad h \in g([\![L]\!]s, [\![E]\!]s)
\end{aligned}$$

The defining equations of $g$ give rise to equivalences between assertions involving $\mathit{reachRightChildInList}$:

$$\begin{aligned}
& \mathit{reachRightChildInList}(\mathsf{lnil}, E) \\
& \quad \iff \mathsf{F} \\
& \mathit{reachRightChildInList}((E_0, E_l, E_r, B_c) :: L, E) \\
& \quad \iff \mathit{reach}(E_r, E) \vee \mathit{reachRightChildInList}(L, E)
\end{aligned}$$

The predicate $allocated(E)$ means that $E$ denotes an allocated cell, $markedR$ that all allocated heap cells are marked and $unmarkedR$ that no allocated heap cells are marked. They are defined by the following equivalences:

$$\begin{aligned} allocated(E) &\iff (E \hookrightarrow ?, ?, ?, ?) \\ markedR &\iff \forall x\colon \mathsf{loc}.\ allocated(x) \Rightarrow (x \hookrightarrow ?, ?, ?, \mathsf{true}) \\ unmarkedR &\iff \forall x\colon \mathsf{loc}.\ allocated(x) \Rightarrow (x \hookrightarrow ?, ?, ?, \mathsf{false}) \end{aligned}$$

The following two predicates are about dangling pointers; $noDangling(E)$ says that $E$ is not a dangling pointer and $noDanglingR$ that no dangling pointers are stored in any allocated heap cell. They are defined by the following equivalences:

$$\begin{aligned} noDangling(E) &\iff (E = \mathsf{nil}) \lor allocated(E) \\ noDanglingR &\iff \forall x, l, r\colon \mathsf{loc}. \\ &\qquad\quad (x \hookrightarrow l, r, ?, ?) \Rightarrow noDangling(l) \land noDangling(r) \end{aligned}$$

Stating the specification mentioned in the beginning of this section in the **BI** assertion language now becomes straightforward once we express that all nodes reachable from *root* have the same values for *left* and *right* fields before and after the execution. We use a spanning tree *STree* for the graph of all reachable nodes to store values of *left* and *right* fields of all reachable nodes.

$$\begin{gathered} \{spansR(STree, root) \land unmarkedR \land noDanglingR \land noDangling(root)\} \\ GraphMarking(root) \\ \{spansR(STree, root) \land markedR\} \end{gathered}$$

Informally, the specification says that if before executing *GraphMarking(root)*,

- all allocated heap cells are reachable from *root*, and *STree* is a spanning tree of those allocated heap cells;
- all allocated heap cells are not marked;
- there are no dangling pointers stored in any allocated heap cells; and
- *root* doesn't contain a dangling pointer,

then after the execution,

- $STree$ is still a spanning tree of all allocated heap cells; and

- all allocated heap cells are marked.

Notice that for two heaps $h$ and $h'$, if they have the same $STree$ as a spanning tree, $h$ and $h'$ have the same allocated heap cells and the contents of those allocated heap cells can be different only in the third and fourth fields. That is, if $(s,h) \models spansR(STree, root)$ and $(s,h') \models spansR(STree, root)$, then $dom(h) = dom(h')$ and for all $l \in dom(h)$, we have $h(l).1 = h'(l).1$ and $h(l).2 = h'(l).2$. Therefore, the mere fact that $STree$ is a spanning tree before and after the execution implies that no new heap cells are allocated or deallocated during the execution and that the *left* and *right* fields of all heap cells have the same values before and after the execution.

## 7.3 Verification

In this section, we use **BI** pointer logic to verify that the implementation of the Schorr-Waite graph marking in Section 7.2 satisfies the specification. We only show that the `while` loop in the implementation preserves a loop invariant because showing such preservation is the most complex step and best illustrates the advantage of local reasoning.

### 7.3.1 Outline

The following code fragment with assertions inserted gives the outline of the proof:

$\{Precond\}$

$r := root;$

$p :=$ nil;

$\{\exists Stack\colon \mathsf{list}.\ preLoopInvR(Stack, p, r, STree, root)\}$

if $(r \neq$ nil$)$ then $(rm := r.mark)$ else $(rm :=$ true$)$;

$\{(\exists Stack\colon \mathsf{list}.\ preLoopInvR(Stack, p, r, STree, root)) \wedge (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm))\}$

while $(p \neq \mathsf{nil} \vee (r \neq \mathsf{nil} \wedge \neg(rm)))$

  do

    if $\neg(r \neq \mathsf{nil} \wedge \neg(rm))$

    then $pc := p.check;$

      if $(pc)$ then $q := r;$

          $r := p;$

          $p := p.right;$

          $r.right := q$

      else $q := r;$

        $r := p.right;$

        $pl := p.left;$

        $p.right := pl;$

        $p.left := q;$

        $p.check :=$ true

    else $q := p;$

      $p := r;$

      $r := r.left;$

      $p.left := q;$

      $p.mark :=$ true;

      $p.check :=$ false;

    $\{\exists Stack\colon \mathsf{list}.\ preLoopInvR(Stack, p, r, STree, root)\}$

    if $(r \neq$ nil$)$ then $(rm := r.mark)$ else $(rm :=$ true$)$

    $\{(\exists Stack\colon \mathsf{list}.\ preLoopInvR(Stack, p, r, STree, root)) \wedge (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm))\}$

  od

$\{Postcond\}$

where *Precond* and *Postcond* are macros for pre- and post- conditions in Section 7.2:

$$\begin{aligned}
\mathit{Precond} &\triangleq \mathit{spansR}(\mathit{STree}, \mathit{root}) \wedge \mathit{unmarkedR} \\
&\quad \wedge \mathit{noDanglingR} \wedge \mathit{noDangling}(\mathit{root}) \\
\mathit{Postcond} &\triangleq \mathit{spansR}(\mathit{STree}, \mathit{root}) \wedge \mathit{markedR}
\end{aligned}$$

and the predicate *preLoopInvR*, which is the main part of the loop invariant, is defined by the following equivalences:

$$\begin{aligned}
&\mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root}) \\
&\Longleftrightarrow \\
&\mathit{noDanglingR} \wedge \mathit{noDangling}(r) \wedge \mathit{noDangling}(p) \\
&\wedge \mathit{markedListR}(\mathit{Stack}, p) * (\mathit{restoredListR}(\mathit{Stack}, r) -\!\!* \; \mathit{spansR}(\mathit{STree}, \mathit{root})) \\
&\wedge \mathit{markedR} * \begin{pmatrix} \mathit{unmarkedR} \\ \wedge \forall x\colon \mathsf{loc}.\ \mathit{allocated}(x) \Rightarrow \begin{pmatrix} \mathit{reach}(r, x) \\ \vee \mathit{reachRightChildInList}(\mathit{Stack}, x) \end{pmatrix} \end{pmatrix}
\end{aligned}$$

Informally, $\mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root})$ says that

- $p$ and $r$ are not dangling, and any allocated heap cell doesn't contain dangling pointers;
- $p$ is the head of the stack data structure implemented by pointer reversals, *Stack* represents the contents of the stack data structure and all nodes in the stack data structure are marked;
- if the reversed links of all nodes in the stack data structure are restored back, where the first reversed link is restored to $r$, then all link fields, i.e., *left* and *right* fields, of every node in the heap have initial values, which is stored in *STree*; and
- all unmarked allocated nodes are reachable from $r$ or the *right* field of a node stored in the stack data structure (*Stack*) going only though unmarked nodes.

Notice that $\mathit{markedListR}(\mathit{Stack}, p) * (\mathit{restoredListR}(\mathit{Stack}, r) -\!\!* \; \mathit{spansR}(\mathit{STree}, \mathit{root}))$ expresses the second and the third points at the same time in a natural way using $*$ and $-\!\!*$. It states that a heap can be divided into two parts; the first part of the heap implements the stack data structure with $p$ as its head node; if the stack data structure is taken away from the heap, thereby leaving only the second part, and then the stack data structure with all reversed links restored back is added, the *left* and *right* fields of all allocated nodes have initial values stored in the spanning tree $STree$.

Let init, body and setrm denote the following code fragments:

```
init ≜ r := root; p := nil

body ≜ if ¬(r ≠ nil ∧ ¬(rm))
         then pc := p.check;
              if (pc) then q := r;
                           r := p;
                           p := p.right;
                           r.right := q
              else q := r;
                   r := p.right;
                   pl := p.left;
                   p.right := pl;
                   p.left := q;
                   p.check := true
         else q := p;
              p := r;
              r := r.left;
              p.left := q;
              p.mark := true;
              p.check := false

setrm ≜ if (r ≠ nil) then (rm := r.mark) else (rm := true)
```

Let $Cond$ denote the condition of the while loop in the program. Then, it suffices to show the following three:

- the loop invariant is established initially;

$$\{\mathit{Precond}\}$$
$$\mathsf{init}; \mathsf{setrm}$$
$$\left\{\begin{array}{l}(\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root}))\\ \wedge\ (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm))\end{array}\right\}$$

- the loop invariant is preserved by the loop body; and

$$\left\{\begin{array}{l}(\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root}))\\ \wedge\ (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm)) \wedge \mathit{Cond}\end{array}\right\}$$
$$\mathsf{body}; \mathsf{setrm}$$
$$\left\{\begin{array}{l}(\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root}))\\ \wedge\ (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm))\end{array}\right\}$$

- the loop invariant implies the postcondition.

$$\left(\begin{array}{l}(\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root}))\\ \wedge\ (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm)) \wedge \neg \mathit{Cond}\end{array}\right) \implies \mathit{Postcond}$$

By the axiom for sequencing, the first two can be derived from the following three Hoare triples.

$$\{\mathit{Precond}\}\ \mathsf{init}\ \{\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root})\}$$

$$\{\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root})\}$$
$$\mathsf{setrm}$$
$$\{(r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm)) \wedge \exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root})\}$$

$$\left\{\begin{array}{l}(\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root}))\\ \wedge\ (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm)) \wedge \mathit{Cond}\end{array}\right\}$$
$$\mathsf{body}$$
$$\{\exists \mathit{Stack}\colon \mathsf{list}.\ \mathit{preLoopInvR}(\mathit{Stack}, p, r, \mathit{STree}, \mathit{root})\}$$

Since it is the last third case that shows clearly the benefit of local reasoning, we only handle the third case.

### 7.3.2 The Correctness of $\mathsf{body}$

For the correctness of $\mathsf{body}$, we start with an informal argument. The argument consists of three steps: firstly, all initial states in the precondition are classified into three groups, depending on whether the current node needs to be traversed and if it doesn't, which link of the top node in the stack data structure is currently being traversed; secondly, for each group, we calculate what local surgery $\mathsf{body}$ does; and finally, we argue that for each group, such a local surgery produces a final state in the postcondition. While verifying the correctness of $\mathsf{body}$ informally, we point out that all important steps are formally expressible by implications between assertions. And the following formal verification is just a paraphrase of the informal argument in **BI** pointer logic.

Since the condition of the $\mathsf{while}$ loop says that either the stack data structure is not empty or the current node needs to be visited, states satisfying the loop invariant and *Cond* can be divided into three cases: the first case and the second case are when the current node need not be traversed because it is either *nil* or already marked. Additionally, in the first case, the *right* link of the top node is currently being traversed; therefore, the appropriate action is to pop the top node from the stack data structure. In the second case, the *left* child of the top node is being traversed; in this case, the *right* link of the top node should be traversed next. The third case is when the current node needs to be visited and pushed into the stack data structure. Recall that the *check* field of each heap cell indicates whether the *right* link is being traversed or not when the cell is in the stack data structure. Therefore, the three cases are described with assertions as follows:

- the current node need not be visited, and the *right* link of the top node in the stack data structure is currently being traversed:

$$(r = \mathsf{nil} \vee (r \hookrightarrow ?,?,?,\mathsf{true}) \wedge (rm = \mathsf{true})) \wedge (p \hookrightarrow ?,?,\mathsf{true},\mathsf{true})$$

- the current node need not be visited, and the *left* link of the top node in the stack data structure is currently being traversed:

$$(r = \mathsf{nil} \vee (r \hookrightarrow ?,?,?,\mathsf{true}) \wedge (rm = \mathsf{true})) \wedge (p \hookrightarrow ?,?,\mathsf{false},\mathsf{true})$$

- the current node needs to be visited:

$$(r \hookrightarrow ?,?,?,\mathsf{false}) \wedge (rm = \mathsf{false})$$

The following lemma states that such a classification, in fact, holds.

**Lemma 76 (Case Splitting)** *The assertion*

$$(\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root))$$
$$\wedge\, (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm)) \wedge Cond$$

*implies*

$$\Big((r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge (rm = \mathsf{true})) \wedge (p \hookrightarrow ?, ?, \mathsf{true}, \mathsf{true})\Big)$$
$$\vee \Big((r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge (rm = \mathsf{true})) \wedge (p \hookrightarrow ?, ?, \mathsf{false}, \mathsf{true})\Big)$$
$$\vee \Big((r \hookrightarrow ?, ?, ?, \mathsf{false}) \wedge (rm = \mathsf{false})\Big)$$

*Proof:* The proof is shown in Section 7.4.1. □

To argue that the implementation is correct, we have to check whether the implementation does what it is supposed to do in each case. Since each case is handled in a different branch in body, it is easy to figure out the actual action of body in each case. Let $p_0$ and $r_0$ denote the initial values of $p$ and $t$, respectively. Then, the implementation does the appropriate action in each case:

- in the first case, the top node in the stack data structure is popped, and $p$ and $r$ are updated to denote a new top node and a new current node, respectively: the heap cell $p_0$, which contains $(p_l, p_r, \mathsf{true}, \mathsf{true})$ initially, the two variables $p$ and $r$ are updated to $(p_l, r_0, \mathsf{true}, \mathsf{true})$, $p_r$ and $p_0$, respectively;

- in the second case, the right child of the top node becomes the current node, and the two link fields and the check field of the top node are changed appropriately: the heap cell $p_0$, which contains $(p_l, p_r, \mathsf{false}, \mathsf{true})$ initially, and a variable $r$ are updated to $(r_0, p_l, \mathsf{true}, \mathsf{true})$ and $p_r$, respectively; and

- in the third case, the current node is pushed into the stack data structure, and variables $p$ and $r$ are changed to denote the new top node and the new current node, respectively: the heap cell $r_0$, which contains $(r_l, r_r, r_c, \mathsf{false})$ initially, and two variables $p$ and $r$ are updated to $(p_0, r_r, \mathsf{false}, \mathsf{true})$, $r_0$ and $r_l$, respectively.

The last step is to argue that for each case, the action of body doesn't result in any final state which violates the postcondition. For instance, in the first case, it is necessary to show that when the cell $p_0$, which contains $(p_l, p_r, \mathsf{true}, \mathsf{true})$ initially, and two variables $p$ and $r$ are updated to $(p_l, r_0, \mathsf{true}, \mathsf{true})$, $p_r$ and $p_0$, respectively, the resulting state satisfies the postcondition as long as the initial state satisfies the precondition. With the connectives $*, -\!\!*$, this reasoning is expressible by implications between assertions. The key point is that $(x \mapsto l, r, c, m) * ((x \mapsto l', r', c', m') -\!\!* C)$ denotes that if the heap cell $x$ with $(l, r, c, m)$ is updated to contain $(l', r', c', m')$, then $C$ holds. Recall that in the predicate *preLoopInvR*, the first argument is meant to denote the content of the stack data structure, the second the address of the top node in the stack data structure and the third the address of the current node; and that $p$ and $r$ are variables which keep track of addresses of the top node in the stack data structure and the current node, respectively. The following three lemmas express that in each case, the action of $\mathsf{body}$ only produces an appropriate final state:

**Lemma 77** *The assertion*

$$\begin{aligned}&\mathit{preLoopInvR}((p_0, p_l, p_r, \mathsf{true}) :: \mathit{Stack}, p_0, r_0, \mathit{STree}, \mathit{root})\\ &\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))\\ &\wedge (p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})\end{aligned}$$

*implies*

$$\begin{aligned}&(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true})\\ &* ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{preLoopInvR}(\mathit{Stack}, p_r, p_0, \mathit{STree}, \mathit{root}))\end{aligned}$$

*Proof:* The proof is shown in Section 7.4.2. □

**Lemma 78** *The assertion*

$$\begin{aligned}&\mathit{preLoopInvR}((p_0, p_l, p_r, \mathsf{false}) :: \mathit{Stack}, p_0, r_0, \mathit{STree}, \mathit{root})\\ &\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))\\ &\wedge (p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true})\end{aligned}$$

*implies*

$$\begin{aligned}&(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\ &* \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{true}) :: \mathit{Stack}, p_0, p_r, \mathit{STree}, \mathit{root})\end{pmatrix}\end{aligned}$$

*Proof:* The proof is shown in Section 7.4.3. □

**Lemma 79** *The assertion*

$$preLoopInvR(Stack, p_0, r_0, STree, root) \wedge (r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{true})$$

*implies*

$$\begin{array}{l} (r_0 \mapsto r_l, r_r, r_c, \mathsf{true}) \\ * \left( \begin{array}{l} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \\ -\!\!* \, preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root) \end{array} \right) \end{array}$$

*Proof:* The proof is shown in Section 7.4.4. □

In the remainder of the section, we formally prove that if an initial state satisfies the precondition and the execution of body terminates, the resulting state satisfies the postcondition. We use Case Analysis, which is a sound proof rule both for partial correctness and for total correctness:

$$\frac{\{P_1\}\, C\, \{Q\} \quad \{P_2\}\, C\, \{Q\} \quad \{P_3\}\, C\, \{Q\}}{\{P_1 \vee P_2 \vee P_3\}\, C\, \{Q\}} \; \textsc{Case Analysis}$$

Since the states in the precondition are categorized into groups by Lemma 76, it suffices to show the following three Hoare triples:

$$\left\{ \begin{array}{l} (\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)) \\ \wedge\, (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge (rm = \mathsf{true})) \wedge (p \hookrightarrow ?, ?, \mathsf{true}, \mathsf{true}) \end{array} \right\}$$

$$\mathsf{body}$$

$$\{\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)\}$$

$$\left\{ \begin{array}{l} (\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)) \\ \wedge\, (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true}) \wedge (p \hookrightarrow ?, ?, \mathsf{false}, \mathsf{true}) \end{array} \right\}$$

$$\mathsf{body}$$

$$\{\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)\}$$

$$\left\{ \begin{array}{l} (\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)) \\ \wedge\, (rm = \mathsf{false}) \wedge (r \hookrightarrow ?, ?, ?, \mathsf{false}) \end{array} \right\}$$

$$\mathsf{body}$$

$$\{\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)\}$$

Handing each case is the main part of this chapter. For all three cases, the verification proceeds in two stages, one for obtaining a local fact and the other for changing a local fact into a global fact. More concretely, in the first stage, we show a certain "local" specification, given by a Hoare triple, holds for $\mathsf{body}$. Such a specification summarizes what $\mathsf{body}$ does by describing a local surgery in heap storage. So, it only mentions the heap cells actually accessed by $\mathsf{body}$. In the next stage, we use this local fact to derive another Hoare triple, which is what we initially intended to show and might involve heap cells not accessed in $\mathsf{body}$. With the three lemmas (Lemma 77, 78 and 79), which are the main part of an informal correctness argument, the change from a local fact to a global fact becomes straightforward.

#### 7.3.2.1 The first case

The Hoare triple, which corresponds to this case, is the following:

$$\left\{\begin{array}{l}(\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)) \\ \wedge\, (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge (rm = \mathsf{true})) \\ \wedge\, (\exists p_l, p_r\colon \mathsf{loc}.\, (p \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}))\end{array}\right\}$$

$$\mathsf{body}$$

$$\{\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)\}$$

In the first case, $\mathsf{body}$ only accesses the heap cell initially pointed to by $p$. We exploit such locality by first proving a fact for the accessed heap cell and variables, and then using it to derive the intended Hoare triple. The following Hoare triple describes such a fact:

$$\left\{\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \wedge\, (r = \mathsf{nil} \vee rm = \mathsf{true}) \\ \wedge\, (p = p_0) \wedge (r = r_0)\end{array}\right\} \mathsf{body} \left\{\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ \wedge\, (p = p_r) \wedge (r = p_0)\end{array}\right\}$$

Notice that in all states satisfying the precondition in the above Hoare triple, only one heap cell is allocated in the heap, and it is pointed to by $p$. The Hoare triple precisely and only describes how the only heap cell and two variables $p, r$ are updated by $\mathsf{body}$. We call the original Hoare triple *global* and the new one in the above *local*. It is straightforward to verify the local Hoare triple as the following proof outline shows:

$\{(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge (r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0)\}$

if $\neg(r \neq \mathsf{nil} \wedge \neg(rm))$

then $pc := p.check$;

  if $(pc)$

  then

    $\{(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0)\}$

    $q := r; r := p; p := p.right; r.right := q$

  else

    $\{\mathsf{F}\}$

    $q := r; r := p.right; pl := p.left; p.right := pl; p.left := q; p.check := \mathsf{true}$

else

  $\{\mathsf{F}\}$

  $q := p; p := r; r := r.left; p.left := q; p.mark := \mathsf{true}; p.check := \mathsf{false}$

$\{(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \wedge (p = p_r) \wedge (r = p_0)\}$

The most important step in the proof is to change the mode of reasoning from local to global, that is, to derive the global Hoare triple from the local one. It is the Frame Rule and the connectives $*$, $-\!\!*$ that make such a change possible:

$$\frac{\left\{\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \wedge (r = \mathsf{nil} \vee rm = \mathsf{true}) \\ \wedge (p = p_0) \wedge (r = r_0)\end{array}\right\} \mathsf{body} \left\{\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ \wedge (p = p_r) \wedge (r = p_0)\end{array}\right\}}{\left\{\begin{pmatrix}(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \wedge (r = \mathsf{nil} \vee rm = \mathsf{true}) \\ \wedge (p = p_0) \wedge (r = r_0)\end{pmatrix} * \begin{pmatrix}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!*\ preLoopInvR(Stack', p_r, p_0, STree, root)\end{pmatrix}\right\}} \text{FR}$$

$$\mathsf{body}$$

$$\left\{\begin{pmatrix}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ \wedge (p = p_r) \wedge (r = p_0)\end{pmatrix} * \begin{pmatrix}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!*\ preLoopInvR(Stack', p_r, p_0, STree, root)\end{pmatrix}\right\}$$

The remaining step is to obtain the global Hoare triple from the above one. We first eliminate auxiliary variables $p_0$,$p_l$,$p_r$,$r_0$ and $Stack'$ and then derive the global Hoare triple by Consequence. When Auxiliary Variable Elimination is applied, we obtain the following Hoare triple:

$$\dfrac{\vdots}{\begin{array}{c}\left\{\begin{array}{l}\exists p_0, p_l, p_r, r_0\colon \mathsf{loc}, Stack'\colon \mathsf{list}.\\ \left(\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true})\\ \wedge\, (r = \mathsf{nil} \vee rm = \mathsf{true})\\ \wedge\, (p = p_0) \wedge (r = r_0)\end{array}\right) * \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true})\\ -\!\!* \, preLoopInvR(Stack', p_r, p_0, STree, root)\end{array}\right)\end{array}\right\}\\ \\ \mathsf{body}\\ \\ \left\{\begin{array}{l}\exists p_0, p_l, p_r, r_0\colon \mathsf{loc}, Stack'\colon \mathsf{list}.\\ \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true})\\ \wedge\, (p = p_r) \wedge (r = p_0)\end{array}\right) * \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true})\\ -\!\!* \, preLoopInvR(Stack', p_r, p_0, STree, root)\end{array}\right)\end{array}\right\}\end{array}}\ \mathrm{AVE}$$

In order to use Consequence to derive the global Hoare triple from the above one, we have to show that the postcondition of the above Hoare triple implies that of the global one and that the precondition of the global implies that of the above. Before proving such implications, we note one property of *pure* assertions. An assertion $P$ is *pure* if it only depends on stack storage: for all stacks $s$, if there is a heap $h$ with $s, h \models P$, then $s, h' \models P$ for all heaps $h'$. One syntactic criteria to decide whether an assertion is pure or not is to check whether the assertion contains non-pure atomic predicates, which are $\mathsf{emp}$, $(- \mapsto -, -, -, -)$ and all predicates that we defined in Section 7.2. We use a particular property of pure assertions in the following verification, which is that when a pure assertion $P$ is used as a conjunct, it moves into and out of $*$:

$$(Q * R) \wedge P \models (Q \wedge P) * R \;\text{ and }\; (Q \wedge P) * R \models (Q * R) \wedge P$$

The following sequence of implications shows the implication between postconditions.

$$\begin{array}{l}\exists p_0, p_l, p_r, r_0\colon \mathsf{loc}, Stack'\colon \mathsf{list}.\\ \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true})\\ \wedge\, (p = p_r) \wedge (r = p_0)\end{array}\right) * \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true})\\ -\!\!* \, preLoopInvR(Stack', p_r, p_0, STree, root)\end{array}\right)\end{array}$$

$\Longrightarrow (p = p_r) \wedge (r = p_0)$ is pure

$$\begin{array}{l}\exists p_0, p_l, p_r, r_0\colon \mathsf{loc}, Stack'\colon \mathsf{list}.\\ (p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) * \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true})\\ -\!\!* \, preLoopInvR(Stack', p_r, p_0, STree, root)\end{array}\right)\\ \wedge\, (p = p_r) \wedge (r = p_0)\end{array}$$

$\Longrightarrow P * (P -\!\!* Q) \models Q$

$\exists p_0, p_l, p_r, r_0 : \mathsf{loc}, Stack' : \mathsf{list}.\ preLoopInvR(Stack', p_r, p_0, STree, root) \wedge (p = p_r) \wedge (r = p_0)$

$\Longrightarrow$

$\exists Stack' : \mathsf{list}.\ preLoopInvR(Stack', p, r, STree, root)$

We use the following observation to show the implication between preconditions: if $p$ denotes an allocated heap cell containing $(p_l, p_r, \mathsf{true}, \mathsf{true})$ and $p$ is the top node implementing a list $Stack$ of tuples, then $Stack$ is not empty and has $(p, p_l, p_r, \mathsf{true})$ as its head. That is,

**Lemma 80** *The assertion*

$$(p \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge markedListR(Stack, p) * \mathsf{T}$$

*implies*

$$\exists Stack' : \mathsf{list}.\ Stack = (p, p_l, p_r, \mathsf{true}) :: Stack'.$$

*Proof:* Suppose $s, h \models (p \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge markedListR(Stack, p) * \mathsf{T}$. Then, it suffices to show that $[\![Stack]\!]s = ([\![p]\!]s, [\![p_l]\!]s, [\![p_r]\!]s, true)tl$ for some $tl \in List$. Let $f$ be the function in the definition of $markedListR$. Then, the heap $h$ can be divided into two subheaps $h_1, h_2$ ($h = h_1 * h_2$) such that $h_1 \in f([\![Stack]\!]s, [\![p]\!]s)$. Since $[\![p]\!]s \in dom(h)$, cell $[\![p]\!]s$ is not equal to $nil$. Moreover, $f([\![Stack]\!]s, [\![p]\!]s)$ is not empty by assumption. Therefore, $[\![Stack]\!]s$ can't be $\epsilon$. Let $[\![Stack]\!]s$ be $(n_0, n_l, n_r, b)tl$. Since $h_1 \in f([\![Stack]\!]s, [\![p]\!]s)$, we have

$$[\![p]\!]s = n_0 \quad \text{and} \quad h_1(n_0) = (n_l, n_r, b, true)$$

So, $n_l = [\![p_l]\!]s$, $n_r = [\![p_r]\!]s$ and $b = true$ because $h_1(n_0) = h([\![p]\!]s)$. $\square$

Therefore, when both $preLoopInvR(Stack, p, r, STree, root)$ and $(p \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})$ hold in a state, so does $\exists Stack' : \mathsf{list}.\ Stack = (p, p_l, p_r, \mathsf{true}) :: Stack'$ in the state because the loop invariant implies $markedListR(Stack, p) * \mathsf{T}$. Using such a fact and Lemma 77, we show that the precondition of the global Hoare triple implies that of the Hoare triple which we obtained using the Frame Rule and Auxiliary Variable Elimination:

$(\exists Stack{:}\,\mathsf{list}.\ preLoopInvR(Stack, p, r, STree, root))$

$\wedge\ (\exists p_l, p_r{:}\,\mathsf{loc}.\ (p \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}))$

$\wedge\ (r = \mathsf{nil} \vee (r \mapsto ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true})$

$\Longrightarrow$

$\exists Stack{:}\,\mathsf{list}, p_l, p_r{:}\,\mathsf{loc}.$

$preLoopInvR(Stack, p, r, STree, root)$

$\wedge\ (p \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})$

$\wedge\ (r = \mathsf{nil} \vee (r \mapsto ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true})$

$\Longrightarrow$ $preLoopInvR(Stack, p, r, STree, root) \models markedListR(Stack, p) * \mathsf{T}$ and Lemma 80

$\exists Stack, Stack'{:}\,\mathsf{list}, p_l, p_r, p_0, r_0{:}\,\mathsf{loc}.$

$preLoopInvR(Stack, p, r, STree, root))$

$\wedge\ (p \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})$

$\wedge\ (r = \mathsf{nil} \vee (r \mapsto ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true})$

$\wedge\ (Stack = (p_l, p_r, \mathsf{true}, \mathsf{true}) :: Stack') \wedge (p = p_0) \wedge (r = r_0)$

$\Longrightarrow$

$\exists p_l, p_r, p_0, r_0{:}\,\mathsf{loc}, Stack'{:}\,\mathsf{list}.$

$preLoopInvR((p_l, p_r, \mathsf{true}, \mathsf{true}) :: Stack', p_0, r_0, STree, root))$

$\wedge\ (p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})$

$\wedge\ (r_0 = \mathsf{nil} \vee (r_0 \mapsto ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true})$

$\wedge\ (p = p_0) \wedge (r = r_0)$

$\Longrightarrow$ Lemma 77

$\exists Stack{:}\,\mathsf{list}, p_0, r_0, p_l, p_r{:}\,\mathsf{loc}.$

$(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \ preLoopInvR(Stack', p_r, p_0, STree, root))$

$\wedge\ (r = \mathsf{nil} \vee rm = \mathsf{true})$

$\wedge\ (p = p_0) \wedge (r = r_0)$

$\Longrightarrow$ $(r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0)$ is pure

$\exists p_l, p_r, p_0, r_0{:}\,\mathsf{loc}, Stack'{:}\,\mathsf{list}.$

$$\begin{pmatrix} (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \wedge\ (r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0) \end{pmatrix} * \begin{pmatrix} (p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!*\ preLoopInvR(Stack', p_r, p_0, STree, root) \end{pmatrix}$$

#### 7.3.2.2 The second case

In this case, the following global Hoare triple needs to be shown:

$$\left\{\begin{array}{l}(\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)) \\ \wedge\, (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true}) \\ \wedge\, (\exists p_l, p_r\colon \mathsf{loc}.\, (p \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}))\end{array}\right\}$$

$$\mathsf{body}$$

$$\{\exists Stack\colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)\}$$

The verification follows the same pattern as in the first case. We first prove a local Hoare triple which exactly and only describes what body does in this case.

