Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic

Neelakantan R. Krishnaswami

CMU-CS-12-127

July 6, 2011

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Co-chair
John C. Reynolds, Co-chair
Robert Harper
Lars Birkedal, IT University of Copenhagen

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Separation Logic, Design Patterns, Verification, Specification, Type Theory, Denotational Semantics, Program Logic, Program Correctness

I would like to thank my wife, Rachel, for her love, patience, and logistical support.

My parents encouraged me to study computers.
I am not at all sure this was what they had in mind.

Abstract

In this thesis I show is that it is possible to give modular correctness proofs of interesting higher-order imperative programs using higher-order separation logic.

To do this, I develop a model higher-order imperative programming language, and develop a program logic for it. I demonstrate the power of my program logic by verifying a series of examples. This includes both realistic patterns of higher-order imperative programming such as the subject-observer pattern, as well as examples demonstrating the use of higher-order logic to reason modularly about highly aliased data structures such as the union-find disjoint set algorithm.

Acknowledgments

I would like to begin by thanking my advisors for giving me perhaps more rope than was advisable. John and Jonathan were both very tolerant of my sudden enthusiasms, my half-baked ideas, and general tendency to wander off in peculiar directions.

Lars was an invaluable source of support and advice, and after every trip to Denmark I returned significantly more knowledgeable than before I left. I would like to thank Bob for an act he may not even recall any more: as a new graduate student, he informed me that I would be coming to ConCert group meetings, despite technically not even being part of the ConCert project.

As a consequence, I have the need and the pleasure to thank the whiteboard gang for their willingness to talk. Noam Zeilberger and Jason Reed were particularly inspiring examples of people following their mathematical muse, and William Lovas, Daniel Licata, Kevin Watkins, Joshua Dunfield, Carsten Varming, and Rob Simmons were reliable sources of type-theoretic and semantic inspiration. Whenever I needed a different perpsective, the Plaid group was there. I'd like to thank Donna Malayeri, Kevin Bierhof, and Nels Beckman in particular.

Hongseok Yang, Peter O'Hearn, Philippa Gardner, and Matthew Parkinson were all encouraging above and beyond the call of duty to a random student from thousands of miles away.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Programming Language . . . 2
- 1.3 Higher Order Separation Logic . . . 2
- 1.4 Verifying Programs . . . 4
  - 1.4.1 Design Patterns . . . 4
  - 1.4.2 The Union-Find Algorithm . . . 5
- 1.5 Related Work . . . 5
  - 1.5.1 Immediate Predecessors . . . 5
  - 1.5.2 Program Logics and Type Theories . . . 6
  - 1.5.3 Program Proofs for Design Patterns . . . 9

2 Type Structure **13**
- 2.1 Syntax of Types and Kinds . . . 13
  - 2.1.1 The Syntactic Theory of Kinds . . . 15
  - 2.1.2 Semantics of Kinds . . . 16
- 2.2 Semantics of Types . . . 18
  - 2.2.1 The Overall Plan . . . 18
  - 2.2.2 Interpreting Monotypes . . . 19
  - 2.2.3 Interpreting Polytypes . . . 20
  - 2.2.4 Interpreting Heaps and Basic Continuations . . . 21
- 2.3 Solving Domain Equations . . . 22
  - 2.3.1 Local Continuity . . . 22
  - 2.3.2 The Inverse Limit Construction . . . 25
- 2.4 Solving Our Recursive Domain Equation . . . 31
  - 2.4.1 Computations form a Monad . . . 31
- 2.5 The Programming Language . . . 33
- 2.6 Denotational Semantics . . . 35
  - 2.6.1 Proofs . . . 36

3 The Semantics of Separation Logic **61**
- 3.1 BI Algebras . . . 61
  - 3.1.1 Partial Commutative Monoids . . . 62
  - 3.1.2 BI-Algebras over Partial Commutative Monoids . . . 62

3.1.3 Sets of Heaps Form the BI Algebra of Heap Assertions . . . 68
3.2 Challenges in Interpreting Specifications . . . 69
3.2.1 Admissibility and Fixed Point Induction . . . 70
3.2.2 The Frame Rule . . . 70
3.3 Basic Hoare Triples . . . 71
3.3.1 Approximating Postconditions . . . 71
3.3.2 Defining the Basic Hoare Triples . . . 73
3.4 Kripke Hoare Triples . . . 74
3.4.1 World Preorders . . . 74
3.4.2 Heyting Algebras over Preorders . . . 74
3.4.3 Defining Kripke Hoare Triples . . . 78
3.4.4 The Framing Operator . . . 81
3.5 Syntax of Assertions and Specifications . . . 85
3.5.1 Substitution Properties . . . 90
3.6 Semantics . . . 91
3.6.1 Interpretation of Sorts . . . 91
3.6.2 Interpretation of Terms and Specifications . . . 92
3.6.3 Substitution Properties . . . 92
3.7 The Program Logic . . . 96
3.8 Correctness Proofs . . . 101
3.8.1 Substitution Theorems . . . 101
3.8.2 Soundness of Assertion Logic Axioms . . . 131
3.8.3 Soundness of Program Logic Axioms . . . 132
3.8.4 The Syntactic Frame Property . . . 146

4 Proving the Correctness of Design Patterns 151
4.1 Introduction . . . 151
4.2 Iterators and Composites . . . 152
4.2.1 The Iterator Specification . . . 153
4.2.2 Example Implementation . . . 160
4.3 The Flyweight and Factory Patterns . . . 173
4.3.1 Verification of the Implementation . . . 176
4.4 Subject-Observer . . . 178
4.4.1 Correctness Proofs for Subject-Observer . . . 181

5 Proving the Union-Find Disjoint Set Algorithm 185
5.1 Introduction . . . 185
5.2 The Specification . . . 186
5.3 Correctness Proofs . . . 191
5.3.1 Proof of find . . . 191
5.3.2 Proof of newset . . . 195
5.3.3 Proof of union . . . 197

List of Figures

2.1 Syntax of the Types and Kinds of the Programming Language 13
2.2 Kinding Rules for Monotypes 14
2.3 Kinding Rules for Polytypes 15
2.4 Equality Rules for Monotypes 16
2.5 Equality Rules for Polytypes 17
2.6 Locally Continuous Functor Interpreting Monotypes 19
2.7 Locally Continuous Functor Interpreting Polytypes 21
2.8 Syntax of the Programming Language 33
2.9 Typing of the Pure Expressions 34
2.10 Typing of Monadic Expressions 35
2.11 Interpretation of Program Contexts 36
2.12 Interpretation of Pure Terms 37
2.13 Interpretation of Computations 38
2.14 Equality Rules for Sums, Products, Exponentials, and Suspended Computations . 39
2.15 Equality Rules for Numbers, Universals, and Existentials 40
2.16 Equality Rules for Computations 40
2.17 Congruence Rules for Equality 41

3.1 Syntax of Assertions and Specifications 87
3.2 Well-sorting of Sorts and Contexts 88
3.3 Well-sorting of Assertions 89
3.4 Well-sorting of Specifications 90
3.5 Interpretation of Sorts and Contexts 92
3.6 Interpretation of Terms 93
3.7 Interpretation of Specifications 94
3.8 Basic Axioms of Specification Logic 96
3.9 Structural Axioms of the Program Logic 97
3.10 Axioms of Assertion Logic 98
3.11 Equality Judgment for Assertions 98

4.1 Interface to the Iterator Library 155
4.2 Auxilliary Functions Used in the Iterator Specification 156
4.3 Type and Predicate Definitions of the Iterator Implementation 161
4.4 Implementation of Collections and Iterators 162

4.5 Flyweight Specification 174
4.6 Flyweight Factory Specification 174
4.7 Flyweight Implementation 175
4.8 Hash Table Specification 176

5.1 Specification of Union Find Algorithm 187
5.2 Concrete Definition of Union Find Invariants 189
5.3 Implementation of Union-Find Algorithm 190

Chapter 1

Introduction

My thesis is that it is possible to give modular correctness proofs of interesting higher-order imperative programs using higher-order separation logic.

1.1 Motivation

It is more difficult to reason about programs that use aliasing than ones that do not use mutable shared data. It is more difficult to reason about programs that use higher order features than first order programs. Put both together, and matters become both challenging and interesting for formal verification, since the combination yields languages more complex than the sum of their parts.

Techniques to reason about purely functional programs, which extensively use higher-order methods but eschew mutable state, are a well-developed branch of programming language theory, with a long and successful history.

Historically, techniques to reason about mutable state have lagged behind, but some years ago O'Hearn and Reynolds introduced separation logic [42], which has proven successful at enabling correctness proofs of even such intricate imperative programs as garbage collectors [8]. Separation logic has historically focused on low-level programming languages that lack a strong type discipline and allow the use of techniques such as casting pointers to integers.

Even high level languages that allow the use of state are prone to aliasing errors, since (with a few exceptions) type systems do not track interference properties. Some languages, such as Haskell [17], isolate all side-effects within a particular family of monadic types. While this preserves reasoning principles for functional code, we still face the problem that reasoning about monadic code remains as difficult as ever. Haskell's type discipline imprisons mutation and state, but does not rehabilitate them.

Of course, we can combine language features combinatorially, and if this particular combination had no applications, then it would be only of technical interest. In fact, though, higher-order state pervades the design of common programs. For example, graphical user interfaces (GUIs) are typically structured as families of callbacks that operate over the shared data structures representing the interface and program. These callbacks are structured using the subject-observer pattern [12], which uses collections of callbacks to implicitly synchronize mutable data struc-

tures across a program. So higher-order state not only poses a technical challenge, but also offers a set of tantalizingly practical examples to motivate the technical development.

In this dissertation, I develop a model higher-order imperative programming language, and develop a program logic for it. I demonstrate the power of my program logic by verifying a series of interesting examples, culminating in the correctness proof of a library for the union-find algorithm. Despite the fact that this is a program making heavy and essential use of aliasing, clients may nevertheless reason using the simple style of separation logic.

1.2 Programming Language

The first contribution of my dissertation is to give a denotational semantics for a predicative version of System F^ω [13], extended with a monadic type constructor for state in Pfenning-Davies [37] style. This language is an instance of Moggi's [27] monadic metalanguage, as all heap effects and nontermination live within the monadic type constructor.

Higher-order imperative programs usually contain a substantial functional part in addition to whatever imperative operations they perform. As a result, it is convenient to be able to reason equationally about programs when possible (including induction principles for inductive types), and by giving a denotational semantics, I can easily show the soundness of these equational properties.

Ths semantics of this language is the focus of Chapter 2. In particular, I show that a programming language including references to polymorphic values, nevertheless gives rise to a domain equation which can be solved using the classical techniques of Smyth and Plotkin [45].

1.3 Higher Order Separation Logic

The main tool I develop to prove programs correct is a higher order separation logic for a higher-order imperative programming language. The details of the logic are given in Chapter 3, but I will say a few words to set the stage now.

As I mentioned earlier, we can reason about the functional part of programs using the standard $\beta\eta$-theory of the lambda calculus, but to reason about imperative computations, I introduce a specification logic, in the style of Reynolds' specification logic for Algol [41].

In this logic, the basic form of specification is the Hoare triple over an imperative computation $\{P\}c\{a : A.\ Q\}$. Here, P is the precondition, c is the computation that we are specifying, and Q is the post- condition. The binder $a : A$ names the return value of the computation, so that we can mention it in the post-condition.

The assertion logic for the pre- and post-conditions is higher order separation logic [6]. This is a substructural logic that extends ordinary logic with three spatial connectives to enable reasoning about the aliasing behavior of data. The Hoare triples themselves form the atomic propositions of a first-order intuitionistic logic of specifications. The quantifiers range over the sorts of the assertion logic, so that we can universally and existentially quantify over assertions and expressions.

As a teaser example, consider the specification of a counter module.

$$
\begin{array}{l}
\exists \alpha : \star \\
\exists \texttt{create} : \mathbf{1} \to \bigcirc \alpha \\
\exists \texttt{next} : \alpha \to \bigcirc \mathbb{N} \\
\exists \mathit{counter} : \alpha \times \mathbb{N} \Rightarrow \mathsf{prop} \\
\quad \langle \mathsf{emp} \rangle \texttt{create}() \langle a : \alpha.\ \mathit{counter}(a, 0) \rangle \\
\quad \& \\
\quad \forall c : \alpha, n : \mathbb{N}.\ \langle \mathit{counter}(c, n) \rangle \texttt{next}(c) \langle a : \mathbb{N}.\ a = n \wedge \mathit{counter}(c, n+1) \rangle
\end{array}
$$

The idea is that in our program logic, we can assert the existence of an abstract type of counters α, operations `create` and `next` (which create and advance counters, respectively), and a state predicate $\mathit{counter}(c, n)$ (which asserts that the counter c has value n).

The specifications are Hoare triples — the triple for `create` asserts that from an empty heap, calling `create` creates a counter initialized to 0, and the triple for $\texttt{next}(c)$ asserts that if `next` is called on c, in a state where it has value n, then the return value will be n and the state of the counter will be advanced to $n + 1$.

Now, here are two possible implementations for the existential witnesses:

$$
\begin{array}{lcl}
\alpha & \equiv & \mathsf{ref}\ \mathbb{N} \\
\mathit{counter} & \equiv & \lambda(r, n).\ (r \mapsto n) \\
\texttt{create} & \equiv & \lambda().\ [\mathsf{new}_{\mathbb{N}}(0)] \\
\texttt{next} & \equiv & \lambda r.\ [\mathsf{letv}\ n = [!r]\ \mathsf{in}\ \mathsf{letv}\ () = [r := n + 1]\ \mathsf{in}\ n]
\end{array}
$$

and also

$$
\begin{array}{lcl}
\alpha & \equiv & \mathsf{ref}\ \mathbb{N} \\
\mathit{counter} & \equiv & \lambda(r, n).\ (r \mapsto n + 7) \\
\texttt{create} & \equiv & \lambda().\ [\mathsf{new}_{\mathbb{N}}(7)] \\
\texttt{next} & \equiv & \lambda r.\ [\mathsf{letv}\ n = [!r]\ \mathsf{in}\ \mathsf{letv}\ () = [r := n + 1]\ \mathsf{in}\ n - 7]
\end{array}
$$

So in our program logic, the implementations of modules consist of the witnesses to the existential formulas. This is morally an application of Mitchell and Plotkin's identification of data abstraction and existential types [26] — in this view, linking a module with a client is nothing but an application of the existential elimination rule of ordinary logic. Note that instead of abstracting only over types, we also abstract over the *heap*.

In order for a verification methodology to scale up to even modestly-sized programs, it must be modular. There are three informal senses in which I use the word "modular", each of which is supported by different features of this logic, and are (mostly) illustrated in this example.

1. First, we should be able to verify programs library by library. That is, we should be able to give the specification of a library, and prove both that implementations satisfy the specification without knowing anything about the clients that use it, and likewise prove the correctness of a client program without knowing anything about the details of the implementation (beyond what is promised by the specification).

In the example above, I tackle this problem by making use of the fact that the presence of existential specifications in our specification logic lets use the Mitchell-Plotkin encoding of modules as existentials. So once I prove a client against this specification, I can swap between implementations without having to re-verify the client.

2. Related to this is a modular treatment of aliasing. Ordinary Hoare logic becomes non-modular when asked to treat mutable, shared state, because we must explicitly specify the aliasing or non-aliasing of every variable and mutable data structure in our pre- and post-conditions. Besides the quadratic dependence on the size of programs, the relevant set of variables grows whenever a subprogram is embedded in a larger one. Separation logic resolves this problem via the frame rule, a feature which we carry forward in our logic.

   In our example, counters have state, and so the predicates for distinct counters need to be disjoint. So even in the simplest possible example, it is useful to be able to have the power of separation logic available.

3. Finally, it is important to ensure that the abstractions we introduce compose. Benton [5] described a situation where he was able to prove that a memory allocator and the rest of the program interacted only through a particular interface, but when he tried to divide the rest of the program into further modules, he encountered difficulties, because it was unclear how to share the resources and responsibility for upholding the contract with the allocator.

   While the previous example was too simple to illustrate these kinds of problems, I do exploit the expressiveness of my program logic to introduce several new *specification patterns* for verifying these kinds of programs. In the following section, I will describe some of the patterns I discovered.

1.4 Verifying Programs

The final test of a program logic is what we can prove with it, and so accordingly I have devoted a great deal of effort to not only prove metatheorems *about* the logic, but also theorems *in* my logic.

1.4.1 Design Patterns

One prominent source of examples of higher-order state arises in object-oriented programs. One way of translating objects into functional language is by viewing objects as records of functions (methods) accessing common hidden state (fields). As a result, common programming idioms in object-oriented languages can be viewed as patterns of higher-order imperative programs.

Over the years, object-oriented programmers have documented many of idioms which repeatedly arise in practice, calling them *design patterns* [12]. While originally intended to help practitioners communicate with each other, design patterns also offer a nice collection of small, but realistic and widely-occurring, programs to study for verification purposes.

In Chapter 4, I translate many common design patterns into my programming language[1] and then give specifications and correctness proofs for these programs. I want to emphasize that I

[1] Unsurprisingly, they turn into idioms that many ML programmers will find familiar.

view the specifications as even more important a contribution as the correctness proofs themselves: programmers have informal reasons for believing in the correctness of their programs, which are often surprisingly subtle to formalize.

1.4.2 The Union-Find Algorithm

One of the reasons for the success of separation logic (and related substructural logics like linear logic) in program verification is the fact that aliasing turns out to be only used lightly in many programs.

In Chapter 5 of my dissertation, I study the union-find [10] algorithm. This algorithm is very challenging to verify, because its correctness and efficiency relies intimately upon the ability to use shared mutable data to broadcast updates to the whole program.

To deal with this problem, I make use of the expressive power of higher-order separation logic, and introduce a *domain-specific logic* to describe the interface between a union-find library and its clients. This allows the implementation to keep a global view of the union-find data structure, while still permitting clients to be proved via local reasoning.

Furthermore, those operations which have genuinely global effects (such as taking the union of two disjoint sets) can be specified using custom modal operators I call *ramifications*. These operators commute with the custom separating conjunction, and so allow local reasoning even when global effects are in play.

1.5 Related Work

1.5.1 Immediate Predecessors

There are two immediate predecessors to the proof system in this dissertation.

First, there is the work of Reynolds on specification logic for Algol [41]. Idealized Algol [40] combines a call-by-name functional language, together with a type of imperative computations whose operations correspond to the while-language (i.e., it does not have higher-order store). This stratification ensures that Algol features a powerful equational theory which validates both the β- and η-rules of the lambda-calculus.

Specification logic extends Hoare logic [14] to deal with these new features, by viewing Hoare triples as the atomic propositions of a first-order logic of specifications. Then, the user of the logic can specify the behavior of a higher-order function using an implications over triples, using the hypothesis of an implication to specify the behavior of arguments of command type. In short, specification logic can be viewed as a synthesis of LCF [25] with Hoare logic.

One of the motivations for the language design in my dissertation was to see if the analogy between the computation type in Algol and the monadic type discipline could be extended to a full higher-order programming language, with features like higher-order store. This has worked fantastically well. The semantics of the logic of this dissertation is (suprisingly) *simpler* than original specification logic, even though the programming language itself is a much more complex one than Idealized Algol.

In particular, one of the main sources of complexity in usages of specification logic has simply vanished from this logic. Algol has assignable variables (like C or Java, though of course this reverses the chronology), and so specification logic had to account for their update with assertions about so-called "good variable" conditions. Since ML-like languages do not have assignable variables, this means that all aliasing is confined to the heap. As a result, the entire machinery of good variables is simply absent from my system. So all aliasing and interference is confined to the heap, and separation logic works quite well for specifying and reasoning about this interference.

The other line of work I make use is on algebraic models of separation logic. I give a very concrete model of separation logic in Chapter 3, and am able to interpret higher-order quantification due to the fact that the lattice of propositions is complete. This is probably best understood as an instantiation of Biering *et al.*'s hyperdoctrine model of higher-order separation logic. Likewise, my semantic domain for specifications makes use of the techniques introduced by Birkedal and Yang [7] to model higher-order frame rules.

Happily, though, most of this machinery stays "under the hood" to ensure that the logic looks simple to the user. As an example of this phenomenon, we use TT-closure [48] to force the admissibility of Hoare triples. This lets us give a simple rule for fixed-point induction, without having to specify conditions on admissible predicates (which is especially problematic in a setting with predicate variables).

Finally, nested triples are very useful for specifying the properties of higher-order programs. It is very useful to be able put the specification of the contents of a reference into an assertion. In my dissertation, I have given a "cheap and cheerful" embedding of assertions in specifications and vice-versa, where the embedding (in either direction) sends the topmost element of the source lattice to the topmost element of the target lattice, and maps all other lattice elements to the bottom element. This approach has the virtues of first, being very technically easy to implement, and second, sufficient for all of the examples I have considered.

However, more sophisticated approaches are certainly possible. Schwinghammer et al. [43] describe how to unify the assertion and specification languages into a single logic. This offers the technical benefit that it extends the power of the higher-order frame rule by letting it operate in a hereditary way, to act on nested triples contained within assertions. In contrast, the frame rule I give stops at the boundary of an assertion.

Despite looking, though, I have not yet found any programs that seem to need this extra generality to verify. Most of the programs I have considered which might benefit from these features seem to also need further generalizations, such as Liskov and Wing's monotonic predicates. Indeed, this is the case even for the higher-order frame rule: most of the results in this dissertation could have been proven without it. However, the Kripke view made it no harder to support than the first-order frame rule.

1.5.2 Program Logics and Type Theories

While there is a vast literature on verification of imperative programs, three lines of research in particular stand out as most closely related to my own work.

Separation Logic for Java

In his dissertation, Parkinson [34] gives a program logic for Java, which, on the surface, closely follows the standard methodology of separation logic — that is, it is a pure Hoare logic with triples as the fundamental device for structuring program proofs (and no logic of triples, as in specification logic). However, a deeper look reveals a number of innovations which permit an elegant treatment of many aspects of object-oriented programming. Since object-oriented programming is fundamentally higher-order imperative programming, there are many ideas in Parkinson's dissertation which are either related to or simply reminiscent of the ideas in my work.

As in this work, Parkinson builds a high-level separation logic. That is, his model of the heap consists of a collection of structured objects, rather than as a simple giant byte array. As in my work, pointer arithmetic is impossible, and pointer point to arbitrary values.

Unlike in my work, all specifications are Hoare triples (there is no specification logic which treats Hoare triples as atoms), and furthermore his program logic supports the semantic subtyping rule (aka the Liskov substitution principle). Now, it is well-known that the straightforward treatment of semantic subtyping essentially reduces object-oriented programs to first-order programs and prevents writing essentially higher-order functions (such as iter or map).

To avoid this trap, Parkinson made two key innovations. First, he makes use of abstract predicates (i.e., second-order quantification over predicates), which permits writing method specifications which relate the footprint of a method to the footprint of the methods of the objects it receives as an argument.

Second, he gives his second-order predicates a very unusual semantic interpretation. The idea, described in Parkinson and Bierman [35], is to index each predicate symbol by an object, and to permit the concrete definition of a predicate to *dynamically dispatch* based on the class of the receiver object. That is, each class in an object hierarchy may define the implementation of an abstract predicate differently, and the definition used actually depends on the concrete class of the object. As a result, semantic subtyping can be formally respected, without blocking the use of objects as higher-order parameters: a family of objects used as higher-order parameters can simply define different concrete definitions of a common predicate symbol.

This design is both clever and elegant, since it puts the natural style of specifications in alignment with the style of object-oriented programming. Unfortunately, the use of dynamic dispatch means that it is difficult to figure out what higher-kinded predicate symbols ought to mean. This turns out not to be a fundamental problem for Parkinson, since the natural style of Java programming is rather like writing the defunctionalized version of a higher-order program. As a result, he rarely needs fully higher-order specifications — he can attach specifications to each class implementing an interface for a higher-order method.