$\{(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) \wedge (r = \mathsf{nil} \vee (rm = \mathsf{true})) \wedge (p = p_0) \wedge (r = r_0)\}$

if $\neg(r \neq \mathsf{nil} \wedge \neg(rm))$

then $pc := p.check;$

  if $(pc)$

  then

    $\{\mathsf{F}\}$

    $q := r; r := p; p := p.right; r.right := q$

  else

    $\{(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0)\}$

    $q := r; r := p.right; pl := p.left; p.right := pl; p.left := q; p.check := \mathsf{true}$

else

  $\{\mathsf{F}\}$

  $q := p; p := r; r := r.left; p.left := q; p.mark := \mathsf{true}; p.check := \mathsf{false}$

$\{(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \wedge (p = p_0) \wedge (r = p_r)\}$

The local Hoare triple, which is proved above, only states how one heap cell and two variables are changed. In order to change the mode of reasoning from local to global, we incorporate other portions of heap storage to the local Hoare triple by the Frame Rule:

$$\dfrac{\left\{\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\ \wedge (r = \mathsf{nil} \vee rm = \mathsf{true})\\ \wedge (p = p_0) \wedge (r = r_0)\end{array}\right\} \mathsf{body}\ \{(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \wedge (p = p_0) \wedge (r = p_r)\}}{\left\{\begin{pmatrix}(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\ \wedge (r = \mathsf{nil} \vee rm = \mathsf{true})\\ \wedge (p = p_0) \wedge (r = r_0)\end{pmatrix} * \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{false}) :: \mathit{Stack}', p_0, p_r, \mathit{STree}, \mathit{root})\end{pmatrix}\right\}}\ \mathrm{FR}$$

$$\mathsf{body}$$

$$\left\{\begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ \wedge (p = p_0) \wedge (r = p_r)\end{pmatrix} * \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{false}) :: \mathit{Stack}', p_0, p_r, \mathit{STree}, \mathit{root})\end{pmatrix}\right\}$$

And the auxiliary variables are eliminated by applying Auxiliary Variable Elimination.

$$\dfrac{\vdots}{\left\{\begin{array}{l}\exists p_l, p_r, p_0, r_0 \colon \mathsf{loc}, \mathit{Stack}' \colon \mathsf{list}.\\ \begin{pmatrix}(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\ \wedge (r = \mathsf{nil} \vee rm = \mathsf{true})\\ \wedge (p = p_0) \wedge (r = r_0)\end{pmatrix} * \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{false}) :: \mathit{Stack}', p_0, p_r, \mathit{STree}, \mathit{root})\end{pmatrix}\end{array}\right\}}\ \mathrm{AVE}$$

$$\mathsf{body}$$

$$\left\{\begin{array}{l}\exists p_l, p_r, p_0, r_0 \colon \mathsf{loc}, \mathit{Stack}' \colon \mathsf{list}.\\ \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ \wedge (p = p_0) \wedge (r = p_r)\end{pmatrix} * \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{false}) :: \mathit{Stack}', p_0, p_r, \mathit{STree}, \mathit{root})\end{pmatrix}\end{array}\right\}$$

Finally, we derive the desired global Hoare triple from the above using Consequence. In the following, we show that the postcondition of the above Hoare triple implies that of the global one:

$$\begin{array}{l}\exists p_l, p_r, p_0, r_0 \colon \mathsf{loc}, \mathit{Stack}' \colon \mathsf{list}.\\ \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ \wedge (p = p_0) \wedge (r = p_r)\end{pmatrix} * \begin{pmatrix}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{false}) :: \mathit{Stack}', p_0, p_r, \mathit{STree}, \mathit{root})\end{pmatrix}\end{array}$$

$\Longrightarrow (p = p_0) \wedge (r = p_r)$ is pure

$$\begin{array}{l}\exists p_l, p_r, p_0, r_0 \colon \mathsf{loc}, \mathit{Stack}' \colon \mathsf{list}.\\ (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{false}) :: \mathit{Stack}', p_0, p_r, \mathit{STree}, \mathit{root}))\\ \wedge (p = p_0) \wedge (r = p_r)\end{array}$$

$\Longrightarrow P * (P -\!\!* Q) \models Q$

$\exists p_l, p_r, p_0, r_0 \colon \mathsf{loc}, Stack' \colon \mathsf{list}.\ \mathit{preLoopInvR}((p_0, r_0, p_l, \mathsf{false}) :: Stack', p_0, p_r, STree, root) \wedge (p = p_0) \wedge (r = p_r)$

$\Longrightarrow$

$\exists Stack \colon \mathsf{list}.\ \mathit{preLoopInvR}(Stack, p, r, STree, root)$

The implication between preconditions is shown in the following, where we use Lemma 78 and Lemma 80:

$(\exists Stack \colon \mathsf{list}. \mathit{preLoopInvR}(Stack, p, r, STree, root))$
$\wedge (\exists p_l, p_r \colon \mathsf{loc}.\ (p \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}))$
$\wedge (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true})$

$\Longrightarrow$

$\exists Stack \colon \mathsf{list}, p_l, p_r \colon \mathsf{loc}.$
$\mathit{preLoopInvR}(Stack, p, r, STree, root)$
$\wedge (p \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true})$
$\wedge (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true})$

$\Longrightarrow$ $\mathit{preLoopInvR}(Stack, p, r, STree, root) \models \mathit{restoredListR}(Stack, p) * \mathsf{T}$ and Lemma 80

$\exists Stack, Stack' \colon \mathsf{list}, p_l, p_r, p_0, T \colon \mathsf{loc}.$
$\mathit{preLoopInvR}(Stack, p, r, STree, root)$
$\wedge (p \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true})$
$\wedge (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true})$
$\wedge (Stack = (p, p_l, p_r, \mathsf{false}) :: Stack') \wedge (p = p_0) \wedge (r = r_0)$

$\Longrightarrow$

$\exists p_l, p_r, p_0, r_0 \colon \mathsf{loc}, Stack' \colon \mathsf{list}.$
$\mathit{preLoopInvR}((p_0, p_l, p_r, \mathsf{false}) :: Stack', p_0, r_0, STree, root)$
$\wedge (p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true})$
$\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}) \wedge rm = \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0)$

$\Longrightarrow$ Lemma 78

$$\begin{array}{l}\exists p_l, p_r, p_0, r_0 \colon \mathsf{loc}, Stack' \colon \mathsf{list}.\\ (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * \left(\begin{array}{l}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \; preLoopInvR((p_0, r_0, p_l, \mathsf{false}) :: Stack', p_0, r_0, STree, root)\end{array}\right)\\ \wedge (r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0)\end{array}$$

$\Longrightarrow (r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p = p_0) \wedge (r = r_0)$ is pure

$$\begin{array}{l}\exists p_0, p_l, p_r, T \colon \mathsf{loc}, Stack' \colon \mathsf{list}.\\ \left(\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\ \wedge (p = p_0) \wedge (r = r_0)\\ \wedge (r = \mathsf{nil} \vee rm = \mathsf{true})\end{array}\right) * \left(\begin{array}{l}(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true})\\ -\!\!* \; preLoopInvR((p_0, r_0, p_l, \mathsf{false}) :: Stack', p_0, r_0, STree, root)\end{array}\right)\end{array}$$

#### 7.3.2.3 The third case

The final case is when $r$ denotes an unmarked allocated heap cell. The global Hoare triple, which needs to be verified in this case, is the following:

$$\left\{\begin{array}{l}(\exists Stack \colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root))\\ \wedge (rm = \mathsf{false}) \wedge (r \hookrightarrow ?, ?, ?, \mathsf{false})\end{array}\right\}$$
$$\mathsf{body}$$
$$\{\exists Stack \colon \mathsf{list}.\, preLoopInvR(Stack, p, r, STree, root)\}$$

As in previous two cases, we start by showing that a local Hoare triple is valid, which precisely describes what $\mathsf{body}$ does when $r$ denotes an unmarked allocated heap cell.

$$
\begin{array}{l}
\{(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \wedge (p = p_0) \wedge (r = r_0) \wedge (rm = \mathsf{false})\} \\
\mathsf{if}\ \neg(r \neq \mathsf{nil} \wedge \neg(rm)) \\
\mathsf{then}\ pc := p.check; \\
\quad \mathsf{if}\ (pc) \\
\quad \mathsf{then} \\
\quad\quad \{\mathsf{F}\} \\
\quad\quad q := r; r := p; p := p.right; r.right := q \\
\quad \mathsf{else} \\
\quad\quad \{\mathsf{F}\} \\
\quad\quad q := r; r := p.right; pl := p.left; p.right := pl; p.left := q; p.check := \mathsf{true} \\
\mathsf{else} \\
\quad \{(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \wedge (p = p_0) \wedge (r = r_0)\} \\
\quad q := p; p := r; r := r.left; p.left := q; p.mark := \mathsf{true}; p.check := \mathsf{false} \\
\{(r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \wedge (p = r_0) \wedge (r = r_l)\}
\end{array}
$$

Changing the mode of reasoning from local to global follows the same pattern as in the previous two cases: we apply the Frame Rule so as to incorporate other portions of heap storage and eliminate auxiliary variables by Auxiliary Variable Elimination.

$$
\dfrac{\left\{\begin{array}{l}(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \\ \wedge (p = p_0) \wedge (r = r_0) \wedge (rm = \mathsf{false})\end{array}\right\} \mathsf{body}\ \{(r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \wedge (p = r_0) \wedge (r = r_l)\}}{\left\{\begin{array}{l}\exists p_0, r_0, r_l, r_r \colon \mathsf{loc}, r_c \colon \mathsf{bool}, Stack \colon \mathsf{list}. \\ ((r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \wedge (p = p_0) \wedge (r = r_0) \wedge (rm = \mathsf{false})) \\ * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \ preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root))\end{array}\right\}} \ \mathrm{FR, AVE}
$$

$$
\mathsf{body}
$$

$$
\left\{\begin{array}{l}\exists p_0, r_0, r_l, r_r \colon \mathsf{loc}, Stack \colon \mathsf{list}. \\ ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \wedge (p = r_0) \wedge (r = r_l)) \\ * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \ preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root))\end{array}\right\}
$$

It remains to show implications between certain assertions to obtain the global Hoare triple from the above one by Consequence. The following shows the implication from the postcondition in the above Hoare triple to the postcondition of the global one:

$\exists p_0, r_0, r_l, r_r\colon \mathsf{loc}, Stack\colon \mathsf{list}.$
$((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \wedge (p = r_0) \wedge (r = r_l))$
$* ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \mathrel{-\!\!*} preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root))$

$\Longrightarrow (p = r_0) \wedge (r = r_l)$ is pure

$\exists p_0, r_0, r_l, r_r\colon \mathsf{loc}, Stack\colon \mathsf{list}.$
$(r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) * \begin{pmatrix} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \\ \mathrel{-\!\!*} preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root) \end{pmatrix}$
$\wedge (p = r_0) \wedge (r = r_l)$

$\Longrightarrow P * (P \mathrel{-\!\!*} Q) \models Q$

$\exists p_0, r_0, r_l, r_r\colon \mathsf{loc}, Stack, Stack'\colon \mathsf{list}.$
$preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root)$
$\wedge (p = r_0) \wedge (r = r_l) \wedge (Stack' = (r_0, p_0, r_r, \mathsf{false}) :: Stack)$

$\Longrightarrow$

$\exists Stack'\colon \mathsf{list}.\ preLoopInvR(Stack', p, r, STree, root)$

Finally, the following shows that the precondition in the global Hoare triple implies the precondition in the Hoare triple which we just obtained using the Frame Rule and Auxiliary Variable Elimination. We use Lemma 79 in the proof.

$(\exists Stack\colon \mathsf{list}.\ preLoopInvR(Stack, p, r, STree, root))$
$\wedge (rm = \mathsf{false}) \wedge (\exists r_l, r_r\colon \mathsf{loc}, r_c\colon \mathsf{bool}.\ (r \hookrightarrow r_l, r_r, r_c, \mathsf{false}))$

$\Longrightarrow$

$\exists Stack\colon \mathsf{list}, r_l, r_r\colon \mathsf{loc}, r_c\colon \mathsf{bool}.$
$preLoopInvR(Stack, p, r, STree, root)$
$\wedge (rm = \mathsf{false}) \wedge (r \hookrightarrow r_l, r_r, r_c, \mathsf{false})$

$\Longrightarrow$

$\exists Stack\colon \mathsf{list}, r_l, r_r, r_0, p_0\colon \mathsf{loc}, r_c\colon \mathsf{bool}.$
$preLoopInvR(Stack, p_0, r_0, STree, root)$
$\wedge (rm = \mathsf{false}) \wedge (r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{false})$
$\wedge (p = p_0) \wedge (r = r_0)$

$\Longrightarrow$ Lemma 79

$$\begin{array}{l}\exists r_l, r_r, r_0, p_0 \colon \mathsf{loc}, r_c \colon \mathsf{bool}, Stack \colon \mathsf{list}.\\ (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * \left(\begin{array}{l}(r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true})\\ -\!\!* \; preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root)\end{array}\right)\\ \wedge (rm = \mathsf{false}) \wedge (p = p_0) \wedge (r = r_0)\end{array}$$

$\Longrightarrow$ $(rm = \mathsf{false}) \wedge (p = p_0) \wedge (r = r_0)$ is pure

$$\begin{array}{l}\exists r_l, r_r, r_0, p_0 \colon \mathsf{loc}, r_c \colon \mathsf{bool}, Stack \colon \mathsf{list}.\\ ((r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \wedge (p = p_0) \wedge (r = r_0) \wedge (rm = \mathsf{false}))\\ * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root))\end{array}$$

## 7.4 Proofs of Lemmas

This section contains proofs of lemmas in Section 7.3.2. The proofs use properties of *strictly exact* assertions. An assertion $P$ is *strictly exact* iff for any stack $s$, there exists at most one heap $h$ such that $s, h \models P$. We use a conservative syntactic criteria to distinguish such assertions, which is to check whether an assertion only contains $\mapsto$ and $*$. When a strictly exact assertion is used, $*$ distributes over $\wedge$: for any strictly exact assertion $P$,

$$(P * Q) \wedge (P * R) \models P * (Q \wedge R)$$

The following lemma summarizes certain facts about assertions which we use in the proofs.

**Lemma 81** *The following implications hold.*

1. $(x \mapsto l, r, c, \mathsf{true}) * markedR$
   $\models markedR$

2. $(x \hookrightarrow l, r, c, m) \wedge noDanglingR$
   $\models noDangling(l) \wedge noDangling(r)$

3. $(x \hookrightarrow l, r, c, m) \wedge noDangling(y) \wedge noDanglingR$
   $\models (x \mapsto l, r, c, m) * ((x \mapsto l, y, c, m) \vee (x \mapsto y, r, c, m) -\!\!* \; noDanglingR)$

4. $(x \hookrightarrow l, r, c, m) \wedge noDangling(y)$
   $\models (x \mapsto l, r, c, m) * ((x \mapsto l', r', c', m') -\!\!* \; noDangling(y))$

5. $(x \hookrightarrow l, r, c, m)$
$$\models (x \mapsto l, r, c, m) * ((x \mapsto l', r', c', m') \mathrel{-\!\!*} noDangling(x))$$

6. $((x = \mathsf{nil}) \vee (x \hookrightarrow ?, ?, ?, \mathsf{true}))$
$$\wedge\ markedR * (unmarkedR \wedge (\forall y{:}\,\mathsf{loc}.\ allocated(y) \Rightarrow reach(x, y) \vee P))$$
$$\models markedR * (unmarkedR \wedge (\forall y{:}\,\mathsf{loc}.\ allocated(y) \Rightarrow P))$$

7. $(x \hookrightarrow l, r, c, \mathsf{true}) \wedge markedR * (unmarkedR \wedge P)$
$$\models (x \mapsto l, r, c, \mathsf{true}) * markedR * (unmarkedR \wedge P)$$

8. $(x \hookrightarrow l, r, c, \mathsf{false})$
$$\wedge\ markedR * \begin{pmatrix} unmarkedR \\ \wedge\ (\forall y{:}\,\mathsf{loc}.allocated(y) \Rightarrow reach(x, y) \vee reachRightChildInList(lst, y)) \end{pmatrix}$$
$$\models markedR * (x \mapsto l, r, c, \mathsf{false})$$
$$* \begin{pmatrix} unmarkedR \\ \wedge\ (\forall y{:}\,\mathsf{loc}.allocated(y) \Rightarrow reach(l, y) \vee reachI(r, y) \vee reachRightChildInList(lst, y)) \end{pmatrix}$$

9. $(x \hookrightarrow l, r, c, \mathsf{false}) \wedge markedListR(lst, y) * (restoredListR(lst, x) \mathrel{-\!\!*} spansR(stree, root))$
$$\models markedListR(lst, y) * (x \mapsto l, r, c, \mathsf{false})$$
$$* ((x \mapsto l, r, \mathsf{true}, \mathsf{true}) * restoredListR(lst, x) \mathrel{-\!\!*} spansR(stree, root))$$

10. $markedListR(lst, x) * \mathsf{T} \wedge (x \hookrightarrow l, r, c, m) \models m = \mathsf{true}$

We refer to each fact in the lemma by its number. For instance, Rule 1 indicates the implication $(x \mapsto l, r, c, \mathsf{true}) * markedR \models markedR$.

### 7.4.1 Proof of Lemma 76

Since for any state satisfying $Cond$ either $((r = \mathsf{nil}) \vee (rm = \mathsf{true})) \wedge (p \neq \mathsf{nil})$ or $(r \neq \mathsf{nil}) \wedge (rm \neq \mathsf{true})$ holds, it suffices to show the following two implications:

$$\begin{pmatrix} (\exists Stack{:}\,\mathsf{list}.\ preLoopInvR(Stack, p, r, STree, root)) \\ \wedge\ (r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p \neq \mathsf{nil}) \end{pmatrix}$$
$$\Longrightarrow \begin{pmatrix} ((r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p \hookrightarrow ?, ?, \mathsf{true}, \mathsf{true})) \\ \vee\ ((r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p \hookrightarrow ?, ?, \mathsf{false}, \mathsf{true})) \end{pmatrix}$$

$$\begin{pmatrix} (r \neq \mathsf{nil}) \wedge (rm \neq \mathsf{true}) \\ \wedge\ (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm)) \end{pmatrix} \Longrightarrow \Big((rm = \mathsf{false}) \wedge (r \hookrightarrow ?, ?, ?, \mathsf{false})\Big)$$

The following sequence of implications proves the first implication.

$$((r = \mathsf{nil}) \vee (rm = \mathsf{true})) \wedge (p \neq \mathsf{nil}) \wedge (\exists Stack{:}\,\mathsf{list}.preLoopInvR(Stack, p, r, STree, root))$$

$\Longrightarrow$ $preLoopInvR(Stack, p, r, STree, root) \models noDangling(p)$ and $noDangling(p) \triangleq (p = \mathsf{nil}) \vee (p \hookrightarrow ?, ?, ?, ?)$

$$((r = \mathsf{nil}) \vee (rm = \mathsf{true})) \wedge (\exists Stack{:}\,\mathsf{list}.preLoopInvR(Stack, p, r, STree, root)) \wedge (p \hookrightarrow ?, ?, ?, ?)$$

$\Longrightarrow$ $preLoopInvR(Stack, p, r, STree, root) \models markedListR(Stack, p) * \mathsf{T}$

$((r = \mathsf{nil}) \vee (rm = \mathsf{true})) \wedge markedListR(Stack, p) * \mathsf{T} \wedge (p \hookrightarrow ?, ?, ?, ?)$

$\Longrightarrow$ Rule 10 in Lemma 81, which is $markedListR(Lst, p) * \mathsf{T} \wedge (p \hookrightarrow ?, ?, ?, b_m) \models (b_m = \mathsf{true})$

$((r = \mathsf{nil}) \vee (rm = \mathsf{true})) \wedge (p \hookrightarrow ?, ?, ?, \mathsf{true})$

$\Longrightarrow$ $(p \hookrightarrow ?, ?, b_c, ?) \models (b_c = \mathsf{true}) \vee (b_c = \mathsf{false})$

$((r = \mathsf{nil}) \vee (rm = \mathsf{true})) \wedge ((p \hookrightarrow ?, ?, \mathsf{true}, \mathsf{true}) \vee (p \hookrightarrow ?, ?, \mathsf{false}, \mathsf{true}))$

$\Longrightarrow$

$((r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p \hookrightarrow ?, ?, \mathsf{true}, \mathsf{true})) \vee ((r = \mathsf{nil} \vee rm = \mathsf{true}) \wedge (p \hookrightarrow ?, ?, \mathsf{false}, \mathsf{true}))$

And the following sequence of implications proves the second implication.

$(r \neq \mathsf{nil}) \wedge (rm \neq \mathsf{true}) \wedge (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm))$

$\Longrightarrow$ $\mathsf{T} \models (rm = \mathsf{true}) \vee (rm = \mathsf{false})$

$(r \neq \mathsf{nil}) \wedge (rm = \mathsf{false}) \wedge (r = \mathsf{nil} \vee (r \hookrightarrow ?, ?, ?, rm))$

$\Longrightarrow$

$(rm = \mathsf{false}) \wedge (r \hookrightarrow ?, ?, ?, \mathsf{false})$

### 7.4.2 Proof of Lemma 77

The following sequence of implications shows the lemma:

$$
\begin{aligned}
&preLoopInvR((p_0, p_l, p_r, \mathsf{true}) :: Stack, p_0, r_0, STree, root)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))\\
&\wedge (p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})
\end{aligned}
$$

$\Longrightarrow$ Definition of $preLoopInvR$

$$
\begin{aligned}
&noDanglingR \wedge noDangling(p_0) \wedge noDangling(r_0)\\
&\wedge markedListR((p_0, p_l, p_r, \mathsf{true}) :: Stack, p_0)\\
&\quad * (restoredListR((p_0, p_l, p_r, \mathsf{true}) :: Stack, r_0) \mathrel{-\!\!*} spansR(STree, root))\\
&\wedge markedR\\
&\quad * \left(\begin{array}{l} unmarkedR \\ \wedge \forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow \left(\begin{array}{l} reach(r_0, x) \\ \vee\ reachRightChildInList((p_0, p_l, p_r, \mathsf{true}) :: Stack, x) \end{array}\right) \end{array}\right)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))\\
&\wedge (p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})
\end{aligned}
$$

$\Longrightarrow$ Rule 2 in Lemma 81, which is $(p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge noDanglingR \models noDangling(p_r)$

$$
\begin{aligned}
& noDanglingR \wedge noDanglingR(T) \wedge noDanglingR(p_r) \wedge noDangling(p_0) \\
& \wedge\, markedListR((p_0, p_l, p_r, \mathsf{true}) :: Stack, p_0) \\
& \quad *\, (restoredListR((p_0, p_l, p_r, \mathsf{true}) :: Stack, r_0) \mathrel{-\!*} spansR(STree, root)) \\
& \wedge\, markedR \\
& \quad * \begin{pmatrix} unmarkedR \\ \wedge\, \forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow \begin{pmatrix} reach(r_0, x) \\ \vee\, reachRightChildInList((p_0, p_l, p_r, \mathsf{true}) :: Stack, x) \end{pmatrix} \end{pmatrix} \\
& \wedge\, (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true})) \\
& \wedge\, (p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true})
\end{aligned}
$$

$\Longrightarrow$ Rules 3,4,5 in Lemma 81, which are

$$
\left[\begin{array}{l}
(p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge noDangling(r_0) \wedge noDanglingR \\
\qquad \models (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!*} noDanglingR) \\
(p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge noDangling(p_r) \\
\quad \models (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!*} noDangling(p_r)) \\
(p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}) \\
\quad \models (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!*} noDangling(p_0))
\end{array}\right]
$$

$$
\begin{aligned}
& (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!*} noDanglingR) \\
& \wedge\, (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!*} noDangling(p_r)) \\
& \wedge\, (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!*} noDangling(p_0)) \\
& \wedge\, markedListR((p_0, p_l, p_r, \mathsf{true}) :: Stack, p_0) \\
& \quad *\, (restoredListR((p_0, p_l, p_r, \mathsf{true}) :: Stack, r_0) \mathrel{-\!*} spansR(STree, root)) \\
& \wedge\, markedR \\
& \quad * \begin{pmatrix} unmarkedR \\ \wedge\, \forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow \begin{pmatrix} reach(r_0, x) \\ \vee\, reachRightChildInList((p_0, p_l, p_r, \mathsf{true}) :: Stack, x) \end{pmatrix} \end{pmatrix} \\
& \wedge\, (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{aligned}
$$

$\Longrightarrow$ Definitions of $markedListR, restoredListR, reachRightChildInList$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_r))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_0))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * markedListR(Stack, p_r)\\
&\quad * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) * restoredListR(Stack, p_0) \mathrel{-\!\!*} spansR(STree, root))\\
&\wedge markedR\\
&\quad * \left( \begin{aligned} &unmarkedR \\ &\wedge \forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow \left( \begin{aligned} &reach(r_0, x) \\ &\vee reach(p_r, x) \\ &\vee reachRightChildInList((p_0, p_l, p_r, \mathsf{true}) :: Stack, x) \end{aligned} \right) \end{aligned} \right)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{aligned}
$$

$\Longrightarrow P * (Q * R \mathrel{-\!\!*} S) \models Q \mathrel{-\!\!*} (P * (R \mathrel{-\!\!*} S))$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_r))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_0))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true})\\
&\quad * \left( \begin{aligned} &(p_0, p_l, r_0, \mathsf{true}, \mathsf{true}) \\ &\mathrel{-\!\!*} markedListR(Stack, p_r) * (restoredListR(Stack, p_0) \mathrel{-\!\!*} spansR(STree, root)) \end{aligned} \right)\\
&\wedge markedR\\
&\quad * \left( \begin{aligned} &unmarkedR \\ &\wedge \forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow \left( \begin{aligned} &reach(r_0, x) \\ &\vee reach(p_r, x) \\ &\vee reachRightChildInList((p_0, p_l, p_r, \mathsf{true}) :: Stack, x) \end{aligned} \right) \end{aligned} \right)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{aligned}
$$

$\Longrightarrow$ Rule 6 in Lemma 81, which is

$$
\left[ \begin{aligned} &((r_0 = \mathsf{nil}) \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true})) \\ &\wedge markedR * (unmarkedR \wedge (\forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow reach(r_0, x) \vee P)) \\ &\qquad \models markedR * (unmarkedR \wedge (\forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow P)) \end{aligned} \right]
$$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_r))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_0))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true})\\
&\quad * \left( \begin{aligned} &(p_0, p_l, r_0, \mathsf{true}, \mathsf{true}) \\ &\mathrel{-\!\!*} markedListR(Stack, p_r) * (restoredListR(Stack, p_0) \mathrel{-\!\!*} spansR(STree, root)) \end{aligned} \right)\\
&\wedge markedR\\
&\quad * (unmarkedR \wedge (\forall x{:}\mathsf{loc}.\ allocated(x) \Rightarrow reach(p_r, x) \vee reachRightChildInList(Stack, x)))\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{aligned}
$$

$\Longrightarrow$ Rule 7 in Lemma 81, which is

$$\left[\begin{array}{l}(p_0 \hookrightarrow p_l, p_r, \mathsf{true}, \mathsf{true}) \wedge \mathit{markedR} * (\mathit{unmarkedR} \wedge P) \\ \qquad \models (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * \mathit{markedR} * (\mathit{unmarkedR} \wedge P))\end{array}\right]$$

$$\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDanglingR}) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_r)) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_0)) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \quad * \left(\begin{array}{l}(p_0, p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!* \mathit{markedListR}(\mathit{Stack}, p_r) * (\mathit{restoredListR}(\mathit{Stack}, p_0) -\!\!* \mathit{spansR}(\mathit{STree}, \mathit{root}))\end{array}\right) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * \mathit{markedR} \\ \quad * \left(\begin{array}{l}\mathit{unmarkedR} \\ \wedge \forall x{:}\,\mathsf{loc}.\ \mathit{allocated}(x) \Rightarrow \mathit{reach}(p_r, x) \vee \mathit{reachRightChildInList}(\mathit{Stack}, x)\end{array}\right)\end{array}$$

$\Longrightarrow P \models Q -\!\!* (Q * P)$

$$\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDanglingR}) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_r)) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_0)) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \quad * \left(\begin{array}{l}(p_0, p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!* \mathit{markedListR}(\mathit{Stack}, p_r) * (\mathit{restoredListR}(\mathit{Stack}, p_0) -\!\!* \mathit{spansR}(\mathit{STree}, \mathit{root}))\end{array}\right) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \quad * \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!* \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) * \mathit{markedR} \\ * \left(\begin{array}{l}\mathit{unmarkedR} \\ \wedge \forall x{:}\,\mathsf{loc}.\ \mathit{allocated}(x) \Rightarrow \mathit{reach}(p_r, x) \vee \mathit{reachRightChildInList}(\mathit{Stack}, x)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}$$

$\Longrightarrow$ Rule 1 in Lemma 81, which is $(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) * \mathit{markedR} \models \mathit{markedR}$

$$\begin{array}{l}(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDanglingR}) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_r)) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_0)) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \quad * \left(\begin{array}{l}(p_0, p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!* \mathit{markedListR}(\mathit{Stack}, p_r) * (\mathit{restoredListR}(\mathit{Stack}, p_0) -\!\!* \mathit{spansR}(\mathit{STree}, \mathit{root}))\end{array}\right) \\ \wedge (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) \\ \quad * \left(\begin{array}{l}(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) \\ -\!\!* \left(\begin{array}{l}\mathit{markedR} \\ * \left(\begin{array}{l}\mathit{unmarkedR} \\ \wedge \forall x{:}\,\mathsf{loc}.\ \mathit{allocated}(x) \Rightarrow \mathit{reach}(p_r, x) \vee \mathit{reachRightChildInList}(\mathit{Stack}, x)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}$$

$\Longrightarrow (p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true})$ is strictly exact

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true})\\
&* \left( \begin{array}{l}
(p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true})\\
\;-\!\!* \left( \begin{array}{l}
noDanglingR \wedge noDangling(p_r) \wedge noDangling(p_0)\\
\wedge\, markedListR(Stack, p_r)\\
\quad * (restoredListR(Stack, p_0) -\!\!* spansR(STree, root))\\
\wedge\, markedR\\
\quad * \left( \begin{array}{l} unmarkedR\\ \wedge\, \forall x{:}\,\mathsf{loc}.\ allocated(x) \Rightarrow reach(p_r, x) \vee reachRightChildInList(Stack, x) \end{array} \right)
\end{array} \right)
\end{array} \right)
\end{aligned}
$$

$\Longrightarrow$ Definition of *preLoopInvR*

$$(p_0 \mapsto p_l, p_r, \mathsf{true}, \mathsf{true}) * ((p_0 \mapsto p_l, r_0, \mathsf{true}, \mathsf{true}) -\!\!* preLoopInvR(Stack, p_r, p_0, STree, root))$$

### 7.4.3 Proof of Lemma 78

The following sequence of implications shows the lemma:

$$
\begin{aligned}
&preLoopInvR((p_0, p_l, p_r, \mathsf{false}) :: Stack, p_0, r_0, STree, root)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))\\
&\wedge (p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true})
\end{aligned}
$$

$\Longrightarrow$ Definition of *preLoopInvR*

$$
\begin{aligned}
&noDanglingR \wedge noDangling(p_0) \wedge noDangling(r_0)\\
&\wedge markedListR((p_0, p_l, p_r, \mathsf{false}) :: Stack, p_0)\\
&\quad * (restoredListR((p_0, p_l, p_r, \mathsf{false}) :: Stack, r_0) -\!\!* spansR(STree, root))\\
&\wedge markedR\\
&\quad * \left( \begin{array}{l} unmarkedR\\ \wedge\, \forall x{:}\,\mathsf{loc}.allocatedI(x) \Rightarrow \left( \begin{array}{l} reach(r_0, x)\\ \vee\, reachRightChildInList((p_0, p_l, p_r, \mathsf{false}) :: Stack, x) \end{array} \right) \end{array} \right)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))\\
&\wedge (p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true})
\end{aligned}
$$

$\Longrightarrow$ Rule 2 in Lemma 81, which is $(p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}) \wedge noDanglingR \models noDangling(p_r)$

$$
\begin{aligned}
&noDanglingR \wedge noDangling(p_0) \wedge noDangling(r_0) \wedge noDanglingR(p_r)\\
&\wedge markedListR((p_0, p_l, p_r, \mathsf{false}) :: Stack, p_0)\\
&\quad * (restoredListR((p_0, p_l, p_r, \mathsf{false}) :: Stack, r_0) -\!\!* spansR(STree, root))\\
&\wedge markedR\\
&\quad * \left( \begin{array}{l} unmarkedR\\ \wedge\, \forall x{:}\,\mathsf{loc}.allocatedI(x) \Rightarrow \left( \begin{array}{l} reach(r_0, x)\\ \vee\, reachRightChildInList((p_0, p_l, p_r, \mathsf{false}) :: Stack, x) \end{array} \right) \end{array} \right)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{aligned}
$$

$\Longrightarrow$ Rules 3,4,5 in Lemma 81, which are

$$\left[\begin{array}{l}
(p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}) \wedge noDanglingR \wedge noDangling(r_0) \\
\qquad \models (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDanglingR) \\
(p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}) \wedge noDangling(p_r) \\
\qquad \models (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDangling(p_r)) \\
(p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}) \\
\qquad \models ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDangling(p_0))
\end{array}\right]$$

$$\begin{array}{l}
(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDanglingR) \\
\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDangling(p_0)) \\
\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDangling(p_r)) \\
\wedge\, markedListR((p_0, p_l, p_r, \mathsf{false}) :: Stack, p_0) \\
\quad * (restoredListR((p_0, p_l, p_r, \mathsf{false}) :: Stack, r_0) {-\!\!*}\, spansR(STree, root)) \\
\wedge\, markedR \\
\quad * \left(\begin{array}{l} unmarkedR \\ \wedge\, \forall x{:}\mathsf{loc}.allocatedI(x) \Rightarrow \left(\begin{array}{l} reach(r_0, x) \\ \vee\, reachRightChildInList((p_0, p_l, p_r, \mathsf{false}) :: Stack, x) \end{array}\right) \end{array}\right) \\
\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{array}$$

$\Longrightarrow$ Definitions of $markedListR$, $restoredListR$, $reachRightChildInList$

$$\begin{array}{l}
(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDanglingR) \\
\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDangling(p_0)) \\
\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) {-\!\!*}\, noDangling(p_r)) \\
\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * markedListR(Stack, p_l) \\
\quad * (restoredListR((p_0, r_0, p_l, \mathsf{true}) :: Stack, p_r) {-\!\!*}\, spansR(STree, root)) \\
\wedge\, markedR \\
\quad * \left(\begin{array}{l} unmarkedR \\ \wedge\, \forall x{:}\mathsf{loc}.allocatedI(x) \Rightarrow \left(\begin{array}{l} reach(r_0, x) \\ \vee reach(p_r, x) \vee reachRightChildInList(Stack, x) \end{array}\right) \end{array}\right) \\
\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{array}$$

$\Longrightarrow P \models Q {-\!\!*} (Q * P)$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_0))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_r))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\
&\quad * \left( \begin{array}{l} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ \mathrel{-\!\!*} \left( \begin{array}{l} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * markedListR(Stack, p_l) \\ * (restoredListR((p_0, r_0, p_l, \mathsf{true}) :: Stack, p_r) \mathrel{-\!\!*} spansR(STree, root)) \end{array} \right) \end{array} \right)\\
&\wedge markedR\\
&\quad * \left( \begin{array}{l} unmarkedR \\ \wedge \forall x\colon \mathsf{loc}.allocatedI(x) \Rightarrow \left( \begin{array}{l} reach(r_0, x) \\ \vee reach(p_r, x) \vee reachRightChildInList(Stack, x) \end{array} \right) \end{array} \right)\\
&\wedge (r_0 = \mathsf{nil} \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true}))
\end{aligned}
$$

$\Longrightarrow$ Rule 6 in Lemma 81, which is

$$
\left[ \begin{array}{l} ((r_0 = \mathsf{nil}) \vee (r_0 \hookrightarrow ?, ?, ?, \mathsf{true})) \\ \;\wedge markedR * (unmarkedR \wedge (\forall x\colon \mathsf{loc}.allocatedI(x) \Rightarrow reach(r_0, x) \vee P)) \\ \qquad \models \; \wedge markedR * (unmarkedR \wedge (\forall x\colon \mathsf{loc}.allocatedI(x) \Rightarrow P)) \end{array} \right]
$$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_0))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_r))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\
&\quad * \left( \begin{array}{l} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ \mathrel{-\!\!*} \left( \begin{array}{l} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * markedListR(Stack, p_l) \\ * (restoredListR((p_0, r_0, p_l, \mathsf{true}) :: Stack, p_r) \mathrel{-\!\!*} spansR(STree, root)) \end{array} \right) \end{array} \right)\\
&\wedge markedR\\
&\quad * \left( \begin{array}{l} unmarkedR \\ \wedge \forall x\colon \mathsf{loc}.allocatedI(x) \Rightarrow reach(p_r, x) \vee reachRightChildInList(Stack, x) \end{array} \right)
\end{aligned}
$$

$\Longrightarrow$ Rule 7 in Lemma 81, which is

$$
\left[ \begin{array}{l} (p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}) \wedge markedR * (unmarkedR \wedge P) \\ \qquad \models (p_0 \hookrightarrow p_l, p_r, \mathsf{false}, \mathsf{true}) * markedR * (unmarkedR \wedge P) \end{array} \right]
$$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_0))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \mathrel{-\!\!*} noDangling(p_r))\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})\\
&\quad * \left( \begin{array}{l} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ \mathrel{-\!\!*} \left( \begin{array}{l} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * markedListR(Stack, p_l) \\ * (restoredListR((p_0, r_0, p_l, \mathsf{true}) :: Stack, p_r) \mathrel{-\!\!*} spansR(STree, root)) \end{array} \right) \end{array} \right)\\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * markedR\\
&\qquad * \left( \begin{array}{l} unmarkedR \\ \wedge \forall x\colon \mathsf{loc}.allocatedI(x) \Rightarrow reach(p_r, x) \vee reachRightChildInList(Stack, x) \end{array} \right)
\end{aligned}
$$

$\Longrightarrow P \models Q -\!\!* (Q * P)$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDanglingR}) \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_0)) \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_r)) \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) \\
&\quad * \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ -\!\!* \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * \mathit{markedListR}(\mathit{Stack}, p_l) \\ * (\mathit{restoredListR}((p_0, r_0, p_l, \mathsf{true}) :: \mathit{Stack}, p_r) -\!\!* \mathit{spansR}(\mathit{STree}, \mathit{root})) \end{pmatrix} \end{pmatrix} \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) \\
&\quad * \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ -\!\!* \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * \mathit{markedR} \\ * \begin{pmatrix} \mathit{unmarkedR} \\ \wedge \forall x \colon \mathsf{loc}.\mathit{allocatedI}(x) \Rightarrow \mathit{reach}(p_r, x) \vee \mathit{reachRightChildInList}(\mathit{Stack}, x) \end{pmatrix} \end{pmatrix} \end{pmatrix}
\end{aligned}
$$

$\Longrightarrow$ Rule 1 in Lemma 81, which is $(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * \mathit{markedR} \models \mathit{markedR}$

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDanglingR}) \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_0)) \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) * ((p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) -\!\!* \mathit{noDangling}(p_r)) \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) \\
&\quad * \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ -\!\!* \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * \mathit{markedListR}(\mathit{Stack}, p_l) \\ * (\mathit{restoredListR}((p_0, r_0, p_l, \mathsf{true}) :: \mathit{Stack}, p_r) -\!\!* \mathit{spansR}(\mathit{STree}, \mathit{root})) \end{pmatrix} \end{pmatrix} \\
&\wedge (p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) \\
&\quad * \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ -\!\!* \begin{pmatrix} \mathit{markedR} \\ * \begin{pmatrix} \mathit{unmarkedR} \\ \wedge \forall x \colon \mathsf{loc}.\mathit{allocatedI}(x) \Rightarrow \mathit{reach}(p_r, x) \vee \mathit{reachRightChildInList}(\mathit{Stack}, x) \end{pmatrix} \end{pmatrix} \end{pmatrix}
\end{aligned}
$$

$\Longrightarrow$ $(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})$ is strictly exact

$$
\begin{aligned}
&(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true}) \\
&* \begin{pmatrix} (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\ -\!\!* \begin{pmatrix} \mathit{noDanglingR} \wedge \mathit{noDangling}(p_0) \wedge \mathit{noDangling}(p_r) \\ \wedge (p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) * \mathit{markedListR}(\mathit{Stack}, p_l) \\ \quad * (\mathit{restoredListR}((p_0, r_0, p_l, \mathsf{true}) :: \mathit{Stack}, p_r) -\!\!* \mathit{spansR}(\mathit{STree}, \mathit{root})) \\ \wedge \mathit{markedR} \\ \quad * \begin{pmatrix} \mathit{unmarkedR} \\ \wedge \forall x \colon \mathsf{loc}.\mathit{allocatedI}(x) \Rightarrow \mathit{reach}(p_r, x) \vee \mathit{reachRightChildInList}(\mathit{Stack}, x) \end{pmatrix} \end{pmatrix} \end{pmatrix}
\end{aligned}
$$

$\Longrightarrow$ $\vee$-Introduction and Definitions of $\mathit{markedListR}$ and $\mathit{reachRightChildInList}$

$$
(p_0 \mapsto p_l, p_r, \mathsf{false}, \mathsf{true})
* \left(
\begin{array}{l}
(p_0 \mapsto r_0, p_l, \mathsf{true}, \mathsf{true}) \\
-\!\!* \left(
\begin{array}{l}
noDanglingR \wedge noDangling(p_0) \wedge noDangling(p_r) \\
\wedge\ markedListR((p_0, r_0, p_l, \mathsf{true}) :: Stack, p_0) \\
\quad * (restoredListR((p_0, r_0, p_l, \mathsf{true}) :: Stack, p_r) -\!\!* spansR(STree, root)) \\
\wedge\ markedR \\
\quad * \left(
\begin{array}{l}
unmarkedR \\
\wedge\ \forall x{:}\,\mathsf{loc}. \\
\qquad allocatedI(x) \Rightarrow \left(
\begin{array}{l}
reach(p_r, x) \\
\vee\ reachRightChildInList((p_0, r_0, p_l, \mathsf{true}) :: Stack, x)
\end{array}
\right)
\end{array}
\right)
\end{array}
\right)
\end{array}
\right)
$$

$\Longrightarrow$ Definition of *preLoopInvR*

$$
preLoopInvR(Stack, p_0, p_r, STree, root)
$$

### 7.4.4 Proof of Lemma 79

The following sequence of implications shows the lemma:

$$
preLoopInvR(Stack, p_0, r_0, STree, root) \wedge (r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{true})
$$

$\Longrightarrow$ Definition of *preLoopInvR*

$$
\begin{array}{l}
noDanglingR \wedge noDangling(p_0) \wedge noDangling(r_0) \\
\wedge\ markedListR(Stack, p_0) * (restoredListR(Stack, p_0) -\!\!* spansR(STree, root)) \\
\wedge\ markedR \\
\quad * (unmarkedR \wedge (\forall x{:}\,\mathsf{loc}.allocated(x) \Rightarrow reach(r_0, x) \vee reachRightChildInList(Stack, x))) \\
\wedge\ (r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{true})
\end{array}
$$

$\Longrightarrow$ Rule 1 in Lemma 81, which is $(r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{false}) \wedge noDanglingR \models noDanglingR(r_l)$

$$
\begin{array}{l}
noDanglingR \wedge noDangling(p_0) \wedge noDangling(r_0) \wedge noDanglingR(r_l) \\
\wedge\ markedListR(Stack, p_0) * (restoredListR(Stack, p_0) -\!\!* spansR(STree, root)) \\
\wedge\ markedR \\
\quad * (unmarkedR \wedge (\forall x{:}\,\mathsf{loc}.allocated(x) \Rightarrow reach(r_0, x) \vee reachRightChildInList(Stack, x))) \\
\wedge\ (r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{true})
\end{array}
$$

$\Longrightarrow$ Rules 3 and 4 in Lemma 81, which are

$$\left[\begin{array}{l}(r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{false}) \wedge noDanglingR \wedge noDangling(p_0)\\ \qquad \models (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDanglingR)\\ (r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{false}) \wedge noDanglingR(r_l)\\ \qquad \models (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_l))\\ (r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{false})\\ \qquad \models (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_0))\end{array}\right]$$

$$\begin{array}{l}(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDanglingR)\\ \wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_0))\\ \wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_l))\\ \wedge markedListR(Stack, p_0) * (restoredListR(Stack, p_0) -\!\!* \; spansR(STree, root))\\ \wedge markedR\\ \quad * (unmarkedR \wedge (\forall x{:}\,\mathsf{loc}.\, allocated(x) \Rightarrow reach(r_0, x) \vee reachRightChildInList(Stack, x)))\end{array}$$

$\Longrightarrow$ Rule 9 in Lemma 81, which is

$$\left[\begin{array}{l}\left(\begin{array}{l}(r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{false})\\ \wedge markedListR(Stack, p_0)\\ \quad * (restoredListR(Stack, r_0) -\!\!* \; spansR(STree, root))\end{array}\right)\\ \quad \models \left(\begin{array}{l}markedListR(Stack, p_0) * (r_0 \mapsto r_l, r_r, r_c, \mathsf{false})\\ * ((r_0 \mapsto r_l, r_r, \mathsf{true}, \mathsf{true}) * restoredListR(Stack, r_0) -\!\!* \; spansR(STree, root))\end{array}\right)\end{array}\right]$$

$$\begin{array}{l}(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDanglingR)\\ \wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_0))\\ \wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_l))\\ \wedge \left(\begin{array}{l}(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * markedListR(Stack, p_0)\\ * ((r_0 \mapsto r_l, r_r, \mathsf{true}, \mathsf{true}) * restoredListR(Stack, r_0) -\!\!* \; spansR(STree, root))\end{array}\right)\\ \wedge markedR\\ \quad * (unmarkedR \wedge (\forall x{:}\,\mathsf{loc}.\, allocated(x) \Rightarrow reach(r_0, x) \vee reachRightChildInList(Stack, x)))\end{array}$$

$\Longrightarrow$ Rule 8 in Lemma 81, which is

$$\left[\begin{array}{l}\left(\begin{array}{l}(r_0 \hookrightarrow r_l, r_r, r_c, \mathsf{false})\\ \wedge markedR\\ \quad * \left(\begin{array}{l}unmarkedR\\ \wedge \forall x{:}\,\mathsf{loc}.\, allocated(x) \Rightarrow reach(r_0, x) \vee reachRightChildInList(Stack, x)\end{array}\right)\end{array}\right)\\ \quad \models \left(\begin{array}{l}markedR * (r_0 \mapsto r_l, r_r, r_c, \mathsf{false})\\ * \left(\begin{array}{l}unmarkedR\\ \wedge \forall x{:}\,\mathsf{loc}.\, allocated(x) \Rightarrow \left(\begin{array}{l}reach(r_l, x)\\ \vee reachI(r_r, x)\\ \vee reachRightChildInList(Lst, x)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right]$$

$$
\begin{aligned}
&(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \mathrel{-\!\!*} noDangling(r_0))\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \mathrel{-\!\!*} noDangling(r_l))\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * markedListR(Stack, p_0)\\
&\quad * ((r_0 \mapsto r_l, r_r, \mathsf{true}, \mathsf{true}) * restoredListR(Stack, r_0) \mathrel{-\!\!*} spansR(STree, root))\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * markedR\\
&\quad * \left(\begin{array}{l} unmarkedR \\ \wedge \forall x{:}\,\mathsf{loc}. \\ \qquad allocated(x) \Rightarrow reach(r_l, x) \vee reach(r_r, x) \vee reachRightChildInList(Stack, x) \end{array}\right)
\end{aligned}
$$

$\Longrightarrow P \models Q \mathrel{-\!\!*} (Q * P)$

$$
\begin{aligned}
&(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \mathrel{-\!\!*} noDanglingR)\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \mathrel{-\!\!*} noDangling(r_0))\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \mathrel{-\!\!*} noDangling(r_l))\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false})\\
&\quad * \left(\begin{array}{l} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \\ \mathrel{-\!\!*} \left(\begin{array}{l} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) * markedListR(Stack, p_0) \\ * ((r_0 \mapsto r_l, r_r, \mathsf{true}, \mathsf{true}) * restoredListR(Stack, r_0) \mathrel{-\!\!*} spansR(STree, root)) \end{array}\right) \end{array}\right)\\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false})\\
&\quad * \left(\begin{array}{l} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \\ \mathrel{-\!\!*} \left(\begin{array}{l} (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * markedR \\ * \left(\begin{array}{l} unmarkedR \\ \wedge \forall x{:}\,\mathsf{loc}. \\ \qquad allocated(x) \Rightarrow \left(\begin{array}{l} reach(r_l, x) \\ \vee\, reach(r_r, x) \\ \vee\, reachRightChildInList(Stack, x) \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right)
\end{aligned}
$$

$\Longrightarrow$ Definitions of $markedListR$, $restoredListR$, $reachRightChildInList$ and Rule 1, which is

$\left[\ (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * markedR \models markedR\ \right]$

$$
\begin{aligned}
&(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDanglingR) \\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_0)) \\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) * ((r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) -\!\!* \; noDangling(r_l)) \\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \\
&\quad * \left( \begin{array}{l} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \\ -\!\!* \left( \begin{array}{l} markedListR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0) \\ * \, (restoredListR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_l) -\!\!* \; spansR(STree, root)) \end{array} \right) \end{array} \right) \\
&\wedge (r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \\
&\quad * \left( \begin{array}{l} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \\ -\!\!* \left( \begin{array}{l} markedR \\ * \left( \begin{array}{l} unmarkedR \\ \wedge \, \forall x \colon \mathsf{loc}. \\ \qquad allocated(x) \Rightarrow \left( \begin{array}{l} reach(r_l, x) \\ \vee \, reachRightChildInList((r_0, p_0, r_r, \mathsf{false}) :: Stack, x) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)
\end{aligned}
$$

$\Longrightarrow$ $(r_0 \mapsto r_l, r_r, r_c, \mathsf{false})$ is strictly exact

$$
\begin{aligned}
&(r_0 \mapsto r_l, r_r, r_c, \mathsf{false}) \\
&* \left( \begin{array}{l} (r_0 \mapsto p_0, r_r, \mathsf{false}, \mathsf{true}) \\ -\!\!* \left( \begin{array}{l} noDanglingR \wedge noDangling(r_0) \wedge noDangling(r_l) \\ \wedge \, markedListR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0) \\ \quad * \, (restoredListR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_l) -\!\!* \; spansR(STree, root)) \\ \wedge \, markedR \\ \quad * \left( \begin{array}{l} unmarkedR \\ \wedge \, \forall x \colon \mathsf{loc}. \\ \qquad allocated(x) \Rightarrow \left( \begin{array}{l} reach(r_l, x) \\ \vee \, reachRightChildInList((r_0, p_0, r_r, \mathsf{false}) :: Stack, x) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)
\end{aligned}
$$

$\Longrightarrow$ Definition of *preLoopInvR*

$$preLoopInvR((r_0, p_0, r_r, \mathsf{false}) :: Stack, r_0, r_l, STree, root)$$

# CHAPTER 8

# Decidability

**BI** pointer logic uses the spatial connectives, $\mathsf{emp}, *, -\!\!*$, in the assertion language in an essential way. For instance, the forward axiom for $E.1 := E'$, which is $\{(E \mapsto E_1, E_2) * P\}\, E.1 := E'\, \{(E \mapsto E', E_2) * P\}$, no longer becomes sound when $*$ is replaced by $\wedge$. In fact, it is such a use of the spatial connectives that makes possible all elegant axioms in **BI** pointer logic. On the other hand, one might worry that such connectives complicate reasoning about the assertion language; if it were the case, **BI** pointer logic would not solve the problem of reasoning about pointers, but it would just shift difficulty from pointers to the spatial connectives. In this section, we argue that **BI** pointer logic indeed reduces the complexity of reasoning by looking at algorithmic issues of the **BI** assertion language. Particularly, we show that the spatial connectives don't change the decidability of the assertion language.

We start this chapter with an algorithmic analysis of the $\mathsf{emp}, *, -\!\!*$-free fragment of the assertion language. The analysis shows that there is no algorithm which answers whether one assertion implies another even when the assertions don't contain the spatial connectives. The proof of the undecidablity result uses quantification in a critical way so that it points out quantifiers as the main reason for undecidability. And in the next section, we show that as long as assertions don't include quantifiers, there is an algorithm which answers whether one assertion implies another. With the undecidability result in the first section, such a decidability result serves our purpose for an algorithmic analysis of the assertion language: it shows that reasoning with the spatial connectives is not so much more complicated than one without them.

## 8.1 Undecidability

In this section, we show that deciding whether one assertion implies another in the **BI** assertion language is not recursively enumerable even when the spatial connectives, $*$, $\mathsf{emp}$ and $-\!\!*$, and all expressions involving natural numbers and boolean values don't appear in the assertions. We prove the result by encoding the co-halting problem of a two stack machine in the assertion language without using $*$,$\mathsf{emp}$,$-\!\!*$ and any expressions involving natural numbers and boolean values. Since the co-halting problem is not recursively enumerable, the conclusion follows from the encoding.

In this section, we assume that each heap cell has five fields. Since a cell with five fields is implementable with four cells with two fields, the assumption does not limit the result in the section. And we use an intuitionistic variant $\hookrightarrow$ of the points-to predicate $\mapsto$ because only with the $\mapsto$ predicate but without the spatial conjunction $*$, we can't express that a heap cell $l$ is allocated and contains $(l_0, l_1, l_2, l_3, l_4)$ without requiring that $l$ is the only allocated heap cell. The meaning of $(E \hookrightarrow E_0, E_1, E_2, E_3, E_4)$ is $(E \mapsto E_0, E_1, E_2, E_3, E_4) * \mathsf{T}$, which says that a heap cell $E$ is allocated and contains $(E_0, E_1, E_2, E_3, E_4)$ but it need not be the only allocated cell. The main theorem in this section is as follows:

**Theorem 82** *Deciding that one assertion implies another in the* **BI** *assertion language is not recursively enumerable even when assertions are restricted by the following context free grammar:*

$$\begin{array}{rrcl} \textit{Expressions} & E & ::= & x, y \ldots \mid \mathsf{nil} \\ \textit{Assertions} & P & ::= & (E \hookrightarrow E, E, E, E, E) \mid E = E \mid \mathsf{F} \mid P \Rightarrow P \mid \forall x. P \end{array}$$

Note that the theorem indicates that $\models$ is not finitely axiomatizable even for such restricted assertions.

We show the theorem by reducing a well-known non-recursively-enumerable problem to the problem of deciding whether one assertion implies another. We use the co-halting problem of a two stack machine, whose alphabet is the set $\{0, 1\}$ and the set of states is $\{Q_0, \ldots, Q_e\}$ with $Q_0$ as the initial state and $Q_e$ as the unique final state. In the proof, we use two kinds of variables: variables in the assertion language and *meta* variables which range over numbers, states of the two stack machine and even assertions including variables in the assertion language. To avoid confusion between two kinds of variables, variables are written in the sans serif font, such as $\mathsf{x}, \mathsf{y}$, and meta variables in the italic font, such as $x, y$, in this section.

We pick up $e+8$ distinct variables $\mathsf{t}$, $\mathsf{f}$, $\mathsf{q}_0$, ..., $\mathsf{q}_e$, $\mathsf{DH}$, $\mathsf{DN}$, $\mathsf{DS1}$, $\mathsf{DS2}$ and $\mathsf{C}$, which are supposed to hold different values. Variables $\mathsf{t}$ and $\mathsf{f}$ represent elements in the alphabet, 1 and 0, respectively. And $\mathsf{q}_i$ represents the state $Q_i$. And the meaning of the last five variables will be explained later. When we use a meta variable to denote a state or a symbol in the alphabet, the same meta variable is also used to denote the corresponding variable. For instance, a meta variable $a$ denoting 0 is also used to denote the variable $\mathsf{f}$.

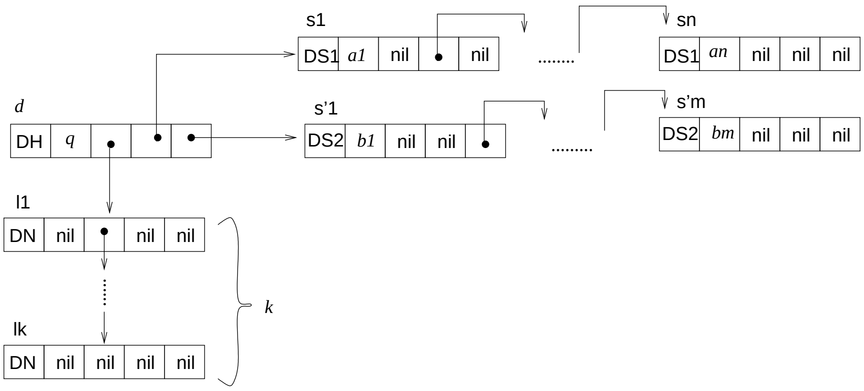
Let $(q, a_1 \ldots a_n, b_1 \ldots b_m)$ be an instantaneous description of a two stack machine, where $q$ denotes the current state, and $a_1 \ldots a_n$ and $b_1 \ldots b_m$ are the contents of two stacks with $a_1$ and $b_1$ their top elements. Suppose the description be the status of the machine at the $k$th execution step. We encode this fact — $(q, a_1 \ldots a_n, b_1 \ldots b_m)$ is the status of the machine at the $k$th step — by the following assertion:

$$\begin{aligned}
&Desc(d, k, (q, a_1 \ldots a_n, b_1 \ldots b_m)) \triangleq \\
&\quad \exists \mathsf{l}_1, \ldots, \mathsf{l}_{k+1}, \mathsf{s}_1, \ldots, \mathsf{s}_{n+1}, \mathsf{s}'_1, \ldots, \mathsf{s}'_{m+1}. \\
&\qquad \wedge (d \hookrightarrow \mathsf{DH}, q, \mathsf{l}_1, \mathsf{s}_1, \mathsf{s}'_1) \\
&\qquad \wedge (\textstyle\bigwedge_{1 \le j \le n} (\mathsf{s}_j \hookrightarrow \mathsf{DS1}, a_j, \mathsf{nil}, \mathsf{s}_{j+1}, \mathsf{nil})) \wedge (\mathsf{s}_{n+1} = \mathsf{nil}) \\
&\qquad \wedge (\textstyle\bigwedge_{1 \le j \le m} (\mathsf{s}'_j \hookrightarrow \mathsf{DS2}, b_j, \mathsf{nil}, \mathsf{nil}, \mathsf{s}'_{j+1})) \wedge (\mathsf{s}'_{m+1} = \mathsf{nil}) \\
&\qquad \wedge (\textstyle\bigwedge_{1 \le j \le k} (\mathsf{l}_j \hookrightarrow \mathsf{DN}, \mathsf{nil}, \mathsf{l}_{j+1}, \mathsf{nil}, \mathsf{nil})) \wedge (\mathsf{l}_{k+1} = \mathsf{nil})
\end{aligned}$$

We take the convention that when the first stack is empty, $n$ becomes 0 and when the second stack is empty, $m$ becomes 0. Recall that $\mathsf{DH}$, $\mathsf{DN}$, $\mathsf{DS1}$ and $\mathsf{DS2}$ are assumed to denote different values. So, for a state satisfying the assertion, any two variables among $d$, $\mathsf{l}_1 \ldots \mathsf{l}_{k+1}$, $\mathsf{s}_1 \ldots \mathsf{s}_{n+1}$ and $\mathsf{s}'_1 \ldots \mathsf{s}'_{m+1}$ must denote different locations. The assertion denotes that the following structure of heap cells exists on heap storage:

| | |
|---|---|
| DH | the node is the head node for the coding of an instantaneous description |
| DN | the node is used for encoding the current computation step number |
| DS1 | the node is used for encoding the first stack |
| DS2 | the node is used for encoding the second stack |
| C | the node is used for encoding a step in a computation sequence |

**Table 8.1** Variables denoting the Role of a Node

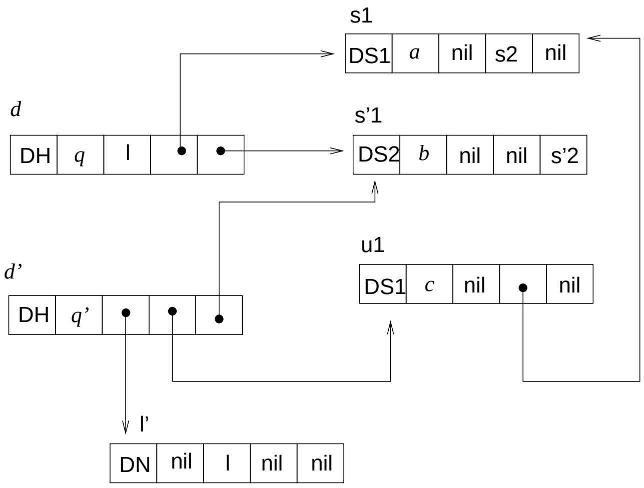



*Desc* is a function which maps an variable, a number and an instantaneous description to an assertion. Therefore, when *Desc* is applied to $(d, k, (q, a_1 \dots a_n, b_1 \dots b_m))$, every free variable in the resulting assertion is among $d$, t, f, $\mathsf{q}_0$, ..., $\mathsf{q}_e$, DH, DN, DS1, DS2, C. In the above picture, the first field of each heap cell denotes what role the cell plays in the encoding. Five distinct variables are used for this purpose, whose meanings are explained in Table 8.1. The second field of the head node, which is pointed to by $d$ in the picture, stores the current state $q$ using the variable $\mathsf{q}_i$ corresponding to $q$. The third field of the head node is the encoding of the current computation step number $k$ with unary representation: the third field points to a singly linked list whose length is $k$. The last two fields in the head node denote two stacks, which are just two singly linked lists. Each node for two stacks uses the second field to store the contents of the stacks, which are sequences of two variables t and f.

Encoding each transition $t$, such as $t = (q, a, b) \to (q', ca, b)$, is trickier; we use aliasing of pointers to denote this transition. Instead of showing the encodings of five possible transitions depending on how two stacks are changed, we deal with only three of them, which are $t = (q, a, b) \to (q', ca, b)$, $t = (q, a, b) \to (q', \ , b)$ and $t = (q, a, b) \to (q', a, b)$; the other two are symmetric to the first two. First, when $t = (q, a, b) \to (q', ca, b)$, the corresponding assertion is as follows:

$$
\begin{aligned}
&\mathit{Trans}(d, d', (q, a, b) \xrightarrow{t} (q', ca, b)) \triangleq \\
&\ \exists \mathsf{l}, \mathsf{l}', \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_1', \mathsf{s}_2', \mathsf{u}_1. \\
&\quad (\mathsf{u}_1 \neq \mathsf{s}_1) \wedge (d \neq d') \\
&\quad \wedge (d \hookrightarrow \mathsf{DH}, q, \mathsf{l}, \mathsf{s}_1, \mathsf{s}_1') \\
&\quad \wedge (\mathsf{s}_1 \hookrightarrow \mathsf{DS1}, a, \mathsf{nil}, \mathsf{s}_2, \mathsf{nil}) \\
&\quad \wedge (\mathsf{s}_1' \hookrightarrow \mathsf{DS2}, b, \mathsf{nil}, \mathsf{nil}, \mathsf{s}_2') \\
&\quad \wedge (d' \hookrightarrow \mathsf{DH}, q', \mathsf{l}', \mathsf{u}_1, \mathsf{s}_1') \\
&\quad \wedge (\mathsf{u}_1 \hookrightarrow \mathsf{DS1}, c, \mathsf{nil}, \mathsf{s}_1, \mathsf{nil}) \\
&\quad \wedge (\mathsf{l}' \hookrightarrow \mathsf{DN}, \mathsf{nil}, \mathsf{l}, \mathsf{nil}, \mathsf{nil})
\end{aligned}
$$

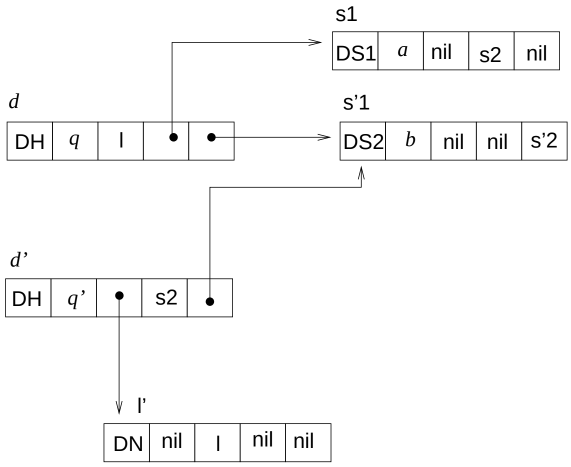
Note that $d, \mathsf{s}_1, \mathsf{s}_1', d', \mathsf{u}_1, \mathsf{l}'$ denote different locations. Therefore, the body part of the existential quantification says that the following structure of heap cells exists in heap storage:




One more node is added in the computation step field in the head node $d'$; the added node represents that the computation step number is increased by one. When $t = (q, a, b) \to (q', \ , b)$, the corresponding assertion is:

$$\begin{aligned}
&\mathit{Trans}(d, d', (q, a, b) \xrightarrow{t} (q', \ \ , b)) \triangleq \\
&\ \exists \mathsf{l}, \mathsf{l}', \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}'_1, \mathsf{s}'_2, \\
&\quad (d \neq d') \\
&\quad \wedge (d \hookrightarrow \mathsf{DH}, q, \mathsf{l}, \mathsf{s}_1, \mathsf{s}'_1) \\
&\quad \wedge (\mathsf{s}_1 \hookrightarrow \mathsf{DS1}, a, \mathsf{nil}, \mathsf{s}_2, \mathsf{nil}) \\
&\quad \wedge (\mathsf{s}'_1 \hookrightarrow \mathsf{DS2}, b, \mathsf{nil}, \mathsf{nil}, \mathsf{s}'_2) \\
&\quad \wedge (d' \hookrightarrow \mathsf{DH}, q', \mathsf{l}', \mathsf{s}_2, \mathsf{s}'_1) \\
&\quad \wedge (\mathsf{l}' \hookrightarrow \mathsf{DN}, \mathsf{nil}, \mathsf{l}, \mathsf{nil}, \mathsf{nil})
\end{aligned}$$

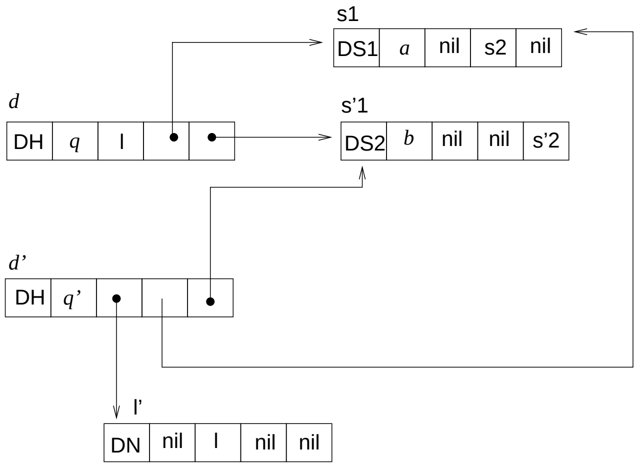
Since $d, d', \mathsf{s}_1, \mathsf{s}'_1, \mathsf{l}'$ denote different locations, the body part of the existential quantification is equivalent to requiring the following structure of heap cells exists in heap storage:




Finally, when $t = (q, a, b) \to (q', a, b)$, the corresponding assertion is:

$$\begin{aligned}
&\mathit{Trans}(d, d', (q, a, b) \xrightarrow{t} (q', a, b)) \triangleq \\
&\ \exists \mathsf{l}, \mathsf{l}', \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}'_1, \mathsf{s}'_2, \\
&\quad (d \neq d') \\
&\quad \wedge (d \hookrightarrow \mathsf{DH}, q, \mathsf{l}, \mathsf{s}_1, \mathsf{s}'_1) \\
&\quad \wedge (\mathsf{s}_1 \hookrightarrow \mathsf{DS1}, a, \mathsf{nil}, \mathsf{s}_2, \mathsf{nil}) \\
&\quad \wedge (\mathsf{s}'_1 \hookrightarrow \mathsf{DS2}, b, \mathsf{nil}, \mathsf{nil}, \mathsf{s}'_2) \\
&\quad \wedge (d' \hookrightarrow \mathsf{DH}, q', \mathsf{l}', \mathsf{s}_1, \mathsf{s}'_1) \\
&\quad \wedge (\mathsf{l}' \hookrightarrow \mathsf{DN}, \mathsf{nil}, \mathsf{l}, \mathsf{nil}, \mathsf{nil})
\end{aligned}$$

The body part of the existential quantification indicates that the following structure of heap cells exists in heap storage:




**Lemma 83** *Let $\delta$ be the set of transitions of a two stack machine. And suppose that $(q, \alpha, \beta)$ is an instantaneous description of the machine. For all states $(s, h)$ and all natural numbers $k$, if*

$$s, h \models (\bigvee_{t \in \delta} \mathit{Trans}(\mathsf{ds}, \mathsf{ds}', t)) \wedge \mathit{Desc}(\mathsf{ds}, k, (q, \alpha, \beta))$$

*then there exist $t$ and $(q', \alpha', \beta')$ such that*

- *$(q', \alpha', \beta')$ is obtained from $(q, \alpha, \beta)$ by applying $t$; and*
- *$s, h \models \mathit{Desc}(\mathsf{ds}', k + 1, (q', \alpha', \beta'))$.*

*Proof:* Let $h$ be a heap satisfying the condition in the lemma. Then, by the interpretation of $\vee$, there exists a transition $t \in \delta$ such that

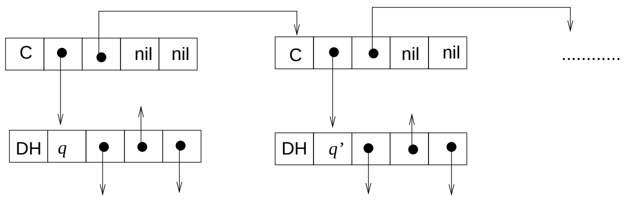
$$s, h \models \mathit{Trans}(\mathsf{ds}, \mathsf{ds}', t) \wedge \mathit{Desc}(\mathsf{ds}, k, (q, \alpha, \beta)).$$

Consider the case when $t = (q, a, b) \to (q', ca, b)$. Since $s, h \models \mathit{Trans}(\mathsf{ds}, \mathsf{ds}', t)$, the first stack of the instantaneous description encoded by nodes from $\mathsf{ds}$ should contain $a$ as a top element and the second stack $b$ as a top element. So, we can rewrite $\alpha$ and $\beta$ as $a\alpha_0$ and $b\beta_0$ for another sequences of characters $\alpha_0$ and $\beta_0$. Then, $(q', ca\alpha_0, b\beta_0)$ can be obtained from $(q, \alpha, \beta)$ by applying $t$. In order to see that the nodes from $\mathsf{ds}'$ encode the instantaneous description $(q', ca\alpha_0, b\beta_0)$, we observe that the nodes denoted by $\mathsf{ds}'$, $\mathsf{ds}'$'s third field and fourth field do not participate in the encoding of $(q, \alpha, \beta)$ by nodes from $\mathsf{ds}$. With this observation and the definitions of $Desc(\mathsf{ds}', k+1, (q', ca\alpha_0, b\beta_0))$ and $Trans(\mathsf{ds}, \mathsf{ds}', (q, a, b) \to (q', ca, b))$, it is easy to see that

$$s, h \models Desc(\mathsf{ds}', k+1, (q', ca\alpha_0, b\beta_0)).$$

All the other cases can be handled similarly. $\square$

The last step of the encoding is to handle a computation sequence. For this purpose, we use nodes tagged with a special variable, $\mathsf{C}$. The second field of such a node denotes the head cell which implements an instantaneous description and the third field the next step in a computation sequence. The following picture shows a typical use of nodes tagged with $\mathsf{C}$:




We encode a computation sequence with three assertions $Init(d, (q_0, \alpha_0, \beta_0))$, $Next$ and $Final$. The first assertion $Init(d, (q_0, \alpha_0, \beta_0))$ says that heap storage contains the encoding of the instantaneous description $(q_0, \alpha_0, \beta_0)$ at step 0. It is defined as follows:

$$\begin{aligned}&Init(c, (q_0, \alpha_0, \beta_0)) \triangleq \\ &\quad \exists \mathsf{ds}_0, \mathsf{nxt}.\, (c \hookrightarrow \mathsf{C}, \mathsf{ds}_0, \mathsf{nxt}, \mathsf{nil}, \mathsf{nil}) \wedge Desc(\mathsf{ds}_0, 0, (q_0, \alpha_0, \beta_0))\end{aligned}$$

For each instantaneous description which appears in the middle of an accepting computation, either there is another description which denotes the next step or the description denotes termination. $Next$ and $Final$ capture this. $Next$ says that for each node which denotes a computation step, if it doesn't denote termination, there is another node for the next step. $Final$ says that if the node denotes termination, its instantaneous description should be in the unique final state $\mathsf{q}_e$.

$$
\begin{array}{l}
\textit{Next} \triangleq \\
\quad \forall \mathsf{cs}_0. \\
\qquad (\exists \mathsf{ds}_0, \mathsf{cs}_1.\, (\mathsf{cs}_0 \hookrightarrow \mathsf{C}, \mathsf{ds}_0, \mathsf{cs}_1, \mathsf{nil}, \mathsf{nil}) \wedge (\mathsf{cs}_1 \neq \mathsf{nil})) \\
\qquad \Rightarrow \begin{pmatrix} \exists \mathsf{ds}_0, \mathsf{ds}_1, \mathsf{cs}_1, \mathsf{cs}_2. \\ \quad (\mathsf{cs}_0 \hookrightarrow \mathsf{C}, \mathsf{ds}_0, \mathsf{cs}_1, \mathsf{nil}, \mathsf{nil}) \wedge (\mathsf{cs}_1 \hookrightarrow \mathsf{C}, \mathsf{ds}_1, \mathsf{cs}_2, \mathsf{nil}, \mathsf{nil}) \wedge (\mathsf{cs}_0 \neq \mathsf{cs}_1) \\ \quad \wedge (\bigvee_{t \in \delta} \textit{Trans}(\mathsf{ds}_0, \mathsf{ds}_1, t)) \end{pmatrix}
\end{array}
$$

$$
\begin{array}{l}
\textit{Final} \triangleq \\
\quad \forall \mathsf{cs}, \mathsf{ds}. \\
\qquad (\mathsf{cs} \hookrightarrow \mathsf{C}, \mathsf{ds}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}) \Rightarrow \exists \mathsf{l}, \mathsf{s}_1, \mathsf{s}_1'.\, (\mathsf{ds} \hookrightarrow \mathsf{DH}, \mathsf{q}_e, \mathsf{l}, \mathsf{s}_1, \mathsf{s}_1')
\end{array}
$$

Notice the local nature of *Next* and *Final*; they only mention each step as opposed to a whole sequence of computation, which is usually the case in the encoding with a list data type or a graph data type in [12, 23]. We need one more formula which says that all $e+8$ variables $\mathsf{t}$, $\mathsf{f}$, $\mathsf{q}_0$, …, $\mathsf{q}_e$, $\mathsf{DH}$, $\mathsf{DN}$, $\mathsf{DS1}$, $\mathsf{DS2}$, $\mathsf{C}$ denote distinct values:

$$
\textit{Disjoint} \triangleq \bigwedge_{v, w \in \{\mathsf{t}, \mathsf{f}, \mathsf{q}_0, \ldots, \mathsf{q}_e, \mathsf{DH}, \mathsf{DN}, \mathsf{DS1}, \mathsf{DS2}, \mathsf{C}\}, v \not\equiv w} (v \neq w)
$$

where we use $\equiv$ to mean that two variables are the same as symbols.