However, I work with a genuinely higher-order programming language, and this solution does not work for me. Instead, I make use of a standard semantics of predicates in separation logic, which makes interpreting predicates of arbitrary order straightforward. As a result, I have no problems with higher-order specifications. A good example of this difference can be seen by contrasting our respective proofs of the subject-observer pattern, which we simultaneously and independently gave correctness proofs for.

From a distance, the specifications look very similar, with the differences appearing on close

inspection. Namely, in Parkinson's version of the subject-observer pattern [33], for each subject class, there is a common observer interface that all concrete observer classes must implement, and the subject predicate is indexed by a list of observers of this interface. He then uses his predicate dynamic dispatch to allow each observer to have a different real invariant. In my version of the subject-observer pattern (in Chapter 4), the subject predicate is directly indexed by a list of observer actions plus their invariants and specifications, directly as a higher-order predicate.

The ability to make use of full higher-order logic turns out to be very important for more complex examples. The ability to freely add new definitions to the logic permits the use of *embedded separation logics* for verifying programs making use of heavily aliased mutable state (as in the union-find example of Chapter 5). There, I make free use of the full machinery of higher-order logic to define custom logics as index domains.

Hoare Type Theory

Nanevski, Morisett and Birkedal have developed Hoare Type Theory [30, 31], a sophisticated dependently-typed functional language that, like our system, uses a monadic discipline to control effects. Unlike my work, HTT takes advantage of type dependency to directly integrate specifications into the types of computation terms, by using an indexed family of monads to represent computations. A single term of computation type has a type $e : \{P\}{-}\{a : A.\ Q(a)\}$, where P and Q are predicates on the heap. Similarly to my own use of existentials, Nanevski, Ahmed, Morisett and Birkedal [28] have also proposed using the existential quantification of their type theory to hide data representations, giving examples such as a malloc/free style memory allocator. Nanevski et al. [29] also give a proof of a modern congruence closure algorithm, illustrating that this system can be used to verify some of the most complex imperative algorithms known.

The key difference between the two approaches can be summarized by saying that my work in this thesis is to construct a higher-order predicate logic *about* programs in a non-dependently programming language, whereas HTT *embeds* imperative programs into a dependently typed language. That is, the distinction lies in whether or not specifications are considered part of the language or not. (Note that the other two combinations are also reasonable: embedding computations into a non-dependent language basically amounts to a monadic type discipline. Specification languages without rich support for quantifiers or higher-order features takes us into the domain of program analysis, in which automated tools prove weak theorems about programs. However, neither of these allow rich specifications of program behavior.)

In principle, embedding computations into a dependently-typed language should offer greater flexibility than separating specifications from programs. However, the specific choices made in the design of HTT mean that this promise goes unrealized: there are natural classes of higher-order imperative program which are impossible to write in HTT. Furthermore, I give correctness proofs of such programs, with not especially difficult proofs, which illustrates that the difficulty is not fundamental.

To understand the issue, recall that computations in HTT have types $e : \{P\}{-}\{a : A.\ Q(a)\}$, where P and Q are predicates on the heap. Furthermore, heaps map locations to values, and heap predicates map heaps to propositions. Now, any higher-order imperative program — i.e., any program which manipulates pointers to code — needs to refer to terms of computation type

in its pre- and post-conditions. In the original formulation of HTT, which is predicative, this causes size issues to arise, since a heap at universe level n can only contain commands which manipulate heaps at universe level $n - 1$ or lower. As a result, examples such as the subject-observer example in Chapter 4 cannot be written in predicative HTT.

This difficulty has been overcome by Petersen *et al.*, who give an impredicative model of Hoare Type Theory. However, Svendsen [personal communication] says that in an impredicative setting, the weaker elimination rules for impredicative existentials made porting the proofs in my system to HTT much more difficult. I do not presently know whether the problem is fundamental, or whether there is a more intricate encoding which can avoid the obstacles.

These issues simply do not arise in my system, since specifications are strictly separated from the types. As a result, there are never any size issues arising from the problem of storing computations in the heap.

(This also suggests that an interesting future direction would be to study a version of Hoare Type Theory in which there is an ordinary monadic type of computations, together with a predicate on terms of monadic type which indexes them with pre- and post-conditions.)

Regional Logic and Ownership

In addition to systems based on separation, there is also a line of research based on the concept of object invariants and ownership. The Java modeling language (JML) [21] and the Boogie methodology [4] are two of the most prominent systems based on this research stream. In Boogie, each object tracks its owner object with a ghost field, and the ownership discipline enforces that the heap have a tree structure. This allows the calculation of frame properties without explosions due to aliasing, even though the specification language remains ordinary first-order logic. Banerjee et al. [2] give a logic, which they name "regional logic", which formalizes these ideas in a small core logic more tractable than industrial-strength systems like JML or Boogie.

For "by-hand" proofs, approaches based on ownership tend to be more onerous than proofs using separation logic, since footprints have to be tracked explicitly in pre- and post-conditions. However, the flip side of this is that there are no substructural quantifiers, and this can enable an easier use of existing theorem provers for automation.

From the semantic point of view, one of the most interesting features of this line of work is that Banerjee and Naumann [1] were able to use an ownership discipline to prove a representation independence (i.e., relational parametricity) result for Java-like programs. This is something that neither my system, nor any of the other ones described earlier is capable of. The equality relation I use is simply the equality inherited from the (highly non-abstract) denotational semantics I give.

1.5.3 Program Proofs for Design Patterns

Iterators

In his dissertation [34], Parkinson gave as an example a simple iterator protocol, lacking the integration with composites we have exhibited. Subsequently, we formalized a similar account of iterators [20], again lacking the integration with composites.

Jacobs, Meijer, Piessens and Schulte [16] extend Boogie with new rules for the coroutine constructs C# uses to define iterators. Their solution typifies the difficulties ownership-based approaches face with iterators, which arise from the fact that iterators must have access to the private state of a collection but may have differing lifetimes. This work builds on Barnett and Naumann's generalization of ownership to friendship [3], which allows object invariants to have some dependency on non-owned objects.

Unlike the work in my thesis, this required an extension to the core logic, rather than being a formula provable within the logic.

Flyweights

Pierik, Clarke and de Boer [39] formalize another extension to the Boogie framework which they name *creation guards*, specifically to handle flyweights. They consider flyweights an instance of a case where object invariants can be invalidated by the allocation of new objects, and add guards to their specifications to control allocation to the permitted cases.

Again, this required an extension to the core logic, though in this case the extension was quite modest, since footprints are already explicitly tracked.

The Subject-Observer Pattern

The subject-observer pattern has been the focus of a great deal of effort, given its prominence in important applications. Simultaneously with our own initial formulation, Parkinson gave an example of verifying the subject-observer protocol [33]. Recently, Parkinson and Distefano [36] have implemented a tool to verify these programs, and have demonstrated several examples including a verification of a subject-observer pattern specified along these lines. The tool includes automatic generation of loop invariants.

The style of invariant in our work and Parkinson's is very similar, and subject to similar limitations. Since each subject needs to know what its observers are, verifying programs with chains of subject-observers is extremely cumbersome. This is especially problematic given that GUI programs — which are one of the primary uses of the subject-observer pattern — rely upon chains of subjects and observers.

The work of Barnett and Naumann is also capable of reasoning about the subject-observer pattern, but only if all of the possible observers are known at verification. Leino and Schulte [22] extended Boogie with Liskov and Wing's concept of history invariants or monotonic predicates [23] to give a more modular solution. The idea in this work is to require the changes that a subject makes to "increase the truth" of the observer's predicate along some partial order. This is less flexible than the approach we took, though perhaps a little easier to use when it is applicable. Unfortunately, this work is not merely heavyweight, but entirely inapplicable for event-driven programs such as GUIs, since there is no natural partial order on user actions.

More recently, Shaner, Naumann and Leavens [44] gave a "gray-box" treatment of the subject-observer pattern. Instead of tracking the specifications of the observers in the predicate, they give a model program that should approximates the behavior of any actual notification method. This works as long as the runtime dependencies are known statically enough to include them in the model programs — again, a limitation which is problematic in the case of GUIs.

In a paper not part of this thesis [19], I use the logic developed here to give a better specification of callbacks which is modular and handles chains of notifications gracefully.

Chapter 2

Type Structure

2.1 Syntax of Types and Kinds

In this section, I will describe the type structure of the programming language I will use in this dissertation. The language is a pure, total, predicatively polymorphic programming language (with quantification over higher kinds), augmented with a monadic type constructor that permits nontermination and higher-order state. The syntax of types is given in figure 2.1.

Kinds	κ	$::=$	$\star \mid \kappa \to \kappa$
Monotypes	τ	$::=$	$\mathbf{1} \mid \tau \times \tau \mid \tau \to \tau \mid \tau + \tau \mid \mathbb{N} \mid$ $\mathsf{ref}\ A \mid \bigcirc \tau \mid$ $\alpha \mid \tau\ \tau \mid \lambda\alpha : \kappa.\ \tau$
Polytypes	A, B	$::=$	$\mathbf{1} \mid A \times B \mid A \to B \mid A + B \mid \mathbb{N} \mid$ $\mathsf{ref}\ A \mid \bigcirc A \mid \tau \mid$ $\forall\alpha : \kappa.\ A \mid \exists\alpha : \kappa.\ A$
Type Contexts	Θ	$::=$	$\cdot \mid \Theta, \alpha : \kappa$

Figure 2.1: Syntax of the Types and Kinds of the Programming Language

The basic kind structure of the language is given with the kinds κ, which range over the kind $\star$, the kind of ground monotypes, and higher kinds $\kappa \to \kappa$ built from it. We write all of these constructors with the letters τ and σ. The monotype constructors are the unit type $\mathbf{1}$, pair types $\tau \times \sigma$, sums $\tau + \sigma$, function space $\tau \to \sigma$, natural numbers $\mathbb{N}$, computation types $\bigcirc\tau$, and finally (ML-style) references $\mathsf{ref}\ A$. Also, within open types we may also use type variables α to refer to monotype constructors, and we can also define lambda-abstractions and applications to inhabit the higher kinds of this language.

The type $\mathsf{ref}\ A$ is not merely a pointer to a value of monomorphic type; it also permits storing a pointer to a value of polymorphic types A. This seemingly violates the usual stratification between monotypes and polytypes, since quantifiers can occur within A. The intuition for viewing

ref A as a monotype is that a reference to a value of polymorphic type is itself merely a location with no additional structure, and so it is safe to treat a reference to a value of polymorphic type as a monomorphic value. Our denotational semantics of references will formalize this intuition and make it precise.

The polytypes A themselves extend the monotypes with universal quantification $\forall\alpha : \kappa.\ A$ as well as existential types $\exists\alpha : \kappa.\ A$. Each of the simple type constructors — sums, products, functions, computations — also may contain polymorphic types as subexpressions within it. However, this is actually only a modest generalization of classical ML-style type schemes. Because the universal and existential quantifiers range over the kinds κ, it is impossible to instantiate them with a polytype, thereby limiting us to predicative polymorphism. Nevertheless, being able to quantifier over higher kinds and instantiate quantifiers with them is sufficient to model many useful idioms (for example, quantifying not just over the element type of a list, but also quantifying over the collection type constructor). For the occasional cases we need to write impredicatively polymorphic programs, we will simulate true impredicativity by passing references.

The kinding judgments $\Theta \vdash \tau : \kappa$ and $\Theta \vdash A : \bigstar$ determine well-formedness of monotypes and polytypes, respectively. The judgment $\Theta \vdash \tau : \kappa$ asserts that the monotype constructor τ has the kind κ, and as can be seen in Figure 2.2, the type constructors all have the expected structure. The rule for references calls out to the judgment $\Theta \vdash A : \bigstar$, defined in Figure 2.3, which gives well-formedness conditions for polytypes. The reason we need a second judgment is that there is no kind of polytypes, and so we simply directly judge whether a polytype is well-formed.

These judgments, and all others in this thesis, follow the usual Barendregt variable convention, in that variable names do not occur repeated in contexts, and that bound variables are renamed (according to the usual rules of alpha-equivalence) so that they are different from the free variables.

$$\frac{}{\Theta \vdash \mathbf{1} : \star}\ \textsc{KUnit} \qquad \frac{\Theta \vdash \tau : \star \qquad \Theta \vdash \sigma : \star}{\Theta \vdash \tau \times \sigma : \star}\ \textsc{KProd} \qquad \frac{\Theta \vdash \tau : \star \qquad \Theta \vdash \sigma : \star}{\Theta \vdash \tau \to \sigma : \star}\ \textsc{KArrow}$$

$$\frac{\Theta \vdash \tau : \star \qquad \Theta \vdash \sigma : \star}{\Theta \vdash \tau + \sigma : \star}\ \textsc{KSum} \qquad \frac{}{\Theta \vdash \mathbb{N} : \star}\ \textsc{KNat} \qquad \frac{\Theta \vdash A : \bigstar}{\Theta \vdash \mathsf{ref}\ A : \star}\ \textsc{KRef}$$

$$\frac{\Theta \vdash \tau : \star}{\Theta \vdash \bigcirc\tau : \star}\ \textsc{KComp}$$

$$\frac{\alpha : \kappa \in \Theta}{\Theta \vdash \alpha : \kappa}\ \textsc{KHyp} \qquad \frac{\Theta \vdash \tau : \kappa' \to \kappa \qquad \Theta \vdash \tau' : \kappa'}{\Theta \vdash \tau\ \tau' : \kappa}\ \textsc{KApp} \qquad \frac{\Theta, \alpha : \kappa' \vdash \tau : \kappa}{\Theta \vdash \lambda\alpha : \kappa'.\ \tau : \kappa' \to \kappa}\ \textsc{KLam}$$

Figure 2.2: Kinding Rules for Monotypes

$$\frac{}{\Theta \vdash \mathbf{1} : \bigstar}\ \textsc{KUnit} \qquad \frac{\Theta \vdash A : \bigstar \qquad \Theta \vdash B : \bigstar}{\Theta \vdash A \times B : \bigstar}\ \textsc{KProd}$$

$$\frac{\Theta \vdash A : \bigstar \qquad \Theta \vdash B : \bigstar}{\Theta \vdash A \to B : \bigstar}\ \textsc{KArrow} \qquad \frac{\Theta \vdash A : \bigstar \qquad \Theta \vdash B : \bigstar}{\Theta \vdash A + B : \bigstar}\ \textsc{KSum}$$

$$\frac{}{\Theta \vdash \mathbb{N} : \bigstar}\ \textsc{KNat} \qquad \frac{\Theta \vdash A : \bigstar}{\Theta \vdash \bigcirc A : \bigstar}\ \textsc{KComp}$$

$$\frac{\Theta \vdash \tau : \star}{\Theta \vdash \tau : \bigstar}\ \textsc{KMono}$$

$$\frac{\Theta, \alpha : \kappa \vdash A : \bigstar}{\Theta \vdash \forall \alpha : \kappa.\ A : \bigstar}\ \textsc{KForall} \qquad \frac{\Theta, \alpha : \kappa \vdash A : \bigstar}{\Theta \vdash \exists \alpha : \kappa.\ A : \bigstar}\ \textsc{KExists}$$

Figure 2.3: Kinding Rules for Polytypes

2.1.1 The Syntactic Theory of Kinds

Types also support a pair of equality judgments $\Theta \vdash \tau \equiv \tau' : \kappa$ and $\Theta \vdash A \equiv B : \bigstar$, shown in Figures 2.4 and 2.5. The equality judgment for monotypes implements the β- and η-equality principles of the lambda calculus, along with congruence rules for all of the type constructors of our language. The only rules we have for the equality judgment for polytypes are simple congruence rules, plus a recursive call back to the other equality judgment whenever we need to compare monotyped terms.

Proposition 1. *(Weakening) If* $\Theta \vdash \tau : \kappa$ *then* $\Theta, \alpha : \kappa' \vdash \tau : \kappa$.

Proof. This follows from structural induction on the derivation of $\Theta \vdash \tau : \kappa$. □

Proposition 2. *(Substitution) Suppose* $\Theta \vdash \tau' : \kappa'$. *Then*

- *If* $\Theta, \alpha : \kappa' \vdash \tau : \kappa$, *then* $\Theta \vdash [\tau'/\alpha]\tau : \kappa$.
- *If* $\Theta, \alpha : \kappa' \vdash A : \bigstar$, *then* $\Theta \vdash [\tau'/\alpha]A : \bigstar$.

Proof. This follows from mutual structural induction on the derivation of $\Theta, \alpha : \kappa' \vdash \tau : \kappa$ and $\Theta, \alpha : \kappa' \vdash A : \bigstar$. □

Proposition 3. *(Well-Kindedness of Equality) If we have that* $\Theta \vdash \tau \equiv \sigma : \kappa$, *then we know that* $\Theta \vdash \tau : \kappa$ *and* $\Theta \vdash \sigma : \kappa$. *Likewise,* $\Theta \vdash A \equiv B : \bigstar$, *then we know that* $\Theta \vdash a : \bigstar$ *and* $\Theta \vdash b : \bigstar$.

Proof. This follows from mutual structural inductions on the derivation of $\Theta \vdash \tau \equiv \sigma : \kappa$ and $\Theta \vdash A \equiv B : \bigstar$. □

Proposition 4. *(Substitution into Equality) If we have that* $\Theta \vdash \sigma \equiv \sigma' : \kappa_1$, *then*

$$\frac{}{\Theta \vdash 1 \equiv 1 : \star} \qquad \frac{\Theta \vdash \tau \equiv \tau' : \star \qquad \Theta \vdash \sigma \equiv \sigma' : \star}{\Theta \vdash \tau \times \sigma \equiv \tau' \times \sigma' : \star} \qquad \frac{\Theta \vdash \tau \equiv \tau' : \star \qquad \Theta \vdash \sigma \equiv \sigma' : \star}{\Theta \vdash \tau \to \sigma \equiv \tau' \to \sigma' : \star}$$

$$\frac{\Theta \vdash \tau \equiv \tau' : \star \qquad \Theta \vdash \sigma \equiv \sigma' : \star}{\Theta \vdash \tau + \sigma \equiv \tau' + \sigma' : \star} \qquad \frac{}{\Theta \vdash \mathbb{N} \equiv \mathbb{N} : \star} \qquad \frac{\Theta \vdash A \equiv A' : \bigstar}{\Theta \vdash \mathsf{ref}\ A \equiv \mathsf{ref}\ A' : \star}$$

$$\frac{\Theta \vdash \tau \equiv \tau' : \star}{\Theta \vdash \bigcirc \tau \equiv \bigcirc \tau' : \star} \qquad \frac{\alpha : \kappa \in \Theta}{\Theta \vdash \alpha \equiv \alpha : \kappa} \qquad \frac{\Theta \vdash \tau \equiv \sigma : \kappa' \to \kappa \qquad \Theta \vdash \tau' \equiv \sigma' : \kappa'}{\Theta \vdash \tau\ \tau' \equiv \sigma\ \sigma' : \kappa}$$

$$\frac{\Theta, \alpha : \kappa' \vdash \tau \equiv \sigma : \kappa}{\Theta \vdash \lambda\alpha : \kappa'.\ \tau \equiv \lambda\alpha : \kappa'.\ \sigma : \kappa' \to \kappa} \qquad \frac{\Theta \vdash (\lambda\alpha : \kappa'.\ \tau)\ \tau' : \kappa}{\Theta \vdash (\lambda\alpha : \kappa'.\ \tau)\ \tau' \equiv [\tau'/\alpha]\tau : \kappa}$$

$$\frac{\Theta, \alpha : \kappa' \vdash \tau\ \alpha \equiv \sigma\ \alpha : \kappa \qquad \alpha \notin \mathrm{FV}(\tau, \sigma)}{\Theta \vdash \tau \equiv \sigma : \kappa' \to \kappa}$$

$$\frac{\Theta \vdash \tau : \kappa}{\Theta \vdash \tau \equiv \tau : \kappa} \qquad \frac{\Theta \vdash \tau \equiv \sigma : \kappa}{\Theta \vdash \sigma \equiv \tau : \kappa} \qquad \frac{\Theta \vdash \tau \equiv \tau' : \kappa \qquad \Theta \vdash \tau' \equiv \sigma : \kappa}{\Theta \vdash \tau \equiv \sigma : \kappa}$$

Figure 2.4: Equality Rules for Monotypes

- *If* $\Theta, \alpha : \kappa_1 \vdash \tau \equiv \tau' : \kappa_2$*, then* $\Theta \vdash [\sigma/\alpha]\tau \equiv [\sigma'/\alpha]\tau' : \kappa_2$.
- *If* $\Theta, \alpha : \kappa_1 \vdash A \equiv B : \bigstar$*, then* $\Theta \vdash [\sigma/\alpha]A \equiv [\sigma'/\alpha]B : \bigstar$.

Proof. This follows by mutual structural induction on the two derivations of the equality judgment. □

2.1.2 Semantics of Kinds

Because types and kinds form an instance of the simply typed lambda calculus, we would like to argue that there is a unique β-normal, η-long form for each well-kinded type expression. If we consider only the monotype constructors excluding the reference types, this is immediate. However, the presence of quantifiers in reference types slightly complicates this story. Luckily, redices can only occur in subterms of polytypes which are monotype constructors, and hence by an induction on the structure of a polytype we can deduce that it has unique normal forms as well.

As a result, when we quotient the set of well-kinded terms by the equality judgment, we know that each equivalence class contains a single long normal term, which we can take as the canonical representative of that class. We can formalize this by giving the set-theoretic semantics of the kinds as follows.

$$[\![\kappa]\!]^s = \{\tau \mid \tau \text{ is } \beta\text{-normal, } \eta\text{-long and } \cdot \vdash \tau : \kappa\}$$
$$[\![\bigstar]\!]^s = \{A \mid A \text{ is } \beta\text{-normal, } \eta\text{-long and } \cdot \vdash A : \bigstar\}$$

$$\frac{}{\Theta \vdash 1 \equiv 1 : \bigstar} \qquad \frac{\Theta \vdash A \equiv A' : \bigstar \qquad \Theta \vdash B \equiv B' : \bigstar}{\Theta \vdash A \times B \equiv A' \times B' : \bigstar}$$

$$\frac{\Theta \vdash A \equiv A' : \bigstar \qquad \Theta \vdash B \equiv B' : \bigstar}{\Theta \vdash A \to B \equiv A' \to B' : \bigstar} \qquad \frac{\Theta \vdash A \equiv A' : \bigstar \qquad \Theta \vdash B \equiv B' : \bigstar}{\Theta \vdash A + B \equiv A' + B' : \bigstar}$$

$$\frac{}{\Theta \vdash \mathbb{N} \equiv \mathbb{N} : \bigstar} \qquad \frac{\Theta \vdash A \equiv A' : \bigstar}{\Theta \vdash \mathsf{ref}\ A \equiv \mathsf{ref}\ A' : \bigstar} \qquad \frac{\Theta \vdash A \equiv A' : \bigstar}{\Theta \vdash \bigcirc A \equiv \bigcirc A' : \bigstar} \qquad \frac{\Theta \vdash \tau \equiv \tau' : \star}{\Theta \vdash \tau \equiv \tau' : \bigstar}$$

$$\frac{\Theta, \alpha : \kappa \vdash A \equiv B : \bigstar}{\Theta \vdash \forall \alpha : \kappa.\ A \equiv \forall \alpha : \kappa.\ B : \bigstar} \qquad \frac{\Theta, \alpha : \kappa \vdash A \equiv B : \bigstar}{\Theta \vdash \exists \alpha : \kappa.\ A \equiv \exists \alpha : \kappa.\ B : \bigstar}$$

$$\frac{\Theta \vdash A : \bigstar}{\Theta \vdash A \equiv A : \bigstar} \qquad \frac{\Theta \vdash A \equiv B : \bigstar \qquad \Theta \vdash B \equiv C : \bigstar}{\Theta \vdash A \equiv C : \bigstar} \qquad \frac{\Theta \vdash A \equiv B : \bigstar}{\Theta \vdash B \equiv A : \bigstar}$$

Figure 2.5: Equality Rules for Polytypes

So we take the meaning of a kind κ to be exactly the closed, β-normal, η-long terms of that kind. Then, we can give a semantics of typing derivations as follows. First, we define the interpretation of kinding contexts to be a tuple of kinds, in the usual way:

$$\begin{array}{lcl} [\![\cdot]\!]^s & = & \mathbf{1} \\ [\![\Theta, \alpha : \kappa]\!]^s & = & [\![\Theta]\!]^s \times [\![\kappa]\!]^s \end{array}$$

Then, the meaning of a derivation $\Theta \vdash \tau : \kappa$ is a set-theoretic function taking the environment into the interpretation of kinds:

$$\begin{array}{lcl} [\![\Theta \vdash \tau : \kappa]\!]^s & \in & [\![\Theta]\!]^s \to [\![\kappa]\!]^s \\ [\![\Theta \vdash \tau : \kappa]\!]^s & = & \lambda \theta \in [\![\Theta]\!]^s.\ \mathsf{nf}(\theta_\Theta(\tau)) \end{array}$$

$$\begin{array}{lcl} [\![\Theta \vdash A : \bigstar]\!]^s & \in & [\![\Theta]\!]^s \to [\![\bigstar]\!]^s \\ [\![\Theta \vdash A : \bigstar]\!]^s & = & \lambda \theta \in [\![\Theta]\!]^s.\ \mathsf{nf}(\theta_\Theta(A)) \end{array}$$

Here, $\theta(\tau)$ is a tuple in $[\![\Theta]\!]$, and the notation θ_Θ denotes a function which turns it back into a substitution:

$$\begin{array}{lcl} ().\ & = & [] \\ (\theta, \tau)_{\Theta, \alpha:\kappa} & = & [\theta_\Theta, \tau/\alpha] \end{array}$$

The substitution θ_Θ can be used to substitute for each of the free variables in τ (or A), and $\mathsf{nf}(\tau)$ computes the normal form of the type constructor τ.

This means that the following theorems about the equality theory on type constructors are true:

Proposition 5. *(Equality Judgment is Sound) We have that*

1. *If* $\Theta \vdash \tau \equiv \tau' : \kappa$*, then* $\tau =_{\beta\eta} \tau'$

2. *If* $\Theta \vdash A \equiv A' : \bigstar$, *then* $A =_{\beta\eta} A'$

Proof. This proof is an easy structural induction on the equality judgment. □

2.2 Semantics of Types

Now, we want to give a semantics for these types, which we will then use to interpret terms of our programming language. Repeating the design criteria mentioned earlier, we need to obey the two principles below:

- Our interpretation of types should make all non-monadic types pure.
- In particular, I want to treat even nontermination as a side effect, in addition to the more-obvious effects of control and state.

The purpose of this choice is twofold. First, it will give us a rich subset of the language which is total and pure, which will be convenient when writing assertions about programs — we will be able to use any pure function directly in assertions, without having to worry about side effects in predicates. Second, purity means that we will get a very rich equality theory for the language — both the β and η laws will hold for all of the types of the programming language, which will facilitate equational reasoning about the pure part of the programming language.

Because we count nontermination as an effect, our denotational semantics is in CPO, the category of complete partial orders and continuous functions between them. In particular, we do not demand that all domains have least elements — that is, we only require that the objects of this category be *predomains*, rather than domains. This permits us to model pure types as predomains lacking a bottom element.

2.2.1 The Overall Plan

The central problem in giving the semantics of types is the interpretation of the monadic type constructor. Since the monadic type is the type of imperative computations, it must be a kind of heap or predicate transformer. However, heaps contain values of arbitrary polymorphic type, which defeats simple attempts to give semantics by induction on types.

Our strategy to resolve this circularity is to give successive domain equations for monotypes, polytypes, and heaps, suitably parametrized so as to let us avoid reference to the semantics of a later stage in an earlier one. Then, we will solve the domain equation, and tie the knot.

1. For each ground monotype τ, we give a functor in $CPO_\perp \times CPO_\perp^{op} \to CPO$. The parameters to this functor will eventually be the basic continuations.
2. Using this interpretation of monotypes, for each polymorphic kinding judgment, we will give another functor in $CPO_\perp \times CPO_\perp^{op} \to CPO$. Again, the parameters to this functor will eventually be the basic continuations.
3. For heaps, a functor $CPO_\perp \times CPO_\perp^{op} \to CPO$. Again, the parameters to this functor will eventually be the basic continuations.
4. A domain equation in $CPO_\perp \times CPO_\perp^{op} \to CPO_\perp$, whose least fixed point we will take as our domain of basic continuations.

$$[\![-]\!]^{\mathrm{m}} : \text{Monotype} \to CPO_{\perp} \times CPO_{\perp}^{op} \to CPO$$

$$\begin{array}{lcl}
\text{Monotype} & = & [\![\star]\!]^s \\
Loc & = & \mathbb{N} \times [\![\bigstar]\!]^s \\
Loc(A) & = & \{(n, B) \in Loc \mid A = B\} \\
 & & \\
[\![\cdot \vdash \mathbf{1} : \star]\!]^{\mathrm{m}}(K_+, K_-) & = & \{*\} \\
[\![\cdot \vdash \mathbb{N} : \star]\!]^{\mathrm{m}}(K_+, K_-) & = & \mathbb{N} \\
[\![\cdot \vdash \tau \times \sigma : \star]\!]^{\mathrm{m}}(K_+, K_-) & = & [\![\cdot \vdash \tau : \star]\!]^{\mathrm{m}}(K_+, K_-) \times [\![\cdot \vdash \sigma : \star]\!]^{\mathrm{m}}(K_+, K_-) \\
[\![\cdot \vdash \tau + \sigma : \star]\!]^{\mathrm{m}}(K_+, K_-) & = & [\![\cdot \vdash \tau : \star]\!]^{\mathrm{m}}(K_+, K_-) + [\![\cdot \vdash \sigma : \star]\!]^{\mathrm{m}}(K_+, K_-) \\
[\![\cdot \vdash \tau \to \sigma : \star]\!]^{\mathrm{m}}(K_+, K_-) & = & [\![\cdot \vdash \tau : \star]\!]^{\mathrm{m}}(K_-, K_+) \to [\![\cdot \vdash \sigma : \star]\!]^{\mathrm{m}}(K_+, K_-) \\
[\![\cdot \vdash \mathsf{ref}\ A : \star]\!]^{\mathrm{m}}(K_+, K_-) & = & Loc(\mathsf{nf}(A)) \\
[\![\cdot \vdash \bigcirc\tau : \star]\!]^{\mathrm{m}}(K_+, K_-) & = & ([\![\cdot \vdash \tau : \star]\!]^{\mathrm{m}}(K_+, K_-) \to K_-) \to K_+
\end{array}$$

Figure 2.6: Locally Continuous Functor Interpreting Monotypes

Roughly speaking, the basic continuations will be the (continuous) maps from heaps to primitive observations (the 2-point Sierpinski space), and by interpreting the monadic type as a CPS type with answer type equal to the basic continuations, we can view monadic terms as heap transformers.

2.2.2 Interpreting Monotypes

In Figure 2.6, we give an interpretation of the closed, canonical monotypes (i.e., monotypes of kind $\star$, with no free occurrences of type variables within them, and no β-redexes within them) as a functor in $CPO_{\perp} \times CPO_{\perp}^{op} \to CPO$. This semantics is parametrized with two arguments K^+ and K^-, which separate the positive and negative occurrences of the basic continuations.

Before explaining the clauses in detail, I will explain why it is well-defined at all. First, because we are considering closed terms of kind $\star$, the normalization theorem tells us that any such term will normalize to one of the cases listed above. In particular, we will never bottom out at a variable, because the context is closed. We will never bottom out at a lambda, because we are considering only the kind $\star$, and we will never bottom out an application, because there will be room for further beta-reduction in this case, and by hypothesis we are only considering the normal forms.

This means we cover all of the possibilities in this definition, and furthermore we know it is well-founded, because all of the recursive calls to $[\![-]\!]^{\mathrm{m}}$ are always on immediate subterms of the type.

Most of the clauses of this definition should be relatively straightforward — the main mystery is that we have parametrized this interpretation by two arguments K_+ and K_-, whose meaning I will explain when we reach the monadic type constructor. We interpret the unit type as the one-element, discretely ordered predomain, the natural number type as the natural numbers with a discrete order, pairs as the categorical products of CPO, sums as coproducts, and functions via the exponentials of CPO.

Reference types ref A are interpreted as pairs consisting of natural numbers and the representative syntactic object A. The intuition is that a reference is just a number, together with a type tag saying what type of value the reference will point to. It is important that we do *not* interpret the type tag A in this definition — a ref cell is a number plus the purely syntactic object A, acting as a label. This is because we have no interpretation function that can interpret the quantifiers yet: the current definition interprets only the monotypes.

The first time we use K is when we interpret the type $\bigcirc\tau$. The monadic type $\bigcirc\tau$ is interpreted in continuation-passing style, as $(\llbracket\tau\rrbracket^{\mathrm{m}}(K_+, K_-) \to K_-) \to K_+$, and the the K_+ and K_- arguments are revealed as the positive and negative occurrences of the "answer type" of the continuation. One minor fact worth pointing out is that the positive and negative occurrences do *not* trade places on the recursive call to $\llbracket\tau\rrbracket$, since it occurs on the left-hand-side of the left-hand-side of a function space (which is hence a positive occurrence).

Note how this works: $\bigcirc\tau$ is to be the type of stateful computations, but there is (apparently) no state in this definition; it seems like an ordinary continuation semantics. The way that we will re-introduce state is in the definition of K. We will ultimately interpret the answer type K as maps from heaps H to the two-point Sierpinski lattice $O = \{\top, \bot\}$, so that $K = H \to O$. Then, the monadic type will mean $(\llbracket\tau\rrbracket \to (H \to O)) \to (H \to O)$, which can be understood as a sort of predicate transformer.

2.2.3 Interpreting Polytypes

At this stage of the definition, we cannot yet define what heaps mean, since heaps can contain references to values of polymorphic type, and we have not yet defined the semantics of polymorphic types. This is why we have carefully parametrized our functorial semantics of monotypes so that we do not needed to mention them explicitly.

To continue closing this gap, we will give an interpretation of polymorphic types as an indexed function from a context of closed mono-kinded type constructors to a CPO. As before, we parametrize by the continuation arguments, and again define a functor. This definition is given in Figure 2.7.

As explained earlier, the interpretation of the type constructor context $\llbracket\Theta\rrbracket$ are the tuples of the interpretations of each kind in the environment Θ, which are in turn interpreted as merely the closed canonical forms of that kind. (So $\star$ are just the closed monotypes, $\star \to \star$ the closed lambda-terms of that kind, and so on.)

This definition is also well-founded, since it is defined by a structural recursion over the derivation of the kinding derivation of $\Theta \vdash A : \bigstar$.

Whenever we reach a variable or application case, we can apply the substitution and invoke the interpretation function for the monotypes. Universal types $\forall\alpha : \kappa.\ A$ are interpreted as set-indexed predomains or dependent functions from the set of closed canonical objects of the kind κ into predomains. Existential types $\exists\tau : \kappa.\ A$ are interpreted as pairs or dependent sums: we pair a syntactic monotype with the predomain interpreting the second component.

The remaining cases essentially repeat the clauses of the definitions for monotypes, to allow for the possibility that there may be sub-components of pairs/sums/functions/etc that contain universal or existential types.

$$
\begin{array}{lcl}
[\![\Theta \vdash A : \bigstar]\!] & \in & [\![\Theta]\!]^s \to (CPO_\perp \times CPO_\perp^{op}) \to CPO \\
\\
[\![\Theta \vdash \tau : \bigstar]\!]\ \theta\ (K_+, K_-) & = & [\![\cdot \vdash \theta(\tau) : \star]\!]^{\mathrm{m}}(K_+, K_-) \\
[\![\Theta \vdash \forall\alpha : \kappa.\ A : \bigstar]\!]\ \theta\ (K_+, K_-) & = & \Pi\tau : \kappa.\ [\![\Theta, \alpha : \kappa \vdash A : \bigstar]\!]\ (\theta, \tau)\ (K_+, K_-) \\
[\![\Theta \vdash \exists\alpha : \kappa.\ A : \bigstar]\!]\ \theta\ (K_+, K_-) & = & \Sigma\tau : \kappa.\ [\![\Theta, \alpha : \kappa \vdash A : \bigstar]\!]\ (\theta, \tau)\ (K_+, K_-) \\
[\![\Theta \vdash A \times B : \bigstar]\!]\ \theta\ (K_+, K_-) & = & [\![\Theta \vdash A : \bigstar]\!]\ \theta\ (K_+, K_-) \times [\![\Theta \vdash B : \bigstar]\!]\ \theta\ (K_+, K_-) \\
[\![\Theta \vdash A + B : \bigstar]\!]\ \theta\ (K_+, K_-) & = & [\![\Theta \vdash A : \bigstar]\!]\ \theta\ (K_+, K_-) + [\![\Theta \vdash B : \bigstar]\!]\ \theta\ (K_+, K_-) \\
[\![\Theta \vdash A \to B : \bigstar]\!]\ \theta\ (K_+, K_-) & = & [\![\Theta \vdash A : \bigstar]\!]\ \theta\ (K_-, K_+) \to [\![\Theta \vdash B : \bigstar]\!]\ \theta\ (K_+, K_-) \\
[\![\Theta \vdash \mathbf{1} : \bigstar]\!]\ \theta\ (K_+, K_-) & = & \{*\} \\
[\![\Theta \vdash \mathbb{N} : \bigstar]\!]\ \theta\ (K_+, K_-) & = & \mathbb{N} \\
[\![\Theta \vdash \bigcirc A : \bigstar]\!]\ \theta\ (K_+, K_-) & = & ([\![\Theta \vdash A : \bigstar]\!]\ \theta\ (K_+, K_-) \to K_-) \to K_+ \\
[\![\Theta \vdash \mathsf{ref}\ A : \star]\!]\ \theta\ (K_+, K_-) & = & \mathit{Loc}([\![\Theta \vdash A : \bigstar]\!]^s\ \theta)
\end{array}
$$

Figure 2.7: Locally Continuous Functor Interpreting Polytypes

As a result, we need to verify that this definition is coherent, For example, a type like $\mathbb{N} + \mathbb{N} : \bigstar$ can be derived two ways, depending on whether the "+" is viewed as constructor combining polytypes or monotypes. Fortunately, the lack of interesting structure on polytypes makes this easy.

Proposition 6. *(Coherence of Polymorphic Type Interpretation) For any two derivations of* $D, D' :: \Theta \vdash A : \bigstar$*, we have that* $[\![D]\!] = [\![D']\!]$.

Proof. This follows via induction on A. For each A, there are at most two rules from which it can be derived. □

2.2.4 Interpreting Heaps and Basic Continuations

We are finally in a position to define heaps and the recursive domain equation we would like to solve:

$$
\begin{array}{lcl}
H(K_+, K_-) & = & \Sigma L \subseteq^{fin} \mathit{Loc}.\ (\Pi(l, A) \in L.\ [\![\cdot \vdash A : \bigstar]\!](\cdot)\ (K_+, K_-)) \\
\mathcal{K}(K_+, K_-) & = & H(K_-, K_+) \to O
\end{array}
$$

We use H to define what heaps mean. A heap is a dependent sum whose first component is a finite set of allocated locations, together with a map that takes each location in the set of allocated locations, and returns a value of the appropriate type.

The continuation type is defined by the solution to the equation $\mathcal{K}$, which of type $(CPO_\perp^{op} \times CPO_\perp) \to CPO_\perp$. We know that this goes into $CPO_\perp$, since it defines a map into the two point domain, and a function space into a domain is itself a domain. So if we can solve the equation $K \simeq \mathcal{K}(K, K)$, then we can plug K into all our other definitions to interpret all of the types in our language.

Filling this in, we can understand how basic continuations work: they receive a heap, and then either loop or terminate. The monadic type $\bigcirc\tau$ now can be seen as the state-and-continuation monad, which combines the effects of the continuation monad and the state monad via a domain

which looks like $(\tau \to H \to O) \to H \to O$. (Though our model supports it, we will never actually use the possibly of adding control operators to the language in this thesis.)

To show that this equation actually has a solution, it suffices to show that F is a locally-continuous functor. We will prove this by induction over $[\![-]\!]$ and $[\![-]\!]^{\mathrm{m}}$, and then at each case appeal to a series of lemmas showing that each construction we use preserves local continuity.

2.3 Solving Domain Equations

2.3.1 Local Continuity

A functor $F : CPO_\perp \times CPO_\perp^{op} \to CPO$ is *locally continuous* if it preserves the order structure on its arguments. That is, it must be monotone — if $f \sqsubseteq f'$ and $g \sqsubseteq g'$, then $F(f,g) \sqsubseteq F(f',g')$ — and it must also preserve limits — given a pair of chains f_i, g_i, $\bigsqcup_i F(f_i, g_i) = F(\bigsqcup_i f_i, \bigsqcup_i g_i)$.

Now, we will show that the functors we used earlier are all locally continuous. All of this is standard, but I give the proofs to make the presentation self-contained.

Lemma 1. *Local Continuity*

1. *If $F, G : CPO_\perp \times CPO_\perp^{op} \to CPO$ are locally continuous, then $\lambda A, B.\ F(A,B) \times G(A,B)$ is locally continuous.*
2. *If $F, G : CPO_\perp \times CPO_\perp^{op} \to CPO$ are locally continuous, then $\lambda A, B.\ F(A,B) + G(A,B)$ is locally continuous.*
3. *If $F, G : CPO_\perp \times CPO_\perp^{op} \to CPO$ are locally continuous, then $\lambda A, B.\ F(A,B) \to G(A,B)$ is locally continuous.*
4. *The constant functor K_C is locally continuous.*
5. *If X is a set, and $F(x) : CPO_\perp \times CPO_\perp^{op} \to CPO$ is an X-indexed family of locally continuous functors, then $\lambda(A,B).\ \Pi x : X.\ F(x)(A,B)$ is a locally continuous functor.*
6. *If X is a set, and $F(x) : CPO_\perp \times CPO_\perp^{op} \to CPO$ is an X-indexed family of locally continuous functors, then $\lambda(A,B).\ \Sigma x : X.\ F(x)(A,B)$ is a locally continuous functor.*

Proof.

We elide proofs of monotonicity, and give only the proofs of preservation of limits. The notations $\langle f; g\rangle$ and notation $[f; g]$ are the unique mediating maps for products and sums, and mean:

$$\begin{aligned} [f;g] &= \lambda x : A + B.\ \begin{cases} f\ a & \text{when } x = \mathsf{inl}\ a \\ g\ b & \text{when } x = \mathsf{inl}\ b \end{cases} \\ \langle f;g\rangle &= \lambda x : A \times B.\ (f\ x, g\ x) \end{aligned}$$

We also lift these to n-ary versions.

1. Suppose F and G are locally continuous. Now, for A and B, we have the functor which takes (A,B) to $F(A,B) \times G(A,B)$ on the object part, and which takes (f,g) to $F(f,g) \times G(f,g)$ on the arrow part.

   Next, suppose that we have a chain of functions $\langle f_i\rangle : A \to B$ and $\langle g_i\rangle : X \to Y$. Now, we calculate:

$$\begin{array}{rcll}
\sqcup_i(F \times G(f_i, g_i)) & = & \sqcup_i(F(f_i, g_i) \times G(f_i, g_i)) & \\
& = & \sqcup_i \langle F(f_i, g_i) \circ \pi_1; G(f_i, g_i) \circ \pi_2 \rangle & \\
& = & \langle \sqcup_i(F(f_i, g_i) \circ \pi_1); \sqcup_i(G(f_i, g_i) \circ \pi_2) \rangle & (*) \\
& = & \langle \sqcup_i F(f_i, g_i) \circ \sqcup_i \pi_1; \sqcup_i G(f_i, g_i) \circ \sqcup_i \pi_2 \rangle & \\
& = & \langle \sqcup_i F(f_i, g_i) \circ \pi_1); \sqcup_i G(f_i, g_i) \circ \pi_2) \rangle & \\
& = & (\sqcup_i F(f_i, g_i)) \times (\sqcup_i G(f_i, g_i)) &
\end{array}$$

The interesting step is marked with (*); it is justified by the fact that we know that $\pi_j \circ (\sqcup_i \langle h_i^1; h_i^2 \rangle) = \sqcup_i(\pi_j \circ \langle h_i^1; h_i^2 \rangle) = \sqcup_i h_i^j$, and that $\pi_j \circ \langle \sqcup_i h_i^1; \sqcup_i h_i^2 \rangle = \sqcup_i h_i^j$, and that the mediating morphism is unique.

2. Suppose F and G are locally continuous. Now, for A and B we have the functor which takes (A, B) to $F(A, B) + G(A, B)$ on the object part, and which takes (f, g) to $F(f, g) + G(f, g)$ on the arrow part.

   Next, suppose that we have a chain of functions $\langle f_i \rangle : C \to A$ and $\langle g_i \rangle : B \to D$. Now, we calculate:

$$\begin{array}{rcll}
\sqcup_i(F + G(f_i, g_i)) & = & \sqcup_i(F(f_i, g_i) + G(f_i, g_i)) & \\
& = & \sqcup_i [\mathsf{inl} \circ F(f_i, g_i); \mathsf{inr} \circ G(f_i, g_i)] & \\
& = & [\mathsf{inl} \circ \sqcup_i(F(f_i, g_i)); \mathsf{inr} \circ \sqcup_i(G(f_i, g_i))] & (*) \\
& = & [\sqcup_i \mathsf{inl} \circ F(f_i, g_i); \sqcup_i \mathsf{inr} \circ G(f_i, g_i)] & \\
& = & [\mathsf{inl} \circ \sqcup_i F(f_i, g_i)); \mathsf{inr} \circ \sqcup_i G(f_i, g_i))] & \\
& = & (\sqcup_i F(f_i, g_i)) + (\sqcup_i G(f_i, g_i)) &
\end{array}$$

   The interesting step is marked with (*); it is justified by the fact that we know that $(\sqcup_i [h_i^1; h_i^2]) \circ \mathsf{in}_j = \sqcup_i([h_i^1; h_i^2] \circ \mathsf{in}_j) = \sqcup_i h_i^j$, and that $\langle \sqcup_i h_i^1; \sqcup_i h_i^2 \rangle \circ \mathsf{in}_j = \sqcup_i h_i^j$, and that the mediating morphism is unique.

3. Suppose F and G are locally continuous. Now, for A and B we have the functor which takes (A, B) to $F(B, A) \to G(A, B)$ on the object part, and which takes (f, g) to $F(g, f) \to G(f, g)$ on the arrow part.

   Next, suppose that we have a chain of functions $\langle f_i \rangle : A \to C$ and $\langle g_i \rangle : D \to B$. Now, we calculate:

$$\begin{array}{rcl}
\sqcup_i[F \to G](f_i, g_i) & = & F(g_i, f_i) \to G(f_i, g_i) \\
& = & \sqcup_i \lambda h.\ [G(f_i, g_i) \circ h \circ F(g_i, f_i)] \\
& = & \lambda h.\ \sqcup_i [G(f_i, g_i) \circ h \circ F(g_i, f_i)] \\
& = & \lambda h.\ [(\sqcup_i G(f_i, g_i)) \circ h \circ (\sqcup_i F(g_i, f_i))] \\
& = & \lambda h.\ [G(\sqcup_i f_i, \sqcup_i g_i) \circ h \circ F(\sqcup_i g_i, \sqcup_i f_i)]
\end{array}$$

4. The constant functor is locally continuous because it maps all morphisms to the identity morphism, and hence trivially preserves limits.