**Lemma 84** *Let $\delta$ be the set of transitions of the two stack machine. And let $(q_0, \alpha_0, \beta_0)$ be the initial instantaneous description of the machine. For all states $(s, h)$, if*

$$
\begin{array}{ll}
s, h \models & \forall \mathsf{t}, \mathsf{f}, \mathsf{q}_0, \ldots, \mathsf{q}_e, \mathsf{DH}, \mathsf{DN}, \mathsf{DS1}, \mathsf{DS2}, \mathsf{C}. \\
& \qquad \textit{Disjoint} \Rightarrow \exists \mathsf{cs}_0.\, \textit{Init}(\mathsf{cs}_0, (q_0, \alpha_0, \beta_0)) \wedge \textit{Next} \wedge \textit{Final}
\end{array}
$$

*then there exists $n$ such that*

$$
\begin{array}{ll}
s, h \models & \forall \mathsf{t}, \mathsf{f}, \mathsf{q}_0, \ldots, \mathsf{q}_e, \mathsf{DH}, \mathsf{DN}, \mathsf{DS1}, \mathsf{DS2}, \mathsf{C}. \\
& \quad \textit{Disjoint} \Rightarrow \\
& \qquad \exists \mathsf{cs}_0 \ldots \mathsf{cs}_{n+1}, \mathsf{ds}_0 \ldots \mathsf{ds}_n. \\
& \qquad\quad \textit{Desc}(\mathsf{ds}_0, (q_0, \alpha_0, \beta_0)) \\
& \qquad\quad \wedge (\bigwedge_{0 \le i \le n} (\mathsf{cs}_i \hookrightarrow \mathsf{C}, \mathsf{ds}_i, \mathsf{cs}_{i+1}, \mathsf{nil}, \mathsf{nil})) \wedge (\mathsf{cs}_{n+1} = \mathsf{nil}) \\
& \qquad\quad \wedge (\bigwedge_{0 \le i < n} (\bigvee_{t \in \delta} \textit{Trans}(\mathsf{ds}_i, \mathsf{ds}_{i+1}, t))) \\
& \qquad\quad \wedge \exists \mathsf{l}, \mathsf{s}_1, \mathsf{s}_1'.\, (\mathsf{ds}_n \hookrightarrow \mathsf{DH}, \mathsf{q}_e, \mathsf{l}, \mathsf{s}_1, \mathsf{s}_1')
\end{array}
$$

*Proof:* The only thing which we have to show is, in fact, the second line in the body of the existential quantification. That is,

$$\begin{aligned} s, h \models\ & \forall \mathsf{t}, \mathsf{f}, \mathsf{q}_0, \ldots, \mathsf{q}_e, \mathsf{DH}, \mathsf{DN}, \mathsf{DS1}, \mathsf{DS2}, \mathsf{C}. \\ & \qquad \mathit{Disjoint} \Rightarrow \\ & \qquad\ \exists \mathsf{cs}_0 \ldots \mathsf{cs}_{n+1}, \mathsf{ds}_0 \ldots \mathsf{ds}_n. \\ & \qquad\quad (\textstyle\bigwedge_{0 \leq i \leq n} (\mathsf{cs}_i \hookrightarrow \mathsf{C}, \mathsf{ds}_i, \mathsf{cs}_{i+1}, \mathsf{nil}, \mathsf{nil})) \wedge \mathsf{cs}_{n+1} = \mathsf{nil} \end{aligned}$$

All the other lines in the body follow from the assumption of the lemma and the above. In order to see why the above holds, we note two facts: $h$ is a finite partial function, and when a heap cell is tagged with $\mathsf{C}$ and contains $\mathsf{nil}$ in its fourth and fifth fields, it should be in a linked list. So, it suffices to show that the linked list from the node $\mathsf{cs}_0$ doesn't contain a cycle. This becomes the case because each node in the linked list contains a link to an instantaneous description whose computation step number is increased by one as the linked list proceeds. □

Now the main fact in this encoding is the following:

**Proposition 85** *Let $\delta$ be the set of transitions of the two stack machine. And let $(q_0, \alpha_0, \beta_0)$ be the initial instantaneous description of the machine. The machine accepts $(q_0, \alpha_0, \beta_0)$ iff there exists a state $(s, h)$ such that*

$$\begin{aligned} s, h \models\ & \forall \mathsf{t}, \mathsf{f}, \mathsf{q}_0, \ldots, \mathsf{q}_e, \mathsf{DH}, \mathsf{DN}, \mathsf{DS1}, \mathsf{DS2}, \mathsf{C}. \\ & \qquad \mathit{Disjoint} \Rightarrow \exists \mathsf{cs}_0. \mathit{Init}(\mathsf{cs}_0, (q_0, \alpha_0, \beta_0)) \wedge \mathit{Next} \wedge \mathit{Final} \end{aligned}$$

*Proof:* It is easy to see that the only-if direction holds because we can build a heap which only contains heap cells for encoding an accepting computation of $(q_0, \alpha_0, \beta_0)$. And Lemma 83 and 84 give the other direction. □

The following corollary shows the reduction.

**Corollary 86** *Let $\delta$ be the set of transitions of the two stack machine. And let $(q_0, \alpha_0, \beta_0)$ be the initial instantaneous description of the machine. The machine doesn't accept $(q_0, \alpha_0, \beta_0)$ iff*

$$\left( \begin{array}{l} \forall \mathsf{t}, \mathsf{f}, \mathsf{q}_0, \ldots, \mathsf{q}_e, \mathsf{DH}, \mathsf{DN}, \mathsf{DS1}, \mathsf{DS2}, \mathsf{C}. \\ \quad \mathit{Disjoint} \Rightarrow \exists \mathsf{cs}_0. \mathit{Init}(\mathsf{cs}_0, (q_0, \alpha_0, \beta_0)) \wedge \mathit{Next} \wedge \mathit{Final} \end{array} \right) \models \mathsf{F}$$

*Proof:* This corollary follows from the proposition because for an assertion $P$,

$$P \models \mathsf{F} \iff \text{there are no states } (s,h) \text{ such that } s,h \models P$$

$\square$

## 8.2 Decidable Fragment

In the previous section, we have shown that the problem of deciding whether one assertion implies another is not recursively enumerable even when the assertions don't contain $*$ and $-\!*$. Quantification is used in an essential way in showing the result. In this section, we show that if quantification is taken out from the **BI** assertion language, the assertion language becomes decidable even with the spatial connectives, $*, -\!*, \mathsf{emp}$. Therefore, this section provides real algorithmic insight of the **BI** assertion language because the spatial connectives, $*$, $-\!*$ and $\mathsf{emp}$, are the main features of the **BI** assertion language.

In this section, we assume that expressions are either variables or $\mathsf{nil}$ in order to concentrate on connectives of the assertion language. The main theorem in the section shows that there is an algorithm which decides whether one assertion implies another as long as the assertions contain neither quantification nor expressions other than variables, $\mathsf{nil}$.

**Theorem 87** *Deciding whether one assertion implies another in the* **BI** *assertion language is algorithmically decidable as long as the assertions are instances of the following grammar:*

$$\begin{array}{llcl} \textit{Expressions} & E & ::= & x, y, \ldots \mid \mathsf{nil} \\ \textit{Assertions} & P & ::= & (E \mapsto E, E) \mid E = E \mid \mathsf{F} \mid P \Rightarrow P \mid \mathsf{emp} \mid P * P \mid P -\!* P \end{array}$$

The theorem is equivalent to the existence of an algorithm which takes an assertion following the grammar in the theorem and answers whether the assertion holds for all states. We show that such an algorithm exists. The main observation is that each assertion determines a finite set of states so that if the assertion holds for all states in the set, it indeed holds for all states, which are not necessarily in the set. We proceed the proof in three steps: firstly, we consider the case that an assertion $P$ and a state $s, h$ are given so that an algorithm is supposed to answer whether $s, h \models P$; secondly, we show that there is an algorithm which takes an assertion $P$ and a stack $s$ as inputs and answers whether $s, h \models P$ for all heaps $h$; finally, we construct an algorithm which, given an assertion $P$, answers whether $s, h \models P$ for all states $(s, h)$. In the remainder of the section, we assume that all assertions follow the grammar in the theorem.

The problem of algorithmically deciding whether $s, h \models P$ holds given $P, s, h$ as inputs is not as straightforward as it seems because of $-\!\!*$ : when $P$ is of the form $Q -\!\!* R$, the interpretation of $s, h \models P$ involves quantification over all heaps, which might require to check infinite possibilities. So, the decidability proof is mainly for showing that there is a finite boundary algorithmically determined by $Q$ and $R$. We first define the *size* of an assertion, which is used to give an algorithm to determine the boundary.

**Definition 88 (size of $P$)** *For an assertion $P$, we define* size *of $P$, $|P|$, as follows:*

$$
\begin{array}{rcl}
|(E \mapsto E_1, E_2)| & = & 1 \\
|E_1 = E_2| & = & 0 \\
|\mathsf{F}| & = & 0 \\
|P \Rightarrow Q| & = & max(|P|, |Q|) \\
|P * Q| & = & |P| + |Q| \\
|P -\!\!* Q| & = & |Q| \\
|\mathsf{emp}| & = & 1
\end{array}
$$

The size of $P$ intuitively determines how many heap cells which are not directly pointed to by variables in $P$ are relevant to the truth of $P$. For instance, the size of $(\neg\mathsf{emp}) * (\neg\mathsf{emp})$ is 2 and indicates that if at least two cells are already allocated in the heap, allocating more cells on the heap doesn't affect the truth of the assertion. Note that $(\neg\mathsf{emp}) * (\neg\mathsf{emp})$ holds for a state when at least two heap cells are allocated in the state.

The following proposition claims that there is a finite boundary for heaps to check in the interpretation of $s, h \models Q -\!\!* R$; the decidability result is just an immediate corollary. For a stack $s$ and a set of variables $X$, let $s(X)$ be $\{s(x) \mid x \in X\}$. And let $\mathsf{ord}$ be a bijection from **Nat** to *Loc*, which gives a particular enumeration of locations in *Loc*.

**Proposition 89** *Given a state $(s, h)$ and assertions $Q, R$, let $X$ be $FV(Q) \cup FV(R)$ and $B$ a finite set consisting of the first $max(|Q|, |R|)$ locations in $Loc - (dom(h) \cup s(X))$ where the ordering is given by* $\mathsf{ord}$. *Pick a value $v \in Val - s(X) - \{nil\}$. Then, $(s, h) \models Q -\!\!* R$ holds if for all $h_1$ such that*

- *$h \# h_1$;*

- $(s, h_1) \models Q$;
- $dom(h_1) \subseteq B \cup s(X)$; and
- for all $l \in dom(h_1)$, $h_1(l) \in (s(X) \cup \{nil, v\}) \times (s(X) \cup \{nil, v\})$,

*we have that*

$$(s, h * h_1) \models R.$$

To see why the proposition implies the decidability result, notice that there are only finitely many $h_1$'s satisfying four conditions because both $B \cup s(X)$ and $s(X) \cup \{nil, v\}$ are finite. Since all other cases in the definition of $s, h \models P$ involve only finite possibilities of satisfying $s, h \models P$, the exhaustive search gives the decision algorithm.

For a stack $s$, a natural number $n$ and a finite set $X$ of variables, we define an equivalence relation $\sim_{s,n,X}$ between heaps, which plays an important role in showing Proposition 89 as well as the decidability of checking whether $s, h \models P$ for all heaps h. Intuitively, two heaps are related if they are essentially the same as long as variables "in interest", i.e., those in $X$, are concerned in the state $s$ and they have the same amount of "garbage heap cells" upto $n$. The important property of the equivalence relation is that when two heaps $h, h'$ are related ($h \sim_{s,n,X} h'$) and an assertion $P$ satisfies certain conditions regarding its size and free variables, $P$ can't distinguish $h$ from $h'$: $s, h \models P$ iff $s, h' \models P$. We explain more on this in Proposition 95. The equivalence relation is defined as follows:

**Definition 90 ($=_S$)** *Given a set of values $S \subseteq Val$, $=_S$ is a relation between pairs of values such that for any $(v_0, v_1), (v'_0, v'_1) \in Val \times Val$, $(v_0, v_1) =_S (v'_0, v'_1)$ iff for $i = 0, 1$,*

- *if $v_i \in S \cup \{nil\}$, then $v_i = v'_i$; and*
- *if $v_i \notin S \cup \{nil\}$, then $v'_i \notin S \cup \{nil\}$.*

**Definition 91 ($\sim_{s,n,X}$)** *Given a stack $s$, a natural number $n$ and a set $X$ of variables, $\sim_{s,n,X}$ is a relation between heaps such that $h \sim_{s,n,X} h'$ iff*

- *$s(X) \cap dom(h) = s(X) \cap dom(h')$;*
- *for all $l \in s(X) \cap dom(h)$, $h(l) =_{s(X)} h'(l)$;*

- *if* $|dom(h) - s(X)| < n$*, then* $|dom(h) - s(X)| = |dom(h') - s(X)|$*; and*

- *if* $|dom(h) - s(X)| \geq n$*, then* $|dom(h') - s(X)| \geq n$*.*

Intuitively, $h \sim_{s,n,X} h'$ holds if $h$ and $h'$ are the "same" as far as variables in $X$ are concerned in the state $s$ and, moreover, $h$ and $h'$ have the same amount of heap cells upto $n$ which are not directly pointed to by any variables in $X$ in $s$. Notice that $\sim_{s,n,X}$ is an equivalence relation and that if $h \sim_{s,m,X} h'$ and $n \leq m$, $h \sim_{s,n,X} h'$. These two properties of $\sim_{s,n,X}$ are used frequently in the rest of the section.

The equivalence relation is preserved by splitting a heap and adding a new heap.

**Lemma 92** *If* $m = n_1 + n_2$ *and* $h \sim_{s,m,X} h'$ *and* $h = h_1 * h_2$*, then there exist* $h'_1, h'_2$ *such that*

1. $h'_1 \# h'_2$*;*

2. $h' = h'_1 * h'_2$*;*

3. $h_1 \sim_{s,n_1,X} h'_1$*; and*

4. $h_2 \sim_{s,n_2,X} h'_2$*.*

*Proof:* Let $\{A, B\}$ be a partition of $dom(h') - s(X)$ such that

- when $|dom(h) - s(X)| < m$, $|A| = |dom(h_1) - s(X)|$ and $|B| = |dom(h_2) - s(X)|$;

- when $|dom(h) - s(X)| \geq m$ and $|dom(h_2) - s(X)| \geq n_2$, $|A| = \min(|dom(h_1) - s(X)|, n_1)$; and

- when $|dom(h) - s(X)| \geq m$ and $|dom(h_2) - s(X)| < n_2$, $|B| = |dom(h_2) - s(X)|$.

Notice that the first case is possible because when $|dom(h) - s(X)| < m$, $|dom(h') - s(X)| = |dom(h) - s(X)|$ and $|dom(h) - s(X)| = |dom(h_1) - s(X)| + |dom(h_2) - s(X)|$. Also, notice that the second and the third cases are also possible because when $|dom(h) - s(X)| \geq m$, $|dom(h') - s(X)| \geq m \geq max(n_1, n_2)$.

Let $h'_A$ and $h'_B$ be restrictions of $h'$ to $A \cup (dom(h_1) \cap s(X))$ and $B \cup (dom(h_2) \cap s(X))$, respectively. We show that $h'_A$ and $h'_B$ satisfy the required properties.

1. $h'_A \# h'_B$:

$$dom(h'_A) \cap dom(h'_B)$$
$$=$$
$$(A \cup (dom(h_1) \cap s(X))) \cap (B \cup (dom(h_2) \cap s(X)))$$
$$=$$
$$(A \cap B) \cup (dom(h_1) \cap dom(h_2) \cap s(X))$$
$$\cup (dom(h_1) \cap s(X) \cap B) \cup (dom(h_2) \cap s(X) \cap A)$$
$$= \quad \because A \cap B = \emptyset \ \wedge \ dom(h_1) \cap dom(h_2) = \emptyset$$
$$(dom(h_1) \cap s(X) \cap B) \cup (dom(h_2) \cap s(X) \cap A)$$
$$= \quad \because B \cap s(X) \subseteq (dom(h') - s(X)) \cap s(X) = \emptyset$$
$$dom(h_2) \cap s(X) \cap A$$
$$= \quad \because A \cap s(X) \subseteq (dom(h') - s(X)) \cap s(X) = \emptyset$$
$$\emptyset$$

2. $h' = h'_A * h'_B$:

Since $h'_A \# h'_B$ and $h'_A$ and $h'_B$ are restrictions of $h'$, it suffices to show that $dom(h'_A) \cup dom(h'_B) = dom(h')$.

$$dom(h'_A) \cup dom(h'_B)$$
$$=$$
$$(A \cup (dom(h_1) \cap s(X))) \cup (B \cup (dom(h_2) \cap s(X)))$$
$$=$$
$$(A \cup B) \cup ((dom(h_1) \cup dom(h_2)) \cap s(X))$$
$$= \quad \because h = h_1 * h_2$$
$$(A \cup B) \cup (dom(h) \cap s(X))$$
$$= \quad \because h \sim_{s,m,X} h'$$
$$(A \cup B) \cup (dom(h') \cap s(X))$$
$$= \quad \because A \cup B = dom(h') - s(X)$$
$$(dom(h') - s(X)) \cup (dom(h') \cap s(X))$$
$$=$$
$$dom(h')$$

3. $h_1 \sim_{s,n_1,X} h'_A$:

To handle this condition and the last condition, we reply on the simplification of $\sim$ for heaps obtained by restricting two already related heaps:

> **Fact:** Suppose that $h \sim_{s,n,X} h'$ and that $h_0$ and $h'_0$ are restrictions of $h$ and $h'$, respectively. If $s(X) \cap dom(h_0) = s(X) \cap dom(h'_0)$, then $h_0(l) =_{s(X)} h'_0(l)$ for all $l \in s(X) \cap dom(h_0)$.

Therefore, it suffices to show that

- $dom(h_1) \cap s(X) = dom(h'_A) \cap s(X)$;
- if $|dom(h_1) - s(X)| < n_1$, then $|dom(h_1) - s(X)| = |dom(h'_A) - s(X)|$; and
- if $|dom(h_1) - s(X)| \geq n_1$, then $|dom(h'_A) - s(X)| \geq n_1$.

For the first bullet,

$$
\begin{array}{ll}
 & dom(h'_A) \cap s(X) \\
= & \\
 & (A \cup (dom(h_1) \cap s(X))) \cap s(X) \\
= & \\
 & (A \cap s(X)) \cup (dom(h_1) \cap s(X)) \\
= & \quad \because A \subseteq dom(h') - s(X) \\
 & dom(h_1) \cap s(X)
\end{array}
$$

For the other two bullets, we note that $dom(h'_A) - s(X) = A$ because:

$$
\begin{array}{ll}
 & dom(h'_A) - s(X) \\
= & \\
 & (A \cup (dom(h_1) \cap s(X))) - s(X) \\
= & \\
 & A - s(X) \\
= & \quad \because A \subseteq dom(h') - s(X) \\
 & A
\end{array}
$$

We divide the proof into two cases, when $|dom(h) - s(X)| < m$ and when $|dom(h) - s(X)| \geq m$. In the first case, $|A| = |dom(h_1) - s(X)|$ holds by the choice of $A$; therefore, the requirement is satisfied. For the second, we consider two subcases depending on whether

$|dom(h_2) - s(X)| \geq n_2$ or not. If $|dom(h_2) - s(X)| \geq n_2$, $|A| = \min(|dom(h_1) - s(X)|, n_1)$ by the choice of $A$. Therefore, it meets the requirement. The only remaining case is when $|dom(h) - s(X)| \geq m$ and $|dom(h_2) - s(X)| < n_2$. In this case, $|B|$ is smaller than $n_2$ by the choice of $B$. Therefore,

$$\begin{array}{rcl} |A| & = & |dom(h) - s(X)| - |B| \\ & \geq & n_1 + n_2 - |B| \\ & \geq & n_1 \end{array}$$

Since $|dom(h_1) - s(X)|$ is greater than or equal to $n_1$ in this case, the requirement is met, too.

4. $h_2 \sim_{s,n_2,X} h'_B$:

   Since $h_2$ and $h'_B$ are restrictions of already related heaps, it suffices to show that

   - $dom(h_2) \cap s(X) = dom(h'_B) \cap s(X)$;
   - if $|dom(h_2) - s(X)| < n_2$, then $|dom(h_2) - s(X)| = |dom(h'_B) - s(X)|$; and
   - if $|dom(h_2) - s(X)| \geq n_2$, then $|dom(h'_B) - s(X)| \geq n_2$.

   For the first bullet,

$$\begin{array}{ll} & dom(h'_B) \cap s(X) \\ = & \\ & (B \cup (dom(h_2) \cap s(X))) \cap s(X) \\ = & \\ & (B \cap s(X)) \cup (dom(h_2) \cap s(X)) \\ = & \quad \because B \subseteq dom(h') - s(X) \\ & dom(h_2) \cap s(X) \end{array}$$

   For the other two, we notice that $dom(h'_B) - s(X) = B$.

$$
\begin{array}{ll}
& dom(h'_B) - s(X) \\
= & \\
& (B \cup (dom(h_2) \cap s(X))) - s(X) \\
= & \\
& B - s(X) \\
= & \qquad \because B \subseteq dom(h') - s(X) \\
& B
\end{array}
$$

When $|dom(h) - s(X)| < m$ or $|dom(h_2) - s(X)| < n_2$, $|B| = |dom(h_2) - s(X)|$. Therefore, the two requirements are satisfied. The only remaining case is when $|dom(h) - s(X)| \geq m$ and $|dom(h_2) - s(X)| \geq n_2$. We show that $|B| \geq n_2$. Note that $|A| = min(|dom(h_1) - s(X)|, n_1)$ by the choice of $A$ in this case. Therefore,

$$
\begin{array}{lll}
|B| & = & |dom(h) - s(X)| - |A| \\
& \geq & m - |A| \\
& \geq & m - n_1 \\
& \geq & n_2
\end{array}
$$

□

**Lemma 93** *If $h \sim_{s,n,X} h'$, $h \# h_1$ and $h' \# h_1$, then $h * h_1 \sim_{s,n,X} h' * h_1$.*

*Proof:* We check each condition in the definition of $h * h_1 \sim_{s,n,X} h' * h_1$.

1. $s(X) \cap dom(h * h_1) = s(X) \cap dom(h' * h_1)$:

$$
\begin{array}{ll}
& s(X) \cap dom(h * h_1) \\
= & \\
& (s(X) \cap dom(h)) \cup (s(X) \cap dom(h_1)) \\
= & \qquad \because h \sim_{s,n,X} h' \\
& (s(X) \cap dom(h')) \cup (s(X) \cap dom(h_1)) \\
= & \\
& s(X) \cap dom(h' * h_1)
\end{array}
$$

2. for all $l \in s(X) \cap dom(h * h_1)$, $(h * h_1)(l) =_{s(X)} (h' * h_1)(l)$:

   For such $l$, $l \in s(X) \cap dom(h)$ or $l \in s(X) \cap dom(h_1)$. In the first case, $(h * h_1)(l) = h(l)$ and $(h' * h_1)(l) = h'(l)$, and $h(l) =_{s(X)} h'(l)$ since $h \sim_{s,n,X} h'$. Therefore, $(h * h_1)(l) =_{s(X)}$

$(h' * h_1)(l)$. In the second case, $(h * h_1)(l) = h_1(l) = (h' * h_1)(l)$. Since $=_{s(X)}$ is reflexive, $(h * h_1)(l) =_{s(X)} (h' * h_1)(l)$ also holds.

3. if $|dom(h * h_1) - s(X)| < n$, then $|dom(h * h_1) - s(X)| = |dom(h' * h_1) - s(X)|$:

   When $|dom(h*h_1) - s(X)| < n$, $|dom(h) - s(X)| < n$. Since $h \sim_{s,n,X} h'$, $|dom(h) - s(X)| = |dom(h') - s(X)|$. The conclusion easily follows from this as shown below:

$$
\begin{array}{ll}
 & |dom(h * h_1) - s(X)| \\
= & \\
 & |dom(h) - s(X)| + |dom(h_1) - s(X)| \\
= & \\
 & |dom(h') - s(X)| + |dom(h_1) - s(X)| \\
= & \\
 & |dom(h' * h_1) - s(X)|
\end{array}
$$

4. if $|dom(h * h_1) - s(X)| \geq n$, then $|dom(h' * h_1) - s(X)| \geq n$:

   There are two cases: $|dom(h) - s(X)| < n$ or $|dom(h) - s(X)| \geq n$. In the first case, since $h \sim_{s,n,X} h'$, $|dom(h' * h_1) - s(X)| = |dom(h * h_1) - s(X)| \geq n$. In the second case, $|dom(h' * h_1) - s(X)| \geq |dom(h') - s(X)| \geq n$ because $h \sim_{s,n,X} h'$.

$\square$

The equivalence relation is also preserved with relocating "garbage heap cells", removing some of them and overwriting an "uninteresting value" of each heap cell by another "uninteresting value". Given a subset $V$ of $Val$ and $v$ in $Val - V - \{nil\}$, we use $prun_{V,v}$ to denote a function from $Val \times Val$ to $Val \times Val$ which overwrites "uninteresting values", i.e., values not in $V \cup \{nil\}$, by another "uninteresting value" $v$:

$$
prun_{V,v}(v_0, v_1) = \begin{cases} (v, v) & \text{if } v_0 \notin (V \cup \{nil\}) \text{ and } v_1 \notin (V \cup \{nil\}) \\ (v, v_1) & \text{if } v_0 \notin (V \cup \{nil\}) \text{ and } v_1 \in (V \cup \{nil\}) \\ (v_0, v) & \text{if } v_0 \in (V \cup \{nil\}) \text{ and } v_1 \notin (V \cup \{nil\}) \\ (v_0, v_1) & \text{if } v_0, v_1 \in (V \cup \{nil\}) \end{cases}
$$

**Lemma 94** *For all $(s, h) \in State$, $m \in \mathbf{Nat}$, $v \in Val - s(X) - \{nil\}$ and all finite subsets $B$ of Loc such that $B \cap s(X) = \emptyset$ and $|B| = m$, there exists $h'$ with the following properties:*

- $h \sim_{s,m,X} h'$;
- $dom(h') - s(X) \subseteq B$;
- *for all* $l \in dom(h') - s(X)$, $h'(l) = (v, v)$; *and*
- *for all* $l \in dom(h') \cap s(X)$, $prun_{s(X),v}(h(l)) = h'(l)$

*Proof:* Pick up a subset $B_0$ of $B$ such that $|B_0| = min(|dom(h) - s(X)|, m)$. Then, define $h'$ such that

$$h'(l) = \begin{cases} (v, v) & \text{when } l \in B_0 \\ prun_{s(X),v}(h(l)) & \text{when } l \in dom(h) \cap s(X) \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Notice that $h'$ is well-defined because $B_0$ and $(dom(h) \cap s(X))$ are disjoint. We show that all conditions for $h \sim_{s,m,X} h'$ are satisfied.

- $dom(h) \cap s(X) = dom(h') \cap s(X)$:

  It holds since $B_0 \cap s(X) = \emptyset$.

- for all $l \in dom(h) \cap s(X)$, $h(l) =_{s(X)} h'(l)$:

  By the definition of $h'$, $prun_{s(X),v}(h(l)) = h'(l)$ for all such $l$. $h(l) =_{s(X)} h'(l)$ follows from this since $prun_{s(X),v}(h(l)) =_{s(X)} h(l)$.

- if $|dom(h) - s(X)| < m$, then $|dom(h) - s(X)| = |dom(h') - s(X)|$; and if $|dom(h) - s(X)| \geq m$, then $|dom(h') - s(X)| \geq m$:

  Since $B_0$ is defined to be $min(|dom(h) - s(X)|, m)$, it suffices to show that $dom(h') - s(X) = B_0$, which is, in fact, the case by the construction of $h'$.

□

The most important property of the equivalence relation is that any assertions can't tell apart two equivalent heaps:

**Proposition 95** . *Given heaps* $h, h'$, *a stack* $s$, *a predicate* $P$ *and a set* $X$ *of variables with* $FV(P) \subseteq X$, *if* $h \sim_{s,|P|,X} h'$ *and* $(s, h) \models P$, *then* $(s, h') \models P$.

*Proof:* We use induction on the structure of $P$.

- $P \equiv (E \mapsto E_1, E_2)$:

$$
\begin{array}{ll}
 & (s,h) \models (E \mapsto E_1, E_2) \\
\Longrightarrow & \\
 & dom(h) = \{[\![E]\!](s)\} \ \wedge \ h([\![E]\!](s)) = ([\![E_1]\!](s), [\![E_2]\!](s)) \\
\Longrightarrow & \qquad \because [\![E]\!](s) \in s(X) \ \wedge \ [\![E_i]\!](s) \in s(X) \cup \{nil\} \ \wedge \ h \sim_{s,1,X} h' \\
 & dom(h') = \{[\![E]\!](s)\} \ \wedge \ h'([\![E]\!](s)) = ([\![E_1]\!](s), [\![E_2]\!](s)) \\
\Longrightarrow & \\
 & (s,h') \models (E \mapsto E_1, E_2)
\end{array}
$$

- $P \equiv (E_1 = E_2)$:

$$
\begin{array}{ll}
 & (s,h) \models (E_1 = E_2) \\
\Longrightarrow & \\
 & [\![E_1]\!](s) = [\![E_2]\!](s) \\
\Longrightarrow & \\
 & (s,h') \models (E_1 = E_2)
\end{array}
$$

- $P \equiv \mathsf{F}$:

  In this case, $(s,h) \models \mathsf{F}$ never holds.