5. Suppose X is a set, and $F(x)$ is an X-indexed family of locally continuous functors.

   First, given objects (A, B), we have the object part of this as the dependent function space $\Pi x : X.\ F(x)(A, B)$, with elements $u \sqsubseteq v$ if and only if for all $x \in X$, $u(x) \sqsubseteq_{F(x)(A,B)}$

$v(x)$.

This object is a true product over X. Given $\Pi x : X.\ F(x)(A, B)$, we can define the x-th projection as $\pi_x = \lambda f.\ f(x)$. Then, it's clear that given a family of morphisms $f_{x \in X} : Y \to F(x)(A, B)$, we can define a function $\langle f_{x \in X} \rangle : Y \to \Pi x : X.\ F(x)(A, B) = \lambda y.\ \lambda x.\ f_x(y)$, which means that for all x, and that $\pi_x \circ \langle f_{x \in X} \rangle = f_x$.

We show uniqueness by supposing that that there is some g such that $\pi_x \circ g = f_x$. Then, we know that $g = \lambda x.\ \pi_x \circ g$, which means that $g = \lambda x.\ f_x$, which is exactly $\langle f_{x \in X} \rangle$.

Next, given morphisms $f \in A \to C$ and $g \in D \to B$, we have $[\Pi x : X.\ F(x)](f, g) \in [\Pi x : X.\ F(x)](A, B) \to [\Pi x : X.\ F(x)](C, D)$ as:

$$[\Pi x : X.\ F(x)](f, g) = \lambda x.\ F(x)(f, g)$$

Clearly, this preserves identities and composition, and is hence a functor.

Now, suppose that $(f, g) \sqsubseteq (f', g')$, and that x is an arbitrary element of X. Then $[\Pi x : X.\ F(x)](f, g) = F(x)(f, g)$ and $[\Pi x : X.\ F(x)](f', g') = F(x)(f', g')$. Since $F(x)$ is a locally continuous functor, we know that $F(x)(f, g) \sqsubseteq F(x)(f', g')$, and so $[\Pi x : X.\ F(x)]$ preserves ordering.

Now, suppose that (f_i, g_i) form a chain. So, we know that

$$\begin{array}{rcll} \sqcup_i([\Pi x : X.\ F(x)](f_i, g_i)) & = & \sqcup_i(\lambda x : X.\ (F(x)(f_i, g_i))) & \\ & = & \lambda x : X.\ (\sqcup_i(F(x)(f_i, g_i))) & (*) \\ & = & \lambda x : X.\ (F(x)(\sqcup_i(f_i, g_i))) & \\ & = & [\Pi x : X.\ F(x)](\sqcup_i(f_i, g_i)) & \end{array}$$

The interesting step is (*); it is justified by the fact that we know that $\pi_x \circ \sqcup_i \langle h_i^x \rangle = \sqcup_i(\pi_x \circ_i \langle h_i^x \rangle) = \sqcup_i h_i^x$, and that $\pi_x \circ \langle \sqcup_i h_i^x \rangle = \sqcup_i h_i^x$, and that the mediating morphism is unique.

As a result, we can conclude that this functor is locally continuous.

6. Suppose X is a set, and $F(x)$ is an X-indexed family of locally continuous functors.

   First, given objects (A, B), we have the object part of the functor yielding the dependent sum $\Sigma x : X.\ F(x)(A, B)$. Ordering is given pairwise, equipping the set X with the trivial ordering. That is $(x, o) \sqsubseteq (x', o')$ if and only if $x = x'$ and $o \sqsubseteq_{F(x)(A,B)} o'$.

   This is a true coproduct over X. Given $\Sigma x : X.\ F(x)(A, B)$, we can define the injections $\iota_x \in F(x)(A, B) \to \Sigma x : X.\ F(x)(A, B)$ as $\lambda v.(x, v)$. Next, suppose we have a family of functions $f_x : F(x)(A, B) \to Y$. We can define a function $[f_{x \in X}] \in (\Sigma x : X.\ F(x)(A, B)) \to Y$ as $\lambda(x, v).\ (f_x\ v)$. It's clear that $[f_{x \in X}] \circ \iota_i = f_i$.

   Finally, we can establish uniqueness as follows. Suppose that there is a g such that $g \circ \iota_i = f_i$. Next, we know that $g = \lambda(x, v).\ g(x, v) = \lambda(x, v).\ (g \circ \iota_x)(v)$, which is clearly $\lambda(x, v).\ (f_x\ v)$, which is just $[f_{x \in X}]$

   Next, given morphisms $f \in A \to C$ and $g \in D \to B$, we have $[\Sigma x : X.\ F(x)](f, g) \in [\Pi x : X.\ F(x)](A, B) \to [\Sigma x : X.\ F(x)](C, D)$ as:

$$[\Sigma x : X.\ F(x)](f,g) = \lambda(x,v).\ (x, F(x)(f,g)(v))$$

This clearly preserves identities and composition, and hence defines a functor.

Now, suppose that $(f,g) \sqsubseteq (f',g')$ and that (x,v) is an element of $\Sigma x : X.\ F(x)(A,B)$. Then we have that $[\Sigma x : X.\ F(x)](f,g)](x,v) = (x, F(x)(f,g)(v))$ and that $[\Sigma x : X.\ F(x)](f',g')](x,v) = (x, F(x)(f',g')(v))$. So we know that $x = x$, and by the local continuity of $F(x)$, we know that $F(x)(f,g)(v) \sqsubseteq F(x)(f',g')(v)$. So this functor preserves ordering.

Finally, suppose (f_i, g_i) form a chain.

$$\begin{array}{rcll} \sqcup_i[\Sigma x : X.\ F(x)](f_i,g_i) & = & \sqcup_i \lambda(x,v).\ (F(x)(f_i,g_i)(v)) & \\ & = & \lambda(x,v).\ \sqcup_i\ (F(x)(f_i,g_i)(v)) & (*) \\ & = & \lambda(x,v).\ F(x)(\sqcup_i f_i, \sqcup_i g_i))(v) & \\ & = & [\Sigma x : X.\ F(x)](\sqcup_i f_i, \sqcup_i g_i) & \end{array}$$

The interesting step is (*); it is justified by the fact that we know that $(\sqcup_i[h_i^x]) \circ \iota_x = \sqcup_i([h_i^x] \circ \iota_x) = \sqcup_i h_i^x$, and that $[\sqcup_i h_i^x] \circ \iota_x = \sqcup_i h_i^x$, and that the mediating morphism is unique.

□

2.3.2 The Inverse Limit Construction

Once we have a locally continuous functor, we would like to find a solution to the fixed point equation it defines.

Proposition 7. *(Smyth and Plotkin) Any locally-continuous functor $F : CPO_\perp \times CPO_\perp^{op} \to CPO_\perp$ has a solution to the equation $X \cong F(X,X)$. Moreover, there is also a* minimal *isomorphism solving this equation.*

The existence of a solution follows from Scott's inverse limit construction, together with Smyth and Plotkin's characterization of such solution [45]. We give an explicit construction of their solution in the following subsection, in which we will always take F to be a locally-continuous functor of the type mentioned above.

Embeddings and Projections

First, recall that an *embedding* $e : C \to D$ between pointed CPOs is a continuous function such that there exists a function $p : D \to C$ (called a *projection*) with the properties that $p \circ e = id_C$ and $e \circ p \sqsubseteq id_D$.

Now, we'll introduce the category $CPO_\perp^O$, which is the category whose objects are the pointed domains, and whose morphisms from D to E are the embedding-projection pairs. The identity morphism from domain D to D is the pair $\langle id, id\rangle$, and the composition operation on $\langle e,p\rangle$ and $\langle e',p'\rangle$ is $\langle e' \circ e, p \circ p'\rangle$. To verify that this is indeed a category, we check that:

- The identity $\langle id, id\rangle : D \to D$ is an embedding-projection pair because $id \circ id = id$ and $id \sqsubseteq id$.

- The composition $\langle e, p\rangle \circ \langle e', p'\rangle$ is an embedding-projection pair because it is defined to be equal to $\langle e \circ e', p' \circ p\rangle$, and we have that embedding followed by projection is:

$$\begin{aligned}(p' \circ p) \circ (e \circ e') &= p' \circ (p \circ e) \circ e' \\ &= p' \circ id \circ e' \\ &= p' \circ e' \\ &= id\end{aligned}$$

  and likewise we have for a projection followed by an embedding:

$$\begin{aligned}(e \circ e') \circ (p' \circ p) &= e \circ (e' \circ p') \circ p \\ &\sqsubseteq e \circ id \circ p \\ &\sqsubseteq e \circ p \\ &\sqsubseteq id\end{aligned}$$

- Finally, it's clear that composition is associative and has identities as units because it inherits these properties from the underlying composition operations.

Now, consider the one-point domain $\emptyset_\perp = \{\perp\}$, and the sequence of domains X_i, defined inductively by $X_0 = \emptyset_\perp$ and $X_{i+1} = F(X_i, X_i)$. Next, we will define embeddings and projections $e_i : X_i \to X_{i+1}$ and $p_i : X_{i+1} \to X_i$ as follows:

$$\begin{aligned}e_0 &: X_0 \to X_1 &&= \lambda x.\ \perp \\ e_{i+1} &: X_{i+1} \to X_{i+2} &&= F(e_i, p_i) \\ p_0 &: X_1 \to X_0 &&= \lambda x.\ \perp \\ p_{i+1} &: X_{i+2} \to X_{i+1} &&= F(p_i, e_i)\end{aligned}$$

Lemma 2. *(Embeddings and Projections) Each $\langle e_i, p_i\rangle$ forms an arrow from X_i to X_{i+1} in $CPO^O_\perp$.*

Proof. This proof proceeds by induction on i.

- Case $i = 0$: Obviously $e_0 \circ p_0 = id$, since $X_0 = \{\perp\}$. Likewise, since $p_0(e_0(x)) = \perp$, and $\perp \sqsubseteq x$, it follows that $p_0 \circ e_0 \sqsubseteq id$.
- Case $i = n + 1$:

  First, we'll show that $e_i \circ p_i$ is the identity:

$$\begin{aligned}e_i \circ p_i &= e_{n+1} \circ p_{n+1} && \text{Def.} \\ &= F(e_n, p_n) \circ F(p_n, e_n) && \text{Def.} \\ &= F(e_n \circ p_n, e_n \circ p_n) && \text{Functor property} \\ &= F(id, id) && \text{Ind. hyp.} \\ &= id && \text{Functor property}\end{aligned}$$

  Now, we'll show that $p_i \circ e_i \sqsubseteq id$:

$$\begin{aligned}p_i \circ e_i &= p_{n+1} \circ e_{n+1} && \text{Def.} \\ &= F(p_n, e_n) \circ F(e_n, p_n) && \text{Def.} \\ &= F(p_n \circ e_n, p_n \circ e_n) && \text{Functor property}\end{aligned}$$

By induction, we know that $p_n \circ e_n \sqsubseteq id$, and because locally continuous functors are also monotone, we know that $F(p_n \circ e_n, p_n \circ e_n) \sqsubseteq F(id, id) \equiv id$.

□

Construction of the Domain

Now, we'll define the domain X to be the domain with the underlying set:

$$X \equiv \{x \in (\Pi n : \mathbb{N}.\ X_n) \mid \forall m : \mathbb{N}.\ x_m = p_m(x_{m+1})\}$$

with the ordering being the usual component-wise ordering. (As a notational convenience, we will write x_n to indicate the n-th component of x, or $x(n)$.) To be in $CPO_\perp$, it needs a least element, which is just $\lambda n : \mathbb{N}.\ \perp$.

We claim that this pointed CPO X is the colimit of the chain of domains X_i in $CPO^O_\perp$. To prove it, we must proceed in two stages.

Lemma 3. *(X is a cocone) X is a cocone of the diagram $X_0 \longrightarrow X_1 \longrightarrow \ldots$.*

Proof. To show this, we must give morphisms $\langle \hat{e}_i, \hat{p}_i \rangle : X_i \to X$. To do so, we'll define:

$$\hat{e}_n : X_n \to X \equiv \lambda x : X_n.\ \lambda m : \mathbb{N}.\ \begin{cases} p_{m,n}(x) & \text{if } m < n \\ x & \text{if } m = n \\ e_{n,m}(x) & \text{if } m > n \end{cases}$$

We define $e_{i,j}$ to be the composition $e_{j-1} \circ e_{j-2} \circ \ldots \circ e_i$, which will have the type $X_i \to X_j$. Likewise, we define $p_{i,j}$ to be the composition $p_i \circ \ldots \circ p_{j-1}$, which will have the type $X_j \to X_i$. The projection $\hat{p}_n : X \to X_n$ is much simpler. It's just

$$\hat{p}_n : X \to X_n \equiv \lambda x : X.\ x_n$$

Now, we'll verify that these do form an embedding-projection pair.

- First, we'll show that $\hat{p}_n \circ \hat{e}_n = id$.

$$\begin{aligned} \hat{p}_n \circ \hat{e}_n &= \lambda x : X_n.\ (\hat{p}_n \circ \hat{e}_n)\ x \\ &= \lambda x : X_n.\ \hat{p}_n(\hat{e}_n\ x) \\ &= \lambda x : X_n.\ (\hat{e}_n\ x)\ n \\ &= \lambda x : X_n.\ x \\ &= id \end{aligned}$$

- Now, we'll show that $\hat{e}_n \circ \hat{p}_n \sqsubseteq id$.

$$\begin{aligned} \hat{e}_n \circ \hat{p}_n &= \lambda x : X.\ (\hat{e}_n \circ \hat{p}_n)\ x \\ &= \lambda x : X.\ \hat{e}_n(\hat{p}_n\ x) \\ &= \lambda x : X.\ \hat{e}_n(x_n) \end{aligned}$$

Now, when applied to an argument $x \in X$, it's clear that the result element is component-wise equal to x for the components less than or equal to n, and less than that for components bigger than n, which makes the result smaller than x.

This establishes that there are morphisms $\langle \hat{e}_i, \hat{p}_i \rangle : X_i \to X$. Now, we need 1) to show that the equation $\langle \hat{e}_i, \hat{p}_i \rangle : X_i \to X = \langle \hat{e}_{i+1}, \hat{p}_{i+1} \rangle \circ \langle e_i, p_i \rangle$ holds, and 2) that $\bigsqcup_i \hat{e}_i \circ \hat{p}_i = id$, which will establish that the diagram commutes appropriately.

Expanding the definition of composition, we want to show that $\langle \hat{e}_i, \hat{p}_i \rangle = \langle \hat{e}_{i+1} \circ e_i, p_i \circ \hat{p}_{i+1} \rangle$. So, we have that

$$\begin{aligned}
\hat{e}_{i+1} \circ e_i &= \lambda x : X_i.\ \lambda m : \mathbb{N}. \begin{cases} p_{m,i+1}(e_i\ x) & \text{if } m < i+1 \\ e_i\ x & \text{if } m = i+1 \\ e_{i+1,m}(e_i\ x) & \text{if } m > i+1 \end{cases} \\
&= \lambda x : X_i.\ \lambda m : \mathbb{N}. \begin{cases} p_{m,i}(p_i(e_i\ x)) & \text{if } m < i+1 \\ e_{i,i+1}\ x & \text{if } m = i+1 \\ e_{i+1,m}(e_i\ x) & \text{if } m > i+1 \end{cases} \\
&= \lambda x : X_i.\ \lambda m : \mathbb{N}. \begin{cases} p_{m,i}(x) & \text{if } m < i+1 \\ e_{i,m}(x) & \text{if } m > i \end{cases} \\
&= \lambda x : X_i.\ \lambda m : \mathbb{N}. \begin{cases} p_{m,i}(x) & \text{if } m < i \\ x & \text{if } m = i \\ e_{i,m}(x) & \text{if } m > i \end{cases} \\
&= \hat{e}_i
\end{aligned}$$

In the other direction, we show that

$$\begin{aligned}
p_i \circ \hat{p}_{i+1} &= \lambda x : X.\ p_i(x_{i+1}) \\
&= \lambda x : X.\ x_i \\
&= \hat{p}_i
\end{aligned}$$

The second step follows from the definition of X.

Now, we need to show that $\bigsqcup_i \hat{e}_i \circ \hat{p}_i = id$.

$$\begin{aligned}
\bigsqcup_i \hat{e}_i \circ \hat{p}_i &= \bigsqcup_i \lambda x : X.\ \hat{e}_i(\hat{p}_i\ x) \\
&= \bigsqcup_i \lambda x : X.\ \hat{e}_i(x_i) \\
&= \bigsqcup_i \lambda x : X.\ \lambda m : \mathbb{N}. \begin{cases} p_{m,i}(x_i) & \text{if } m < i \\ x_i & \text{if } m = i \\ e_{i,m}(x_i) & \text{if } m > i \end{cases} \\
&= \bigsqcup_i \lambda x : X.\ \lambda m : \mathbb{N}. \begin{cases} x_m & \text{if } m \leq i \\ e_{i,m}(x_i) & \text{if } m > i \end{cases} \\
&= \bigsqcup_i \lambda x : X.\ \lambda m : \mathbb{N}. \begin{cases} x_m & \text{if } m \leq i \\ e_{i,m}(p_{i,m}\ x_m) & \text{if } m > i \quad (*) \end{cases} \\
&= \lambda x : X.\ \lambda m : \mathbb{N}.\ x_m \\
&= id
\end{aligned}$$

In $(*)$, we use the definition of X to see that the components greater than i are smaller than x's component at that index. So for each given i, the first i components of $\hat{e}_i \circ \hat{p}_i$ are the identity function, and below the identity for anything bigger than that. Thus, the limit as i goes to infinity is the identity function for all components.

□

To show that X is the colimit of this diagram, we need to show there is a unique map from it to any other cocone.

Lemma 4. *(Universality of X) Suppose that there is a Y with morphisms $\langle f_n, q_n\rangle : X_n \to Y$ and $q_n : Y \to X_n$, forming a cocone over $X_0 \longrightarrow X_1 \longrightarrow \ldots$. Then, there is a unique $\langle h_e, h_p\rangle : X \to Y$ such that for all n, $\langle f_n, q_n\rangle = \langle h_e, h_p\rangle \circ \langle \hat{e}_n, \hat{p}_n\rangle$.*

Proof. To show this, we need to explicitly construct h_e and h_p, and show that they form an embedding-projection pair. We'll define $h_e : X \to Y = \bigsqcup_i f_i \circ \hat{p}_i$, and define $h_p : Y \to X = \lambda y : Y.\ \lambda i : \mathbb{N}.\ q_i\ y$.

Before we can proceed any further, we need to establish that h_e actually defines a morphism — that is, we have to establish that $f_i \circ \hat{p}_i$ is a chain in i. So, assume we have some arbitrary i, and some arbitrary $x : X$.

1. Now, by the properties of embedding-projection pairs, we know $e_i(p_i\ x_{i+1}) \sqsubseteq x_{i+1}$,
2. By the continuity of f_{i+1}, this means $f_{i+1}(e_i(p_i\ x_{i+1})) \sqsubseteq f_{i+1}(x_{i+1})$.
3. By the fact that Y is a cocone, this means $f_i(p_i\ x_{i+1}) \sqsubseteq f_{i+1}(x_{i+1})$.
4. By the definition of X, this is the same as showing $f_i(x_i) \sqsubseteq f_{i+1}(x_{i+1})$.
5. By the definition of $\hat{p}$, this is the same as $f_i(\hat{p}_i\ x) \sqsubseteq f_{i+1}(\hat{p}_{i+1}\ x)$.
6. Since this holds for all x, we have shown $f_i \circ \hat{p}_i \sqsubseteq f_{i+1} \circ \hat{p}_{i+1}$.

Next, let's establish that h_e and h_p form an embedding-projection pair. To show that $h_e \circ h_p \sqsubseteq id$, we use equational reasoning:

$$\begin{array}{rcl} h_e \circ h_p & = & (\bigsqcup_i f_i \circ \hat{p}_i) \circ (\lambda y : Y.\ \lambda i : \mathbb{N}.\ (q_i\ y)) \\ & = & \bigsqcup_i (f_i \circ \hat{p}_i \circ (\lambda y : Y.\ \lambda i : \mathbb{N}.\ (q_i\ y)) \\ & = & \bigsqcup_i (\lambda y : Y.\ f_i(\hat{p}_i\ (\lambda i : \mathbb{N}.\ (q_i\ y)))) \\ & = & \bigsqcup_i (\lambda y : Y.\ f_i(q_i\ y)) \\ & \sqsubseteq & \bigsqcup_i \lambda y : Y.\ y \\ & \sqsubseteq & \lambda y : Y.\ y \end{array}$$

In other direction, we need to show $h_p \circ h_e = id$.

$$\begin{array}{rcl} h_p \circ h_e & = & (\lambda y : Y.\ \lambda j : \mathbb{N}.\ (q_j\ y)) \circ (\bigsqcup_i f_i \circ \hat{p}_i) \\ & = & \bigsqcup_i ((\lambda y : Y.\ \lambda j : \mathbb{N}.\ (q_j\ y)) \circ f_i \circ \hat{p}_i) \\ & = & \lambda x : X.\ \bigsqcup_i ((\lambda y : Y.\ \lambda j : \mathbb{N}.\ (q_j\ y)) \circ f_i \circ \hat{p}_i))\ x \\ & = & \lambda x : X.\ \bigsqcup_i ((\lambda y : Y.\ \lambda j : \mathbb{N}.\ (q_j\ y))\ (f_i\ (\hat{p}_i\ x))) \\ & = & \lambda x : X.\ \bigsqcup_i (\lambda j : \mathbb{N}.\ (q_j\ (f_i\ (\hat{p}_i\ x)))) \\ & = & \lambda x : X.\ \lambda j : \mathbb{N}.\ \bigsqcup_i ((q_j\ (f_i\ (\hat{p}_i\ x)))) \end{array}$$

To finish this calculation, consider an arbitrary $x : X$ and $j : \mathbb{N}$. Now, consider the tail of the chain, where $i > j$. Now, since we know that $q_k = p_k \circ q_{k+1}$, it follows that:

$$\begin{array}{rcl} q_j\ (f_i\ (\hat{p}_i\ x)) & = & p_{j,i}(q_i\ (f_i\ (\hat{p}_i\ x))) \\ & = & p_{j,i}(\hat{p}_i\ x) \\ & = & \hat{p}_j\ x \\ & = & x_j \end{array}$$

Which means that the least upper bound of the chain has to be $\lambda x : X.\ \lambda j : \mathbb{N}.\ x_j$ – which means that it is the identity.

So we have established that $\langle h_e, h_p \rangle$ is a morphism between X and Y. Next, let's see whether it commutes: $\langle f_n, q_n \rangle = \langle h_e, h_p \rangle \circ \langle \hat{e}_n, \hat{p}_n \rangle$. Unfolding the definition of composition, we get two proof obligations. First,

$$\begin{array}{rcl} h_e \circ \hat{e}_n & = & (\bigsqcup_i f_i \circ \hat{p}_i) \circ \hat{e}_n \\ & = & \bigsqcup_i (f_i \circ \hat{p}_i \circ \hat{e}_n) \\ & = & \lambda x : X_n.\ \bigsqcup_i (f_i (\hat{p}_i\ (\hat{e}_n\ x))) \\ & = & \lambda x : X_n.\ \bigsqcup_i (f_i (\hat{e}_n\ x\ i)) \\ & = & \lambda x : X_n.\ \bigsqcup_i f_i \left(\left\{ \begin{array}{ll} p_{i,n}(x) & \text{if } i < n \\ x & \text{if } i = n \\ e_{n,i}(x) & \text{if } i > n \end{array} \right. \right) \end{array}$$

To find the limit of this chain, consider any $i > n$. Because $f_{k+1} \circ e_k = f_k$, we can see that $f_i(e_{n,i}\ x) = f_n\ x$, which means that the limit is $f_n\ x$, and hence $h_e \circ \hat{e}_n = f_n$.