- $P \equiv Q \Rightarrow R$:

$$
\begin{array}{ll}
 & (s,h) \models Q \Rightarrow R \\
\Longrightarrow & \\
 & (s,h) \models Q \text{ implies } (s,h) \models R \\
\Longrightarrow & \qquad \because \text{the induction hypothesis and } \sim_{s,|P|,X} \text{ is symmetric} \\
 & (s,h') \models Q \text{ implies } (s,h') \models R \\
\Longrightarrow & \\
 & (s,h') \models Q \Rightarrow R
\end{array}
$$

- $P \equiv Q * R$:

$(s,h) \models Q * R$

$\Longrightarrow$

$\exists h_0, h_1.\ h_0 \# h_1\ h = h_0 * h_1,\ (s,h_0) \models Q$ and $(s,h_1) \models R$

$\Longrightarrow \quad \because |P| = |Q| + |R| \ \wedge$ Lemma 92

$\exists h'_0, h'_1, h_0, h_1.$

$h'_0 \# h'_1 \ \wedge \ h' = h'_0 * h'_1$

$\wedge \ h_0 \sim_{s,|Q|,X} h'_0 \ \wedge \ h_1 \sim_{s,|R|,X} h'_1$

$\wedge \ (s,h_0) \models Q \ \wedge \ (s,h_1) \models R$

$\Longrightarrow \quad \because$ the induction hypothesis

$\exists h'_0, h'_1.\ h'_0 \# h'_1,\ h' = h'_0 * h'_1,\ (s,h'_0) \models Q$ and $(s,h'_1) \models R$

$\Longrightarrow$

$(s,h') \models Q * R$

- $P \equiv Q -\!\!* R$:

For all heaps $h'_1$ such that $(s,h'_1) \models Q$ and $h'_1 \# h'$, we have to show that $(s, h' * h'_1) \models Q$. Given such $h'_1$, let's choose a subset $B$ of $Loc - dom(h) \cup dom(h') \cup s(X)$ such that $|B| = max(|Q|,|R|)$, which is possible because $Loc$ is a countably infinite set and $dom(h) \cup dom(h') \cup s(X)$ is finite. By Lemma 94, there exists $h_1$ such that $h_1 \sim_{s,max(|Q|,|R|),X} h'_1$ and $dom(h_1) - s(X) \subseteq B$. Note that $(dom(h_1) - s(X)) \cap (dom(h) \cup dom(h')) = \emptyset$ by the choice of $B$, which we use in the following:

$$
\begin{aligned}
& dom(h_1) \cap (dom(h) \cup dom(h')) \\
= \quad & \\
& ((dom(h_1) \cap s(X)) \cup (dom(h_1) - s(X))) \cap (dom(h) \cup dom(h')) \\
= \quad & \quad \because (dom(h_1) - s(X)) \cap (dom(h) \cup dom(h')) = \emptyset \\
& (dom(h_1) \cap s(X)) \cap (dom(h) \cup dom(h')) \\
= \quad & \quad \because h_1 \sim_{s,max(|Q|,|R|),X} h'_1 \\
& (dom(h'_1) \cap s(X)) \cap (dom(h) \cup dom(h')) \\
= \quad & \\
& (dom(h'_1) \cap s(X) \cap dom(h)) \cup (dom(h'_1) \cap s(X) \cap dom(h')) \\
= \quad & \quad \because h \sim_{s,|P|,X} h' \\
& (dom(h'_1) \cap s(X) \cap dom(h')) \cup (dom(h'_1) \cap s(X) \cap dom(h')) \\
= \quad & \quad \because h' \# h'_1 \\
& \emptyset
\end{aligned}
$$

That is, $h \# h_1$ and $h' \# h_1$. Therefore, we have $h * h_1 \sim_{s,|R|,X} h' * h_1$ by Lemma 93. We also have $h' * h_1 \sim_{s,|R|,X} h' * h'_1$ by the same lemma because $h_1 \sim_{s,|R|,X} h'_1$, $h' \# h_1$ and $h' \# h'_1$. So, $h * h_1 \sim_{s,|R|,X} h' * h'_1$. Now, for obtaining the goal, which is to show $(s, h' * h'_1) \models R$, we only need to prove that $(s, h * h_1) \models R$; then, the conclusion follows by induction. Since $h_1 \sim_{s,|Q|,X} h'_1$, by the induction hypothesis, $(s, h_1) \models Q$. Now, it follows from $(s, h) \models Q \mathrel{-\!\!*} R$ that $(s, h * h_1) \models R$.

- $P \equiv \mathsf{emp}$:

$$
\begin{aligned}
& (s, h) \models \mathsf{emp} \\
\Longrightarrow \quad & \\
& dom(h) = \emptyset \\
\Longrightarrow \quad & \quad \because h \sim_{s,1,X} h' \\
& dom(h') \cap s(X) = \emptyset \ \wedge \ |dom(h') - s(X)| = 0 \\
\Longrightarrow \quad & \\
& dom(h') = \emptyset \\
\Longrightarrow \quad & \\
& (s, h') \models \mathsf{emp}
\end{aligned}
$$

$\square$

We have developed enough properties of the equivalence relation $\sim_{s,n,X}$ to show Proposition 89, which says that in the interpretation of $(s,h) \models Q \mathrel{-\!\!*} R$, it suffices to consider heaps only in a certain finite set. We show that for all $h_0$ with $h \# h_0$ and $(s, h_0) \models Q$, there is a heap $h_1$ in the finite set, i.e. satisfying four conditions in the proposition, such that $h_1 * h$ is related to $h_0 * h$ by $\sim_{s,|R|,X}$. Then, Proposition 95 gives the conclusion: $(s, h * h_1) \models R$ by the assumption, so $(s, h*h_0) \models R$ because $h_0 * h \sim_{s,|R|,X} h_1 * h$ holds (Proposition 95). We use Lemma 94 to get $h_1$ which is related to $h_0$ by $\sim_{s,max(|Q|,|R|),X}$ and satisfies the third and the fourth conditions in the proposition, which are $dom(h_1) - s(X) \subseteq B$ and $h_1(l) \in (s(X) \cup \{nil, v\}) \times (s(X) \cup \{nil, v\})$ for all $l \in dom(h_1)$. Note that $h_1$ is related to $h_0$ by both $\sim_{s,|Q|,X}$ and $\sim_{s,|R|,X}$ since both $|Q|$ and $|R|$ are less than or equal to $max(|Q|, |R|)$. So, $(s, h_1) \models Q$ by Proposition 95, i.e., $h_1$ satisfies the second condition in the proposition. In order to show that $h_1$ is in the finite set, it remains to prove the first condition, $h_1 \# h$. Note again that the additional requirement, $h_0 * h \sim_{s,|R|,X} h_1 * h$, also follows from $h_1 \# h$ because $h_0 \sim_{s,|R|,X} h_1$ and $h_0 \# h$ (Lemma 93) The following shows that $h_1 \# h$ holds:

$$\begin{array}{ll}
 & dom(h_1) \cap dom(h) \\
= & \\
 & \Big((dom(h_1) - s(X)) \cup (dom(h_1) \cap s(X))\Big) \cap dom(h) \\
= & \quad \because h_0 \sim_{s,|Q|,X} h_1 \\
 & \Big((dom(h_1) - s(X)) \cup (dom(h_0) \cap s(X))\Big) \cap dom(h) \\
\subseteq & \quad \because dom(h_1) - s(X) \subseteq B \ \wedge \ h_0 \# h \\
 & B \cap dom(h) \\
= & \quad \because B \subseteq Loc - (dom(h) \cup s(X)) \\
 & \emptyset
\end{array}$$

We move on to the next question: given a stack $s$, is it decidable to check $(s,h) \models P$ holds for all $h$? The following lemma answers it.

**Proposition 96** *Given a stack $s$ and an assertion $P$, let $X$ be $FV(P)$ and $B$ a finite set consisting of the first $|P|$ elements of $Loc - s(X)$ where the ordering is given by* ord. *Pick $v \in Val - \{nil\} - s(X)$. Then, in order to see that $(s,h) \models P$ holds for all $h$, it suffices only to consider the following $h$'s:*

- *$dom(h) \subseteq B \cup s(X)$; and*

- *for all $l$ in $dom(h)$, $h(l)$ is in $(s(X) \cup \{nil, v\}) \times (s(X) \cup \{nil, v\})$.*

*Proof:* Suppose that $(s, h) \not\models P$ for a heap $h$. Then, by Lemma 94, there is $h'$ with two conditions in the lemma such that $h' \sim_{s,|P|,X} h$. Then, by Proposition 95, $(s, h') \not\models P$, either. This contradicts to the assumption. $\square$

The proposition implies the decidability result since we can write a program which takes $s$ and $P$ as an input and checks whether $(s, h) \models P$ for all heaps satisfying two conditions in the proposition.

**Corollary 97** *Given a stack $s$ and an assertion $P$, checking $(s, h) \models P$ for all $h$ is decidable.*

The final step is to show the decidability of checking $(s, h) \models P$ for all states $(s, h)$. We observe that the actual values of variables are not relevant to the truth of an assertion as long as the "relationship" of the values remains the same. In the following series of lemmas, we present the observation precisely. For two functions $f : A \to B$ and $g : C \to D$, we use $f \times g$ to denote a function from $A \times C$ to $B \times D$ given by $(f \times g)(a, b) = (f(a), g(b))$. We define a relation $\approx_X$. to capture the word "relationship" above. Intuitively, two states are related by $\approx_X$ iff the relationship of the values, which are stored in variables in $X$ or in heap cells, are the same for the two states.

**Definition 98 ($\approx_X$)** *For states $(s, h)$ and $(s', h')$ and a subset $X$ of $Var$, $(s, h) \approx_X (s', h')$ iff there exists a bijection $r$ from $Val$ to $Val$ such that*

- *$r(nil) = nil$ and $r$ restricts to a bijection from $Loc$ to $Loc$;*
- *for all $x \in X$, $r(s(x)) = s'(x)$; and*
- *$(r \times r) \circ h \circ (r|_{Loc})^{-1} = h'$.*

Notice that $\approx_X$ is an equivalence relation. Also, note that the last condition is equivalent to any of the following three because $r$ is a bijection:

- $(r \times r) \circ h = h' \circ (r|_{Loc})$;
- $h = (r^{-1} \times r^{-1}) \circ h' \circ (r|_{Loc})$; and
- $h \circ (r|_{Loc})^{-1} = (r^{-1} \times r^{-1}) \circ h'$

**Lemma 99** *Given states* $(s,h),(s',h')$ *and an expression* $E$*, if* $(s,h) \approx_Y (s',h')$ *by a bijection* $r\colon Val \to Val$ *and* $Y \supseteq FV(E)$*, then* $r(\llbracket E \rrbracket(s)) = \llbracket E \rrbracket(s')$*.*

**Proposition 100** *For all states* $(s,h)$ *and* $(s',h')$ *and all assertions* $P$ *such that* $(s,h) \approx_{FV(P)} (s',h')$*, if* $(s,h) \models P$*, then* $(s',h') \models P$*.*

*Proof:* Let $r$ be the bijection in the definition of $(s,h) \approx_{FV(P)} (s',h')$. We use induction on the structure of $P$.

- $P \equiv (E \mapsto E_1, E_2)$:

$$\begin{array}{ll} & (s,h) \models (E \mapsto E_1, E_2) \\ \Longrightarrow & \\ & dom(h) = \{\llbracket E \rrbracket(s)\} \ \wedge \ h(\llbracket E \rrbracket(s)) = (\llbracket E_1 \rrbracket(s), \llbracket E_2 \rrbracket(s)) \\ \Longrightarrow & \qquad \because (s,h) \approx_{FV(P)} (s',h') \\ & dom(h') = \{r(\llbracket E \rrbracket(s))\} \ \wedge \ h'(r(\llbracket E \rrbracket(s))) = (r(\llbracket E_1 \rrbracket(s)), r(\llbracket E_2 \rrbracket(s))) \\ \Longrightarrow & \qquad \because \text{Lemma 99} \\ & dom(h') = \{\llbracket E \rrbracket(s')\} \ \wedge \ h'(\llbracket E \rrbracket(s')) = (\llbracket E_1 \rrbracket(s'), \llbracket E_2 \rrbracket(s')) \end{array}$$

- $P \equiv (E_1 = E_2)$:

$$\begin{array}{ll} & (s,h) \models E_1 = E_2 \\ \Longrightarrow & \\ & \llbracket E_1 \rrbracket(s) = \llbracket E_2 \rrbracket(s) \\ \Longrightarrow & \\ & r(\llbracket E_1 \rrbracket(s)) = r(\llbracket E_2 \rrbracket(s)) \\ \Longrightarrow & \qquad \because \text{Lemma 99} \\ & \llbracket E_1 \rrbracket(s') = \llbracket E_2 \rrbracket(s') \end{array}$$

- $P \equiv \mathsf{F}$:

  $(s,h) \models \mathsf{F}$ never holds.

- $P \equiv Q \Rightarrow R$:

$$(s,h) \models Q \Rightarrow R$$

$$\Longrightarrow$$

$$(s,h) \models Q \text{ implies } (s,h) \models R$$

$$\Longrightarrow \qquad \because \text{the induction hypothesis and the symmetry of } \approx_{FV(Q)}$$

$$(s',h') \models Q \text{ implies } (s',h') \models R$$

$$\Longrightarrow$$

$$(s',h') \models Q \Rightarrow R$$

- $P \equiv \mathsf{emp}$:

  In this case, we should show that $dom(h') = \emptyset$. Since $(s,h) \approx_{FV(P)} (s',h')$ by $r$, it suffices show that $dom((r \times r) \circ h \circ (r|_{Loc})^{-1}) = \emptyset$. Notice that $r \times r$ and $(r|_{Loc})^{-1}$ are bijective. Therefore, the above is equivalent to $dom(h) = \emptyset$, which is the case since $(s,h) \models \mathsf{emp}$.

- $P \equiv Q * R$:

  Since $(s,h) \models Q * R$, there exist $h_1, h_2$ such that $h_1 \# h_2$, $h = h_1 * h_2$, $(s,h_1) \models Q$ and $(s,h_2) \models R$. Let $h'_i$ be $(r \times r) \circ h_i \circ (r|_{Loc})^{-1}$ for $i = 1,2$. Then, it suffices to show that $h'_1 \# h'_2$, $h = h'_1 * h'_2$, $(s,h_1) \approx_{FV(Q)} (s',h'_1)$ by $r$ and $(s,h_2) \approx_{FV(R)} (s',h'_2)$ by $r$. The conclusion follows by the induction hypothesis.

  - $h'_1 \# h'_2$:

    $$dom(h'_1) \cap dom(h'_2)$$

    $$=$$

    $$dom((r \times r) \circ h_1 \circ (r|_{Loc})^{-1}) \cap dom((r \times r) \circ h_2 \circ (r|_{Loc})^{-1})$$

    $$= \qquad \because r \times r \text{ is total}$$

    $$dom(h_1 \circ (r|_{Loc})^{-1}) \cap dom(h_2 \circ (r|_{Loc})^{-1})$$

    $$= \qquad \because r \text{ is bijective}$$

    $$r(dom(h_1)) \cap r(dom(h_2))$$

    $$= \qquad \because r \text{ is bijective}$$

    $$r(dom(h_1) \cap dom(h_2))$$

    $$=$$

    $$\emptyset$$

  - $h = h'_1 * h'_2$:

Note that $dom(h_1') \cup dom(h_2') = r(dom(h_1) \cup dom(h_2))$ by the similar calculation as above, and so, $dom(h_1') \cup dom(h_2') = r(dom(h)) = dom(h')$. Therefore, it suffices to show that for all $l \in dom(h_1')$, $h_1'(l) = h'(l)$ and for all $l \in dom(h_2')$, $h_2'(l) = h'(l)$. We only show the first; the other is similar.

$$\begin{array}{rcl} h_1'(l) & = & (r \times r)(h_1((r|_{Loc})^{-1}(l))) \\ & = & (r \times r)(h((r|_{Loc})^{-1}(l))) \\ & = & h'(l) \end{array}$$

– $(s, h_1) \approx_{FV(Q)} (s', h_1')$ by $r$:

Since $(s, h) \approx_{FV(Q*R)} (s', h')$, the conditions on stacks hold. And by definition of $h_1'$, $h_1' = (r \times r) \circ h_1 \circ (r|_{Loc})^{-1}$.

– $(s, h_2) \approx_{FV(Q)} (s', h_2')$ by $r$:

The case is similar to the previous one.

- $P \equiv Q \mathrel{-\!\!*} R$:

Let $h_1'$ be a heap such that $h' \# h_1'$ and $(s', h_1') \models Q$. We should show that $(s', h' * h_1') \models Q$. Define $h_1$ as $(r^{-1} \times r^{-1}) \circ h_1' \circ (r|_{Loc})$. Then, $(s', h_1') \approx_{FV(Q)} (s, h_1)$ by $r$ and so $(s, h_1) \models Q$ by the induction hypothesis. Note that $h \# h_1$ holds because

$$\begin{array}{ll} & dom(h) \cap dom(h_1) \\ = & \\ & dom((r^{-1} \times r^{-1}) \circ h' \circ (r|_{Loc})) \cap dom((r^{-1} \times r^{-1}) \circ h_1' \circ (r|_{Loc})) \\ = & \quad \because r^{-1} \times r^{-1} \text{ is total} \\ & dom(h' \circ (r|_{Loc})) \cap dom(h_1' \circ (r|_{Loc})) \\ = & \quad \because r^{-1} \text{ is bijective} \\ & r^{-1}(dom(h')) \cap r^{-1}(dom(h_1')) \\ = & \quad \because r^{-1} \text{ is bijective} \\ & r^{-1}(dom(h') \cap dom(h_1')) \\ = & \\ & \emptyset \end{array}$$

So, we also know that $(s, h * h_1) \models R$. Now, it suffices to show $(s, h * h_1) \approx_{FV(R)} (s', h' * h_1')$ by $r$; then, the conclusion follows by the induction hypothesis. We note that all requirements on stacks hold since $(s, h) \approx_{FV(Q \mathrel{-\!\!*} R)} (s', h')$ by $r$ by assumption. So, the only step which we need to show is that

$$h' * h'_1 = (r \times r) \circ (h * h_1) \circ (r|_{Loc})^{-1}$$

The following shows it:

$$
\begin{array}{ll}
& h' * h'_1 \\
= & \\
& \Big((r \times r) \circ h \circ (r|_{Loc})^{-1}\Big) * \Big((r \times r) \circ h_1 \circ (r|_{Loc})^{-1}\Big) \\
= & \quad \because (r \times r) \text{ is total} \\
& (r \times r) \circ \Big((h \circ (r|_{Loc})^{-1}) * (h_1 \circ (r|_{Loc})^{-1})\Big) \\
= & \quad \because (r|_{Loc})^{-1} \text{ is bijective} \\
& (r \times r) \circ (h * h_1) \circ (r|_{Loc})^{-1}
\end{array}
$$

□

**Lemma 101** *Given a state $(s, h)$ and an assertion $P$, let $B$ be the set consisting of the first $|FV(P)|$ locations in Loc, where the ordering is given by* ord*, and all natural numbers from* 1 *to $|FV(P)|$. Then, there exists a state $(s', h')$ such that*

- $s'(Var - FV(P)) \subseteq \{nil\}$*;*
- $s'(FV(P)) \subseteq B \cup \{nil\}$*; and*
- $(s, h) \approx_{FV(P)} (s', h')$*.*

*Proof:* Since $|s(FV(P))| \leq |FV(P)| = (|B|/2)$, there is a bijection $r$ from *Val* to *Val* which maps all values in $s(FV(P))$ to values in $B \cup \{nil\}$, restricts to a bijection from *Loc* to *Loc* and maps *nil* to itself. Define $s'$ and $h'$ as follows:

$$h' = (r \times r) \circ h \circ (r|_{Loc})^{-1} \text{ and } s'(x) = \begin{cases} r(s(x)) & \text{if } x \in FV(P) \\ nil & \text{otherwise} \end{cases}$$

Then, $s'(FV(P)) \subseteq B \cup \{nil\}$ by the choice of $r$, and $s'(Var - FV(P)) \subseteq \{nil\}$ by the definition of $s'$. We also have $(s, h) \approx_{FV(P)} (s', h')$ by $r$ because for all $x \in FV(P)$, $s'(x) = r(s(x))$. □

The decidability result follows from the above lemma. To see the reason, we note that because of the lemma, for all assertions $P$, there is a finite set of stacks such that if for all stacks $s$ in the set and all heaps $h$, $(s, h) \models P$, then $P$ holds for all states whose stack is not necessarily in the set. Therefore, a decision algorithm is obtained by exhaustively checking for each stack $s$ in such a finite set whether $(s, h) \models P$ holds for all heaps $h$ using the algorithm in Corollary 97.

**Corollary 102** *Given an assertion $P$, checking $(s, h) \models P$ for all states $(s, h)$ is decidable.*

# PART III

# Local Reasoning for Programs with Procedures and Pointers

# CHAPTER 9

## Programs with Procedures and Pointers

Part III concerns local reasoning for programming languages with both a procedure mechanism and language constructs for heap storage. Recall that the idea of local reasoning is realized in program verification mainly by having a rule to change the mode of reasoning from local to global. When a programming language doesn't have a procedure mechanism, the Frame Rule in Chapter 5 is such a rule: it allows one to derive a global fact from a local one by incorporating a property for an unaccessed portion of the store as an invariant. It is:

Frame Rule

$$\frac{\{P\}\,C\,\{Q\}}{\{P*R\}\,C\,\{Q*R\}}\;(\mathit{ModifyOnly}(C)\cap FV(R)=\emptyset)$$

However, the presence of a procedure mechanism makes the Frame Rule in Part II unsound; it is because the simple syntactic checking in the side condition no longer ensures that $C$ doesn't write to any variables which might affect the truth of $R$. For instance, when identifiers $x$ and $y$ are aliased so that they denote the same variable, $x := 1$ changes the truth of $y = 2$ even though $\mathit{ModifyOnly}(x := 1)\cap FV(y = 2) = \emptyset$. In this chapter, we formulate the Frame Rule for programs with procedures and pointers and justify it with a formal semantics. The main idea is to use the symmetric noninterference predicate in Part I to ensure that $C$ doesn't modify any variables which might affect the truth of $R$. The rule becomes:

Frame Rule

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

One might worry that upon combining the symmetric noninterference $\perp$ and the spatial conjunction $*$, certain important reasoning principles for either of them become unsound because of their undesirable interaction. The formal semantics in this chapter shows that this is not the case: all important reasoning principles for the symmetric noninterference predicate and the **BI** assertion language hold; and all axioms in **BI** pointer logic still hold with slight modifications. The formal semantics is obtained by combining ideas in Part I and II: as in Chapters 3 and 4, it uses sheaves to obtain a set of variables which are actually accessed by a "program"; assertions are interpreted in the same way as in Chapter 5; and following Chapter 5, the interpretation of a Hoare triple has an additional requirement to ensure that a well-specified program doesn't go wrong. From a technical point of view, the semantics also improves upon that in Chapters 3 and 4 in two aspects: firstly, we consider the issue of passivity so that two program phrases $M$ and $N$ do not interfere even when $M$ and $N$ access the same variable as long as the variable is only read by them; secondly, the general recursion operator is included in the programming language so that the framework is shifted from sets to cpos. Both issues are not included in Chapters 3 and 4 in order to simplify the presentation. However, the techniques in this chapter are equally applicable to Chapters 3 and 4.

We start this chapter by explaining Idealized Algol with heap storage, which is obtained by combining Idealized Algol from Chapter 2 with **WHILE**$\mathsf{h}$ in Chapter 5. In the same section, we describe an assertion language and a specification language: the assertion language is a straightforward adaptation of the **BI** assertion language in Chapter 5, and the specification language is a fragment of Reynolds' specification logic. In Section 9.2, we consider reasoning principles for the specification language in a natural deduction style. Section 9.3 includes a categorical semantics of the programming language, the assertion language and the specification language, which validate all reasoning principles in Section 9.2. We focus on important reasoning principles for local reasoning, which are axioms for the symmetric noninterference predicate and the Frame Rule. In this chapter, we assume that the reader is familiar with basic notions of category theory and sheaf theory. A brief review of sheaf theory can be found in Chapter 4.

## 9.1 Syntax

In this chapter, we consider a programming language with both procedures and pointers. The language is essentially Idealized Algol in Chapter 2 with the additional constant terms of **WHILE**$\mathsf{h}$ in Chapter 5 for handling heap storage. That is, the language is a call-by-name typed lambda calculus with imperative base types, such as $\mathsf{com}$, and imperative constant terms, which include constants for reading, updating or creating either variables or heap cells. We call the language *Idealized Algol with heaps* (in short *IAh*). The syntax of IAh is shown in Table 9.1. As in IA, types are stratified into two layers: data types $\delta$ for storable values and phrase types $\theta$ for program terms. In IAh, we take $\mathsf{loc}$, which denotes the set of addresses of all possible heap cells, as the only data type in order to avoid an unnecessary complication for handling recursive definitions of data types. Expression and variable types are chosen accordingly: $\mathsf{exp}$ denotes location expressions and $\mathsf{var}$ variables holding locations. Note that a term of $\mathsf{var}$ type denotes a variable not a heap cell; each heap cell in memory is accessed by an expression, which returns the address of the cell when applied to a state. In addition to the usual imperative base types, IAh has type $\mathsf{val}$ for location values; a term of $\mathsf{val}$ type denotes a location without depending on states. All other phrase types are constructed from these base types, $\mathsf{exp}, \mathsf{var}, \mathsf{com}, \mathsf{val}$, with function and product type constructors. Terms in IAh are the standard ones in a typed lambda calculus with recursion except constant terms for handling states. The constants $\mathsf{get}, \_ := \_, \mathsf{new}$ are for reading, updating and allocating a variable, respectively. The corresponding constants for heap cells are $\_ := \_.i, \_.i := \_$ and $\_ := \mathsf{cons}(\_, \_)$: command $V := E.i$ dereferences a heap cell $E$ and assigns to a variable $V$ the value of the $i$th field of the cell; $E.i := E'$ updates the $i$th field of the heap cell $E$ by $E'$; and $V := \mathsf{cons}(E_1, E_2)$ allocates a new heap cell holding $(E_1, E_2)$ and stores its address in a variable $V$. Note that all such constant terms for accessing heap storage have the command type as their result type. So, only terms of command type can depend on the heap. The constant $\mathsf{nil}$ is a particular "location" which should not be used to allocate a heap cell. The conditional statement has equality built-in (therefore, its name $\mathsf{ifeq}$) because $\mathsf{loc}$ is the only data type. As in IA, we have the usual restriction on two branches of the conditional in order to ensure that only terms of $\mathsf{com}$ type or $\mathsf{exp}$ type depend on states. Finally, $\mathsf{coerce}$ is a coercion of a location value to a location expression. We often omit $\mathsf{get}$ and $\mathsf{coerce}$ whenever the omission is clear from the context. When a term $M$ has a type $\theta$ in a context $\Gamma$, which is just a finite function from identifiers to phrase types, we write $\Gamma \triangleright M : \theta$.

The assertion language is essentially the same as in Chapter 5. It is first-order classical logic with equality, a points-to relation and the spatial connectives $\mathsf{emp}, *$ and $-\!\!*$. Assertion $(E \mapsto E_1, E_2)$ denotes that a heap cell $E$ is allocated and contains $(E_1, E_2)$, and moreover it is the only allocated heap cell; $\mathsf{emp}$ denotes an empty heap; $P * Q$ holds in a heap if the heap

**Types:**

| | | | |
|---|---|---|---|
| data type | $\delta$ | $::=$ | $\mathsf{loc}$ |
| phrase type | $\theta$ | $::=$ | $\mathsf{exp} \mid \mathsf{var} \mid \mathsf{com} \mid \mathsf{val} \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2$ |

**Preterms:**

$$M \quad ::= \quad c \mid x \mid \lambda x{:}\,\theta.\, M \mid M_0\, M_1 \mid \langle M_0,\, M_1\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M$$

**Constant Terms:**

$$\begin{array}{rcl}
\mathsf{Y} & : & (\theta \to \theta) \to \theta \\
\mathsf{skip} & : & \mathsf{com} \\
_;_ & : & \mathsf{com} \times \mathsf{com} \to \mathsf{com} \\
\mathsf{get} & : & \mathsf{var} \to \mathsf{exp} \\
_ := _ & : & \mathsf{var} \times \mathsf{exp} \to \mathsf{com} \\
\mathsf{new} & : & (\mathsf{var} \to \mathsf{com}) \to \mathsf{com} \\
_ := _.1 & : & \mathsf{var} \times \mathsf{exp} \to \mathsf{com} \\
_ := _.2 & : & \mathsf{var} \times \mathsf{exp} \to \mathsf{com} \\
_.1 := _ & : & \mathsf{exp} \times \mathsf{exp} \to \mathsf{com} \\
_.2 := _ & : & \mathsf{exp} \times \mathsf{exp} \to \mathsf{com} \\
_ := \mathsf{cons}(_,_) & : & \mathsf{var} \times \mathsf{exp} \times \mathsf{exp} \to \mathsf{com} \\
\mathsf{nil} & : & \mathsf{exp} \\
\mathsf{ifeq} & : & \mathsf{exp} \times \mathsf{exp} \times \beta \times \beta \to \beta \\
\mathsf{coerce} & : & \mathsf{val} \to \mathsf{exp}
\end{array}$$

where $\beta = \mathsf{exp}$ or $\mathsf{com}$.

**Table 9.1** Syntax of Idealized Algol with Heaps

$$\frac{\Gamma \triangleright M\colon \mathsf{exp} \quad \Gamma \triangleright M_0\colon \mathsf{exp} \quad \Gamma \triangleright M_1\colon \mathsf{exp}}{\Gamma \triangleright (M \mapsto M_0, M_1)\colon \mathsf{assert}} \qquad \frac{\Gamma \triangleright M_0\colon \mathsf{exp} \quad \Gamma \triangleright M_1\colon \mathsf{exp}}{\Gamma \triangleright M_0 = M_1\colon \mathsf{assert}}$$

$$\frac{}{\Gamma \triangleright \mathsf{F}\colon \mathsf{assert}} \qquad \frac{\Gamma \triangleright P_0\colon \mathsf{assert} \quad \Gamma \triangleright P_1\colon \mathsf{assert}}{\Gamma \triangleright P_0 \Rightarrow P_1\colon \mathsf{assert}} \qquad \frac{\Gamma, x\colon \mathsf{val} \triangleright P\colon \mathsf{assert}}{\Gamma \triangleright \forall x.\, P\colon \mathsf{assert}}$$

$$\frac{}{\Gamma \triangleright \mathsf{emp}\colon \mathsf{assert}} \qquad \frac{\Gamma \triangleright P_0\colon \mathsf{assert} \quad \Gamma \triangleright P_1\colon \mathsf{assert}}{\Gamma \triangleright P_0 * P_1\colon \mathsf{assert}} \qquad \frac{\Gamma \triangleright P_0\colon \mathsf{assert} \quad \Gamma \triangleright P_1\colon \mathsf{assert}}{\Gamma \triangleright P_0 \mathrel{-\!\!*} P_1\colon \mathsf{assert}}$$

**Table 9.2** Syntax of the Assertion Language

can be split into two so that $P$ holds for one part and $Q$ for the other; and $P \mathrel{-\!\!*} Q$ holds in a heap if adding a new heap doesn't make $Q$ false as long as the new heap satisfies $P$. We write $\Gamma \triangleright P\colon \mathsf{assert}$ to denote that $P$ is a well-formed assertion in a context $\Gamma$; the question of well-formedness arises because in the points-to and the equality predicates, all terms are required to be of $\mathsf{exp}$ type. The syntax for assertions is shown in Table 9.2. Note that other connectives in first-order classical logic are definable from $\Rightarrow, \mathsf{F}, \forall x$: $\neg P \triangleq (P \Rightarrow \mathsf{F})$, $\mathsf{T} \triangleq \neg \mathsf{F}$, $P \wedge Q \triangleq \neg(P \Rightarrow \neg Q)$, $P \vee Q \triangleq \neg(\neg P \wedge \neg Q)$ and $\exists x. P \triangleq \neg \forall x. \neg P$.

A specification about programs is given following Reynolds' specification logic, which is first-order intuitionistic logic with Hoare triples, the symmetric noninterference predicate and the equality predicate as atomic formulas. Note that a specification describes a property of *programs* while an assertion describes a property of *states*. The specification language is shown in Table 9.3, where we write $\Gamma \triangleright \varphi\colon \mathsf{spec}$ to denote that $\varphi$ is a well-formed specification in a context $\Gamma$. The noninterference predicate states that two program phrases use disjoint sets of variables with an exception for a passive use: they might access the same variable as long as the variable is only read by both of them. Note that the noninterference predicate is defined not only for programs but also for assertions. The Hoare triple $\{P\}\, C\, \{Q\}$ is a partial correctness Hoare triple with an additional requirement, as in Chapter 5: it specifies that when started in a state satisfying the precondition, the execution of $C$ doesn't generate memory faults, and moreover, if $C$ ever terminates, the final state satisfies $Q$.

When $\tau$ is either $\theta$ or $\mathsf{assert}$

$$\frac{\Gamma \triangleright M{:}\,\tau \quad \Gamma \triangleright M'{:}\,\tau}{\Gamma \triangleright M \equiv_\tau M'{:}\,\mathsf{spec}}$$

$$\frac{\Gamma \triangleright P{:}\,\mathsf{assert} \quad \Gamma \triangleright P'{:}\,\mathsf{assert} \quad \Gamma \triangleright C{:}\,\mathsf{com}}{\Gamma \triangleright \{P\}\, C\, \{P'\}{:}\,\mathsf{spec}} \qquad \frac{\Gamma \triangleright M{:}\,\tau \quad \Gamma \triangleright M'{:}\,\tau'}{\Gamma \triangleright M \perp_{\tau,\tau'} M'{:}\,\mathsf{spec}}$$

$$\frac{\Gamma \triangleright \varphi{:}\,\mathsf{spec} \quad \Gamma \triangleright \psi{:}\,\mathsf{spec}}{\Gamma \triangleright \varphi \wedge \psi{:}\,\mathsf{spec}} \qquad \frac{\Gamma \triangleright \varphi{:}\,\mathsf{spec} \quad \Gamma \triangleright \psi{:}\,\mathsf{spec}}{\Gamma \triangleright \varphi \Longrightarrow \psi{:}\,\mathsf{spec}} \qquad \frac{\Gamma, x{:}\,\theta \triangleright \varphi{:}\,\mathsf{spec}}{\Gamma \triangleright \forall x{:}\,\theta.\, \varphi{:}\,\mathsf{spec}}$$

**Table 9.3** Syntax of the Specification Language

## 9.2 Reasoning Principles

To reason about specifications, one can freely use all rules in intuitionistic logic with equality or in the equational theory of the call-by-name typed lambda calculus, such as beta and eta equivalences. All rules in **BI** pointer logic are also incorporated in the current framework. However, they need to be modified slightly in order to handle assignment to a *term* of type $\mathsf{var}$ and prevent undesirable sharing of variables. For instance, the axiom for assignment to a variable is changed to

$$\frac{(FV(P) - \{x\}) \perp V}{\{P[E/x]\}\, V := E\, \{P[V/x]\}}$$

where $(FV(P) - \{x\}) \perp V$ is an abbreviation for $x_1 \perp V \wedge \ldots \wedge x_n \perp V$ where $FV(P) - \{x\} = \{x_1, \ldots, x_n\}$. The rule uses an additional identifier $x$ to explain that a term $V$ is replaced by $E$ in the precondition, and it also has a condition to prevent sharing of variables between the assertion $P$ and the variable $V$. Table 9.4 shows modified axioms of **BI** pointer logic in a natural deduction style as well as axioms for the symmetric noninterference predicate.

The Passivity axiom for the noninterference predicate uses passive types, which intuitively denote program phrases which don't modify any variables. So, the rule says that when both program phrases don't write to any variables, they don't interfere symmetrically. Passive types are defined by the following grammar:

$$\text{passive type} \quad \zeta \quad ::= \quad \mathsf{val} \mid \mathsf{exp} \mid \theta \to \zeta \mid \zeta \times \zeta$$

In this chapter, we concentrate on the Frame Rule and axioms for the symmetric noninterference predicates, because it is these rules that formalize the informal idea of local reasoning.

The Frame Rule uses the symmetric noninterference predicate in the premise to ensure that $C$ doesn't update any variables which might affect the truth of $R$. As in Chapter 5, the spatial conjunction and the additional requirement in the Hoare triple $\{P\}\,C\,\{Q\}$ guarantee the other part: $C$ doesn't change any heap cells which $R$ depends on. For the Frame Rule to be effective, reasoning about the symmetric noninterference predicate must be much easier than reasoning about a Hoare triple. Note that the axioms for the noninterference predicate fulfill such a requirement since they only involve simple syntactic decomposition and checking. The axioms also capture properties of programs: Noninterference Decomposition indicates that a term can only access variables which are already accessible by free identifiers in the term; New says that a newly declared variable can not be accessed directly by any nonlocal program phrases; Passivity indicates that when a term eventually returns an expression or a value, it doesn't change any variables. From the axioms for the noninterference predicate, we can derive the first principle of the symmetric noninterference predicate in Chapter 2, which says that if neither a procedure $P$ nor an argument $M$ interferes with $N$, $P(M)$ doesn't interfere with $N$, either.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x\bot N \wedge y\bot N]}{x\bot N} \qquad \dfrac{[x\bot N \wedge y\bot N]}{y\bot N}}{x(y)\bot N}}{x\bot N \wedge y\bot N \Longrightarrow x(y)\bot N}}{\forall x, y.\, x\bot N \wedge y\bot N \Longrightarrow x(y)\bot N}}{P\bot N \wedge M\bot N \Longrightarrow P(M)\bot N}$$

Before closing this section, we illustrate the use of the Frame Rule. The example is from [24]. Suppose that we are given a procedure which sorts a linked list but uses an external procedure *swap* so that the procedure satisfies the following specification:

$$\forall x\colon \mathsf{var}.\, \mathit{swap}\bot x \Longrightarrow \{\mathit{list}(x)\}\, \mathit{sort}(x) \{\mathit{sorted}(x)\}$$

Suppose that both $\mathit{list}(x)$ and $\mathit{sorted}(x)$ don't interfere with a term $M$ if $x\bot M$ and that *sort*'s only free identifier is *swap*. With such specifications, it is possible to derive that if *sort* is called twice for different linked lists, both linked lists are going to be sorted.

$$\dfrac{\dfrac{\dfrac{x\bot \mathit{swap}}{\{\mathit{list}(x)\}\ \mathit{sort}(x)\ \{\mathit{sorted}(x)\}} \quad \dfrac{\dfrac{y\bot x \quad y\bot \mathit{swap}}{y\bot \mathit{sort}(x)}}{\mathit{list}(y)\bot \mathit{sort}(x)}}{\{\mathit{list}(x) * \mathit{list}(y)\}\ \mathit{sort}(x)\ \{\mathit{sorted}(x) * \mathit{list}(y)\}} \quad \dfrac{\dfrac{\mathit{swap}\bot y}{\{\mathit{list}(y)\}\ \mathit{sort}(y)\ \{\mathit{sorted}(y)\}} \quad \dfrac{\dfrac{x\bot y \quad x\bot \mathit{swap}}{x\bot \mathit{sort}(y)}}{\mathit{sorted}(x)\bot \mathit{sort}(y)}}{\{\mathit{sorted}(x) * \mathit{list}(y)\}\ \mathit{sort}(y)\ \{\mathit{sorted}(x) * \mathit{sorted}(y)\}}}{\{\mathit{list}(x) * \mathit{list}(y)\}\ \mathit{sort}(x); \mathit{sort}(y)\ \{\mathit{sorted}(x) * \mathit{sorted}(y)\}}$$

Note that we have conditions on $x$ and $y$ and $swap$, which say that any two of them do not interfere with each other symmetrically. These conditions are important because if $swap$ assigns an arbitrary value to $x$, assertion $sorted(x)$ can't be an invariant for the procedure call $sort(y)$.

## 9.3 Semantics

In this section, we develop a semantics of IAh, the assertion language and the specification language. The challenge is to validate reasoning principles, especially those for local reasoning: the Frame Rule and axioms for the symmetric noninterference predicate. Recall that the soundness of such axioms is subtle even in a simpler setting as noted in Chapters 5 and 3. We show that the ideas in Chapters 5 and 3 still work even in a complex setting, where a programming language includes both procedures and constructs for heap storage and the noninterference predicate allows passive sharing. We start with describing the overall structure of the semantics by explaining the categories used for the semantics. Then, we interpret types and terms and validate important reasoning principles.

### 9.3.1 Semantic Categories

To interpret types in IAh, we use functors from a category of "worlds" to the category of directed complete posets with monotone functions as morphisms, which we call **mCpo**. Each object in the source category, which we call *world*, models the set of variables currently allocated in stack storage, and a morphism between worlds represents a change in the shape of allocated variables, such as allocating new variables or renaming existing variables. So, the interpretation of IAh types specifies what semantic values are available when certain variables are allocated in the stack and how such values are changed when more variables are allocated or existing variables are renamed. We impose conditions on functors to interpret recursion and the symmetric noninterference predicate properly. Each term in IAh is interpreted as a natural transformation between such functors. Similarly, the types for assertions and specifications, $\mathsf{assert}$ and $\mathsf{spec}$, are interpreted as functors from the same category of worlds but to the category **Set** of sets instead of **mCpo**. We use sets because assertions and specifications are not computable entities. In fact, the interpretation of a type in IAh can be seen as one of such functors by composing with a forgetful functor $U$ from **mCpo** to **Set**. Assertions and

Simple Assignment

$$\frac{(FV(P) - \{x\}) \perp V}{\{P[E/x]\}\, V := E\, \{P[V/x]\}}$$

Forward Cons

($x'$ is not free in $P, E_1, E_2$)

$$\frac{(FV(P) \cup FV(E_1) \cup FV(E_2) - \{x\}) \perp V}{\{P[V/x]\}\, V := \mathsf{cons}(E_1[V/x], E_2[V/x])\, \{\exists x'.\, P[x'/x] * (V \mapsto E_1[x'/x], E_2[x'/x])\}}$$

Backward Cons

($x'$ is not free in $P, E_1, E_2$)

$$\frac{(FV(P) - \{x\}) \perp V}{\{\forall x'.(x' \mapsto E_1, E_2) \mathrel{-\!\!*} P[x'/x]\}\, V := \mathsf{cons}(E_1, E_2)\, \{P[V/x]\}}$$

Component Lookup

($x_1, x_2$ are different and not free in $P, E$)

$$\frac{(FV(P) - \{x\}) \perp V}{\{\exists x_1.\, P[x_1/x] \wedge \exists x_2.\, (E \mapsto x_1, x_2) * \mathsf{T}\}\, V := E.1\, \{P[V/x]\}}$$

Frame Rule

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

Noninterference Decomposition

$$\frac{FV(M) \perp N}{M \perp N}$$

Noninterference Symmetry

$$\frac{M \perp N}{N \perp M}$$

New

($x$ is not free in $M, P, Q$)

$$\frac{\forall x\colon \mathsf{var}.\Big((FV(P) \cup FV(Q) \cup FV(M)) \perp x\Big) \Longrightarrow \{P\}\, M(x)\, \{Q\}}{\{P\}\, \mathsf{new}\ M\, \{Q\}}$$

Passivity

(Both $\theta$ and $\theta'$ are passive types)

$$M \perp_{\theta,\theta'} N$$

**Table 9.4** Axioms for Specifications

specifications are interpreted as natural transformations between functors from the category of worlds to **Set**.

The category of worlds, $\mathbf{W}$, models the shape of stack storage and its change. Each object in $\mathbf{W}$, which we call *world*, is a pair $(X_0, X_w)$ of finite sets of variables with $X_w \subseteq X_0$. The set $X_0$ denotes variables currently allocated in the stack, and $X_w$ specifies what variables can be altered. So, all variables in $X_0 - X_w$ are supposed to be used only in a read-only fashion. A morphism in $\mathbf{W}$ stands for a change in the shape of the stack. In $\mathbf{W}$, there are three kinds of changes: renaming variables, allocating new variables and allowing certain read-only variables as modifiable. Formally, each morphism $f: Y \to X$ is an injection from $X_0$ to $Y_0$ such that for all $x$ in $X_w$, $f(x) \in Y_w$; all variables in $Y_0 - f(X_0)$ represent newly allocated variables; $f: X_0 \to f(X_0)$ denotes renaming; and $(Y_w - f(X_w)) \cap f(X_0)$ is the set of variables whose access permission is changed from read-only to modifiable. Notice that the direction of $f$ is the opposite to the change; it is because we interpret each type as a functor from $\mathbf{W}^{op}$.

All types ($\theta$, $\mathsf{assert}$ and $\mathsf{spec}$) are interpreted as sheaves from $\mathbf{W}^{op}$ to **Set** when they are viewed as functors from $\mathbf{W}^{op}$ to **Set**, if necessary, by composing with the forgetful functor $U$ from **mCpo** to **Set**. Such an additional requirement for each type being a sheaf allows a more accurate analysis of program phrases and assertions regarding their use of variables, which leads to the proper notion of symmetric noninterference. For such an interpretation, we use the following Grothendieck topology on the category $\mathbf{W}$. For an object $X$, a sieve $S$ on $X$ is a covering ($S \in J(X)$) iff

- $S$ is nonempty; and
- $X_w = \{x \in X_0 \mid \forall f: Y \to X \in S.\, f(x) \in Y_w\}$.

Note that the second condition says that all future worlds in $S$ combined should precisely explain which variables in $X$ are modifiable.

**Lemma 103** *$J$ satisfies all axioms for a Grothendieck topology.*

*Proof:* For any object $X$ in $\mathbf{W}$, the maximal sieve $t_X = \{f \mid cod(f) = X\}$ satisfies two conditions to be in $J(X)$. So, it remains to show that $J$ satisfies stability and transitivity axioms.

- *Stability*: for all sieves $S$ in $J(X)$ and a morphism $f: Y \to X$, we must show that $f^*(S)$ is in $J(Y)$. We first describe a construction parameterized by a morphism $h: W \to X$ in $S$.

Given a morphism $h: W \to X$ in $S$, we consider an object $K$ and morphisms $g: K \to Y$ and $k: K \to W$ such that

- $K_0$ consists of $|Y_0| + |W_0| - |X_0|$ elements;
- for $y \in Y_0$ and $w \in W_0$, $g(y) = k(w)$ if and only if $y = f(x)$ and $w = h(x)$ for some $x \in X$; and
- $K_w = g(Y_w) \cup k(W_w)$.

Intuitively, the second bullet says that $g$ and $k$ overlap only as much as to make $f \circ g = h \circ k$ hold. Note that such $K$ and two morphisms to $Y$ and $W$ always exist. We also note that $g \in f^*(S)$ because $f \circ g = h \circ k$ and $h \circ k$ in $S$. Another important fact for the construction is that $g(y) \in K_w$ only when either $y \in Y_w$ or there is some $x \in X$ such that $f(x) = y$ and $h(x) \in W_w$; the reason is that if $y$ is not in $Y_w$, $g(y)$ must be in $k(W_w)$. Coming back to the proof of stability, the construction shows that if $S$ is nonempty, so is $f^*(S)$ because we can always find a morphism in $f^*(S)$ from one in $S$ by the construction just described. And for the other condition, suppose that $y$ is mapped to an element in $K'_w$ by all $g': K' \to Y$ in $f^*(S)$ but $y$ itself is not in $Y_w$. Then, for all $h': W' \to X$ in $S$, we can show by applying the construction for $h'$ that there is $x_{h'} \in X_w$ such that $y = f(x_{h'})$ and $h'(x_{h'}) \in W'_w$. Since $S$ is nonempty and $f$ is injective, there is $x \in X$ such that $f(x) = y$ and $h'(x) \in W'_w$ for all $h': W' \to X$ in $S$. Since $S$ is a covering sieve, $x \in X_w$, and since $f(X_w) \subseteq Y_w$, $y$ is in $Y_w$, which leads to a contradiction.

- *Transitivity*: For a covering sieve $S$ in $J(X)$ and a sieve $S'$ on $X$ such that $f^*(S') \in J(dom(f))$ for all $f$ in $S$, we have to show that $S'$ is a covering sieve, too. Since $S$ is nonempty and all covering sieves are nonempty, $S'$ is also nonempty. To see that the other condition for a covering sieve holds, suppose $x$ is in $X$ such that $f'(x) \in Y'_w$ for all $f': Y' \to X$ in $S'$. Since $S$ is a covering sieve, it suffices to show that $f(x) \in Y_w$ for all $f: Y \to X$ in $S$. Let $f$ be a morphism from $Y$ to $X$ in $S$. Then, for all $g: K \to Y \in f^*(S')$, $g(f(x))$ is in $K_w$. Since $f^*(S')$ is in $J(Y)$, $f(x)$ is in $Y_w$.

□

We use a subcategory of the category of functors from $\mathbf{W}^{op}$ to $\mathbf{mCpo}$ to interpret each type $\theta$ in IAh. Before explaining in detail the subcategory, we consider the cartesian closed structure of the category of functors from $\mathbf{W}^{op}$ to $\mathbf{mCpo}$, which we denote by $\mathbf{mCpo}^{\mathbf{W}^{op}}$. The structure is very similar to that of a presheaf category. Let $F$ and $G$ be functors in $\mathbf{mCpo}^{\mathbf{W}^{op}}$. The products and terminal objects are given pointwise:

$$\begin{array}{rcl} 1(X) & = & \{*\} \\ 1(X) & = & \mathsf{id}_{\{*\}} \end{array}$$

$$\begin{array}{rcll} (F \times G)(X) & = & F(X) \times G(X) & \text{(ordered pointwise)} \\ (F \times G)(f) & = & F(f) \times G(f) & \end{array}$$

The exponential $F \Rightarrow G$ of two functors $F, G$ is also similar to one in a presheaf category. It is defined as a family of *monotone* functions indexed by "future worlds" satisfying the naturality condition.

$$\begin{array}{rcl} (F \Rightarrow G)(X) & = & \{m \in \Pi_{f:Y \to X \in \mathbf{W}}(F(Y) \to G(Y)) \\ & & \quad \mid \forall f, g \in \mathbf{W}.\, G(g) \circ m_f = m_{f \circ g} \circ F(g)\} \\ & & \qquad\qquad \text{(ordered pointwise)} \\ (F \Rightarrow G)(f) & = & \lambda m.\, \lambda g.\, m_{f \circ g} \end{array}$$

The application map $\mathsf{ap}$ from $(F \Rightarrow G) \times F$ to $G$ is the natural transformation which instantiates the family in the first argument at the current world and applies it to the second argument:

$$\mathsf{ap}_W(m, a) \quad = \quad m_{\mathsf{id}_W}(a)$$

**Lemma 104** *The category of functors from* $\mathbf{W}^{op}$ *to* $\mathbf{mCpo}$ *is cartesian closed.*

The subcategory $\mathbf{C}$ of $\mathbf{mCpo}^{\mathbf{W}^{op}}$, which is used to interpret each type in IAh, is a full subcategory of $\mathbf{mCpo}^{\mathbf{W}^{op}}$ consisting of following functors $F$:

- for all objects $X$ in $W$, $F(X)$ is pointed, and for all morphisms $f$ in $W$, $F(f)$ is continuous and strict;
- for all morphisms $f$ in $W$, $F(f)$ is an order embedding: $F(f)$ is injective and reflects ordering; and
- when $F$ is composed with the forgetful functor $U$ from $\mathbf{mCpo}$ to $\mathbf{Set}$, functor $U \circ F$ becomes a sheaf on the site $(\mathbf{W}, J)$.

The last condition is equivalently expressed as a usual condition for being a sheaf on a site: for a covering sieve $S \in J(V)$, a matching family is a set $\{x_f \in F(dom(f))\}_{f \in S}$ with $x_{f \circ g} = x_f \cdot g$ for all $f, f \circ g \in S$[1]; an amalgamation of a matching family $\{x_f \in F(dom(f))\}_{f \in S}$ is an element $x$ in $F(V)$ with $x \cdot f = x_f$ for all $f \in S$; and $F$ becomes a sheaf if all matching families for all covering sieves have unique amalgamations.

All three rules are critical either in interpreting recursion or the symmetric noninterference predicate properly. The first condition is related to recursion. We note a well-known fact, sometimes called Plotkin lemma, about the transfer of fixed points:

**Lemma 105** *Given pointed directed complete posets $C, D$, monotone functions $f: C \to C$, $g: D \to D$ and a strict continuous function $h: C \to D$, if the following diagram commutes:*

$$\begin{tikzcd} C \arrow[r, "f"] \arrow[d, "h"'] & C \arrow[d, "h"] \\ D \arrow[r, "g"] & D \end{tikzcd}$$

*then $h(\mathsf{fix}\ f) = \mathsf{fix}\ g$ where $\mathsf{fix}$ returns the least fixed point for a monotone function.*[2]

So, when a functor $F \in \mathbf{mCpo}^{\mathbf{W}^{op}}$ satisfies the first condition, we obtain a natural transformation $\mathsf{nfix}$ from $(F \Rightarrow F)$ to $F$ using $\mathsf{fix}$ as follows:

$$\mathsf{nfix}_W(m) \quad \triangleq \quad \mathsf{fix}\ (m_{\mathsf{id}_W})$$

To see that $\mathsf{nfix}$ is natural, let $f$ be a morphism from $X$ to $W$ in $\mathbf{W}$. Since $F(W)$ and $F(X)$ are pointed and $F(f)$ is strict and continuous, the Plotkin lemma says that we only need to show that the following diagram commutes:

$$\begin{tikzcd} F(W) \arrow[r, "m_{\mathsf{id}_W}"] \arrow[d, "F(f)"'] & F(W) \arrow[d, "F(f)"] \\ F(X) \arrow[r, "m_f"] & F(X) \end{tikzcd}$$

[1]For a functor $F$ either in $\mathbf{mCpo}^{\mathbf{W}^{op}}$ or in $\mathbf{Set}^{\mathbf{W}^{op}}$, $x$ in $F(V)$ and a morphism $f: X \to V$ in $\mathbf{W}$, we write $x \cdot f$ to denote $F(f)(x)$.

[2]For all monotone functions $f$ from a pointed directed-complete poset $D$ to $D$, the least fixed point of $f$ always exists [1].

which is just the naturality condition for $m$; so, it always holds.

The other two conditions in the interpretation of IAh types are related to *minimal resource property*, which is essential in interpreting the symmetric noninterference predicate properly. We consider a functor $F$ in $\mathbf{Set}^{\mathbf{W}^{op}}$ because a functor in $\mathbf{mCpo}^{\mathbf{W}^{op}}$ can be considered as one in $\mathbf{Set}^{\mathbf{W}^{op}}$ by composing with the forgetful functor $U$ from $\mathbf{mCpo}$ to $\mathbf{Set}$. For an element $a \in F(X)$, the *support* of $a$ ($\mathsf{supp}(a)$) is an object $X'$ in $\mathbf{W}$ such that $X'_0 \subseteq X_0$, $X'_w \subseteq X_w$ and $F(\mathsf{inc}_{X'_0,X_0})(a') = a$ for some $a' \in F(X')$, and $X'$ is the least such with respect to the subset ordering. Intuitively, it denotes what variables in $X_0$ are accessed by $a$ and what variables in $X_w$ are modified by $a$. Note that the support of $a$ may or may not exist. We say that $F$ has a *minimal resource property* when for all $a \in F(X)$, the support of $a$ always exists. For functors with the minimal resource property, we can define the notion of symmetric noninterference. Recall that two terms do not interfere symmetrically if all variables used by both terms are only read by the terms. When functors $F$ and $G$ in $\mathbf{Set}^{\mathbf{W}^{op}}$ have the minimal resource property, such an intuition is easily formalized: for $a \in F(X)$ and $b \in G(X)$, we say $a$ and $b$ do not interfere symmetrically ($a \perp b$) if

$$(\mathsf{supp}(a)_0 \cap \mathsf{supp}(b)_0) \quad \subseteq \quad (\mathsf{supp}(a)_0 - \mathsf{supp}(a)_w) \cap (\mathsf{supp}(b)_0 - \mathsf{supp}(b)_w)$$

The left hand side denotes all variables accessed by both $a$ and $b$, and the right hand side denotes all variables only read by both $a$ and $b$.

The last two conditions in the interpretation of IAh types imply the minimal resource property, thereby allowing symmetric noninterference in the above to be defined. In the following, we prove a slightly stronger result: if a functor $F \in \mathbf{Set}^{\mathbf{W}^{op}}$ is a sheaf with respect to the topology $J$ and $F(f)$ is injective for all $f$ in $\mathbf{W}$, $F$ has the minimal resource property. To see why the result is stronger, note that when a functor in $\mathbf{mCpo}^{\mathbf{W}^{op}}$ is considered as one in $\mathbf{Set}^{\mathbf{W}^{op}}$ by composing the forgetful functor $U$, the second condition in the interpretation of IAh types implies that the functor maps a morphism in $W$ to an injection in $\mathbf{Set}$ and the third condition says that the functor is a sheaf. Following [49], we start by proving that if $F \in \mathbf{Set}^{\mathbf{W}^{op}}$ maps a pullback in $\mathbf{W}^{op}$ to a pullback in $\mathbf{Set}$, $F$ has the minimal resource property; then, we prove that if $F$ is an injective sheaf, it also preserves pullbacks.

**Lemma 106** *The following commutative diagram in $\mathbf{W}^{op}$ is a pullback*

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, "h"'] & Y \arrow[d, "g"] \\ W \arrow[r, "k"'] & K \end{tikzcd}$$

*iff*

1. *for all $y \in Y_0$ and $w \in W_0$, $g(y) = k(w)$ implies that $f(x) = y$ and $h(x) = w$ for some $x \in X_0$; and*

2. *$X_w = \{x \in X_0 \mid f(x) \in Y_w \wedge h(x) \in W_w\}$.*

**Lemma 107** *If a functor $F : \mathbf{W}^{op} \to \mathbf{Set}$ maps a pullback in $\mathbf{W}^{op}$ to a pullback in $\mathbf{Set}$, then $F$ has a minimal resource property.*

*Proof:* In the proof, we consider the category $\mathbf{W}^{op}$ not $\mathbf{W}$. The proof is from [49]. For all $a \in F(X)$, let $Y, Z$ be objects of $\mathbf{W}^{op}$ such that $Y_0, Z_0 \subseteq X_0$, $Y_w, Z_w \subseteq X_w$, $F(\mathsf{inc}_{Y_0,X_0})(b) = a$ for $b \in F(Y)$ and $F(\mathsf{inc}_{Z_0,X_0})(c) = a$ for $c \in F(Z)$. Let $K$ be $(Y_0 \cap Z_0, Y_w \cap Z_w)$. Then, the following diagram becomes a pullback in $\mathbf{W}^{op}$:

$$\begin{tikzcd} K \arrow[r, "\mathsf{inc}_{K_0,Y_0}"] \arrow[d, "\mathsf{inc}_{K_0,Z_0}"'] \arrow[dr, phantom, "\lrcorner", very near start] & Y \arrow[d, "\mathsf{inc}_{Y_0,X_0}"] \\ Z \arrow[r, "\mathsf{inc}_{Z_0,X_0}"'] & X \end{tikzcd}$$

Therefore, there exists $d \in F(K)$ which is mapped to $a$ by $F(\mathsf{inc}_{K_0,X_0})$. Since $X_0$ is finite, the minimal resource can be obtained by repeating the above construction. $\square$

**Lemma 108** *For all functors $F : \mathbf{W}^{op} \to \mathbf{Set}$, if $F(f)$ is injective for all $f$ and $F$ is a sheaf with respect to the topology $J$, then $F$ preserves pullbacks.*

*Proof:* During the proof, we consider the category $\mathbf{W}^{op}$ not $\mathbf{W}$. Therefore, the assumption for $F$ being a sheaf is rephrased dually. For instance, we deal with co-sieves and covering co-sieves instead of sieves and covering sieves. We show that such $F$ preserves two special types of pullbacks. Consider a pullback in $\mathbf{W}^{op}$:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "h"'] \arrow[dr, phantom, "\lrcorner", very near start] & Y \arrow[d, "g"] \\
W \arrow[r, "k"'] & K
\end{tikzcd}$$

The pullback is of the first type when both $g$ and $h$ are identity functions even though they need not be identity morphisms in $\mathbf{W}^{op}$. And it is of the second type when $Y_w = \{y \mid g(y) \in K_w\}$ and $W_w = \{w \mid k(w) \in K_w\}$. We show that $F$ preserves such types of pullbacks. In order to see that the preservation of such particular pullbacks implies that $F$ preserves pullbacks in general, we note that all pullback diagrams can be decomposed into four pullbacks which are either of the first or of the second type. Consider the pullback in the beginning of the proof. It can be decomposed as follows: let $Y', W', X', X'', X'''$ be objects in $\mathbf{W}^{op}$ given by:

$$\begin{aligned}
Y' &\triangleq (Y, \{y \mid g(y) \in K_w\}) \\
W' &\triangleq (W, \{w \mid k(w) \in K_w\}) \\
X' &\triangleq (X, \{x \mid g(f(x)) \in K_w\}) \\
X'' &\triangleq (X, \{x \mid h(x) \in W_w\}) \\
X''' &\triangleq (X, \{x \mid f(x) \in Y_w\})
\end{aligned}$$

Then, the above pullback diagram is decomposed into the following four pullbacks:

$$\begin{tikzcd}
X \arrow[r, "id_{X_0}"] \arrow[d, "id_{X_0}"'] \arrow[dr, phantom, "\lrcorner", very near start] & X''' \arrow[r, "f"] \arrow[d, "id_{X_0}"'] \arrow[dr, phantom, "\lrcorner", very near start] & Y \arrow[d, "id_{Y_0}"] \\
X'' \arrow[r, "id_{X_0}"] \arrow[d, "h"'] \arrow[dr, phantom, "\lrcorner", very near start] & X' \arrow[r, "f"] \arrow[d, "h"'] \arrow[dr, phantom, "\lrcorner", very near start] & Y' \arrow[d, "g"] \\
W \arrow[r, "id_{W_0}"] & W' \arrow[r, "k"] & K
\end{tikzcd}$$

where $id$ is an identity function but is not necessarily an identity morphism in the category $\mathbf{W}^{op}$. Note that all four pullbacks are either of the first or of the second type.

We first show that $F$ preserves pullbacks of the first type. Suppose that $a \in F(W)$ and $b \in F(Y)$ are mapped to the same element by $F(k)$ and $F(g)$. It suffices to show that there is

$c \in F(X)$ such that $F(f)(c) = b$ and $F(h)(c) = a$; uniqueness comes from injectivity of $F(f)$. Let $S$ be a co-sieve on $X$ consisting of all morphisms which can factor through either $f$ or $h$. Since $S$ contains $f, h$ and $X$ is a pullback, $S$ satisfies two conditions to be a covering co-sieve. We define a matching family for $S$ which contains $a$ and $b$; then, its amalgamation is what we are looking for. Define a family $\{d_j\}_{j \in S}$ as: for all $j \in S$,

$$d_j \;=\; \begin{cases} F(j')(a) & \text{if } j = j' \circ h \\ F(j')(b) & \text{if } j = j' \circ f \end{cases}$$

It suffices to show that the family is well-defined; then, the family becomes a matching family by definition. There are two possibilities that the family is not well-defined. The first possibility is that $j' \circ h$ and $j'' \circ f$ are morphisms from $X$ to $Z$ in $\mathbf{W}^{op}$ and they are the same but $F(j')(a) \neq F(j'')(b)$. We show that it is not the case. Consider a morphism $id_{Z_0}\colon (Z_0, Z_w) \to (Z_0, Z_0)$ in $\mathbf{W}^{op}$ which, viewed as a function, is an identity map. When $j''$ is considered as a morphism from $K$ to $(Z_0, Z_0)$, we have the following commutative diagram because, when viewed as functions, both $g$ and $h$ are identity functions and $f$ and $k$ are the same function:

$$\begin{tikzcd}
Y \arrow[r, "j''"] \arrow[d, "g"'] & Z \arrow[d, "id_{Z_0}"] \\
K \arrow[r, "j''"] & (Z_0, Z_0) \\
W \arrow[u, "k"] \arrow[r, "j'"'] & Z \arrow[u, "id_{Z_0}"']
\end{tikzcd}$$

Therefore, $F(id_{Z_0} \circ j')(a) = F(id_{Z_0} \circ j'')(b)$, so $F(j')(a) = F(j'')(b)$ by injectivity of $F(id_{Z_0})$. The second possibility is that $j' \circ f$ and $j'' \circ f$ are morphisms from $X$ to $Z$ in $\mathbf{W}^{op}$ and they are the same but $F(j')(b) \neq F(j'')(b)$. To see why such a case never happens, consider the same morphism as in the first case: a morphism $id_{Z_0}\colon (Z_0, Z_w) \to (Z_0, Z_0)$ in $\mathbf{W}^{op}$ which, viewed as a function, is an identity function. Then, the following diagram commutes:

$$
\begin{tikzcd}
Y \arrow[r, "j''"] \arrow[d, "g"'] & Z \arrow[d, "id_{Z_0}"] \\
K \arrow[r, "j''"] & (Z_0, Z_0) \\
W \arrow[u, "k"] \arrow[r, "k"'] & K \arrow[u, "j'"']
\end{tikzcd}
$$

because, when viewed as functions, $k$ and $f$ are the same function and $g$ is an identity map. Therefore, $F(id_{Z_0} \circ j'')(b) = F(j' \circ k)(a)$. We consider another commutative diagram in $\mathbf{W}^{op}$, which shows that $F(id_{Z_0} \circ j')(b) = F(j' \circ k)(a)$.

$$
\begin{tikzcd}
Y \arrow[r, "j'"] \arrow[d, "g"'] & Z \arrow[d, "id_{Z_0}"] \\
K \arrow[r, "j'"] & (Z_0, Z_0) \\
W \arrow[u, "k"] \arrow[r, "k"'] & K \arrow[u, "j'"']
\end{tikzcd}
$$

Therefore, $F(id_{Z_0} \circ j')(b) = F(id_{Z_0} \circ j'')(b)$. Now, since $F(id_{Z_0})$ is injective, $F(j')(b) = F(j'')(b)$.

To show that $F$ preserves pullbacks of the second type, we use a proof in [26]. It suffices to show that for all $a \in F(W)$, $b \in F(Y)$ with $F(k)(a) = F(g)(b)$, there is $c$ in $F(X)$ such that $F(f)(c) = b$ and $F(h)(c) = a$. Consider a co-sieve $\{j \circ g \circ f \mid j : K \to Z \wedge (|Z_0| > |K_0| + |Y_0| + |W_0|)\}$ on $X$ and call it $S$. Then, $S$ becomes a covering co-sieve because it is nonempty and $(g \circ f)(x) \in K_w$ implies $x \in X_w$. Moreover, $\{F(j \circ g)(b)\}_{j \circ g \circ f \in S}$ is a matching family for $S$: for any $j, j' : K \to Z$ such that $j \circ g \circ f = j' \circ g \circ f \in S$, we have $F(j \circ g)(b) = F(j' \circ g)(b)$ because there are sequences $j_0, j_1, \ldots, j_n$ such that $j_0 = j$, $j_n = j'$ and $(j_i \circ g) = (j_{i+1} \circ g)$ or $(j_i \circ k) = (j_{i+1} \circ k)$ for all $0 \leq i \leq n-1$. Since $F$ is a sheaf, there is a unique amalgamation $c \in F(X)$ of the matching family. Because of injectivity, $c$ becomes an element in $F(X)$ which is mapped to $b$ by $F(f)$ and $a$ by $F(h)$. $\square$

In the following, we show that the three conditions in the interpretation of IAh types are closed under taking a terminal object, products and exponentials in $\mathbf{mCpo}^{\mathbf{W}^{op}}$. That is, the category $\mathbf{C}$ is a sub-CCC of $\mathbf{mCpo}^{\mathbf{W}^{op}}$. Such a result indicates that all types in IAh can be interpreted in $\mathbf{C}$ as long as base types in IAh are interpreted in $\mathbf{C}$.

**Lemma 109** *The full subcategory $\mathbf{C}$ of $\mathbf{mCpo}^{\mathbf{W}^{op}}$ consisting of all functors with three requirements in the interpretation of IAh types forms a sub-CCC of $\mathbf{mCpo}^{\mathbf{W}^{op}}$.*

*Proof:* Since showing that the subcategory $\mathbf{C}$ is closed under a terminal object and products is straightforward, we concentrate on showing that $\mathbf{C}$ is closed under taking exponentials. Suppose that $F$ and $G$ are object in $\mathbf{C}$.

1. For all $Y$ and $f: X \to Y$ in $\mathbf{W}$, $(F \Rightarrow G)(Y)$ is pointed and $(F \Rightarrow G)(f)$ is strict and continuous:
   The least $m$ in $(F \Rightarrow G)(Y)$ is a family of functions which map an element of $F$ to the least element of $G$: $m_g(a) = \perp$. Note that the family $m$ satisfies the naturality condition because $G(g)$ is strict for all $g: Z \to Y$ in $\mathbf{W}$. The strictness of $(F \Rightarrow G)(f)$ follows from such a charaterization of the least family $m$. To see the continuity, let $D$ be a directed subset of $(F \Rightarrow G)(Y)$. The least upper bounds of $D$ and $\{m \cdot f \mid m \in D\}$ are obtained pointwise. So, for all morphisms $k: Z \to X$, we have
$$\begin{aligned} (\sqcup\{m \cdot f \mid m \in D\})_k &= \sqcup\{m_{f \circ k} \mid m \in D\} \\ &= ((\sqcup D) \cdot f)_k \end{aligned}$$

2. For all $f: X \to Y$ in $\mathbf{W}$, $(F \Rightarrow G)(f)$ is an order embedding:
   Note that it suffices to show that $(F \Rightarrow G)(f)$ reflects an order because equality can be written by $\sqsubseteq$ and $\sqsupseteq$ (antisymmetry). Suppose that $m \cdot f \sqsubseteq m' \cdot f$ for $m, m' \in (F \Rightarrow G)(Y)$. We will show that for all $k: K \to Y$, $m_k \sqsubseteq m'_k$. Let $k$ be a morphism from $K$ to $Y$ in $\mathbf{W}$. Then, there are $g, h$ such that the following diagram commutes:
$$\begin{tikzcd} Z \arrow[r, "h"] \arrow[d, "g"'] & K \arrow[d, "k"] \\ X \arrow[r, "f"] & Y \end{tikzcd}$$

Then,

$$\begin{array}{rcccccc} G(h) \circ m_k & = & m_{k \circ h} \circ F(h) & = & m_{f \circ g} \circ F(h) & & \\ & \sqsubseteq & m'_{f \circ g} \circ F(h) & = & m'_{k \circ h} \circ F(h) & = & G(h) \circ m'_k \end{array}$$

The conclusion follows since $G(h)$ reflects an order.