Next, consider $\hat{p}_n \circ h_p$:

$$\begin{array}{rcl} \hat{p}_n \circ h_p & = & \hat{p}_n \circ (\lambda y : Y.\ \lambda i : \mathbb{N}.\ (q_i\ y)) \\ & = & \lambda y : Y.\ \hat{p}_n (\lambda i : \mathbb{N}.\ (q_i\ y)) \\ & = & \lambda y : Y.\ (q_n\ y) \\ & = & q_n \end{array}$$

At this point, we have established that X is a weak colimit – there's a morphism from it to any other cone, but we still have yet to show that it is a unique morphism. So, suppose that we have some other mediating morphism $\langle h'_e, h'_p \rangle : X \to Y$.

For the embedding h'_e, we proceed as follows:

1. Now, it must be the case that $\langle h'_e, h'_p \rangle \circ \langle \hat{e}_n, \hat{p}_n \rangle = \langle f_n, q_n \rangle$.
2. So $h'_e \circ \hat{e}_n = f_n$.
3. Composing both sides with $\hat{p}_n$, we get $h'_e \circ \hat{e}_n \circ \hat{p}_n = f_n \circ \hat{p}_n$.
4. Taking limits of chains on both sides, we get $h'_e \circ \bigsqcup_n \hat{e}_n \circ p_n = \bigsqcup_n f_n \circ \hat{p}_n$
5. Simplifying, we get $h'_e = h_e$.

For the projection h'_p, we have

1. We have $\hat{p}_n \circ h'_p = q_n$.
2. Composing on both sides with $\hat{e}_n$, we have $\hat{e}_n \circ \hat{p}_n \circ h'_p = \hat{e}_n \circ q_n$.
3. Taking limits on both sides, we have $h'_p = \bigsqcup_n \hat{e}_n \circ q_n$.
4. Simplifying the limit expression, we get $h'_p = \lambda y : Y.\ \lambda n : \mathbb{N}.\ q_n(y)$.
5. So $h'_p = h_p$.

□

Showing X is a Solution to $F(X, X) \cong X$

Lemma 5. *(X is a fixed point) The isomorphism $F(X, X) \cong X$ is valid.*

Proof. First, note that applying F to each of the X_i and $\langle e_i, p_i \rangle$ yields X_{i+1} and $\langle e_{i+1}, p_{i+1} \rangle$. In other words, applying F to our old diagram gives us the same thing as before, only with the first element chopped off.

Therefore, X is still a colimit for this diagram, because if we replicate the colimit construction for this diagram, we can establish an isomorphism between the "new" construction and X, since the leading elements of the infinite product are determined by the requirement that $x_i = p_i(x_{i+1})$.

Since F is a locally continuous functor, it preserves colimits of chains $\bot \longrightarrow F(\bot) \longrightarrow \ldots$, so $F(X, X)$ is itself a colimiting object.

Since colimits are unique up to isomorphism, it follows that $F(X, X) \cong X$. $\square$

2.4 Solving Our Recursive Domain Equation

Given that we know that the basic operations we use in our interpretation are locally continuous, we can show that our interpretation function gives rise to a locally continuous functor.

Lemma 6. *Functoriality of* $[\![-]\!]^{\mathrm{m}}$ *and* $[\![-]\!]$

1. *For all canonical derivations* $\cdot \vdash \tau : \star$, $[\![\cdot \vdash \tau : \star]\!]^{\mathrm{m}}$ *is locally continuous.*
2. *For all canonical derivations* $\cdot \vdash A : \bigstar$, $[\![\cdot \vdash A : \bigstar]\!]$ *is locally continuous.*
3. H *is a locally continuous functor*
4. $\mathcal{K}$ *is a locally continuous functor.*

Proof. The proof of the first case follows by structural induction on the canonical derivations of monotypes. This is then used as a lemma in the proof of the second case, which is done via a structural induction on the canonical derivations of polytypes. This then lets us prove the third case, that H is a locally-continuous functor, because we can work from the inside out, using the fact that set-indexed sums and products of locally-continuous functors are themselves locally continuous. Finally, since $\mathcal{K}$ is just $H \to O$, we know it is locally continuous also. $\square$

Observe that $\mathcal{K}$, applied to any arguments, yields a pointed domain, since the Sierpinski domain is pointed, and a continuous function space into a pointed domain is itself pointed. Hence our functor K is also a functor into $CPO_{\bot}$, the category of complete *pointed* partial orders and continuous functions. Now, we can appeal to the existence of solutions to our recursive domain equation to solve for the solution to the equation

$$K \cong \mathcal{K}(K, K)$$

2.4.1 Computations form a Monad

Most of the base type constructors are obviously interpreted in terms of the underlying categorical constructions: pair types are categorical products, sums are categorical sums, functions are exponentials, and natural numbers are interpreted as a natural numbers object. However, we will need to check that the type of computations actually forms a monad.

Lemma 7. *(Computations form a Monad) The functor $T(A) = (A \to K) \to K$ forms a monad in CPO.*

Proof. We need to give a unit (a family of arrows $\eta_A : A \to T(A)$) and a lift operations (given $f : A \to T(B)$, we need to give $f^* : T(A) \to T(B)$, such that the following equations hold:

1. $\eta_A^* = id_{T(A)}$
2. $f^* \circ \eta_A = f$
3. $f^* \circ g^* = (f^* \circ g)^*$

(Technically, these conditions are the conditions for a Kleisli triple, which is equivalent to a monad.) We can now define η_A and f^* in terms of the internal language of CPO as follows:

$$\begin{array}{lcl} \eta_A(a) & = & \lambda k.\ k\ a \\ f^* & = & \lambda a' : (A \to K) \to K . \lambda k_b : (B \to K).\ a'(\lambda a.\ f\ a\ k_b) \end{array}$$

1. The proof of the first equation is as follows:

$$\begin{array}{rcl} \eta_A^* & = & \lambda a'.\ \lambda k_A.\ a'(\lambda a.\ \eta_A\ a\ k_A) \\ & = & \lambda a'.\ \lambda k_A.\ a'(\lambda a.\ k_A\ a) \\ & = & \lambda a'.\ \lambda k_A.\ a'\ k_A \\ & = & \lambda a'.\ a' \\ & = & id_{T(A)} \end{array}$$

2. The proof of the second equation is as follows:

$$\begin{array}{rcl} f^* \circ \eta_A & = & \lambda a.\ f^*(\eta_A(a)) \\ & = & \lambda a.\ f^*(\lambda k.\ k\ a) \\ & = & \lambda a.\ \lambda k_a.\ (\lambda k.\ k\ a)\ (\lambda a.\ f\ a\ k_a) \\ & = & \lambda a.\ \lambda k_a.\ (\lambda a.\ f\ a\ k_a)\ a \\ & = & \lambda a.\ \lambda k_a.\ f\ a\ k_a \\ & = & \lambda a.\ f\ a \\ & = & f \end{array}$$

3. The proof of the third equation is as follows:

$$\begin{array}{rcl} f^* \circ g^* & = & \lambda a''.\ f^*(g^*(a'')) \\ & = & \lambda a''.\ f^*(\lambda k.\ a''(\lambda a.\ g\ a\ k)) \\ & = & \lambda a''.\ \lambda k.\ (\lambda k.\ a''(\lambda a.\ g\ a\ k))\ (\lambda a_1.\ f\ a\ k) \\ & = & \lambda a''.\ \lambda k.\ a''(\lambda a.\ g\ a\ (\lambda a_1.\ f\ a_1\ k)) \\ & = & \lambda a''.\ \lambda k.\ a''(\lambda a.\ f^*\ (ga)\ k) \\ & = & \lambda a''.\ \lambda k.\ a''(\lambda a.\ (f^* \circ g)\ a\ k) \\ & = & (f^* \circ g)^* \end{array}$$

□

Pure expressions	e	$::=$	$\langle\rangle \mid \langle e, e'\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid$ $\mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}(e_0,\ x_1.\ e_1,\ x_2.\ e_2) \mid$ $\mathsf{z} \mid \mathsf{s}(e) \mid \mathsf{iter}(e, e_0, x.\ e_1)$ $x \mid \lambda x : A.\ e \mid e\ e' \mid$ $\Lambda\alpha : \kappa.\ e \mid e\ \tau \mid$ $\mathsf{pack}(\tau, e) \mid \mathsf{unpack}(\alpha, x) = e\ \mathsf{in}\ e' \mid$ $[c] \mid \mathsf{fix}\ x : D.\ e$
Computations	c	$::=$	$e \mid \mathsf{letv}\ x = e\ \mathsf{in}\ c \mid \mathsf{new}_A(e) \mid\ !e \mid e := e'$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Pointed Types	D	$::=$	$\bigcirc A \mid A \to D \mid \forall\alpha : \kappa.\ D \mid \mathbf{1} \mid D \times D$

Figure 2.8: Syntax of the Programming Language

2.5 The Programming Language

We have given the semantics of types in domain-theoretic terms. Now, we'll give the syntax and typing of the programming language. Then, we'll use the domain-theoretic semantics just given to first give a denotational semantics for the programming language, and second, to give an interesting equality theory for it. The syntax of terms is given in figure 2.8.

The pure terms of the language include the unit value $\langle\rangle$; pairs $\langle e, e'\rangle$ and projections $\mathsf{fst}\ e$ and $\mathsf{snd}\ e$; injections into sum types $\mathsf{inl}\ e$ and $\mathsf{inr}\ e$, and a case form $\mathsf{case}(e,\ x.\ e',\ y.\ e'')$; lambda abstractions $\lambda x : A.\ e$ and applications $e\ e'$; type abstraction $\Lambda\alpha : \kappa.\ e$ and type application $e\ \tau$; and existential packing $\mathsf{pack}(\tau, e)$ and unpacking $\mathsf{unpack}(\alpha, x) = e\ \mathsf{in}\ e'$. Suspended monadic computations $[c]$ are terms of the type $\bigcirc A$. Natural numbers are given by zero z and successor $\mathsf{s}(e)$ constructors, and are eliminated by primitive iteration $\mathsf{iter}(e, e_0, x.\ e_1)$. The restriction to primitive iteration ensures that infinitely looping programs cannot be defined by eliminating natural numbers, while still permitting us to define total functions such as addition and multiplication. (We will make free use of other inductive datatypes in following chapters, using the naturals as a prototypical example of how to handle them.)

For general recursion, we have a term-level fixed point operator, $\mathsf{fix}\ x : D.\ e$. It is not defined over all types; it is only permitted to range over *well-pointed* types. That is, fixed points are only defined over types whose interpretations are domains with least elements. The pointed types include the monadic types, functions into pointed types, products of pointed types, and polymorphic quantification over pointed types. Functions into pointed types corresponds to ML's usual fixed point, and taking fixed points of products corresponds to mutual recursion. Taking fixed points of polymorphic types corresponds to *polymorphic recursion*. The absence of nontermination (or any other effect) at other types ensures that the full beta and eta rules will be available for reasoning with them.

The typing rules for all of these forms are given in figure 2.9, and the computation forms of the language are given in figure 2.10.

$$\frac{}{\Theta;\ \Gamma \vdash \langle\rangle : \mathbf{1}}\ \textsc{EUnit} \qquad \frac{\Theta;\ \Gamma \vdash e_1 : A \qquad \Theta;\ \Gamma \vdash e_2 : B}{\Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A \times B}\ \textsc{EPair}$$

$$\frac{\Theta;\ \Gamma \vdash e : A \times B}{\Theta;\ \Gamma \vdash \mathsf{fst}\ e : A}\ \textsc{EFst} \qquad \frac{\Theta;\ \Gamma \vdash e : A \times B}{\Theta;\ \Gamma \vdash \mathsf{snd}\ e : B}\ \textsc{ESnd}$$

$$\frac{\Theta;\ \Gamma \vdash e : A}{\Theta;\ \Gamma \vdash \mathsf{inl}\ e : A + B}\ \textsc{EInl} \qquad \frac{\Theta;\ \Gamma \vdash e : B}{\Theta;\ \Gamma \vdash \mathsf{inr}\ e : A + B}\ \textsc{EInr}$$

$$\frac{\Theta;\ \Gamma \vdash e : A + B \qquad \Theta;\ \Gamma, x : A \vdash e_1 : C \qquad \Theta;\ \Gamma, y : B \vdash e_2 : C}{\Theta;\ \Gamma \vdash \mathsf{case}(e,\ x.\ e_1,\ y.\ e_2) : C}\ \textsc{ECase}$$

$$\frac{}{\Theta;\ \Gamma \vdash \mathsf{z} : \mathbb{N}}\ \textsc{EZero} \qquad \frac{\Theta;\ \Gamma \vdash e : \mathbb{N}}{\Theta;\ \Gamma \vdash \mathsf{s}(e) : \mathbb{N}}\ \textsc{ESucc}$$

$$\frac{\Theta;\ \Gamma \vdash e : \mathbb{N} \qquad \Theta;\ \Gamma \vdash e_0 : A \qquad \Theta;\ \Gamma, x : A \vdash e_1 : A}{\Theta;\ \Gamma \vdash \mathsf{iter}(e, e_0, x.\ e_1) : A}\ \textsc{EIter}$$

$$\frac{x : A \in \Gamma}{\Theta;\ \Gamma \vdash x : A}\ \textsc{EVar} \qquad \frac{\Theta;\ \Gamma, x : A \vdash e : B}{\Theta;\ \Gamma \vdash \lambda x : A.\ e : A \to B}\ \textsc{ELam}$$

$$\frac{\Theta;\ \Gamma \vdash e : A \to B \qquad \Theta;\ \Gamma \vdash e' : A}{\Theta;\ \Gamma \vdash e\ e' : B}\ \textsc{EApp}$$

$$\frac{\Theta, \alpha : \kappa;\ \Gamma \vdash e : A \qquad \Theta \vdash \Gamma}{\Theta;\ \Gamma \vdash (\Lambda\alpha : \kappa.\ e) : (\forall \alpha : \kappa.\ A)}\ \textsc{ETLam} \qquad \frac{\Theta;\ \Gamma \vdash e : \forall \alpha : \kappa.\ A \qquad \Theta \vdash \tau : \kappa}{\Theta;\ \Gamma \vdash e\ \tau : [\tau/\alpha]A}\ \textsc{ETApp}$$

$$\frac{\Theta;\ \Gamma \vdash \tau : \kappa \qquad \Theta, \alpha : \kappa \vdash A : \bigstar \qquad \Theta;\ \Gamma \vdash e : [\tau/\alpha]A}{\Theta;\ \Gamma \vdash \mathsf{pack}(\tau, e) : \exists \alpha : \kappa.\ A}\ \textsc{EPack}$$

$$\frac{\Theta;\ \Gamma \vdash e : \exists \alpha : \kappa.\ A \qquad \Theta, \alpha : \kappa;\ \Gamma, x : A \vdash e' : B \qquad \Theta \vdash B : \bigstar}{\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = e\ \mathsf{in}\ e' : B}\ \textsc{EUnpack}$$

$$\frac{\Theta;\ \Gamma \vdash c \div A}{\Theta;\ \Gamma \vdash [c] : \bigcirc A}\ \textsc{EMonad} \qquad \frac{\Theta;\ \Gamma, x : D \vdash e : D}{\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e : D}\ \textsc{EFix}$$

$$\frac{\Theta;\ \Gamma \vdash e : A \qquad \Theta \vdash A \equiv B : \bigstar}{\Theta;\ \Gamma \vdash e : B}\ \textsc{EKeq}$$

Figure 2.9: Typing of the Pure Expressions

$$\frac{\Theta;\ \Gamma \vdash e : A}{\Theta;\ \Gamma \vdash e \div A}\ \textsc{CReturn} \qquad \frac{\Theta;\ \Gamma \vdash e : \bigcirc A \qquad \Theta;\ \Gamma, x : A \vdash c \div B}{\Theta;\ \Gamma \vdash \mathsf{letv}\ x = e\ \mathsf{in}\ c \div B}\ \textsc{CLet}$$

$$\frac{\Theta;\ \Gamma \vdash e : \mathsf{ref}\ A}{\Theta;\ \Gamma \vdash\ !e \div A}\ \textsc{CGet} \qquad \frac{\Theta;\ \Gamma \vdash e : \mathsf{ref}\ A \qquad \Theta;\ \Gamma \vdash e' : A}{\Theta;\ \Gamma \vdash e := e' \div \mathbf{1}}\ \textsc{CSet}$$

$$\frac{\Theta;\ \Gamma \vdash e : A}{\Theta;\ \Gamma \vdash \mathsf{new}_A(e) \div \mathsf{ref}\ A}\ \textsc{CNew}$$

Figure 2.10: Typing of Monadic Expressions

One abbreviation we will make use of is to write $\mathsf{run}(e)$ for $\mathsf{letv}\ [x] = e\ \mathsf{in}\ x$. Just as $[c]$ embeds a computation of type A into an expression term of type $\bigcirc A$, the abbreviation $\mathsf{run}(e)$ does the reverse, giving a computation of type A from an expression of type $\bigcirc A$.

2.6 Denotational Semantics

We give the semantics of the expression and command languages with functions $[\![\Theta;\ \Gamma \vdash e : A]\!]^e$ and $[\![\Theta;\ \Gamma \vdash c \div A]\!]^c$. Since we have two contexts, we will need two environments, one a type environment (as before, a tuple of closed type expressions), and the other, a value environment (consisting of a tuple of values of the appropriate type). In other words, the interpretation of a term is a type-indexed family of morphisms in CPO.

We give the interpretation of contexts in Figure 2.11. In this definition, we take the usual liberties in not explicitly giving the isomorphisms necessary to implement structural rules like Exchange. The definitions of the two mutually recursive interpretation functions, $[\![\Theta;\ \Gamma \vdash e : A]\!]^e$ and $[\![\Theta;\ \Gamma \vdash c \div A]\!]^c$, which interpret pure expressions and computations respectively, are given in figure 2.12 and figure 2.13.

In the types I give for these interpretation, for readability I elide occurences of (K, K), since they never change and should be obvious from context. Furthemore, I confuse n-ary and iterated products, and sometimes suppress contexts (e.g., write $[\![A]\!]\ \theta$ instead of $[\![\Theta \vdash A : \bigstar]\!]\ \theta$) when it would either be cluttered or run off the page.

We also give a set of rules for deriving equalities between program expressions, in the figures from 2.14 to 2.17. These rules include the β- and η-equalities of the lambda calculus, for sums, products, and function spaces, the monad laws for computation types, as well as the β- and η-equalities for numbers, existentials and universals. The η-rule for numbers justifies reasoning about iterative programs via induction on the natural numbers. The η-rule for universals and existentials arises from simple extensionality over types: since our model is not parametric, it does not justify parametric principles of reasoning.

Below, we collect the theorems describing the properties of these two judgments, and give their proofs in the following subsection.

Lemma 8. *(Soundness of Weakening) Suppose* $\Theta;\ \Gamma \vdash e : A$.

$$\frac{}{\Theta \vdash \cdot}\ \textsc{CtxNil} \qquad \frac{\Theta \vdash \Gamma \qquad \Theta \vdash A : \bigstar}{\Theta \vdash \Gamma, x : A}\ \textsc{CtxCons}$$

$$\begin{array}{lcl} [\![\Theta \vdash \cdot]\!]\ \theta & = & 1 \\ [\![\Theta \vdash \Gamma, x : A]\!]\ \theta & = & [\![\Theta \vdash \Gamma]\!]\ \theta \times [\![\Theta \vdash A : \bigstar]\!]\theta \end{array}$$

Figure 2.11: Interpretation of Program Contexts

1. *It is the case that* $[\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : A]\!]^e\ (\theta, \tau)\ \gamma$ *is equal to* $[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma$.
2. *It is the case that* $[\![\Theta;\ \Gamma, x : B \vdash e : A]\!]^e\ \theta\ (\gamma, v)$ *is equal to* $[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma$.

Lemma 9. *(Soundness of Type Substitution) If we know that* $\Theta \vdash \tau : \kappa$*, then*

1. *If* $\Theta, \alpha : \kappa;\ \Gamma \vdash e : A$*, then* $[\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : A]\!]^e\ (\theta, [\![\tau]\!]^s\ \theta)\ \gamma$ *is equal to* $[\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]e : [\tau/\alpha]A]\!]^e\ \theta\ \gamma$
2. *If* $\Theta, \alpha : \kappa;\ \Gamma \vdash c \div A$*, then* $[\![\Theta, \alpha : \kappa;\ \Gamma \vdash c \div A]\!]^c\ (\theta, [\![\tau]\!]^s\ \theta)\ \gamma$ *is equal to* $[\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]c \div [\tau/\alpha]A]\!]^c\ \theta\ \gamma$

Lemma 10. *(Soundness of Substitution)*

1. *If we know that* $\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B$ *and* $\Theta;\ \Gamma \vdash e' : A$ *and* $\Theta \vdash \theta$*, then* $[\![\Theta;\ \Gamma \vdash [e'/y]e : B]\!]^e\ \theta\ (\gamma, \gamma') = [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B]\!]^e\ \theta\ \ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$
2. *If we know that* $\Theta;\ \Gamma, y : A, \Gamma' \vdash c \div B$ *and* $\Theta;\ \Gamma \vdash e' : A$ *and* $\Theta \vdash \theta$*, then* $[\![\Theta;\ \Gamma \vdash [e'/y]c : B]\!]^c\ \theta\ (\gamma, \gamma') = [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash c \div B]\!]^c\ \theta\ \ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^c\ \theta\ \gamma, \gamma')$

Lemma 11. *(Soundness of Equality Rules) We have that:*

1. *If* $\Theta;\ \Gamma \vdash e \equiv e' : A$*, then* $\Theta;\ \Gamma \vdash e : A$ *and* $\Theta;\ \Gamma \vdash e' : A$ *and* $[\![\Theta;\ \Gamma \vdash e : A]\!]^e = [\![\Theta;\ \Gamma \vdash e' : A]\!]^e$.
2. *If* $\Theta; \Gamma \vdash c \equiv c' \div A$*, then* $\Theta;\ \Gamma \vdash c \div A$ *and* $\Theta;\ \Gamma \vdash c' \div A$ *and* $[\![\Theta;\ \Gamma \vdash c \div A]\!]^c = [\![\Theta;\ \Gamma \vdash c' \div A]\!]^c$.

2.6.1 Proofs

Lemma. *(8, page 35: Soundness of Weakening) Suppose* $\Theta;\ \Gamma \vdash e : A$.

1. *It is the case that* $[\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : A]\!]^e\ (\theta, \tau)\ \gamma$ *is equal to* $[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma$.
2. *It is the case that* $[\![\Theta;\ \Gamma, x : B \vdash e : A]\!]^e\ \theta\ (\gamma, v)$ *is equal to* $[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma$.

Proof. The proof is by induction on the typing derivation of $\Theta;\ \Gamma \vdash e : A$. □

$$
\begin{array}{lcl}
[\![\Theta;\ \Gamma \vdash e : A]\!]^e & \in & \Pi\ \theta : [\![\Theta]\!]^s.\ [\![\Theta \vdash \Gamma]\!]\ \theta \to [\![\Theta \vdash A : \bigstar]\!]\ \theta \\
\\
[\![\Theta;\ x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i]\!]^e\ \theta\ \gamma & = & \pi_{x_1:A_1,\ldots,x_n:A_n \vdash x_i:A_i}(\gamma) \\
[\![\Theta;\ \Gamma \vdash \lambda x : A.\ e : A \to B]\!]^e\ \theta\ \gamma & = & \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e : B]\!]^e\ \theta\ (\gamma, v) \\
[\![\Theta;\ \Gamma \vdash e\ e' : B]\!]^e\ \theta\ \gamma & = & ([\![\Theta;\ \Gamma \vdash e : A \to B]\!]^e\ \theta\ \gamma)\ ([\![\Theta;\ \Gamma \vdash e' : A]\!]^e \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \langle\rangle : \mathbf{1}]\!]^e\ \theta\ \gamma & = & * \\
[\![\Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A_1 \times A_2]\!]^e\ \theta\ \gamma & = & ([\![\Theta;\ \Gamma \vdash e_1 : A_1]\!]^e\ \theta\ \gamma, [\![\Theta;\ \Gamma \vdash e_2 : A_2]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \mathsf{fst}\ e : A_1]\!]^e\ \theta\ \gamma & = & \pi_1([\![\Theta;\ \Gamma \vdash e : A_1 \times A_2]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \mathsf{snd}\ e : A_2]\!]^e\ \theta\ \gamma & = & \pi_2([\![\Theta;\ \Gamma \vdash e : A_1 \times A_2]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \mathsf{inl}\ e : A_1 + A_2]\!]^e\ \theta\ \gamma & = & \iota_1([\![\Theta;\ \Gamma \vdash e : A_1]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \mathsf{inr}\ e : A_1 + A_2]\!]^e\ \theta\ \gamma & = & \iota_2([\![\Theta;\ \Gamma \vdash e : A_2]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \mathsf{case}(e,\ x.\ e_1,\ y.\ e_2) : C]\!]^e\ \theta\ \gamma & = & \text{let } a = [\![\Theta;\ \Gamma \vdash e : A_1 + A_2]\!]^e \theta\ \gamma \text{ in} \\
 & & \text{let } f_1 = \lambda v_1.\ [\![\Theta;\ x : A_1, \Gamma \vdash e_1 : C]\!]^e \theta\ (\gamma, v_1) \text{ in} \\
 & & \text{let } f_2 = \lambda v_2.\ [\![\Theta;\ y : A_2, \Gamma \vdash e_2 : C]\!]^e \theta\ (\gamma, v_2) \text{ in} \\
 & & [f_1, f_2](a) \\
[\![\Theta;\ \Gamma \vdash \mathsf{z} : \mathbb{N}]\!]^e\ \theta\ \gamma & = & 0 \\
[\![\Theta;\ \Gamma \vdash \mathsf{s}(e) : \mathbb{N}]\!]^e\ \theta\ \gamma & = & 1 + ([\![\Theta;\ \Gamma \vdash e : \mathbb{N}]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \mathsf{iter}(e, e_0, x.\ e_1) : A]\!]^e\ \theta\ \gamma & = & \text{let } a = [\![\Theta;\ \Gamma \vdash e : \mathbb{N}]\!]^e\ \theta\ \gamma \text{ in} \\
 & & \text{let } i = [\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ \gamma \text{ in} \\
 & & \text{let } s = \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, v) \text{ in} \\
 & & iter_A[i, s](a) \\
[\![\Theta;\ \Gamma \vdash [c] : \bigcirc A]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma \\
[\![\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e : D]\!]^e\ \theta\ \gamma & = & fix(\lambda v.\ ([\![\Theta;\ \Gamma, x : D \vdash e : D]\!]^e\ \theta\ (\gamma, v))) \\
[\![\Theta;\ \Gamma \vdash \Lambda\alpha : \kappa.\ e : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma & = & \lambda\tau : [\![\kappa]\!].\ ([\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : A]\!]^e\ (\theta, \tau)\ \gamma \\
[\![\Theta;\ \Gamma \vdash e\ \tau : A[\tau/\alpha]]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash e : \forall\alpha : \kappa.\ A]\!]\ \theta\ \gamma\ [\theta(\tau)] \\
[\![\Theta;\ \Gamma \vdash \mathsf{pack}(\tau, e) : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma & = & ([\theta(\tau)], [\![\Theta;\ \Gamma \vdash e : A[\tau/\alpha]]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = e\ \mathsf{in}\ e' : B]\!]^e\ \theta\ \gamma & = & (\lambda\,\langle\tau, v\rangle\,.\ [\![\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash e' : B]\!]^e\ (\theta, \tau)\ (\gamma, v)) \\
 & & ([\![\Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma) \\
[\![\Theta;\ \Gamma \vdash e : B]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma \text{ when } \Theta \vdash A \equiv B : \bigstar \text{ by EKeq} \\
\\
iter_A[i, f](0) & = & i \\
iter_A[i, f](n+1) & = & f(iter_A[i, f](n)) \\
\\
\pi_{x_i} & = & \text{project the component corresponding to } x_i \text{ in a context } x_1 : A_1,
\end{array}
$$

Figure 2.12: Interpretation of Pure Terms

$$
\begin{array}{lcl}
[\![\Theta;\ \Gamma \vdash c \div A]\!]^c & \in & \Pi\, \theta : [\![\Theta]\!]^s.\ [\![\Theta \vdash \Gamma]\!]\ \theta \to T([\![\Theta \vdash A : \bigstar]\!]\ \theta) \\
\\
[\![\Theta;\ \Gamma \vdash e \div A]\!]^c\ \theta & = & \eta_{[\![A]\!]\ \theta} \circ ([\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta) \\
[\![\Theta;\ \Gamma \vdash \mathsf{letv}\ x = e\ \mathsf{in}\ c \div B]\!]^c\ \theta\ \gamma & = & \text{let } m : [\![\bigcirc A]\!]\theta = [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]^e\ \theta\ \gamma \text{ in} \\
 & & \text{let } f : [\![A]\!]\theta \to [\![\bigcirc B]\!]\theta = \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash c : B]\!]^e\ \theta\ (\gamma, v) \text{ in} \\
 & & f^*(m) \\
[\![\Theta;\ \Gamma \vdash\ !e \div A]\!]^c\ \theta\ \gamma & = & \text{let } l = [\![\Theta;\ \Gamma \vdash e : \mathsf{ref}\ A]\!]^e\ \theta\ \gamma \text{ in} \\
 & & \lambda k.\ \lambda(L, h).\ \begin{cases} k\ (h\ l)\ (L, h) & \text{when } l \in L \\ \top & \text{otherwise} \end{cases} \\
[\![\Theta;\ \Gamma \vdash e := e' \div \mathbf{1}]\!]^c\ \theta\ \gamma & = & \text{let } l = [\![\Theta;\ \Gamma \vdash e : \mathsf{ref}\ A]\!]^e\ \theta\ \gamma \text{ in} \\
 & & \text{let } v = [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma \text{ in} \\
 & & \lambda k.\ \lambda(L, h).\ \begin{cases} k\ \langle\rangle\ (L, [h|l : v]) & \text{when } l \in L \\ \top & \text{otherwise} \end{cases} \\
[\![\Theta;\ \Gamma \vdash \mathsf{new}_A(e) \div \mathsf{ref}\ A]\!]^c\ \theta\ \gamma & = & \text{let } v = [\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma \text{ in} \\
 & & \lambda k.\ \lambda(L, h). \\
 & & \qquad \text{let } l = newloc(L, A) \text{ in} \\
 & & \qquad k\ l\ (L \cup \{l\}, [h|l : v]) \\
\\
newloc(L, A) & = & (1 + \sup\{n \in \mathbb{N} \mid \exists B.\ (n, B) \in L\}, A)
\end{array}
$$

Figure 2.13: Interpretation of Computations

Lemma. *(9, page 36: Soundness of Type Substitution) If we know that* $\Theta \vdash \tau : \kappa$*, then*

- *If* $\Theta, \alpha : \kappa;\ \Gamma \vdash e : A$*, then* $[\![\Theta, \alpha : \kappa, \Theta';\ \Gamma \vdash e : A]\!]^e\ (\theta, [\![\tau]\!]^s\ \theta, \theta')\ \gamma$ *is equal to* $[\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]e : [\tau/\alpha]A]\!]^e\ \theta\ \gamma$
- *If* $\Theta, \alpha : \kappa;\ \Gamma \vdash c \div A$*, then* $[\![\Theta, \alpha : \kappa;\ \Gamma \vdash c \div A]\!]^c\ (\theta, [\![\tau]\!]^s\ \theta, \theta')\ \gamma$ *is equal to* $[\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]c \div [\tau/\alpha]A]\!]^c\ \theta\ \gamma$

Proof. The proof is by induction on the structure of the typing derivation. The interesting case is:

- case EHyp:

  $[\![\Theta, \alpha : \kappa, \Theta';\ \Gamma \vdash x_i : A_i]\!]^e\ (\theta, [\![\tau]\!]^s(\theta), \theta')\ \gamma$

  $= \pi_{x_i}(\gamma)$ Semantics

  Now, observe that the tuple γ is an element of $[\![\Theta, \alpha : \kappa, \Theta' \vdash \Gamma]\!]\ (\theta, [\![\tau]\!]^s\ \theta, \theta')$.

  $[\![\Theta, \alpha : \kappa \vdash \Gamma]\!]\ (\theta, [\![\tau]\!]^s\ \theta) =$

  $$
  \begin{array}{ll}
  = & [\![[\![\Theta, \alpha : \kappa, \Theta' \vdash A_1 : \bigstar]\!]\ (\theta, [\![\tau]\!]^s\ \theta, \theta') \times \ldots \times [\![\Theta, \alpha : \kappa \vdash A_n : \bigstar]\!]\ (\theta, [\![\tau]\!]^s\ \theta, \theta')]\!] \\
  = & [\![\Theta \vdash [\tau/\alpha]A_1 : \bigstar]\!]\ (\theta, \theta') \times \ldots \times [\![\Theta \vdash [\tau/\alpha]A_n : \bigstar]\!]\ (\theta, \theta') \\
  = & [\![\Theta \vdash [\tau/\alpha]\Gamma]\!]\ (\theta, \theta')
  \end{array}
  $$

$$\frac{\Theta;\ \Gamma \vdash e : \mathbf{1} \qquad \Theta;\ \Gamma \vdash e' : \mathbf{1}}{\Theta;\ \Gamma \vdash e \equiv e' : \mathbf{1}}\ \textsc{EqUnit}$$

$$\frac{\Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A_1 \times A_2}{\Theta;\ \Gamma \vdash \mathsf{fst}\ \langle e_1, e_2\rangle \equiv e_1 : A_1}\ \textsc{EqPairFst} \qquad \frac{\Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A_1 \times A_2}{\Theta;\ \Gamma \vdash \mathsf{snd}\ \langle e_1, e_2\rangle \equiv e_2 : A_2}\ \textsc{EqPairSnd}$$

$$\frac{\Theta;\ \Gamma \vdash e : A_1 \times A_2}{\Theta;\ \Gamma \vdash e \equiv \langle \mathsf{fst}\ e, \mathsf{snd}\ e\rangle : A_1 \times A_2}\ \textsc{EqPairEta}$$

$$\frac{\Theta;\ \Gamma, x : A \vdash e\ x \equiv e'\ x : B \qquad x \notin \mathrm{FV}(e, e')}{\Theta;\ \Gamma \vdash e \equiv e' : A \to B}\ \textsc{EqFunEta}$$

$$\frac{\Theta;\ \Gamma \vdash (\lambda x : A.\ e)\ e' : B \qquad x \notin \Gamma}{\Theta;\ \Gamma \vdash (\lambda x : A.\ e)\ e' \equiv [e'/x]e : B}\ \textsc{EqFunBeta}$$

$$\frac{\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inl}\ e,\ x.\ e_1,\ y.\ e_2) : C}{\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inl}\ e,\ x.\ e_1,\ y.\ e_2) \equiv [e/x]e_1 : C}\ \textsc{EqSumInlBeta}$$

$$\frac{\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inr}\ e,\ x.\ e_1,\ y.\ e_2) : C}{\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inr}\ e,\ x.\ e_1,\ y.\ e_2) \equiv [e/y]e_2 : C}\ \textsc{EqSumInrBeta}$$

$$\frac{\Theta;\ \Gamma \vdash e : A + B \qquad \Theta;\ \Gamma, z : A + B \vdash e' : C}{\Theta;\ \Gamma \vdash [e/z]e' \equiv \mathsf{case}(e,\ x.\ [\mathsf{inl}\ x/z]e',\ y.\ [\mathsf{inr}\ y/z]e') : C}\ \textsc{EqSumEta}$$

$$\frac{\Theta; \Gamma \vdash c \equiv c' \div A}{\Theta;\ \Gamma \vdash [c] \equiv [c'] : \bigcirc A}\ \textsc{EqMonad} \qquad \frac{\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e : D}{\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e \equiv [\mathsf{fix}\ x : D.\ e/x]e : D}\ \textsc{EqFix}$$

Figure 2.14: Equality Rules for Sums, Products, Exponentials, and Suspended Computations

$$\frac{\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{z}, e_0, x.\ e_1) : A}{\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{z}, e_0, x.\ e_1) \equiv e_0 : A}\ \textsc{EqNatZBeta}$$

$$\frac{\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{s}(e), e_0, x.\ e_1) : A}{\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{s}(e), e_0, x.\ e_1) \equiv [\mathsf{iter}(e, e_0, x.\ e_1)/x]e_1 : A}\ \textsc{EqNatSBeta}$$

$$\frac{\begin{array}{c}\Theta;\ \Gamma \vdash e_0 : A \qquad \Theta;\ \Gamma, x : A \vdash e_1 : A \qquad \Theta;\ \Gamma \vdash e : A \\ \Theta;\ \Gamma \vdash [\mathsf{z}/n]e \equiv e_0 : A \qquad \Theta;\ \Gamma, m : \mathbb{N} \vdash [\mathsf{s}(m)/n]e \equiv [[m/n]e/x]e_1 : A\end{array}}{\Theta;\ \Gamma, n : \mathbb{N} \vdash e \equiv \mathsf{iter}(n, e_0, x.\ e_1) : A}\ \textsc{EqNatEta}$$

$$\frac{\Theta;\ \Gamma \vdash \Lambda\alpha : \kappa.\ e : \forall\alpha : \kappa.\ A \qquad \Theta \vdash \tau : \kappa}{\Theta;\ \Gamma \vdash (\Lambda\alpha : \kappa.\ e)\ \tau \equiv [\tau/\alpha]e : [\tau/\alpha]A}\ \textsc{EqAllBeta}$$

$$\frac{\begin{array}{c}\Theta, \alpha : \kappa;\ \Gamma \vdash e\ \alpha \equiv e'\ \alpha : A \\ \Theta;\ \Gamma \vdash e : \forall\alpha : \kappa.\ A \qquad \Theta;\ \Gamma \vdash e' : \forall\alpha : \kappa.\ A \qquad \Theta \vdash \Gamma\end{array}}{\Theta;\ \Gamma \vdash e \equiv e' : \forall\alpha : \kappa.\ A}\ \textsc{EqAllEta}$$

$$\frac{\Theta;\ \Gamma \vdash \mathsf{pack}(\tau, e) : \exists\alpha : \kappa.\ A \qquad \Theta, \alpha : \kappa;\ \Gamma, x : A \vdash e' : B}{\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = \mathsf{pack}(\tau, e)\ \mathsf{in}\ e' \equiv [\tau/\alpha][e/x]e' : B}\ \textsc{EqExistsBeta}$$

$$\frac{\Theta;\ \Gamma, z : \exists\alpha : \kappa.\ A \vdash e' : B \qquad \Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A}{\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = e\ \mathsf{in}\ [\mathsf{pack}(\alpha, x)/z]e' \equiv [e/z]e' : B}\ \textsc{EqExistsEta}$$

Figure 2.15: Equality Rules for Numbers, Universals, and Existentials

$$\frac{\Theta;\ \Gamma \vdash c \div A}{\Theta; \Gamma \vdash c \equiv \mathsf{letv}\ x = [c]\ \mathsf{in}\ x \div A}\ \textsc{EqCommandEta}$$

$$\frac{\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [e]\ \mathsf{in}\ c \div A}{\Theta; \Gamma \vdash \mathsf{letv}\ x = [e]\ \mathsf{in}\ c \equiv [e/x]c \div A}\ \textsc{EqCommandBeta}$$

$$\frac{\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [\mathsf{letv}\ y = e\ \mathsf{in}\ c_1]\ \mathsf{in}\ c_2 \div A}{\Theta; \Gamma \vdash \mathsf{letv}\ x = [\mathsf{letv}\ y = e\ \mathsf{in}\ c_1]\ \mathsf{in}\ c_2 \equiv \mathsf{letv}\ y = e\ \mathsf{in}\ \mathsf{letv}\ x = [c_1]\ \mathsf{in}\ c_2 \div A}\ \textsc{EqCommandAssoc}$$

Figure 2.16: Equality Rules for Computations

$$\frac{\Theta;\ \Gamma \vdash e : A}{\Theta;\ \Gamma \vdash e \equiv e : A}\ \textsc{EqRefl} \qquad \frac{\Theta;\ \Gamma \vdash e \equiv e' : A}{\Theta;\ \Gamma \vdash e' \equiv e : A}\ \textsc{EqSymm}$$

$$\frac{\Theta;\ \Gamma \vdash e \equiv e' : A \qquad \Theta;\ \Gamma \vdash e' \equiv e'' : A}{\Theta;\ \Gamma \vdash e \equiv e'' : A}\ \textsc{EqTrans}$$

$$\frac{\Theta;\ \Gamma, x : A \vdash e_1 \equiv e_2 : B \qquad \Theta;\ \Gamma \vdash e_1' \equiv e_2' : A}{\Theta;\ \Gamma \vdash [e_1'/x]e_1 \equiv [e_2'/x]e_2 : B}\ \textsc{EqSubst}$$

$$\frac{\Theta;\ \Gamma \vdash c \div A}{\Theta; \Gamma \vdash c \equiv c \div A}\ \textsc{EqCommandRefl} \qquad \frac{\Theta; \Gamma \vdash c \equiv c' \div A}{\Theta; \Gamma \vdash c' \equiv c \div A}\ \textsc{EqCommandSymm}$$

$$\frac{\Theta; \Gamma \vdash c \equiv c' \div A \qquad \Theta; \Gamma \vdash c' \equiv c'' \div A}{\Theta; \Gamma \vdash c \equiv c'' \div A}\ \textsc{EqCommandTrans}$$

$$\frac{\Theta; \Gamma, x : A \vdash c_1 \equiv c_2 \div B \qquad \Theta;\ \Gamma \vdash e_1 \equiv e_2 : A}{\Theta; \Gamma \vdash [e_1/x]c_1 \equiv [e_2/x]c_2 \div B}\ \textsc{EqCommandSubst}$$

Figure 2.17: Congruence Rules for Equality

So γ is also an element of $[\![\Theta, \Theta' \vdash [\tau/\alpha]\Gamma]\!]\ (\theta, \theta')$ as well, and so
$[\![\Theta;\ [\tau/\alpha]\Gamma \vdash x_i : [\tau/\alpha]A_i]\!]^e\ \theta\ \gamma =$

$$\begin{array}{lll} = & \pi_{x_i}(\gamma) & \text{Semantics} \\ = & [\![\Theta, \alpha : \kappa, \Theta';\ \Gamma \vdash x_i : A_i]\!]^e\ (\theta, [\![\theta]\!]^s\ \tau, \theta')\ \gamma & \end{array}$$

- other cases: These all follow the structure of the derivation.

□

Lemma. *(10, page 36: Soundness of Substitution)*

1. *If we know that* $\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B$ *and* $\Theta;\ \Gamma \vdash e' : A$ *and* $\Theta \vdash \theta$*, then*
   $[\![\Theta;\ \Gamma \vdash [e'/y]e : B]\!]^e\ \theta\ (\gamma, \gamma') = [\![\Theta;\ \Gamma, y : A \vdash e : B]\!]^e\ \theta\ \ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$
2. *If we know that* $\Theta;\ \Gamma, y : A, \Gamma' \vdash c \div B$ *and* $\Theta;\ \Gamma \vdash e' : A$ *and* $\Theta \vdash \theta$*, then*
   $[\![\Theta;\ \Gamma \vdash [e'/y]c : B]\!]^c\ \theta\ (\gamma, \gamma') = [\![\Theta;\ \Gamma, y : A \vdash c \div B]\!]^c\ \theta\ \ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$

Proof. This property follows by a mutual structural induction on the derivations $\Theta;\ \Gamma, y : A \vdash e : B$ and $\Theta;\ \Gamma, y : A \vdash c \div C$.

First, we'll do the cases for pure terms. (When there's no confusion, we'll write $[\![e]\!]^e$ for $[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta$.)

- case EVar: There are two cases, depending on whether the $e = x_i$, or $e = y$

1. If $e = x_i$ for some i, then $[e'/y]x_i = x_i$, and so we have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]x_i : B]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash x_i : B]\!]^e\ \theta\ (\gamma, \gamma') & \text{Subst.} \\
= & \pi_{x_i}(\gamma, \gamma') & \text{Semantics} \\
= & \pi_{x_i}(\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma') & \text{Adjusting } i \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash x_i : B]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta, \gamma') & \text{Semantics}
\end{array}
$$

2. If $e = y$, then we have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]y : A\ \theta\ (\gamma, \gamma')]\!]^e$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash e' : A]\!]^e\ \theta\ (\gamma, \gamma') & \text{Subst.} \\
= & [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma & \text{Since } FV(e') \cap \Gamma = \emptyset \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash y : A]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash y : A]\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics}
\end{array}
$$

- case ELam: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](\lambda x : B.\ e) : B \to B'\ \theta\ (\gamma, \gamma')]\!]^e$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \lambda x : B.\ [e'/y]e : B \to B']\!]^e\ \theta\ (\gamma, \gamma') & \text{Subst.} \\
= & \lambda v.\ [\![\Theta;\ \Gamma, \Gamma', x : B \vdash [e'/y]e : B']\!]^e\ \theta\ (\gamma, \gamma', v) & \text{Semantics} \\
= & \lambda v.\ [\![\Theta;\ \Gamma, y : A, \Gamma', x : B \vdash e : B']\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma', v) & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \lambda x : B.\ e : B \to B']\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics}
\end{array}
$$

  Note that we need to use the Barendregt convention, implicitly renaming x to ensure the avoidance of captures. We will not mention this point further in remaining proofs.