3. $(F \Rightarrow G)$ is a sheaf:

   Let $U$ be the forgetful functor from **mCpo** to **Set**. Then, $U \circ (F \Rightarrow G)$ is a subfunctor of $(U \circ F) \Rightarrow (U \circ G)$, which is a sheaf as a functor from $\mathbf{W}^{op}$ to **Set**. So, for all matching families $\{m^f\}_{f \in S}$ of elements of $F \Rightarrow G$ for a covering sieve $S \in J(X)$, the family can be considered as a matching family of elements of $(U \circ F) \Rightarrow (U \circ G)$. Since $(U \circ F) \Rightarrow (U \circ G)$ is a sheaf from $\mathbf{W}^{op}$ to **Set**, there exists a unique amalgamation $m$ in $((U \circ F) \Rightarrow (U \circ G))(X)$ for the family. So, it suffices to show that the $m$ is, indeed, in the subfunctor $U \circ (F \Rightarrow G)$, which amounts to showing that $m_g$ is monotone for all $g{:}\, Y \to X \in \mathbf{W}$. Let $g$ be a morphism of type $Y \to X$ in $\mathbf{W}$ and $a, a'$ elements in $F(Y)$ such that $a \sqsubseteq a'$. Pick $k \in g^*(S)$. Then,

$$\begin{array}{ll} G(k)(m_g(a)) = (m \cdot g \circ k)_{\mathsf{id}}(a \cdot k) = m^{g \circ k}_{\mathsf{id}}(a \cdot k) & \\ \qquad \sqsubseteq m^{g \circ k}_{\mathsf{id}}(a' \cdot k) & = (m \cdot g \circ k)_{\mathsf{id}}(a' \cdot k) = G(k)(m_g(a')) \end{array}$$

   Since $G(k)$ reflects ordering, $m_g(a) \sqsubseteq m_g(a')$.

□

In presenting reasoning principles in Section 9.2, we introduced a subclass of IAh types: passive types. A program phrase of passive type is supposed not to write to any variables. We consider a subcategory $\mathbf{P}$ of $\mathbf{C}$, which captures such an intuition. The category $\mathbf{P}$ is a full subcategory of $\mathbf{C}$ consisting of $F$'s in $\mathbf{C}$ satisfying:

> for all $f{:}\, X \to Y$ in $\mathbf{W}$, if $f$ is a bijection as a function from $Y_0$ to $X_0$, then $F(f)$ is an isomorphism.

In order to see how such a definition captures the intuition on passive types, let $F$ be a functor in $\mathbf{P}$. Pick $a \in F(X)$. Then, the modifiable portion of the support of $a$, which is $\mathsf{supp}(a)_w$, is always empty; it is because whenever $a$ is expanded from $a'$ in $F(X'_0, X'_w)$, the inverse of the isomorphism $F(id_{X'_0}{:}\,(X'_0, X_w) \to (X'_0, \emptyset))$ gives $a''$ in $F(X'_0, \emptyset)$ which is also expanded to $a$.

Such a fact indicates that $a$ doesn't modify any variables. The following result shows that the syntactic criteria to discern passive types with the context free grammar in Section 9.2 is, in fact, correct.

**Lemma 110** *For functors $F, G \in \mathbf{C}$,*

1. *if $F, G \in \mathbf{P}$, then $F \times G \in \mathbf{P}$; and*

2. *if $G \in \mathbf{P}$, then $F \Rightarrow G \in \mathbf{P}$.*

*Proof:* The case for products is straightforward. We only deal with the second case. Let $G, F$ be functors as in the lemma. Then, $F \Rightarrow G$ is in $\mathbf{C}$. In particular, $(F \Rightarrow G)(f)$ is an order embedding for all $f$ in $\mathbf{W}$. Therefore, it suffices to show that $(F \Rightarrow G)(f)$ is surjective for certain $f$ in $\mathbf{W}$. Let $f\colon Y \to X$ be a morphism in $\mathbf{W}$ such that it becomes a bijection when viewed as a function from $X_0$ to $Y_0$. To show that $(F \Rightarrow G)(f)$ is surjective, pick $m$ in $(F \Rightarrow G)(Y)$. We construct a natural family $\{n_g\colon F(Z) \to G(Z)\}_{g:Z\to X}$ of morphisms indexed by a morphism $g\colon Z \to X$ in $\mathbf{W}$. Let $Z'$ be $(Z_0, Z_0)$ and $id_{Z_0}$ a morphism from $Z'$ to $Z$ which is an identity function when viewed as a function from $Z_0$ to $Z'_0$. Let $k_g$ be a morphism from $Z'$ to $Y$ such that when viewed as a function from $Y_0$ to $Z'_0$, $k_g(y) = g(f^{-1}(y))$. Then, we have the following commutative diagram in $\mathbf{W}$:

$$\begin{tikzcd}
Y \arrow[d, "f"'] & Z' \arrow[l, "k_g"'] \arrow[d, "id_{Z_0}"] \\
X & Z \arrow[l, "g"]
\end{tikzcd}$$

Since $G(id_{Z_0})$ is an isomorphism, we can define

$$n_g \triangleq G(id_{Z_0})^{-1} \circ m_{k_g} \circ F(id_{Z_0})$$

To see the family satisfies the naturality condition, consider $g: Z \to X$ and $h: W \to Z$ in $\mathbf{W}$. Then, the following diagram commutes:

$$\begin{tikzcd}
Y \arrow[d, equal] & & W' \arrow[ll, "k_{g \circ h}"'] \arrow[d, equal] \\
Y \arrow[d, "f"'] & Z' \arrow[l, "k_g"'] \arrow[d, "id_{Z_0}"] & W' \arrow[l, "h'"'] \arrow[d, "id_{W_0}"] \\
X & Z \arrow[l, "g"'] & W \arrow[l, "h"']
\end{tikzcd}$$

where $W'$, $id_{W_0}$ and $k_{g \circ h}$ are constructed from $W$, $g \circ h$ and $f$ just as $Z'$, $id_{Z_0}$ and $k_g$ are constructed from $Z$, $g$ and $f$; and $h'$ is the same as $h$ as a function. Then,

$$\begin{aligned}
G(h) \circ n_g &= G(h) \circ G(id_{Z_0})^{-1} \circ m_{k_g} \circ F(id_{Z_0}) \\
&= G(id_{W_0})^{-1} \circ G(h') \circ m_{k_g} \circ F(id_{Z_0}) \\
&= G(id_{W_0})^{-1} \circ m_{k_g \circ h'} \circ F(h') \circ F(id_{Z_0}) \\
&= G(id_{W_0})^{-1} \circ m_{k_{g \circ h}} \circ F(id_{W_0}) \circ F(h) \\
&= n_{g \circ h} \circ F(h)
\end{aligned}$$

Finally, in order to see that $n \cdot f = m$, consider $g: Z \to Y$. Then, the following diagram commutes by the construction of $k_{f \circ g}$:

$$\begin{tikzcd}
Y \arrow[r, equal] \arrow[d, "f"'] & Y \arrow[d, equal] & Z' \arrow[l, "k_{f \circ g}"'] \arrow[d, "id_{Z_0}"] \\
X & Y \arrow[l, "f"'] & Z \arrow[l, "g"']
\end{tikzcd}$$

So,

$$\begin{aligned}
n_{f \circ g} &= G(id_{Z_0})^{-1} \circ m_{k_{f \circ g}} \circ F(id_{Z_0}) \\
&= G(id_{Z_0})^{-1} \circ G(id_{Z_0}) \circ m_g \\
&= m_g
\end{aligned}$$

□

To interpret types for assertions and specifications, we use a category $\mathbf{Sh}(\mathbf{W}, J)$ of sheaves on a site $(\mathbf{W}, J)$. Note that the interpretation of an IAh type in $\mathbf{C}$ can also be considered as a sheaf in $\mathbf{Sh}(\mathbf{W}, J)$ because the third requirement for a functor $F$ to be in $\mathbf{C}$ is that $U \circ F$ is a sheaf. It is well-known that the category $\mathbf{Sh}(\mathbf{W}, J)$ forms a topos and, thus, provides an enough structure to interpret not only typed-lambda calculus but also higher-order intuitionistic logic. Before closing this section, we give a concrete description of an object $\Omega$ for truth values in $\mathbf{Sh}(\mathbf{W}, J)$, i.e., a subobject classifier in $\mathbf{Sh}(\mathbf{W}, J)$, because such an object becomes a natural choice for the interpretation of spec.

$$\begin{aligned} \Omega(X) &= \{\mathit{Var}\} \cup \{D \mid X_w \subseteq D \subseteq X_0\} \\ \Omega(f{:}Y \to X)(D) &= \begin{cases} f(D) \cup Y_w & \text{if } D \neq \mathit{Var} \\ \mathit{Var} & \text{if } D = \mathit{Var} \end{cases} \end{aligned}$$

In $\Omega(X)$, a smaller set represents more truth, so $\mathit{Var}$ denotes the falsity and $X_w$ the truth. The truth map true, therefore, maps $*$ to $X_w$ when instantiated at $X$.

### 9.3.2 The Interpretation of Types

Types in IAh are interpreted as objects in $\mathbf{C}$, and assert and spec are interpreted as objects in the category $\mathbf{Sh}(\mathbf{W}, J)$ of sheaves on the site $(\mathbf{W}, J)$. Such categories have a structure, which is useful in defining semantics: the category $\mathbf{C}$ is a sub-CCC of $\mathbf{mCpo}^{\mathbf{W}^{op}}$ so that function and product types in IAh are easily interpreted by exponentials and products in $\mathbf{C}$; the category $\mathbf{Sh}(\mathbf{W}, J)$ of sheaves is a topos so that spec is naturally interpreted as a subobject classifier in $\mathbf{Sh}(\mathbf{W}, J)$ because the subobject classifier is an object for truth values. In this section, we concentrate on the interpretation of base types in IAh and assert.

In introducing IAh in Section 9.1, we assumed that only locations are storable and that each heap cell has two fields. Such assumptions can be clarified by semantic domains for storage. Let $\mathit{Loc}$ be a countable set which is disjoint from $\mathit{Var}$, and let $\mathit{nil}$ be an element which doesn't appear either in $\mathit{Loc}$ or in $\mathit{Var}$. And let $X$ be a world in $\mathbf{W}$, i.e., a pair $(X_0, X_w)$ of finite sets of variables with $X_w \subseteq X_0$. Then, the domain for storage in the world $X$ is given as follows:

$$\begin{aligned} \mathit{Val} &= \mathit{Loc} \cup \{\mathit{nil}\} \\ \mathit{Stack}(X) &= X_0 \to \mathit{Val} \\ \mathit{Heap} &= \mathit{Loc} \rightharpoonup_{\mathit{fin}} \mathit{Val} \times \mathit{Val} \\ \mathit{State}(X) &= \mathit{Stack}(X) \times \mathit{Heap} \end{aligned}$$

In interpreting com and exp, we frequently have to handle replacing a stack in a bigger world by another stack in a smaller world or projecting a stack in a bigger world into one in a smaller world. We formalize such operations by defining the replacement $\rho_f: Stack(X) \times Stack(X') \to Stack(X)$ and the projection $\varphi_f: Stack(X) \to Stack(X')$ for every morphism $f: X \to X'$ in $\mathbf{W}$:

$$\begin{aligned} \rho_f(s, s')(x) &= \begin{cases} s'(f^{-1}(x)) & \text{if } x \in f(X_0') \\ s(x) & \text{otherwise} \end{cases} \\ \varphi_f(s)(x) &= s(f(x)) \end{aligned}$$

A denotation for a command at a world $W$ maps a state to a set which may include states, a memory fault (*fault*) or divergence ($\perp$). In order to capture a map to such sets, we use the Egli-Milner powerdomain of the flat cpo $(State(W) \cup \{fault\})_\perp$. The powerdomain $\mathcal{P}_{EM}((State(X) \cup \{fault\})_\perp)$ is:

$$\mathcal{P}_{EM}((State(X) \cup \{fault\})_\perp) \;=\; \{S \mid S \subseteq State(W) \cup \{fault, \perp\}\}.$$

It is ordered as follows:

$$\begin{aligned} S \sqsubseteq S' \text{ iff } \;& (\perp \notin S \text{ implies } S = S') \\ & \text{and } (\perp \in S \text{ implies } S - \{\perp\} \subseteq S') \end{aligned}$$

Intuitively, $S$ is less than $S'$ if we can replace $\perp$ in $S$ by a set to obtain $S'$.

A denotation $c$ for a command at a world $X$ is required to have Write Locality and the other locality conditions for heap storage. Recall that the motivation to define a world as a pair is to specify access permission for each variable in the world. The Write Locality condition fulfills such intention by requiring that $c$ should not change read-only variables in $X_0 - X_w$. The other locality conditions are about how $c$ uses heap storage. It says that the execution of $c$ doesn't depend on "unaccessed" heap cells. To define them precisely, we use two notations $\#$ and $*$ for heaps: $h_1 \# h_2$ holds if and only if $dom(h_1)$ and $dom(h_2)$ are disjoint; and when $h_1 \# h_2$, heap $h_1 * h_2$ is defined to be the union of $h_1$ and $h_2$.

- *Write Locality*: for all $(s, h), (s', h') \in State(X)$, if $(s', h') \in c(s, h)$, then $s(x) = s'(x)$ for all $x \in X_0 - X_w$; and
- for all stacks $s$ in $Stack(X)$ and all heaps $h_1, h_2$ in *Heap* such that $h_1 \# h_2$,
  1. *Safety Monotonicity*: if $fault \notin c(s, h_1)$, then $fault \notin c(s, h_1 * h_2)$;
  2. *Termination Monotonicity*: if neither $\perp$ nor *fault* is in $c(s, h_1)$, then neither $\perp$ nor *fault* is in $c(s, h_1 * h_2)$; and

3. *Heap Locality*: when $fault \notin c(s, h_1)$, for all $(s', h') \in c(s, h_1 * h_2)$, there exists $h'_1$ such that $h' = h'_1 * h_2$ and $(s', h'_1) \in c(s, h_1)$.

The command type is interpreted as follows:

$$
\begin{array}{lcl}
[\![\mathsf{com}]\!](X) & \triangleq & \{c \in \mathit{State}(X) \to \mathcal{P}_{EM}((\mathit{State}(X) \cup \{\mathit{fault}\})_\perp) \\
 & & \quad \mid c \text{ satisfies all four locality conditions}\} \\
[\![\mathsf{com}]\!](f)(c)(s, h) & \triangleq & \{(\rho_f(s, s''), h'') \mid (s'', h'') \in c(\varphi_f(s), h)\} \\
 & & \cup \{\mathit{fault} \mid \mathit{fault} \in c(\varphi_f(s), h)\} \\
 & & \cup \{\perp \mid \perp \in c(\varphi_f(s), h)\}
\end{array}
$$

Value, expression and variable types are interpreted as follows:

$$
\begin{array}{lcl}
[\![\mathsf{val}]\!](X) & = & \mathit{Val}_\perp \\
[\![\mathsf{val}]\!](f) & = & \mathit{id}_\perp \\
 & & \\
[\![\mathsf{exp}]\!](X) & = & [\mathit{Stack}(X) \to \mathit{Val}_\perp] \\
[\![\mathsf{exp}]\!](f) & = & \lambda e.\, \lambda s'.\, e(\varphi_f(s')) \\
 & & \\
[\![\mathsf{var}]\!](X) & = & (X_w)_\perp \\
[\![\mathsf{var}]\!](f) & = & f_\perp
\end{array}
$$

Note that the interpretation indicates that each value doesn't depend on states, while each expression might depend on them; an expression, however, accesses only variables not any heap cells. The variable type is interpreted differently from one in Chapters 3 and 4: $[\![\mathsf{var}]\!](X)$ denotes a set of updatable variables currently allocated in the stack, while in Chapters 3 and 4, $[\![\mathsf{var}]\!](X)$ denotes a set of objects with two methods for updating and reading. In fact, such an object-oriented point of view could have been used. Note that $[\![\mathsf{var}]\!]$ is isomorphic to the following $F$:

$$
\begin{array}{lcl}
F(X) & = & \{(\lambda v.\, \lambda(s, h).\, \{(s[x \mapsto v], h)\},\, \lambda s.\, s(x)) \mid x \in X_w\} \cup \{(\perp, \perp)\} \\
F(f) & = & \lambda x.\, (\lambda v.\, [\![\mathsf{com}]\!](f)(\mathsf{fst}(x)(v)),\, [\![\mathsf{exp}]\!](f)(\mathsf{snd}(x)))
\end{array}
$$

The assertion type $\mathsf{assert}$ and the specification type $\mathsf{spec}$ are interpreted as functors in $\mathbf{Sh}(\mathbf{W}, J)$.

$$\begin{array}{rcl} [\![\mathsf{assert}]\!](X) & = & \mathcal{P}(\mathit{State}(X)) \\ [\![\mathsf{assert}]\!](f) & = & \lambda p.\, \{(s,h) \mid (\varphi_f(s), h) \in p\} \end{array}$$

$$\begin{array}{rcl} [\![\mathsf{spec}]\!](X) & = & \{\mathit{Var}\} \cup \{D \mid X_w \subseteq D \subseteq X_0\} \\ [\![\mathsf{spec}]\!](f{:}\,Y \to X) & = & \lambda D. \begin{cases} \mathit{Var} & \text{if } D = \mathit{Var} \\ f(D) \cup Y_w & \text{otherwise} \end{cases} \end{array}$$

The set $[\![\mathsf{spec}]\!](X)$ indicates that in a world $X$, either each specification is false ($\mathit{Var}$) or it specifies the amount of variables whose access permission should be relaxed in order to make the specification true. Note that the sheaf $[\![\mathsf{spec}]\!]$ is, indeed, a subobject classifier in the topos $\mathbf{Sh}(\mathbf{W}, J)$.

**Lemma 111** *$[\![\mathsf{com}]\!]$, $[\![\mathsf{exp}]\!]$, $[\![\mathsf{val}]\!]$ and $[\![\mathsf{var}]\!]$ are functors in $\mathbf{C}$.*

*Proof:* We only deal with the case of $[\![\mathsf{com}]\!]$. The expression type is handled similarly, and it is straightforward to show that $[\![\mathsf{val}]\!]$ and $[\![\mathsf{var}]\!]$ are functors in $\mathbf{C}$. Since it is also straightforward to show that $[\![\mathsf{com}]\!]$ is a well-defined functor in $\mathbf{mCpo}^{\mathbf{W}^{op}}$, we concentrate on showing that $[\![\mathsf{com}]\!]$ is in $\mathbf{C}$.

- for all $W$ in $\mathbf{W}$, $[\![\mathsf{com}]\!](W)$ is pointed, and for all $f{:}\,Y \to X$ in $\mathbf{W}$, $[\![\mathsf{com}]\!](f)$ is strict and continuous:

  The least element in $[\![\mathsf{com}]\!](W)$ is $\lambda(s,h).\,\{\bot\}$ for all $W$ in $\mathbf{W}$. So, by the definition of $[\![\mathsf{com}]\!](f)$, the least element is mapped to the least element by $[\![\mathsf{com}]\!](f)$. For continuity, consider a directed set $D$ of commands in $[\![\mathsf{com}]\!](X)$ and a state $(s_0, h_0)$ in $\mathit{State}(Y)$. The following calculation shows that the least upper bound of $D$ is preserved by $[\![\mathsf{com}]\!](f)$.

$$
\begin{aligned}
& \textstyle\bigsqcup_{c \in D}((c \cdot f)(s_0, h_0)) \\
= \\
& \textstyle\bigsqcup_{c \in D} \left( \begin{array}{l} \{(\rho_f(s_0, s'), h') \mid (s', h') \in c(\varphi_f(s_0), h_0)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c(\varphi_f(s_0), h_0)\} \cup \{\bot \mid \bot \in c(\varphi_f(s_0), h_0)\} \end{array} \right) \\
= \\
& \textstyle\bigcup_{c \in D} \left( \begin{array}{l} \{(\rho_f(s_0, s'), h') \mid (s', h') \in c(\varphi_f(s_0), h_0)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c(\varphi_f(s_0), h_0)\} \end{array} \right) \\
& \cup \{\bot \mid \text{for all } c \in D, \bot \in c(\varphi_f(s_0), h_0)\} \\
= \\
& \{(\rho_f(s_0, s'), h') \mid (s', h') \in \textstyle\bigcup_{c \in D} c(\varphi_f(s_0), h_0)\} \\
& \cup \{\mathit{fault} \mid \mathit{fault} \in \textstyle\bigcup_{c \in D} c(\varphi_f(s_0), h_0)\} \cup \{\bot \mid \bot \in \textstyle\bigsqcup_{c \in D} c(\varphi_f(s_0), h_0)\} \\
= \\
& \{(\rho_f(s_0, s'), h') \mid (s', h') \in \textstyle\bigsqcup_{c \in D} c(\varphi_f(s_0), h_0)\} \\
& \cup \{\mathit{fault} \mid \mathit{fault} \in \textstyle\bigsqcup_{c \in D} c(\varphi_f(s_0), h_0)\} \cup \{\bot \mid \bot \in \textstyle\bigsqcup_{c \in D} c(\varphi_f(s_0), h_0)\} \\
= \\
& ((\sqcup D) \cdot f)(s_0, h_0)
\end{aligned}
$$

- for all $f{:}Y \to X$ in $\mathbf{W}$, $[\![\mathsf{com}]\!](f)$ reflects ordering:

  Consider $c, c' \in [\![\mathsf{com}]\!](X)$ such that $(c \cdot f) \sqsubseteq (c' \cdot f)$. We have to show that $c(s, h) \sqsubseteq c'(s, h)$ for all $(s, h) \in \mathit{State}(X)$. Pick a state $(s, h) \in \mathit{State}(X)$. Then, there is $s_0 \in \mathit{stack}(Y)$ such that $\varphi_f(s_0) = s$. Since $(c \cdot f)(s_0, h) \sqsubseteq (c' \cdot f)(s_0, h)$, we have

$$
\begin{array}{c}
\left( \begin{array}{l} \{(\rho_f(s_0, s'), h') \mid (s', h') \in (c(s, h) - \{\mathit{fault}, \bot\})\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c(s, h)\} \cup \{\bot \mid \bot \in c(s, h)\} \end{array} \right) \\
\sqsubseteq \\
\left( \begin{array}{l} \{(\rho_f(s_0, s'), h') \mid (s', h') \in (c'(s, h) - \{\mathit{fault}, \bot\})\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c'(s, h)\} \cup \{\bot \mid \bot \in c'(s, h)\} \end{array} \right)
\end{array}
$$

When $\perp \notin c(s,h)$, the above $\sqsubseteq$ becomes $=$, and otherwise, we have:

$$\begin{pmatrix} \{(\rho_f(s_0,s'),h') \mid (s',h') \in (c(s,h) - \{\mathit{fault},\perp\})\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c(s,h)\} \end{pmatrix}$$
$$\subseteq$$
$$\begin{pmatrix} \{(\rho_f(s_0,s'),h') \mid (s',h') \in (c'(s,h) - \{\mathit{fault},\perp\})\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c'(s,h)\} \end{pmatrix}$$

In each case, we have $c(s,h) \sqsubseteq c'(s,h)$ because $\rho_f(s_0,-)$ induces a map from subsets of $\mathit{State}(X)$ to subsets of $\mathit{State}(Y)$, which reflects the subset ordering:

> for $S, S' \subseteq \mathit{State}(X)$, if $\{(\rho_f(s_0,s'),h') \mid (s',h') \in S\}$ is a subset of $\{(\rho_f(s_0,s'),h') \mid (s',h') \in S'\}$, then $S$ is a subset of $S'$.

- $[\![\mathsf{com}]\!]$ is a sheaf:

  For all matching families $\{c_f \in [\![\mathsf{com}]\!](\mathit{dom}(f))\}_{f \in S}$ for a cover $S$ of $X$, it suffices to show that there is an amalgamation $c$ of the family; then, uniqueness follows from the fact that $[\![\mathsf{com}]\!]$ maps a morphism to an injection. By the definition of $S$ being a cover, there are $f_1: X_1 \to X, \ldots, f_n: X_n \to X$ in $S$ such that for all $x \in X_0 - X_w$, there exists $i \in \{1 \ldots n\}$ such that $f_i(x) \in (X_i)_0 - (X_i)_w$. Consider an object $Z$ and $g_i: Z \to X_i$ in $\mathbf{W}$ such that

  1. $|Z_0| = \Sigma_{i=1}^{n}(|(X_i)_0| - |X_0|) + |X_0|$;
  2. for all $i, j \in \{1 \ldots n\}$ with $i \neq j$, $g_i(x_i) = g_j(x_j)$ iff there exists $x \in X_0$ such that $x_i = f_i(x)$ and $x_j = f_j(x)$; and
  3. $Z_w = \bigcup_{i \in \{1 \ldots n\}} g_i((X_i)_w)$.

  Note that the second condition implies that for all $i, j \in \{1 \ldots n\}$, $f_i \circ g_i = f_j \circ g_j$. Let $k$ be $f_i \circ g_i$ for some $i \in \{1 \ldots n\}$. Then, $k \in S$, so it is possible to consider $c_k$. Such $c_k$ only writes to $k(X_w)$: for all $(s,h) \in \mathit{State}(Z)$, if $(s',h') \in c_k(s,h)$, then $s(z) = s'(z)$ for all $z \in Z_0 - k(X_w)$. To see the reason, consider two cases: $z \in k(X) - k(X_w)$ and $z \notin k(X)$. In the first case, there is $f_i$ with $z \notin f_i((X_i)_w)$. Since $(c_{f_i} \cdot g_i) = c_k$, $(s',h')$ is also in $(c_{f_i} \cdot g_i)(s,h)$. Then, $s'(z) = s(z)$ because $z \notin f_i((X_i)_w)$. In the second case, we consider an object $K$ and $l, l': K \to Z$ such that

  1. $|K_0| = |Z_0| + |Z_0| - |X_0|$;

2. $l(z) = l'(z)$ iff there exists $x \in X_0$ such that $z = k(x)$; and

3. $K_w = l(Z_w) \cup l'(Z_w)$.

Then, there is $s_0 \in \mathit{Stack}(K)$ such that $\varphi_l(s_0) = s$ and $\varphi_{l'}(s_0) = s$. Since $c_{k \circ l} = c_{k \circ l'}$, we have $(c_k) \cdot l = (c_k) \cdot l'$. So, $(\rho_{l'}(s_0, s'), h') = (\rho_l(s_0, s''), h')$ for some $s'' \in \mathit{Stack}(Z)$ satisfying $(s'', h') \in (c_k)(s, h)$. Therefore, $s'(z) = \varphi_{l'}(\rho_l(s_0, s''))(z)$, which is equal to $s_0(l'(z))$ because $l'(z) \notin l(Z_0)$. Since $s_0(l'(z)) = \varphi_{l'}(s_0)(z) = s(z)$, we have $s'(z) = s(z)$.

In the following, we show that there is a command $c \in [\![\mathsf{com}]\!](X)$ such that $c \cdot k = c_k$. To see why it is enough, consider $c_g$ for $g \in S$. Then, there are $g', k'$ in $\mathbf{W}$, such that $g \circ g' = k \circ k'$. So, $c \cdot g \circ g' = (c \cdot k) \cdot k' = c_k \cdot k'$, which is again equal to $c_g \cdot g'$. The injectivity of $[\![\mathsf{com}]\!](g')$ shows that $c \cdot g = c_g$. We define $c$ as follows: for $(s, h)$ in $\mathit{State}(X)$, pick $s_0 \in \mathit{Stack}(Z)$ such that $\varphi_k(s_0) = s$, and

$$c(s,h) \quad = \quad \begin{pmatrix} \{(\varphi_k(s''), h'') \mid (s'', h'') \in c_k(s_0, h)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c_k(s_0, h)\} \cup \{\perp \mid \perp \in c_k(s_0, h)\} \end{pmatrix}$$

We show that $c(s, h)$ is well-defined by proving that it doesn't depend on the choice of $s_0$. Let $s_1$ be a state in $\mathit{State}(Y)$ such that $\varphi_f(s_1) = s$. Then, there are $l, l' \colon W \to Z$ and $s_2 \in \mathit{State}(W)$ such that $k \circ l = k \circ l'$, $\varphi_l(s_2) = s_0$ and $\varphi_{l'}(s_2) = s_1$. Since $(c_k \cdot l)(s_2, h) = (c_k \cdot l')(s_2, h)$, we have

$$\begin{pmatrix} \{(\varphi_{k \circ l}(\rho_l(s_2, s'')), h'') \mid (s'', h'') \in c_k(s_0, h)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c_k(s_0, h)\} \cup \{\perp \mid \perp \in c_k(s_0, h)\} \end{pmatrix}$$
$$=$$
$$\begin{pmatrix} \{(\varphi_{k \circ l'}(\rho_{l'}(s_2, s'')), h'') \mid (s'', h'') \in c_k(s_1, h)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c_k(s_1, h)\} \cup \{\perp \mid \perp \in c_k(s_1, h)\} \end{pmatrix}$$

Since $\varphi_{k \circ l}(\rho_l(s_2, s'')) = \varphi_k(s'')$ and $\varphi_{k \circ l'}(\rho_{l'}(s_2, s'')) = \varphi_k(s'')$, the choice of $s_0$ or $s_1$ doesn't matter in defining $c(s, h)$.

The defined $c$ belongs to $[\![\mathsf{com}]\!](X)$; it satisfies the Write Locality condition since $c_k$ doesn't write to any variables in $Z - k(X_w)$, and it also satisfies the other three locality conditions for commands because $c_k$ does. And finally, the following calculation shows that $c \cdot k = c_k$: for all states $(s_0, h) \in \mathit{Stack}(Z)$,

$$
\begin{array}{ll}
& (c \cdot k)(s_0, h) \\
= & \\
& \left(\begin{array}{l} \{(\rho_k(s_0, s''), h'') \mid (s'', h'') \in c(\varphi_k(s_0), h)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c(\varphi_k(s_0), h)\} \cup \{\bot \mid \bot \in c(\varphi_k(s_0), h)\} \end{array}\right) \\
= & \\
& \left(\begin{array}{l} \{(\rho_k(s_0, \varphi_k(s'')), h'') \mid (s'', h'') \in c_k(s_0, h)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c_k(s_0, h)\} \cup \{\bot \mid \bot \in c_k(s_0, h)\} \end{array}\right) \\
= & \quad \because (s'', h'') \in c_k(s_0, h) \text{ implies } s''(z) = s_0(z) \text{ for all } z \in Z_0 - k(X_w) \\
& \left(\begin{array}{l} \{(s'', h'') \mid (s'', h'') \in c_k(s_0, h)\} \\ \cup \{\mathit{fault} \mid \mathit{fault} \in c_k(s_0, h)\} \cup \{\bot \mid \bot \in c_k(s_0, h)\} \end{array}\right) \\
= & \\
& c_k(s_0, h)
\end{array}
$$

□

**Lemma 112** *$[\![\mathsf{assert}]\!]$ and $[\![\mathsf{spec}]\!]$ are sheaves in $\mathbf{Sh}(\mathbf{W}, J)$. Moreover, $[\![\mathsf{assert}]\!](f)$ is injective for all $f$, and $[\![\mathsf{spec}]\!]$ is a subobject classifier in the topos $\mathbf{Sh}(\mathbf{W}, J)$.*

The well-definedness of the interpretation of base types has one immediate consequence: the semantic counterpart of the symmetric noninterference predicate in Section 9.3.1 is well-defined because the proviso for the semantic counterpart being well-definedness is met by the interpretation of IAh types and the assertion type.

**Corollary 113** *For all IAh types $\theta$, $[\![\theta]\!]$ has the minimal resource property in Section 9.3.1. And so does $[\![\mathsf{assert}]\!]$.*

*Proof:* All of $U \circ [\![\theta]\!]$ and $[\![\mathsf{assert}]\!]$ are injective sheaves, where $U$ is the forgetful functor from **mCpo** to **Set**. The conclusion follows by Lemma 107 and 108. □

### 9.3.3 The Interpretation of Terms

Each term $\Gamma \triangleright M : \theta$ in IAh is interpreted as a morphism from $[\![\Gamma]\!]$ to $[\![\theta]\!]$ in $\mathbf{C}$, where the typing context $\Gamma$ is interpreted as a named product $\Pi_{x \in dom(\Gamma)} [\![\Gamma(x)]\!]$. The interpretation of terms in

$$
\begin{array}{rcl}
[\![c]\!]_W\eta & = & [\![c]\!]_W(*) \\
[\![x]\!]_W\eta & = & \eta(x) \\
[\![\lambda x{:}\,\theta.\,M]\!]_W\eta & = & \{\lambda a{:}\,[\![\theta]\!]_X.\,[\![M]\!]_X(\eta\cdot f)[x\mapsto a]\}_{f:X\to W\in\mathbf{W}} \\
[\![M_0\ M_1]\!]_W\eta & = & ([\![M_0]\!]_W\eta)_{\mathsf{id}_W}([\![M_1]\!]_W\eta) \\
[\![\langle M_0,\,M_1\rangle]\!]_W\eta & = & ([\![M_0]\!]_W\eta,\,[\![M_1]\!]_W\eta) \\
[\![\mathsf{fst}\ M]\!]_W\eta & = & \mathsf{fst}\ ([\![M]\!]_W\eta) \\
[\![\mathsf{snd}\ M]\!]_W\eta & = & \mathsf{snd}\ ([\![M]\!]_W\eta)
\end{array}
$$

**Table 9.5** Semantics of Terms in IAh

IAh other than constants is given using the cartesian closed structure of **C**. A constant $c{:}\,\theta$ is interpreted as a morphism from a terminal object 1 to $[\![\theta]\!]$. Table 9.5 shows the interpretation of terms in IAh and Table 9.6 the interpretation of constant terms in IAh. In Table 9.6, we often use the isomorphism between $\mathsf{Hom}[1, F \Rightarrow G]$ and $\mathsf{Hom}[F, G]$. So, for a constant $c{:}\,\theta_1 \to \theta_2$, we define $[\![c]\!]$ as a morphism from $[\![\theta_1]\!]$ to $[\![\theta_2]\!]$ instead of a morphism from 1 to $[\![\theta_1]\!] \Rightarrow [\![\theta_2]\!]$.

**Lemma 114** *The interpretation of constant terms in IAh in Table 9.6 is well-defined: all definitions give rise to natural transformations.*

Each assertion $\Gamma \triangleright P{:}\,\mathsf{assert}$ is interpreted as a natural transformation from $U \circ [\![\Gamma]\!]$ to $[\![\mathsf{assert}]\!]$, where $U$ is the forgetful functor from **mCpo** to **Set**. The interpretation is the standard semantics of the **BI** assertion language and is shown in Table 9.7. The detailed explanation can be found in Chapter 5.