- Case EApp: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](e_1\ e_2) : B]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_1\ [e'/y]e_2 : B]\!]^e\ \theta\ (\gamma, \gamma') & \text{Subst.} \\
= & [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_1 : B_2 \to B]\!]^e\ \theta\ (\gamma, \gamma')\ \ [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_2 : B_2]\!]^e\ \theta\ (\gamma, \gamma') & \text{Semantics} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_1 : B_2 \to B]\!]^e\ \theta\ \gamma''\ \ [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_2 : B_2]\!]^e\ \theta\ \gamma'' & \text{IH, IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_1\ e_2 : B]\!]^e\ \theta\ \gamma'' & \text{Semantics}
\end{array}
$$

  (Where $\gamma'' = (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$)

- case EUnit: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\ \langle\rangle : \mathbf{1}]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \langle\rangle : \mathbf{1}]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & * & \text{Semantics} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \langle\rangle : \mathbf{1}]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics}
\end{array}
$$

- case EPair: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\ \langle e_1, e_2\rangle : B]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \langle [e'/y]e_1, [e'/y]e_2 \rangle : B_1 \times B_2]\!]^e\ \theta\ (\gamma, \gamma') & \text{Subst.} \\
= & ([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_1 : B_1]\!]^e\ \theta\ (\gamma, \gamma'), [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_2 : B_2]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
= & ([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_1 : B_1]\!]^e\ \theta\ \gamma'', [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_2 : B_2]\!]^e\ \theta\ \gamma'') & \text{IH, IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \langle e_1, e_2 \rangle : B]\!]^e\ \theta\ \gamma'' & \text{Semantics}
\end{array}
$$

(Where $\gamma'' = (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$)

- Case EFst: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{fst}\ e : B_1]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{fst}\ [e'/y]e : B_1]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & \pi_1([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B_1 \times B_2]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
= & \pi_1([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B_1 \times B_2]\!]^e\ \theta\ \gamma'') & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{fst}\ e : B_1 \times B_2]\!]^e\ \theta\ \gamma'' & \text{Semantics}
\end{array}
$$

(Where $\gamma'' = (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$)

- Case ESnd: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{snd}\ e : B_1]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{snd}\ [e'/y]e : B_2]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & \pi_2([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B_1 \times B_2]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
= & \pi_2([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B_1 \times B_2]\!]^e\ \theta\ \gamma'') & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{snd}\ e : B_1 \times B_2]\!]^e\ \theta\ \gamma'' & \text{Semantics}
\end{array}
$$

(Where $\gamma'' = (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$)

- Case EInl: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{inl}e : B_1 + B_2]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{inl}([e'/y]e) : B_1 + B_2]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & \iota_1([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B_1]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
= & \iota_1([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B_1]\!]^e\ \theta\ \gamma'') & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{inl}e : B_1 + B_2]\!]^e\ \theta\ \gamma'' & \text{Semantics}
\end{array}
$$

(Where $\gamma'' = (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$)

- Case EInr: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{inl}e : B_1 + B_2]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{inr}([e'/y]e) : B_1 + B_2]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & \iota_2([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B_2]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
= & \iota_2([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B_2]\!]^e\ \theta\ \gamma'') & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{inr}e : B_1 + B_2]\!]^e\ \theta\ \gamma'' & \text{Semantics}
\end{array}
$$

(Where $\gamma'' = (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$)

- Case ECase: We have $\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{case}(e,\ x_1.\ e_1,\ x_2.\ e_2) : C$

$$
\begin{array}{lcll}
 & = & \Theta;\ \Gamma, \Gamma' \vdash \mathsf{case}([e'/y]e,\ x_1.\ [e'/y]e_1,\ x_2.\ [e'/y]e_2) : C & \text{Substitution} \\
 & & & \\
(1) & & \Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B_1 + B_2 & \text{Inversion} \\
(2) & & \Theta;\ \Gamma, \Gamma', x_1 : B_1 \vdash [e'/y]e_1 : B_1 & \text{Inversion} \\
(3) & & \Theta;\ \Gamma, \Gamma', x_2 : B_2 \vdash [e'/y]e_2 : B_2 & \text{Inversion} \\
[\![(1)]\!]^e & = & \lambda\theta\ (\gamma, \gamma').\ [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B_1 + B_2]\!]^e\ \theta\ (\gamma, [\![e']\!]^e \theta\gamma, \gamma') & \text{IH} \\
[\![(2)]\!]^e & = & \lambda\theta\ (\gamma, \gamma', v).\ [\![\Theta;\ \Gamma, y : A, \Gamma', x_1 : B_1 \vdash e_1 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e \theta\gamma, \gamma', v) & \text{IH} \\
[\![(3)]\!]^e & = & \lambda\theta\ (\gamma, \gamma', v).\ [\![\Theta;\ \Gamma, y : A, \Gamma', x_2 : B_2 \vdash e_2 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e \theta\gamma, \gamma', v) & \text{IH}
\end{array}
$$

Now, the interpretation $[\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{case}([e'/y]e,\ x_1.\ [e'/y]e_1,\ x_2.\ [e'/y]e_2) : C]\!]^e\ \theta\ (\gamma, \gamma')$ will be equal to $[f_1, f_2]\ a$, where

$$
\begin{array}{rcl}
a & = & [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B_1 + B_2]\!]^e\ \theta\ (\gamma, \gamma') \\
 & = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B_1 + B_2]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') \\
f_1 & = & \lambda v.\ [\![\Theta;\ \Gamma, \Gamma', x_1 : B_1 \vdash e_1 : C]\!]^e\ \theta\ (\gamma, \gamma', v) \\
 & = & \lambda v.\ [\![\Theta;\ \Gamma, y : A, \Gamma', x_1 : B_1 \vdash e_1 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma', v) \\
f_2 & = & \lambda v.\ [\![\Theta;\ \Gamma, \Gamma', x_2 : B_2 \vdash e_2 : C]\!]^e\ \theta\ (\gamma, \gamma', v) \\
 & = & \lambda v.\ [\![\Theta;\ \Gamma, y : A, \Gamma', x_2 : B_2 \vdash e_1 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma', v)
\end{array}
$$

Which means that $[f_1, f_2]\ a = [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{case}(e,\ x_1.\ e_1,\ x_2.\ y_2) : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')$

- case EZero: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{z} : \mathbb{N}]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lcll}
 & = & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{z} : \mathbb{N}]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
 & = & 0 & \text{Semantics} \\
 & = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{z} : \mathbb{N}]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics}
\end{array}
$$

- case ESucc: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{s}(e) : \mathbb{N}]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lcll}
 & = & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{s}([e'/y]e) : \mathbb{N}]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
 & = & s([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : \mathbb{N}]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
 & = & s([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : \mathbb{N}]\!]^e\ \theta\ \gamma'') & \text{IH} \\
 & = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{s}(e) : \mathbb{N}]\!]^e\ \theta\ \gamma'' & \text{Semantics}
\end{array}
$$

(Where $\gamma'' = (\gamma, [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma, \gamma')$)

- case EIter: We have $\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{iter}(e, e_0, x.\ e_1) : C$

$$= \quad \Theta;\ \Gamma, \Gamma' \vdash \mathsf{iter}([e'/y]e, [e'/y]e_0, x.\ [e'/y]e_1) : C \qquad \text{Substitution}$$

$$\begin{array}{lll} (1) & \Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : \mathbb{N} & \text{Inversion} \\ (2) & \Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_0 : C & \text{Inversion} \\ (3) & \Theta;\ \Gamma, \Gamma', x : C \vdash [e'/y]e_1 : C & \text{Inversion} \end{array}$$

$$\begin{array}{lcll} [\![(1)]\!]^e & = & \lambda\theta\ (\gamma, \gamma').\ [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : \mathbb{N}]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{IH} \\ [\![(2)]\!]^e & = & \lambda\theta\ (\gamma, \gamma').\ [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_0 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{IH} \\ [\![(3)]\!]^e & = & \lambda\theta\ (\gamma, \gamma'').\ [\![\Theta;\ \Gamma, y : A, \Gamma', x : C \vdash e_0 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma'') & \text{IH} \end{array}$$

Now, we assume suitable θ, γ, γ', and consider the interpretation of

$$[\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{iter}([e'/y]e, [e'/y]e_0, x.\ [e'/y]e_1) : C]\!]^e\ \theta\ (\gamma, \gamma')$$

This is equal to $iter_C[i, s]\ a$, where:

$$\begin{array}{rcl} a & = & [\![(1)]\!]^e\ \theta\ (\gamma, \gamma') \\ & = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : \mathbb{N}]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') \\ \\ i & = & [\![(2)]\!]^e\ \theta\ (\gamma, \gamma') \\ & = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_0 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') \\ s & = & \lambda v.\ [\![(3)]\!]^e\ \theta\ (\gamma, \gamma', v) \\ & = & \lambda v.\ [\![\Theta;\ \Gamma, y : A, \Gamma', x : C \vdash e_1 : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma', v) \end{array}$$

Which means that

$$iter_C[i, s]\ a = [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{iter}(e, [e'/y]e_0, x.\ [e'/y]e_1) : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')$$

- case EMonad: We have $\Theta;\ \Gamma, \Gamma' \vdash [e'/y][c] : \bigcirc B$.

$$= \quad \Theta;\ \Gamma, \Gamma' \vdash [[e'/y]c] : \bigcirc B \qquad \text{Substitution}$$

$$\text{We have} \qquad \Theta;\ \Gamma, \Gamma' \vdash [e'/y]c \div B \qquad \text{Inversion}$$

By mutual induction, we know that for all suitable θ, γ, γ',

$$[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]c \div B]\!]^c\ \theta\ (\gamma, \gamma') = [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash c : B]\!]^c\ \theta\ \ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')$$

Therefore we know that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y][c] : \bigcirc B]\!]^e\ \theta\ (\gamma, \gamma')$ is

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]c \div B]\!]^c\ \theta\ (\gamma, \gamma) & \text{Semantics} \\ = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash c : B]\!]^c\ \theta\ \ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{See above} \\ = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash [c] : \bigcirc B]\!]^e\ \ \theta\ \ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics} \end{array}$$

- case EFix: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](\mathsf{fix}\ x : D.\ e) : D]\!]^e\ \theta\ (\gamma, \gamma')$ is

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{fix}\ x : D.\ ([e'/y]e) : D]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & fix(\lambda v.\ ([\![\Theta;\ \Gamma, \Gamma', x : D \vdash [e'/y]e : D]\!]^e\ \theta\ (\gamma, \gamma', v))) & \text{Semantics} \\
= & fix(\lambda v.\ ([\![\Theta;\ \Gamma, y : A, \Gamma', x : D \vdash e : D]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma', v))) & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{fix}\ x : D.\ e : D]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma', v) & \text{Semantics}
\end{array}
$$

- case ETLam: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](\Lambda\alpha : \kappa.\ e) : \forall\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, \gamma')$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \Lambda\alpha : \kappa.\ [e'/y]e : \forall\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & \lambda\tau.\ [\![\ \Theta, \alpha : \kappa;\ \Gamma, \Gamma' \vdash [e'/y]e : B]\!]^e\ (\theta, \tau)\ (\gamma, \gamma') & \text{Semantics} \\
= & \lambda\tau.\ [\![\ \Theta, \alpha : \kappa;\ \Gamma, \Gamma' \vdash e : B]\!]^e\ (\theta, \tau)\ (\gamma, [\![e']\!]^e\ (\theta, \tau)\ \gamma, \gamma') & \text{IH} \\
= & \lambda\tau.\ [\![\ \Theta, \alpha : \kappa;\ \Gamma, \Gamma' \vdash e : B]\!]^e\ (\theta, \tau)\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Since } \alpha \notin FTV(e') \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \Lambda\alpha : \kappa.\ e : \forall\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics}
\end{array}
$$

- case ETApp: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](e\ \tau) : [\tau/\alpha]B]\!]^e\ \theta\ (\gamma, \gamma')$ is

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash ([e'/y]e)\ \tau : [\tau/\alpha]B]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & ([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : \forall\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, \gamma'))\ \tau(\theta) & \text{Semantics} \\
= & ([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : \forall\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma'))\ \tau(\theta) & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e\ \tau : [\tau/\alpha]B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Substitution}
\end{array}
$$

- case EPack: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{pack}(\tau, e) : \exists\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, \gamma')$ is

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{pack}(\tau, e) : \exists\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & (\tau(\theta), [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : [\tau/\alpha]B]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
= & (\tau(\theta), [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : [\tau/\alpha]B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')) & \text{IH} \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{pack}(\tau, e) : \exists\alpha : \kappa.\ B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics}
\end{array}
$$

- case EUnpack: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](\mathsf{unpack}(\alpha, x) = e_1\ \mathsf{in}\ e_2) : B]\!]^e\ \theta\ (\gamma, \gamma')$ as

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{unpack}(\alpha, x) = [e'/y]e_1\ \mathsf{in}\ [e'/y]e_2 : B]\!]^e\ \theta\ (\gamma, \gamma') & \text{Substitution} \\
= & (\lambda\,\langle\tau, v\rangle\,.\ ([\![\Theta, \alpha : \kappa;\ \Gamma, \Gamma', x : C \vdash [e'/y]e_2 : B]\!]^e\ (\theta, \tau)\ (\gamma, \gamma', v))) & \\
 & \quad ([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_1 : \exists\alpha : \kappa.\ C]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\
= & (\lambda\,\langle\tau, v\rangle\,.\ ([\![\Theta, \alpha : \kappa;\ \Gamma, y : A, \Gamma', x : C \vdash e_2 : B]\!]^e\ (\theta, \tau)\ (\gamma, [\![e']\!]^e\ (\theta, \tau)\ \gamma, \gamma', v))) & \\
 & \quad ([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_1 : \exists\alpha : \kappa.\ C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')) & \text{IH} \\
= & (\lambda\,\langle\tau, v\rangle\,.\ ([\![\Theta, \alpha : \kappa;\ \Gamma, y : A, \Gamma', x : C \vdash e_2 : B]\!]^e\ (\theta, \tau)\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma', v))) & \\
 & \quad ([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_1 : \exists\alpha : \kappa.\ C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')) & \alpha \notin FTV(e') \\
= & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{unpack}(\alpha, x) = e_1\ \mathsf{in}\ e_2 : B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics}
\end{array}
$$

- case EKeq: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B]\!]^e\ \theta\ (\gamma, \gamma')$

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : C]\!]^e\ \theta\ (\gamma, \gamma') & \text{Semantics, } B = C \\ = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : C]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{IH} \\ = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics} \end{array}$$

Now, the cases for the computation terms follow.

- case CReturn: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e \div B]\!]^c\ \theta\ (\gamma, \gamma')$

$$\begin{array}{lll} = & \eta_{[\![B]\!]\theta}([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : B]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\ = & \eta_{[\![B]\!]\theta}([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')) & \text{Mutual IH} \\ = & \eta_{[\![B]\!]\theta}([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')) & \text{Semantics} \end{array}$$

- case CLet: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](\mathsf{letv}\ x = e\ \mathsf{in}\ c) \div C]\!]^c\ \theta\ (\gamma, \gamma')$

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{letv}\ x = [e'/y]e\ \mathsf{in}\ [e'/y]c \div C]\!]^c\ \theta\ (\gamma, \gamma') & \text{Substitution} \\ = & (\lambda v.\ [\![\Theta;\ \Gamma, \Gamma', x : B \vdash [e'/y]c \div C]\!]^c\ \theta\ (\gamma, \gamma', v))^* & \\ & ([\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : \bigcirc B]\!]^e\ \theta\ (\gamma, \gamma')) & \text{Semantics} \\ = & (\lambda v.\ [\![\Theta;\ \Gamma, y : A, \Gamma', x : B \vdash c \div C]\!]^c\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma', v))^* & \\ & ([\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : \bigcirc B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma')) & \text{IH,IH} \\ = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{letv}\ x = e\ \mathsf{in}\ c \div C]\!]^c\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{Semantics} \end{array}$$

- case CGet: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](!e) \div B]\!]^c\ \theta\ (\gamma, \gamma')$

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, \Gamma' \vdash\ !([e'/y]e) \div B]\!]^c\ \theta\ (\gamma, \gamma') & \text{Substitution} \\ = & \lambda k.\ \lambda(L, h).\ \begin{cases} k\ (h\ l)\ (L, h) & \text{when } l \in L \\ \top & \text{otherwise} \end{cases} & \text{Semantics} \\ & \text{where } l = [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : \mathsf{ref}\ B]\!]^e\ \theta\ (\gamma, \gamma') & \end{array}$$

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash\ !e \div B]\!]^c\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \\ & \text{because } l = [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e : \mathsf{ref}\ B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{IH} \end{array}$$

- case CSet: We have $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y](e_1 := e_2) \div \mathbf{1}]\!]^c\ \theta\ (\gamma, \gamma')$

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e_1 := [e'/y]e_2 \div \mathbf{1}]\!]^c\ \theta\ (\gamma, \gamma') & \text{Substitution} \\ = & \lambda k.\ \lambda(L, h).\ \begin{cases} k\ \langle\rangle\ (L, [h|l : v]) & \text{when } l \in L \\ \top & \text{otherwise} \end{cases} & \text{Semantics} \\ & \text{where } l = [\![\Theta;\ \Gamma, \Gamma; \vdash [e'/y]e_1 : \mathsf{ref}\ B]\!]^e\ \theta\ (\gamma, \gamma') & \\ & \text{where } v = [\![\Theta;\ \Gamma, \Gamma; \vdash [e'/y]e_2 : B]\!]^e\ \theta\ (\gamma, \gamma') & \end{array}$$

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash e_1 := e_2 \div \mathbf{1}]\!]^c\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \\ & \text{because } l = [\![\Theta;\ \Gamma, y : A, \Gamma \vdash e_1 : \mathsf{ref}\ B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{IH} \\ & \text{because } v = [\![\Theta;\ \Gamma, y : A, \Gamma \vdash e_2 : B]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{IH} \end{array}$$

- case CNew: We have that $[\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]\mathsf{new}_B(e) \div \mathsf{ref}\ B]\!]^c\ \theta\ (\gamma, \gamma')$.

$$\begin{array}{lll} = & [\![\Theta;\ \Gamma, \Gamma' \vdash \mathsf{new}_B(([e'/y]e)) \div \mathsf{ref}\ B]\!]^c\ \theta\ (\gamma, \gamma') & \text{Substitution} \\ = & \lambda k.\ \lambda(L, h).\ \left(\begin{array}{l}\text{let } l = newloc(L, A) \text{ in} \\ k\ l\ (L \cup \{l\}, [h|l : v])\end{array}\right) & \\ & \text{where } v = [\![\Theta;\ \Gamma, \Gamma' \vdash [e'/y]e : A]\!]^e\ \theta\ (\gamma, \gamma') & \text{Semantics} \\ = & [\![\Theta;\ \Gamma, y : A, \Gamma' \vdash \mathsf{new}_B(e) \div \mathsf{ref}\ B]\!]^c\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \\ & \text{because } v = [\![\Theta;\ \Gamma, \Gamma' \vdash e : A]\!]^e\ \theta\ (\gamma, [\![e']\!]^e\ \theta\ \gamma, \gamma') & \text{IH} \end{array}$$

□

Lemma. *(11, page 36: Soundness of Equality Rules) We have that:*

1. *If* $\Theta;\ \Gamma \vdash e \equiv e' : A$*, then* $\Theta;\ \Gamma \vdash e : A$ *and* $\Theta;\ \Gamma \vdash e' : A$ *and* $[\![\Theta;\ \Gamma \vdash e : A]\!]^e = [\![\Theta;\ \Gamma \vdash e' : A]\!]^e$.
2. *If* $\Theta; \Gamma \vdash c \equiv c' \div A$*, then* $\Theta;\ \Gamma \vdash c \div A$ *and* $\Theta;\ \Gamma \vdash c' \div A$ *and* $[\![\Theta;\ \Gamma \vdash c : A]\!]^c = [\![\Theta;\ \Gamma \vdash c' \div A]\!]^c$.

Proof. The proof of this theorem is by induction on the derivations of $\Theta;\ \Gamma \vdash e \equiv e' : A$ and $\Theta; \Gamma \vdash c \equiv c' \div A$. So, assuming we have suitable θ and γ, we proceed as follows:

- case EqUnit: From $\Theta;\ \Gamma \vdash e \equiv e' : \mathbf{1}$, we have

$$\begin{array}{ll} \Theta;\ \Gamma \vdash e : \mathbf{1} & \text{By inversion} \\ \Theta;\ \Gamma \vdash e' : \mathbf{1} & \text{By inversion} \end{array}$$

$$\begin{array}{lll} [\![\Theta;\ \Gamma \vdash e : \mathbf{1}]\!]^e\ \theta\ \gamma & = & (*) \qquad\qquad\qquad\qquad\quad \text{Semantics} \\ & = & [\![\Theta;\ \Gamma \vdash e' : \mathbf{1}]\!]^e\ \theta\ \gamma \quad \text{Semantics} \end{array}$$

- case EqPairFst: From $\Theta;\ \Gamma \vdash \mathsf{fst}\ \langle e_1, e_2\rangle \equiv e_1 : A_1$, we have:

$$\begin{array}{lll} 1 & \Theta;\ \Gamma \vdash \mathsf{fst}\ \langle e_1, e_2\rangle \equiv e_2 : A_1 & \text{Hypothesis} \\ 2 & \Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A_1 \times A_2 & \text{By inversion on 1} \\ 3 & \Theta;\ \Gamma \vdash e_1 : A_1 & \text{By inversion on 2} \\ 4 & \Theta;\ \Gamma \vdash \mathsf{fst}\ \langle e_1, e_2\rangle : A_1 & \text{By rule ESnd on 2} \end{array}$$

$[\![\Theta;\ \Gamma \vdash \mathsf{fst}\ \langle e_1, e_2\rangle : A_1]\!]^e\ \theta\ \gamma =$

$$\begin{array}{lll} = & \pi_1([\![\Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A_1 \times A_2]\!]^e\ \theta\ \gamma) & \text{Semantics} \\ = & \pi_1(([\![\Theta;\ \Gamma \vdash e_1 : A_1]\!]^e\ \theta\ \gamma, [\![\Theta;\ \Gamma \vdash e_1 : A_2]\!]^e\ \theta\ \gamma)) & \text{Semantics} \\ = & [\![\Theta;\ \Gamma \vdash e_1 : A_1]\!]^e\ \theta\ \gamma & \text{Products} \end{array}$$

- case EqPairSnd: From $\Theta;\ \Gamma \vdash \mathsf{snd}\ \langle e_1, e_2\rangle \equiv e_1 : A_2$, we have:

$$\begin{array}{lll} 1 & \Theta;\ \Gamma \vdash \mathsf{snd}\ \langle e_1, e_2\rangle \equiv e_1 : A_2 & \text{Hypothesis} \\ 2 & \Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A_1 \times A_2 & \text{By inversion on 1} \\ 3 & \Theta;\ \Gamma \vdash e_2 : A_2 & \text{By inversion on 2} \\ 4 & \Theta;\ \Gamma \vdash \mathsf{snd}\ \langle e_1, e_2\rangle : A_2 & \text{By rule EFst on 2} \end{array}$$

$[\![\Theta;\ \Gamma \vdash \mathsf{snd}\ \langle e_1, e_2\rangle : A_2]\!]^e\ \theta\ \gamma =$

$$\begin{array}{lll} = & \pi_2([\![\Theta;\ \Gamma \vdash \langle e_1, e_2\rangle : A_1 \times A_2]\!]^e\ \theta\ \gamma) & \text{Semantics} \\ = & \pi_2(([\![\Theta;\ \Gamma \vdash e_1 : A_1]\!]^e\ \theta\ \gamma, [\![\Theta;\ \Gamma \vdash e_1 : A_2]\!]^e\ \theta\ \gamma)) & \text{Semantics} \\ = & [\![\Theta;\ \Gamma \vdash e_2 : A_2]\!]^e\ \theta\ \gamma & \text{Products} \end{array}$$

- case EqPairEta:

1	$\Theta;\ \Gamma \vdash e \equiv \langle \mathsf{fst}\ e, \mathsf{snd}\ e\rangle : A_1 \times A_2$	Hypothesis
2	$\Theta;\ \Gamma \vdash e : A_1 \times A_2$	Inversion
3	$\Theta;\ \Gamma \vdash \mathsf{fst}\ e : A_1$	Rule EFst on 2
4	$\Theta;\ \Gamma \vdash \mathsf{snd}\ e : A_2$	Rule ESnd on 2
5	$\Theta;\ \Gamma \vdash \langle \mathsf{fst}\ e, \mathsf{snd}\ e\rangle : A_1 \times A_2$	Rule EPair on 3, 4