Each specification $\Gamma \triangleright \varphi{:}\,\mathsf{spec}$ is interpreted as a natural transformation from $U \circ [\![\Gamma]\!]$ to $[\![\mathsf{spec}]\!]$. Instead of giving a natural transformation, we use the fact that $[\![\mathsf{spec}]\!]$ is a subobject classifier so that a natural transformation from $U \circ [\![\Gamma]\!]$ to $[\![\mathsf{spec}]\!]$ can be represented as a subobject of $U \circ [\![\Gamma]\!]$. Such a presentation is well-known as Kripke-Joyal semantics [32]. The interpretation is shown in Table 9.8, where we use $W, \eta \models \varphi$ to denote that $\eta$ is in $[\![\varphi]\!](W)$ when $[\![\varphi]\!]$ is viewed as a subobject of $U \circ [\![\Gamma]\!]$. Note that the interpretation of a Hoare triple requires that all states satisfying the precondition should not lead to memory faults in addition to the usual partial correctness requirement. Such a requirement guarantees that a state in the precondition doesn't lead to any illegal memory access such as dangling pointer dereference. The symmetric noninterference predicate is interpreted using the semantic version of the symmetric noninterference predicate discussed in Section 9.3.1, which intuitively denotes that when a variable is accessed by two

$$[\![\mathsf{Y}_\theta]\!]_W(m) \triangleq \mathsf{fix}\ m_{\mathsf{id}_W}$$

$$[\![\mathsf{skip}]\!]_W(*) \triangleq \lambda(s,h).\,\{(s,h)\}$$

$$[\![\_;\_]\!]_W(c_1,c_2) \triangleq \lambda(s,h).\begin{pmatrix} \bigcup_{(s',h')\in c_1(s,h)} c_2(s',h') \\ \cup\{\bot \mid \bot \in c_1(s,h)\} \cup \{\mathit{fault} \mid \mathit{fault} \in c_1(s,h)\}\end{pmatrix}$$

$$[\![\mathsf{get}]\!]_W(v) \triangleq \lambda s.\begin{cases}\bot & \text{if } v=\bot\\ s(v) & \text{otherwise}\end{cases}$$

$$[\![\_ := \_]\!]_W(v,e) \triangleq \lambda(s,h).\begin{cases}\{\bot\} & \text{if } e(s)=\bot \vee v=\bot\\ \{(s[v\mapsto e(s)],h)\} & \text{otherwise}\end{cases}$$

$$\begin{aligned}&[\![\mathsf{new}]\!]_W(p)(s,h)\\ &\triangleq \bigcup_{x\notin W_0}\{\pi(p_{\mathsf{inc}}(x)(s',h)) \mid s' \in \mathit{Stack}(W_0\cup\{x\}, W_w\cup\{x\}) \wedge \varphi_{\mathsf{inc}}(s')=s\}\\ &\begin{pmatrix}\text{where } \mathsf{inc}\text{: } (W_0\cup\{x\}, W_w\cup\{x\}) \to W \text{ is an inclusion from } W_0 \text{ to } W_0\cup\{x\}\\ \text{and } \pi(S) \text{ is } \{(\varphi_{\mathsf{inc}}(s''),h'') \mid (s'',h'')\in S\} \cup \{\mathit{fault} \mid \mathit{fault}\in S\} \cup \{\bot \mid \bot \in S\}\end{pmatrix}\end{aligned}$$

$$\begin{aligned}&[\![\_ := \_.1]\!]_W(v,e)(s,h)\\ &\triangleq \begin{cases}\{\bot\} & \text{if } e(s)=\bot \vee v=\bot\\ \{(s[v\mapsto l_1],h)\} & \text{if } h(e(s))=(l_1,l_2)\wedge v\neq\bot\\ \{\mathit{fault}\} & \text{otherwise}\end{cases}\end{aligned}$$

$$\begin{aligned}&[\![\_.1 := \_]\!]_W(e_1,e_2)(s,h)\\ &\triangleq \begin{cases}\{\bot\} & \text{if } e_1(s)=\bot \vee e_2(s)=\bot\\ \{(s,h[e_1(s)\mapsto(l_1',l_2)]\} & \text{if } h(e_1(s))=(l_1,l_2)\wedge e_2(s)=l_1'\neq\bot\\ \{\mathit{fault}\} & \text{otherwise}\end{cases}\end{aligned}$$

$$\begin{aligned}&[\![\_ := \mathsf{cons}(\_,\_)]\!]_W(v,e_1,e_2)(s,h)\\ &\triangleq \begin{cases}\{\bot\} & \begin{aligned}&\text{if } v=\bot\\ &\vee e_1(s)=\bot\\ &\vee e_2(s)=\bot\end{aligned}\\ \{(s[v\mapsto l], h*(l\mapsto e_1(s),e_2(s))) \mid l\in \mathit{Loc}-\mathit{dom}(h)\} & \text{otherwise}\end{cases}\end{aligned}$$

$$[\![\mathsf{nil}]\!]_W(*) \triangleq \lambda s.\,\mathsf{nil}$$

$$[\![\mathsf{ifeq}_{\mathsf{com}}]\!]_W(e_1,e_2,c_1,c_2) \triangleq \lambda(s,h).\begin{cases}\{\bot\} & \text{if } e_1(s)=\bot\vee e_2(s)=\bot\\ c_1(s,h) & \text{if } e_1(s)\neq\bot\wedge e_1(s)=e_2(s)\\ c_2(s,h) & \text{otherwise}\end{cases}$$

$$[\![\mathsf{coerce}]\!]_W(v) \triangleq \lambda s.\,v$$

**Table 9.6** Semantics of Constant Terms

$$
\begin{array}{rcl}
[\![E \mapsto E_0, E_1]\!]_W \eta & = & \{(s,h) \mid \{[\![E]\!]_W \eta(s)\} = dom(h) \text{ and} \\
& & \qquad h([\![E]\!]_W \eta(s)) = ([\![E_0]\!]_W \eta(s), [\![E_1]\!]_W \eta(s))\} \\
[\![E_0 = E_1]\!]_W \eta & = & \{(s,h) \mid [\![E_0]\!]_W \eta(s) = [\![E_1]\!]_W \eta(s)\} \\
[\![\mathsf{F}]\!]_W \eta & = & \emptyset \\
[\![P_0 \Rightarrow P_1]\!]_W \eta & = & (State(W) - [\![P_0]\!]_W \eta) \cup [\![P_1]\!]_W \eta \\
[\![\forall x.P]\!]_W \eta & = & \{(s,h) \mid \forall v \in [\![\mathsf{val}]\!].\, (s,h) \in [\![P]\!]_W (\eta[x \mapsto v])\} \\
[\![\mathsf{emp}]\!]_W \eta & = & \{(s, \emptyset)\} \\
[\![P_0 * P_1]\!]_W \eta & = & \{(s, h_0 * h_1) \mid h_0 \# h_1 \text{ and} \\
& & \qquad (s,h_0) \in [\![P_0]\!]_W \eta \text{ and } (s,h_1) \in [\![P_1]\!]_W \eta\} \\
[\![P_0 -\!\!* P_1]\!]_W \eta & = & \{(s,h) \mid \text{for all } h' \text{ such that } h\#h' \text{ and } (s,h') \in [\![P_0]\!]_W \eta, \\
& & \qquad (s, h * h') \in [\![P_1]\!]_W \eta\}
\end{array}
$$

**Table 9.7** Semantics of Assertions

noninterfering semantic entities, the variable is only read by them. One might worry that the symmetric noninterference predicate might not be well-defined because the semantic version of the noninterference predicate assumes that functors have the minimal resource property. Recall that the noninterference predicate is used only for program phrases or assertions, and that all IAh types and the assertion type are interpreted as functors with the minimal resource properties. Therefore, the interpretation of the noninterference predicate is well-defined. All other connectives in intuitionistic logic are interpreted following Kripke-Joyal semantics.

**Lemma 115** *The interpretation of a specification $\Gamma \rhd \varphi{:}\,\mathsf{spec}$ in Table 9.8 is well-defined in the sense that it induces a subsheaf of $U \circ [\![\Gamma]\!]$; equivalently,*

- *for all $f{:}\, X \to W$ in $\mathbf{W}$ and $\eta \in (U \circ [\![\Gamma]\!])(W)$, if $W, \eta \models \varphi$, then $X, (\eta \cdot f) \models \varphi$; and*
- *for all covering sieves $S \in J(W)$ and $\eta \in (U \circ [\![\Gamma]\!])(W)$, if $dom(f), (\eta \cdot f) \models \varphi$ for all $f \in S$, then $W, \eta \models \varphi$.*

### 9.3.4 Soundness of Reasoning Principles

The semantics of IAh, assertions and specifications validate all reasoning principles for specifications in Section 9.2 as well as axioms for assertions in Chapter 5. In this section, we show that the Frame Rule and axioms for the symmetric noninterference predicate are valid in the semantics. We use the usual notion of validity in Kripke-Joyal semantics in a topos

| | | |
|---|---|---|
| $W, \eta \models M \equiv M'$ | iff | $[\![M]\!]_W \eta = [\![M']\!]_W \eta$ |
| $W, \eta \models \{P\}\, C\, \{P'\}$ | iff | for all $(s,h) \in [\![P]\!]_W \eta$, $(s',h') \in [\![C]\!]_W \eta(s,h)$ implies $(s',h') \in [\![P']\!]_W \eta$ and $\mathit{fault} \notin [\![C]\!]_W \eta(s,h)$ |
| $W, \eta \models M \bot M'$ | iff | $[\![M]\!]_W \eta \bot [\![M']\!]_W \eta$ |
| $W, \eta \models \varphi \wedge \psi$ | iff | $W, \eta \models \varphi$ and $W, \eta \models \psi$ |
| $W, \eta \models \varphi \Rightarrow \psi$ | iff | for all $f : X \to W$ in $\mathbf{W}$, $X, (\eta \cdot f) \models \varphi$ implies $X, (\eta \cdot f) \models \psi$ |
| $W, \eta \models \forall x{:}\, \theta.\, \varphi$ | iff | for all $f : X \to W$ in $\mathbf{W}$ and for all $a \in [\![\theta]\!]_X$, $X, (\eta \cdot f)[x \mapsto a] \models \varphi$ |

**Table 9.8** Semantics of Specifications

$\mathbf{Sh}(\mathbf{W}, J)$: $\Gamma \triangleright \varphi \models \psi$ iff for all $W$ and $\eta \in (U \circ [\![\Gamma]\!])(W)$, $W, \eta \models \varphi$ implies $W, \eta \models \psi$. This section contains a proof of the following claim:

**Proposition 116** *The Frame Rule and axioms for the symmetric noninterference predicate in Table 9.4 are valid in the semantics in the sense that if $\psi$ is derived from $\varphi_1, \ldots, \varphi_n$ and $\Gamma$ includes all free identifiers both in $\psi$ and $\varphi_1, \ldots, \varphi_n$, we have $\Gamma \triangleright \varphi_1 \wedge \ldots \wedge \varphi_n \models \psi$.*

We handle axioms for the noninterference predicate first. Just to avoid any confusion, we say "semantic $\bot$" to denote the semantic version of the noninterference predicate in Section 9.3.1. We first note that the semantic $\bot$ is symmetric, can be decomposed and is preserved by natural transformations.

**Lemma 117** *For functors $F, G, H, F', G'$ in $\mathbf{Set}^{\mathbf{W}^{op}}$ with the minimal resource property, $a \in F(X)$, $b \in G(X)$, $c \in H(X)$ and two natural transformations $\mu : F \to F'$, $\gamma : G \to G'$, we have the following facts:*

1. *If $a \bot b$, then $b \bot a$.*
2. *If $a \bot c$ and $b \bot c$, then $(a, b) \bot c$.*
3. *If $a \bot b$, then $\mu_X(a) \bot \gamma_X(b)$.*

*Proof:* The first is immediate from the definition. For the second, we use the following fact from set theory: for sets $A_0, A_w, B_0, B_w$ with $A_w \subseteq A_0$ and $B_w \subseteq B_0$,

$$A_0 \cap B_0 \subseteq (A_0 - A_w) \cap (B_0 - B_w) \text{ iff } A_0 \cap B_w = \emptyset \text{ and } B_0 \cap A_w = \emptyset.$$

So, it suffices to show that both $\mathsf{supp}(a,b)_0 \cap \mathsf{supp}(c)_w$ and $\mathsf{supp}(c)_0 \cap \mathsf{supp}(a,b)_w$ are empty sets. Such a disjointedness result follows from the fact that $\mathsf{supp}(a,b)_0$ and $\mathsf{supp}(a,b)_w$ are included in $\mathsf{supp}(a)_0 \cup \mathsf{supp}(b)_0$ and $\mathsf{supp}(a)_w \cup \mathsf{supp}(b)_w$, respectively.

For the third, it suffices to show that $\mu_X(a) \perp b$; then, the symmetry of $\perp$ and the similar proof show the remaining half. Because $\mu$ is natural, we have $\mathsf{supp}(\mu_X(a))_0 \subseteq \mathsf{supp}(a)_0$ and $\mathsf{supp}(\mu_X(a))_w \subseteq \mathsf{supp}(a)_w$. So,

$$\mathsf{supp}(a)_0 \cap \mathsf{supp}(b)_0 \quad \subseteq \quad (\mathsf{supp}(a)_0 - \mathsf{supp}(\mu_X(a))_w) \cap (\mathsf{supp}(b)_0 - \mathsf{supp}(b)_w)$$

The conclusion follows by taking the intersection of $\mathsf{supp}(\mu_X(a))_0$ with both sides in the above inclusion. $\square$

Noninterference Decomposition and Noninterference Symmetry follow from properties of the semantic $\perp$ summarized in the above lemma. To see why Noninterference Decomposition is valid, consider $W, \eta$ such that $W, \eta \models FV(M) \perp N$. We should show that $[\![M]\!]_W \eta \perp [\![N]\!]_W \eta$. Let $\eta'$ be the restriction of $\eta$ to $FV(M)$. Then, $\eta' \perp [\![N]\!]_W \eta$ because for all $x \in FV(M)$, $\eta'(x) \perp [\![N]\!]_W \eta$ and the semantic $\perp$ has a decomposition property (Lemma 117). Since the semantic $\perp$ is preserved by a natural transformation (Lemma 117), we have that $[\![M]\!]_W \eta' \perp [\![N]\!]_W \eta$. The conclusion follows since $[\![M]\!]_W \eta' = [\![M]\!]_W \eta$.

Intuitively, the rule of Passivity holds because when both arguments in the interference predicate are of passive type, they don't write to any variables. In Section 9.3.1, we defined a category $\mathbf{P}$, which formalizes such an intuition because for all $F \in \mathbf{P}$ and $a \in F(X)$, $\mathsf{supp}(a)_w = \emptyset$. So, when $F$ and $G$ are in $\mathbf{P}$, $a \perp b$ for all $a \in F(X)$ and $b \in G(X)$. Using such a fact, we prove the soundness of Passivity by verifying that each passive type is interpreted in $\mathbf{P}$. In fact, we only need to show that $\mathsf{val}$ and $\mathsf{exp}$, which are the base cases in the grammar for passive types, are interpreted as functors in $\mathbf{P}$; then, all passive types are interpreted in $\mathbf{P}$ because of Lemma 110.

**Lemma 118** $\mathsf{val}$ *and* $\mathsf{exp}$ *are interpreted as functors in* $\mathbf{P}$*.*

*Proof:* $[\![\mathsf{val}]\!]$ is a constant functor so it is in $\mathbf{P}$. For $[\![\mathsf{exp}]\!]$, let $f$ be a morphism from $X$ to $X'$ in $\mathbf{W}$ which, viewed as a function, is a bijection. Since $[\![\mathsf{exp}]\!]$ is in $\mathbf{C}$, $[\![\mathsf{exp}]\!](f)$ is an order embedding. So, it suffices to show that $[\![\mathsf{exp}]\!](f)$ is surjective. The bottom in $[\![\mathsf{exp}]\!](X)$ is in the image of $[\![\mathsf{exp}]\!](f)$ because $[\![\mathsf{exp}]\!](f)$ is strict. For all other $e \in ([\![\mathsf{exp}]\!](X) - \{\bot\})$, define $e'$ as $\lambda s.\, e(s \circ f^{-1})$. Then, $e' \cdot f = e$. $\square$

The soundness of the rule New relies on showing that all nonlocal entities don't interfere symmetrically with a locally declared variable. That is, for all worlds $W$ in $\mathbf{W}$, an environment $\eta \in (U \circ [\![\Gamma]\!])(W)$ and a new variable $y \notin W_0$, a nonlocal entity $(\eta \cdot \mathsf{inc})(u)$ doesn't interfere with a new local variable $y$ where $u \in \mathit{dom}(\Gamma)$, $\mathsf{inc}\colon (W_0 \cup \{y\}, W_w \cup \{y\}) \to W$ in $\mathbf{W}$ is an inclusion from $W_0$ to $W_0 \cup \{y\}$ and $y$ is considered as an element in $[\![\mathsf{var}]\!]_{(W_0 \cup \{y\}, W_w \cup \{y\})}$; then, the premise says that $(W_0 \cup \{y\}, W_w \cup \{y\}), (\eta \cdot \mathsf{inc})[x \mapsto y] \models \{P\}\, M(x)\, \{Q\}$ holds. From such a fact, we obtain $W, \eta \models \{P\}\, \mathsf{new}\ M\, \{Q\}$ using the freshness of the identifier $x$ and the naturality of $[\![M]\!], [\![P]\!], [\![Q]\!]$.

Showing the soundness of the Frame Rule is similar to the proof of the Frame Rule for **WHILEh** in Chapter 5. Suppose that the premises of the rule hold: $W, \eta \models \{P\}\, C\, \{Q\}$ and $W, \eta \models R \bot C$. We should show that $W, \eta \models \{P * R\}\, C\, \{Q * R\}$. Consider a state $(s, h)$ in $\mathit{State}(W)$ satisfying the precondition $[\![P * R]\!]_W \eta$. By the interpretation of $*$, there are $h_1, h_2$ such that $h = h_1 * h_2$, $(s, h_1) \in [\![P]\!]_W \eta$ and $(s, h_2) \in [\![R]\!]_W \eta$. The first requirement is to show that a memory fault is not generated by $[\![C]\!]_W \eta(s, h_1 * h_2)$. We obtain such a result applying Safety Monotonicity of $[\![C]\!]_W \eta$ to $\mathit{fault} \notin [\![C]\!]_W \eta(s, h_1)$, which itself follows from the fact that the Hoare triple $\{P\}\, C\, \{Q\}$ holds in a world $W$ and an environment $\eta$. The second requirement is to prove that all final states belong to the postcondition $[\![Q * R]\!]_W \eta$. Pick a final state $(s', h')$ in $[\![C]\!]_W \eta(s, h_1 * h_2)$. Since $\mathit{fault} \notin [\![C]\!]_W \eta(s, h_1)$, Heap Locality of $[\![C]\!]_W \eta$ implies that $h'$ should be decomposed into $h_1'$ and $h_2$ such that $(s', h_1') \in [\![C]\!]_W \eta(s, h_1)$. Note that $(s', h_1')$ must be in $[\![Q]\!]_W \eta$ because $W, \eta \models \{P\}\, C\, \{Q\}$ holds. So, it remains to show that $(s', h_2) \in [\![R]\!]_W \eta$; then, $(s', h_1' * h_2) \in [\![P * R]\!]_W \eta$ by definition. To see that $(s', h_2) \in [\![R]\!]_W \eta$, note that for all variables $x$ which $[\![R]\!]_W \eta$ depends on $(x \in \mathsf{supp}([\![R]\!]_W \eta)_0)$, we have $s(x) = s'(x)$ because $[\![R]\!]_W \eta \perp [\![C]\!]_W \eta$. Moreover, $(s, h_2)$ is already in $[\![R]\!]_W \eta$ by the assumption. Therefore, $(s', h_2)$ belongs to $[\![R]\!]_W \eta$.

# CHAPTER 10

## Conclusion and Future Work

In the thesis, we have provided theoretical and practical justification for the promise of local reasoning. We have shown that local reasoning can be successfully realized by the Frame Rule: the rule is a sound and complete method to derive a "global" property of a program from a "local" one, and moreover the rule is easy to use in practice. In particular, the symmetric noninterference predicate and the spatial conjunction connective, which play an important role in the rule, have good reasoning principles. Reasoning principles for symmetric noninterference are mostly based on simple syntactic checking, and they remain valid even when language constructs and reasoning principles for data abstraction are considered. And reasoning about spatial conjunctions can even be automated in certain cases. We believe that such results in the thesis indicate that the idea of local reasoning can bring formal verification closer to the development process of a program in practice.

The development of local reasoning in the thesis is, however, not complete. Firstly, it is not completely resolved how to incorporate the notion of passivity for heap cells in the Frame Rule. When a Hoare triple $\{P * R\}\, C\, \{Q * R\}$ is derived using the Frame Rule, $C$ is not allowed to access any heap cells which $R$ depends on. Such a requirement is too strict because $C$ can read such heap cells without invalidating the reasoing. In fact, such an issue of passivity has already appeared in the verification of the Schorr-Waite graph marking algorithm in Chapter 7. The situation in Chapter 7 is as follows: suppose that $R$ is a property involving a heap cell $(y \mapsto y_1, y_2)$ (more precisely, $R$ implies that $(y \mapsto y_1, y_2) * \mathsf{T}$) and that $C$ accesses the heap cell $y$ and a different cell $(x \mapsto x_1, x_2)$ but only modifies the cell $x$ to contain a pair $(x'_1, x'_2)$. We want to reason as follows:

$$\frac{\{(x \mapsto x_1, x_2) * (y \mapsto y_1, y_2)\}\, C\, \{(x \mapsto x_1', x_2') * (y \mapsto y_1, y_2)\}}{\{(x \mapsto x_1, x_2) * R\}\, C\, \{(x \mapsto x_1', x_2') * R\}}$$

Even though the derivation is sound, it is not a direct instance of the Frame Rule: the cell $y$ is "included" in $R$ in the conclusion.

The tentative solution in Chapter 7 is to use the spatial implication to "take out" certain heap cells from $R$. That is, instead of trying to conjoining $R$ to the given Hoare triple, $(y \mapsto y_1, y_2) \mathrel{-\!\!*} R$ is conjoined. Using the Frame Rule followed by the rule of Consequence, we can derive the intended Hoare triple:

$$\frac{\dfrac{\{(x \mapsto x_1, x_2) * (y \mapsto y_1, y_2)\}\, C\, \{(x \mapsto x_1', x_2') * (y \mapsto y_1, y_2)\}}{\{(x \mapsto x_1, x_2) * (y \mapsto y_1, y_2) * ((y \mapsto y_1, y_2) \mathrel{-\!\!*} R)\}\, C\, \{(x \mapsto x_1', x_2') * (y \mapsto y_1, y_2) * ((y \mapsto y_1, y_2) \mathrel{-\!\!*} R)\}}}{\{(x \mapsto x_1, x_2) * R\}\, C\, \{(x \mapsto x_1', x_2') * R\}}$$

where Consequence uses the following equivalence:

$$(y \mapsto y_1, y_2) * ((y \mapsto y_1, y_2) \mathrel{-\!\!*} R) \iff R$$

However, the solution in Chapter 7 is not satisfactory because it requires that a given Hoare triple is precise enough to specify which heap cells are not changed. For instance, in the example, the given Hoare triple says that the heap cell $y$ is not changed by having $- * (y \mapsto y_1, y_2)$ both in pre- and post- conditions; if it would be

$$\{(x \mapsto x_1, x_2) * (y \mapsto y_1, y_2)\}\, C\, \{(x \mapsto x_1', x_2') * \top\},$$

the idea in Chapter 7 could not have helped the case. In the desired solution, a given Hoare triple should only specify which cells are only read and how a command modifies other heap cells; from such a Hoare triple, one should be able to derive a global Hoare triple. With O'Hearn, we are considering to use a notion of passive (i.e., read-only) heap cells and a passivity modality ! to specify that $!P$ holds in a state if $P$ holds in the state and all allocated heap cells are passive. However, it requires further investigation to see where such an idea leads to.

Secondly, in the thesis, we have not found *all* reasoning principles for the spatial connectives: a complete set of axioms for the quantification-free fragment of the **BI** assertion language hasn't been found. Even though the decidability in Chapter 8 indicates that such axiomatization is possible, finding elegant complete axioms requires another insight into the **BI** assertion language.

Thirdly, the programming language in the thesis doesn't consider stored code. Indeed, giving a decent denotational semantics for programs with code pointers and devising good reasoning principles for such programs have been open problems in the programming language community. Programs with code pointers are difficult to interpret using denotational semantics because the meanings of the store and types depend on one another in a complicated way. Even if it is possible to have a semantics, the semantics often provides no insights into programs and reasoning principles. One approach is to restrict the use of stored code, such as to prevent stored code from referring to itself, in order to break the mutual dependency between the interpretation of the store and that of types [43]. Another approach is to use operational semantics to validate reasoning principles for programs with code pointers [34, 22]. Recently, O'Hearn and Calcagno followed the second line of research in order to validate reasoning principles, including a version of the Frame Rule, for programs with code pointers [8]. However, various issues regarding code pointers, such as how to handle callbacks, are not resolved yet [8].

In the thesis, we formulated the Frame Rule with both the symmetric noninterference predicate and the spatial conjunction. One might wonder whether they can be combined into a single notion because both of them are essentially intended to state that two phrases use disjoint portions of the store. The fourth question for future research is whether we really need two independent ideas or we can combine them into a single notion.

The fifth question is whether **BI** pointer logic is relatively complete in the sense of Cook [11]. The main issue in this question is whether one can express all necessary loop invariants with assertions. For deterministic or finitely nondeterministic programs, the question is usually resolved by showing that all recursive enumerable properties can be expressed with assertions. However, it is not clear whether such an approach is applicable in our case because **WHILE**h has a countably nondeterministic construct cons. For instance, in the case of total correctness, it has been shown that an assertion language should be able to express more than recursively enumerable properties in order for programming logic with such assertions to be relatively complete [1].

Finally, even though the thesis mainly concerns local reasoning in program verification, the idea of local reasoning may as well be applied to a program analysis. For instance, an analysis algorithm can be designed to handle each procedure of a program separately while keeping track of which portion of the store is used; then, when the procedure is called, it instantiates such an information in a context-sensitive manner by incorporating a property for the unaccessed portion of the store as an invariant. Indeed, the idea of getting a generic information of a procedure and instantiating it differently according to the context is used by Rehof and Fähndrich to obtain an efficient but precise flow analysis algorithm [61]. We believe that obtaining context-sensitivity by incorporating a property for the unaccessed portion of the store can also lead to improvements in a certain program analysis.

# REFERENCES


[1] K. Apt and G. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, October 1986.

[2] K. R. Apt. Ten years of Hoare's logic: A survey. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, October 1981.

[3] R.-J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[4] R. J. R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In Bernhard Möller, editor, *Mathematics of Program Construction*. Springer LNCS 957, 1995.

[5] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, Berlin, 1998.

[6] R. Bornat. Proving pointer programs in Hoare Logic. In *Mathematics of Program Construction*, July 2000.

[7] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[8] C. Calcagno. A logic for objects. The third workshop of the ESPRIT Working Group Applied Semantics (APPSEM), March 2001.

[9] C. Calcagno, H. Yang, and P. O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. submitted, 2001.

[10] Jr. Clarke, E. M. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *J. ACM*, 26(1):129–147, January 1979.

[11] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Computing*, 7(1):70–90, 1978.

[12] S. A. Cook and D.C. Oppen. An assertion language for data structures. In *Principles of Programming Languages*, pages 160–166. ACM, 1975.

[13] M. Dawson and P. Taylor, editors. *Hypatia Electronic Library*. Queen Mary and Westfield College, URL //hypatia.dcs.qmw.ac.uk.

[14] D. Detlefs, R. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report Research Report 156, Compaq System Reserch Center, July 1998.

[15] D. Detlefs, R. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq System Reserch Center, December 1998.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[17] W. H. Hesselink. *Programs, Recursion and Unbounded Choice.* Cambridge University Press, Cambridge, 1992.

[18] W. H. Hesselink. Predicate transformers for recursive procedures with local variables. *Formal Aspects of Computing*, 11:616–636, 1999.

[19] C. Hoare and J. He. A trace model for pointers and objects. In R. Guerraoui, editor, *ECOOP '99*, volume 1628 of *LNCS*, pages 1–17. Springer-Verlag, 1999.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–583, 1969.

[21] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symp. Semantics of Algorithmic Languages*, volume 188 of *Lect. Notes Math.*, pages 102–116. Springer-Verlag, 1971.

[22] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable type logic of effects. *Information and Computation*, 119(1):55–90, May 1995.

[23] S. Ishtiaq. BI is strongly expressive. available at http://www.dcs.qmw.ac.uk/˜si/, November 2000.

[24] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, January 2001.

[25] P. T. Johnstone. *Topos Theory.* Academic Press, New York, 1977.

[26] P. T. Johnstone. A topos-theorist looks at dilators. *Journal of Pure and Applied Algebra*, 58:235–249, 1989.

[27] Y. Kinoshita, P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. An axiomatic approach to binary logical relations with applications to data refinement. Manuscript, Queen Mary and Westfield, London, 1997.

[28] T. Kleymann. Hoare Logic and Auxiliary Variables. *Formal Aspects of Computing*, 11:541–566, 1999.

[29] R. Leino. *Toward reliable modular programs.* PhD thesis, California Institute of Technology, 1995.

[30] R. Leino and G. Nelson. Data abstraction and information hiding. Technical Report Research Report 160, Compaq System Reserch Center, November 2000.

[31] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Puting Static Analysis to Work for Verification: A Case Study. In *ISSTA 2000*, August 2000.

[32] S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic.* Springer-Verlag, 1991.

[33] Clare E. Martin. Preordered categories and predicate transformers. Dissertation, Oxford University, 1991.

[34] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Comput. Sci.*, 105(2):167–215, 1992.

[35] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 191–203. ACM, 1988. (Reprinted as Chapter 7 of [54]).

[36] C. Morgan, editor. *Programming from Specifications, 2nd Edition.* Prentice-Hall, 1994.

[37] C. Morgan and T. Vickers, editors. *On the Refinement Calculus.* Springer-Verlag, 1992.

[38] C. C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3), Jul 1988. Reprinted in [37].

[39] C. C. Morgan. The cuppest capjunctive capping, and galois. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honor of C. A. R. Hoare*, pages 1–16. Prentice-Hall International, 1994.

[40] J. M. Morris. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology (Proceedings of the 1981 Marktoberdorf Summer School)*, pages 25–51. Reidel, 1982.

[41] S. Muchnick. *Advanced Compiler Design And Implementation.* Academic Press, New York, 1997.

[42] D. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 2000. To appear.

[43] D. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Science of Computer Programming*, 2000. To appear.

[44] David A. Naumann. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science*, 8(4):351–399, August 1998.

[45] David A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Comput. Sci.*, 2001. To appear.

[46] F. Nielson, H. Nielson, and C. Henkin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[47] P. O'Hearn, J. Reynolds, and H. Yang. Towards local reasoning about programs that alter data structures. (to appear in CSL 2001), 2001.

[48] P. W. O'Hearn. *Semantics of Noninterference: A Natural Approach.* PhD thesis, Queen's University, Kingston, Canada, 1990.

[49] P. W. O'Hearn. A model for syntactic control of interference. *Math. Struct. Comput. Sci.*, 3:435–465, 1993.

[50] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semanatics: Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theor. Comput. Sci.* Elsevier, 1995. (Reprinted as Chapter 18 of [54]).

[51] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 171–184. ACM, 1993.

[52] P. W. O'Hearn and R. D. Tennent. Semantical analysis of specification logic, Part 2. *Inf. Comput.*, 107(1):25–57, 1993. (Reprinted as Chapter 14 of [54]).

[53] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995. (Reprinted as Chapter 16 of [54]).

[54] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages (Two volumes).* Birkhäuser, Boston, 1997.

[55] Peter W. O'Hearn and John C. Reynolds. From algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, January 2000.

[56] P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.

[57] S. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Comm. ACM*, 19(5):279–285, 1976.

[58] G. D. Plotkin. Lambda definability in the full type hierarchy. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, New York, 1980.

[59] U. S. Reddy. When parametricity implies naturality. Electronic manuscript, [13], July 1997.

[60] U. S. Reddy. Objects and classes in Algol-like languages. In *Fifth Intern. Workshop on Foundations of Object-oriented Languages.* electronic proceedings at http://pauillac.inria.fr/~remy/fool/proceedings.html, Jan 1998.

[61] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Principles of Programming Languages*, January 2001.

[62] J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978. (Reprinted as Chapter 10 of [54]).

[63] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall International, London, 1981.

[64] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981. (Reprinted as Chapter 3 of [54]).

[65] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge Univ. Press, 1982. (Reprinted as Chapter 6 of [54]).

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

[67] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*. Palgrave, 2000.

[68] John C. Reynolds. Lectures on reasoning about shared mutable data structure. IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic, Tandil, Argentina, September 6-13, 2000.

[69] N. Rinetzky. Interprocedural shape analysis. Master's thesis, Israel Institute of Technology, 2000.

[70] E. Robinson. Logical relations and data abstraction. available at ftp://ftp.dcs.qmw.ac.uk/pub/lfp/edmundr/data-abstraction.ps, Aug. 1996.

[71] E. Robinson and G. Rosolini. Reflexive graphs and parametric polymorphism. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, July 1994.

[72] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, 200.

[73] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, January 1996.

[74] R. D. Tennent. Semantical analysis of specification logic. *Inf. Comput.*, 85(2):135–162, 1990. (Reprinted as Chapter 13 of [54]).

[75] D. Walker. Mechanical reasoning about low-level programming languages. 15-819A3 Class Notes, available at http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/www15819A3s2001/15819A3desc.html, March 2001.

[76] D. Walker and G. Morrisett. Alias types for recursive data structures. Workshop on Types in Compilation, September 2000.

[77] H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. SPACE 2001 workshop, available at http://www.students.uiuc.edu/~hyang/paper/SchorrWaite.ps.gz, 2001.

[78] H. Yang and U. S. Reddy. On the semantics of refinement calculi. In *Foundations of Software Science and Computation Structures*, pages 359–374. Springer-Verlag, 2000.

[79] H. Yang and U. S. Reddy. Parametric sheaves for modelling store locality. (submitted), July 2000.

# VITA

Hongseok Yang was born May 21, 1974 in Korea. He received a Bachelor of Science in Korea Advanced Institute of Science and Technology, Korea, in February 1996 with major in Computer Science and minor in Mathematics. He entered graduate school at the University of Illinois at Urbana-Champaign in the fall of 1996. His research interest includes programming with states, such as object-oriented or imperative programming; substructural logic, especially, the logic of Bunched Implications; semantics, type system and reasoning methods; category theory and sheaf theory.