$[\![\Theta;\ \Gamma \vdash e : A_1 \times A_2]\!]^e\ \theta\ \gamma =$

$$\begin{array}{lll} = & (\pi_1([\![\Theta;\ \Gamma \vdash e : A_1 \times A_2]\!]^e\ \theta\ \gamma), \pi_2([\![\Theta;\ \Gamma \vdash e : A_1 \times A_2]\!]^e\ \theta\ \gamma)) & \text{Products} \\ = & ([\![\Theta;\ \Gamma \vdash \mathsf{fst}\ e : A_1]\!]^e\ \theta\ \gamma, [\![\Theta;\ \Gamma \vdash \mathsf{snd}\ e : A_2]\!]^e\ \theta\ \gamma) & \text{Semantics} \times 2 \\ = & [\![\Theta;\ \Gamma \vdash \langle \mathsf{fst}\ e, \mathsf{snd}\ e\rangle : A_1 \times A_2]\!]^e\ \theta\ \gamma & \text{Semantics} \end{array}$$

- case EqFunBeta:

1	$\Theta;\ \Gamma \vdash (\lambda x : A.\ e)\ e' \equiv [e'/x]e : B$	Hypothesis
2	$\Theta;\ \Gamma \vdash (\lambda x : A.\ e)\ e' : B$	Inversion on 1
3	$\Theta;\ \Gamma \vdash e' : A$	Inversion on 2
4	$\Theta;\ \Gamma \vdash \lambda x : A.\ e : A \to B$	Inversion on 2
5	$\Theta;\ \Gamma, x : A \vdash e : B$	Inversion on 4
6	$\Theta;\ \Gamma \vdash [e'/x]e : B$	Substitution 3 into 5

$[\![\Theta;\ \Gamma \vdash (\lambda x : A.\ e)\ e' : B]\!]^e\ \theta\ \gamma =$

$$\begin{array}{lll} = & ([\![\Theta;\ \Gamma \vdash \lambda x : A.\ e : A \to B]\!]^e\ \theta\ \gamma)\ ([\![\Theta;\ \Gamma \vdash e' : B]\!]^e\ \theta\ \gamma) & \text{Semantics} \\ = & (\lambda v.\ ([\![\Theta;\ \Gamma, x : A \vdash e : B]\!]^e\ \theta\ (\gamma, v))\ [\![\Theta;\ \Gamma \vdash e' : B\ \theta\ \gamma]\!]^e & \text{Semantics} \\ = & [\![\Theta;\ \Gamma, x : A \vdash e : B]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e' : B]\!]^e\ \theta\ \gamma) & \text{Functions} \\ = & [\![\Theta;\ \Gamma \vdash [e'/x]e : B]\!]^e\ \theta\ \gamma & \text{Substitution} \end{array}$$

- case EqFunEta:

1	$\Theta;\ \Gamma \vdash e \equiv e' : A \to B$	Hypothesis
2	$\Theta;\ \Gamma, x : A \vdash e\ x \equiv e'\ x : B$	Inversion
3	$\Theta;\ \Gamma, x : A \vdash e\ x : B$	Induction
4	$\Theta;\ \Gamma, x : A \vdash e'\ : B$	Induction
5	$\Theta;\ \Gamma \vdash e : A \to B$	Inversion and $x \notin FV(e)$
6	$\Theta;\ \Gamma \vdash e' : A \to B$	Inversion and $x \notin FV(e')$

For arbitrary v,

$$\begin{array}{lcll}
[\![\Theta;\ \Gamma, x : A \vdash e'\ x : B]\!]^e\ \theta\ (\gamma, v) & = & [\![\Theta;\ \Gamma, x : A \vdash e\ x : B]\!]^e\ \theta\ (\gamma, v) & \text{Induction} \\
[\![\Theta;\ \Gamma, x : A \vdash e : A \to B]\!]^e\ \theta\ (\gamma, v)\ v & = & [\![\Theta;\ \Gamma \vdash e : A \to B]\!]^e\ \theta\ \gamma & \text{Semantics, } x \notin FV(e) \\
[\![\Theta;\ \Gamma, x : A \vdash e' : A \to B]\!]^e\ \theta\ (\gamma, v)\ v & = & [\![\Theta;\ \Gamma \vdash e' : A \to B]\!]^e\ \theta\ \gamma & \text{Semantics, } x \notin FV(e') \\
[\![\Theta;\ \Gamma \vdash e' : A \to B]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash e : A \to B]\!]^e\ \theta\ \gamma & \text{Transitivity}
\end{array}$$

- case EqSumInlBeta

1	$\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inl}\ e,\ x.\ e_1,\ y.\ e_2) \equiv [e/x]e_1 : C$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inl}\ e,\ x.\ e_1,\ y.\ e_2) : C$	Inversion on 1
3	$\Theta;\ \Gamma \vdash \mathsf{inl}\ e : A + B$	Inversion on 2
4	$\Theta;\ \Gamma, x : A \vdash e_1 : C$	Inversion on 2
5	$\Theta;\ \Gamma, y : B \vdash e_2 : C$	Inversion on 2
6	$\Theta;\ \Gamma \vdash e : A$	Inversion on 3
7	$\Theta;\ \Gamma \vdash [e/x]e_1 : C$	Substitute 6 into 4

$[\![\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inl}\ e,\ x.\ e_1,\ y.\ e_2) : C]\!]^e\ \theta\ \gamma =$

$$\begin{array}{lcll}
 & = & [f_1, f_2](a) & \text{Semantics} \\
 & & a = [\![\Theta;\ \Gamma \vdash \mathsf{inl}\ e : A + B]\!]^e\ \theta\ \gamma & \\
\text{where} & & \quad = \iota_1([\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma) & \\
 & & f_1 = \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : C]\!]^e\ \theta\ (\gamma, v) & \\
 & & f_2 = \lambda v.\ [\![\Theta;\ \Gamma, y : B \vdash e_2 : C]\!]^e\ \theta\ (\gamma, v) & \\
[f_1, f_2](a) & = & [f_1, f_2](\iota_1([\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma)) & \text{Sums} \\
 & = & f_1([\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma) & \\
 & = & [\![\Theta;\ \Gamma, x : A \vdash e_1 : C]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : A]\!]^e) & \text{Def of } f_1 \\
 & = & [\![\Theta;\ \Gamma \vdash [e/x]e_1 : C]\!]^e\ \theta\ \gamma & \text{Substitutition}
\end{array}$$

- case EqSumInrBeta:

1	$\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inr}\ e,\ x.\ e_1,\ y.\ e_2) \equiv [e/y]e_2 : C$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inr}\ e,\ x.\ e_1,\ y.\ e_2) : C$	Inversion on 1
3	$\Theta;\ \Gamma \vdash \mathsf{inr}\ e : A + B$	Inversion on 2
4	$\Theta;\ \Gamma, x : A \vdash e_1 : C$	Inversion on 2
5	$\Theta;\ \Gamma, y : B \vdash e_2 : C$	Inversion on 2
6	$\Theta;\ \Gamma \vdash e : B$	Inversion on 3
7	$\Theta;\ \Gamma \vdash [e/y]e_2 : C$	Substitute 6 into 5

$[\![\Theta;\ \Gamma \vdash \mathsf{case}(\mathsf{inr}\ e,\ x.\ e_1,\ y.\ e_2) : C]\!]^e\ \theta\ \gamma =$

$$
\begin{array}{lcll}
 & = & [f_1, f_2](a) & \text{Semantics} \\
 & & \quad a = [\![\Theta;\ \Gamma \vdash \mathsf{inr}\ e : A + B]\!]^e\ \theta\ \gamma & \\
\text{where} & & \quad\ = \iota_2([\![\Theta;\ \Gamma \vdash e : B]\!]^e\ \theta\ \gamma) & \\
 & & \quad f_1 = \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : C]\!]^e\ \theta\ (\gamma, v) & \\
 & & \quad f_2 = \lambda v.\ [\![\Theta;\ \Gamma, y : B \vdash e_2 : C]\!]^e\ \theta\ (\gamma, v) & \\
[f_1, f_2](a) & = & [f_1, f_2](\iota_2([\![\Theta;\ \Gamma \vdash e : B]\!]^e\ \theta\ \gamma)) & \text{Sums} \\
 & = & f_2([\![\Theta;\ \Gamma \vdash e : B]\!]^e\ \theta\ \gamma) & \\
 & = & [\![\Theta;\ \Gamma, y : B \vdash e_2 : C]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : B]\!]^e) & \text{Def of } f_1 \\
 & = & [\![\Theta;\ \Gamma \vdash [e/y]e_2 : C]\!]^e\ \theta\ \gamma & \text{Substitution}
\end{array}
$$

- case EqSumEta:

1	$\Theta;\ \Gamma \vdash \mathsf{case}(e,\ x.\ [\mathsf{inl}\ x/z]e',\ y.\ [\mathsf{inr}\ y/z]e') \equiv [e/z]e' : C$	Hypothesis
2	$\Theta;\ \Gamma \vdash e : A + B$	Inversion on 1
3	$\Theta;\ \Gamma, z : A + B \vdash e' : C$	Inversion on 1
4	$\Theta;\ \Gamma, x : A, z : A + B \vdash e' : C$	Weakening on 3
5	$\Theta;\ \Gamma, x : A \vdash \mathsf{inl}\ x : A + B$	By rules
6	$\Theta;\ \Gamma, x : A \vdash [\mathsf{inl}\ x/z]e' : C$	Substitution of 5 into 4
7	$\Theta;\ \Gamma, y : B, z : A + B \vdash e' : C$	Weakening on 3
8	$\Theta;\ \Gamma, y : B \vdash \mathsf{inr}\ y : A + B$	By rules
9	$\Theta;\ \Gamma, y : B \vdash [\mathsf{inr}\ y/z]e' : C$	Substitution of 8 into 7
10	$\Theta;\ \Gamma \vdash \mathsf{case}(e,\ x.\ [\mathsf{inl}\ x/z]e',\ y.\ [\mathsf{inr}\ y/z]e') : C$	By ECase on 2, 6, 9
11	$\Theta;\ \Gamma \vdash [e/z]e' : C$	Substitution of 2 into 3

Now from the semantics, we know that $[\![\Theta;\ \Gamma \vdash e : A + B]\!]^e\ \theta\ \gamma$ is either equal to some $\iota_i(v_A)$ or some $\iota_2(v_B)$.

Suppose it is equal $\iota_1(v_A)$. Then, $[\![\Theta;\ \Gamma \vdash \mathsf{case}(e,\ x.\ [\mathsf{inl}\ x/z]e',\ y.\ [\mathsf{inr}\ y/z]e') : C]\!]^e\ \theta\ \gamma$ is equal to

$$
\begin{array}{lcll}
 & = & \left[\begin{array}{l} \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash [\mathsf{inl}\ x/z]e' : C]\!]^e\ \theta\ (\gamma, v), \\ \lambda v.\ [\![\Theta;\ \Gamma, y : B \vdash [\mathsf{inr}\ y/z]e' : C]\!]^e\ \theta\ (\gamma, v) \end{array}\right](\iota_1(v_A)) & \text{Semantics} \\
 & = & [\![\Theta;\ \Gamma, x : A \vdash [\mathsf{inl}\ x/z]e' : C]\!]^e\ \theta\ (\gamma, v_A) & \text{Sums} \\
 & = & \begin{array}{l} [\![\Theta;\ \Gamma, x : A, z : A + B \vdash e' : C]\!]^e\ \theta \\ \qquad (\gamma, v_A, [\![\Theta;\ \Gamma, x : A \vdash \mathsf{inl}\ x : A + B]\!]^e\ \theta\ (\gamma, v_A)) \end{array} & \text{Substitution} \\
 & = & [\![\Theta;\ \Gamma, x : A, z : A + B \vdash e' : C]\!]^e\ \theta\ (\gamma, v_A, \iota_1(v_A)) & \text{Semantics} \\
 & = & [\![\Theta;\ \Gamma, z : A + B \vdash e' : C]\!]^e\ \theta\ (\gamma, \iota_1(v_A)) & \text{Since } x \notin FV(e') \\
 & = & [\![\Theta;\ \Gamma, z : A + B \vdash e' : C]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : A + B]\!]^e\ \theta\ \gamma) & \text{Meaning of } \iota_1(v_A) \\
 & = & [\![\Theta;\ \Gamma \vdash [e/z]e' : C]\!]^e\ \theta\ \gamma & \text{Substitution}
\end{array}
$$

Suppose it is $\iota_2(v_B)$. Then, $[\![\Theta;\ \Gamma \vdash \mathsf{case}(e,\ x.\ [\mathsf{inl}\ x/z]e',\ y.\ [\mathsf{inr}\ y/z]e') : C]\!]^e\ \ \theta\ \ \gamma$ is

equal to

$$
\begin{array}{lll}
= & \left[\begin{array}{l} \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash [\mathsf{inl}\ x/z]e' : C]\!]^e\ \theta\ (\gamma, v), \\ \lambda v.\ [\![\Theta;\ \Gamma, y : B \vdash [\mathsf{inr}\ y/z]e' : C]\!]^e\ \theta\ (\gamma, v) \end{array}\right] (\iota_2(v_B)) & \text{Semantics} \\
= & [\![\Theta;\ \Gamma, y : B \vdash [\mathsf{inr}\ y/z]e' : C]\!]^e\ \theta\ (\gamma, v_B) & \text{Sums} \\
= & \begin{array}{l} [\![\Theta;\ \Gamma, y : B, z : A + B \vdash e' : C]\!]^e\ \theta \\ \qquad (\gamma, v_B, [\![\Theta;\ \Gamma, y : B \vdash \mathsf{inr}\ y : A + B]\!]^e\ \theta\ (\gamma, v_B)) \end{array} & \text{Substitution} \\
= & [\![\Theta;\ \Gamma, y : B, z : A + B \vdash e' : C]\!]^e\ \theta\ (\gamma, v_B, \iota_2(v_B)) & \text{Semantics} \\
= & [\![\Theta;\ \Gamma, z : A + B \vdash e' : C]\!]^e\ \theta\ (\gamma, \iota_2(v_B)) & \text{Since } x \notin FV(e') \\
= & [\![\Theta;\ \Gamma, z : A + B \vdash e' : C]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : A + B]\!]^e\ \theta\ \gamma) & \text{Meaning of } \iota_2(v_B) \\
= & [\![\Theta;\ \Gamma \vdash [e/z]e' : C]\!]^e\ \theta\ \gamma & \text{Substitution}
\end{array}
$$

- case EqMonad:

1	$\Theta;\ \Gamma \vdash [c] \equiv [c'] : \bigcirc A$	Hypothesis
2	$\Theta; \Gamma \vdash c \equiv c' \div A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash c \div A$	Mutual Induction on 2
4	$\Theta;\ \Gamma \vdash c' \div A$	Induction on 2
5	$\Theta;\ \Gamma \vdash [c] : \bigcirc A$	By rule EMonad on 3
6	$\Theta;\ \Gamma \vdash [c'] : \bigcirc A$	By rule EMonad on 4

$$
\begin{array}{lll}
[\![\Theta;\ \Gamma \vdash [c] : \bigcirc A]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma \quad \text{Semantics} \\
& = & [\![\Theta;\ \Gamma \vdash c' \div A]\!]^c\ \theta\ \gamma \quad \text{Mutual Induction} \\
& = & [\![\Theta;\ \Gamma \vdash [c'] : \bigcirc A]\!]^e\ \theta\ \gamma \quad \text{Semantics}
\end{array}
$$

- case EqFix

1	$\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e \equiv [(\mathsf{fix}\ x : D.\ e)/x]e : D$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e : D$	Inversion on 1
3	$\Theta;\ \Gamma, x : D \vdash e : D$	Inversion on 2
4	$\Theta;\ \Gamma \vdash [(\mathsf{fix}\ x : D.\ e)/x]e : D$	Substitution of 2 into 3

$[\![\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e : D]\!]^e\ \theta\ \gamma$

$$
\begin{array}{lll}
= & fix(\lambda v.\ ([\![\Theta;\ \Gamma, x : D \vdash e : D]\!]^e\ \theta\ (\gamma, v))) & \text{Semantics} \\
= & [\![\Theta;\ \Gamma, x : D \vdash e : D]\!]^e\ \theta\ (\gamma, fix(\lambda v.\ ([\![\Theta;\ \Gamma, x : D \vdash e : D]\!]^e\ \theta\ (\gamma, v)))) & \text{Unroll } fix \\
= & [\![\Theta;\ \Gamma, x : D \vdash e : D]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash \mathsf{fix}\ x : D.\ e : D]\!]^e\ \theta\ \gamma) & \text{Definition} \\
= & [\![\Theta;\ \Gamma \vdash [(\mathsf{fix}\ x : D.\ e)/x]e : D]\!]^e\ \theta\ \gamma & \text{Substitution}
\end{array}
$$

- case EqNatZBeta:

1	$\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{z}, e_0, x.\ e_1) \equiv e_0 : A$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{z}, e_0, x.\ e_1) : A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash \mathsf{z} : \mathbb{N}$	Inversion on 2
4	$\Theta;\ \Gamma, x : A \vdash e_1 : A$	Inversion on 2
5	$\Theta;\ \Gamma \vdash e_0 : A$	Inversion on 2

$[\![\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{z}, e_0, x.\ e_1) : A]\!]^e\ \theta\ \gamma =$

$$
\begin{array}{lll}
= & \begin{array}{l} iter[[\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ \gamma, \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, v)] \\ ([\![\Theta;\ \Gamma \vdash \mathsf{z} : \mathbb{N}]\!]^e\ \theta\ \gamma) \end{array} & \text{Semantics} \\
= & iter[[\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ \gamma, \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, v)](z) & \text{Semantics} \\
= & [\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ \gamma & \text{Iter properties}
\end{array}
$$

- EqNatSBeta

1	$\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{s}(e), e_0, x.\ e_1) \equiv [\mathsf{iter}(e, e_0, x.\ e_1)/x]e_1 : A$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{s}(e), e_0, x.\ e_1) : A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash \mathsf{s}(e) : \mathbb{N}$	Inversion on 2
4	$\Theta;\ \Gamma \vdash e_0 : A$	Inversion on 2
5	$\Theta;\ \Gamma, x : A \vdash e_1 : A$	Inversion on 2
6	$\Theta;\ \Gamma \vdash e : \mathbb{N}$	Inversion on 3
7	$\Theta;\ \Gamma \vdash \mathsf{iter}(e, e_0, x.\ e_1) : A$	Rule EIter on 6, 4, 5

$[\![\Theta;\ \Gamma \vdash \mathsf{iter}(\mathsf{s}(e), e_0, x.\ e_1) : A]\!]^e\ \theta\ \gamma =$

$$
\begin{array}{lll}
= & \begin{array}{l} iter[[\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ \gamma, \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, v)] \\ ([\![\Theta;\ \Gamma \vdash \mathsf{s}(e) : \mathbb{N}]\!]^e\ \theta\ \gamma) \end{array} & \text{Semantics} \\
= & \begin{array}{l} iter[[\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ \gamma, \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, v)] \\ (s([\![\Theta;\ \Gamma \vdash e : \mathbb{N}]\!]^e\ \theta\ \gamma)) \end{array} & \text{Semantics} \\
= & \left(\begin{array}{l} [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta \\ \gamma,\ \begin{array}{l} iter[[\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ \gamma, \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, v)] \\ ([\![\Theta;\ \Gamma \vdash e : \mathbb{N}]\!]^e\ \theta\ \gamma) \end{array} \end{array}\right) & \text{Iter} \\
= & [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta([\![\Theta;\ \Gamma \vdash \mathsf{iter}(e, e_0, x.\ e_1) : A]\!]^e\ \theta\ \gamma) & \text{Semantics} \\
= & [\![\Theta;\ \Gamma \vdash [\mathsf{iter}(e, e_0, x.\ e_1)/x]e_1 : A]\!]^e\ \theta\ \gamma & \text{Substitution}
\end{array}
$$

- case EqNatEta:

1	$\Theta;\ \Gamma, n : \mathbb{N} \vdash \mathsf{iter}(n, e_0, x.\ e_1) \equiv e : A$	Hypothesis
2	$\Theta;\ \Gamma \vdash e : A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash e_0 : A$	Inversion on 1
4	$\Theta;\ \Gamma, n : \mathbb{N} \vdash e_0 : A$	Weakening on 3
5	$\Theta;\ \Gamma, x : A \vdash e_1 : A$	Inversion on 1
6	$\Theta;\ \Gamma, n : \mathbb{N}, x : A \vdash e_1 : A$	Weakening on 5
7	$\Theta;\ \Gamma, n : \mathbb{N} \vdash n : \mathbb{N}$	Rule Hyp
8	$\Theta;\ \Gamma, n : \mathbb{N} \vdash \mathsf{iter}(n, e_0, x.\ e_1) : A$	Rule EIter on 7, 4, 6

Now, assume we have some suitable environment (γ, v). So v is a natural number, and we shall proceed by induction on it.

  - case $v = 0$.

$[\![\Theta;\ \Gamma, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, 0) =$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash \mathsf{z} : \mathbb{N}]\!]^e\ \theta\ \gamma) & \text{Semantics} \\
= & [\![\Theta;\ \Gamma \vdash [\mathsf{z}/n]e : A]\!]^e\ \theta\ \gamma & \text{Substitution} \\
= & [\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma & \text{Induction Hypothesis} \\
= & [\![\Theta;\ \Gamma, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, 0) & \text{Weakening}
\end{array}
$$

- case $v = s(k)$ By induction, we know
  $[\![\Theta;\ \Gamma, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, k) = [\![\Theta;\ \Gamma, n : \mathbb{N} \vdash \mathsf{iter}(n, e_0, x.\ e_1) : A]\!]^e\ \theta\ (\gamma, k)$

$[\![\Theta;\ \Gamma, n : \mathbb{N} \vdash \mathsf{iter}(n, e_0, x.\ e_1) : A]\!]^e\ \theta\ (\gamma, s(k)) =$

$$
\begin{array}{lll}
= & iter \left[\begin{array}{l} [\![\Theta;\ \Gamma, n : \mathbb{N} \vdash e_0 : A]\!]^e\ \theta\ (\gamma, s(k)), \\ \lambda v.\ [\![\Theta;\ \Gamma, n : \mathbb{N}, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, s(k), v) \end{array} \right] (s(k)) & \text{Semantics} \\
= & iter \left[\begin{array}{l} [\![\Theta;\ \Gamma \vdash e_0 : A]\!]^e\ \theta\ (\gamma), \\ \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, v) \end{array} \right] (s(k)) & \text{since } x \notin FV(e_0), FV(e_1) \\
= & iter \left[\begin{array}{l} [\![\Theta;\ \Gamma, m : \mathbb{N} \vdash e_0 : A]\!]^e\ \theta\ (\gamma, k), \\ \lambda v.\ [\![\Theta;\ \Gamma, m : \mathbb{N}, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, k, v) \end{array} \right] (s(k)) & \text{By weakening} \\
= & [\![\Theta;\ \Gamma, n : \mathbb{N}, x : A \vdash e_1 : A]\!]^e\ \theta\ (\gamma, iter[\ldots](k)) & \text{By } iter \\
= & \begin{array}{l} [\![\Theta;\ \Gamma, n : \mathbb{N}, x : A \vdash e_1 : A]\!]^e\ \theta \\ \quad (\gamma, k, [\![\Theta;\ \Gamma, n : \mathbb{N} \vdash \mathsf{iter}(n, e_0, x.\ e_1) : A]\!]^e\ \theta\ (\gamma, k)) \end{array} & \text{Semantics} \\
= & \begin{array}{l} [\![\Theta;\ \Gamma, n : \mathbb{N}, x : A \vdash e_1 : A]\!]^e\ \theta \\ \quad (\gamma, k, [\![\Theta;\ \Gamma, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, k)) \end{array} & \text{Inner Induction} \\
= & [\![\Theta;\ \Gamma, n : \mathbb{N} \vdash [e/x]e_1 : A]\!]^e\ \theta\ (\gamma, k) & \text{Substitution}
\end{array}
$$

$[\![\Theta;\ \Gamma, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, s(k)) =$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma, m : \mathbb{N}, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, k, s(k)) & \text{Weakening} \\
= & [\![\Theta;\ \Gamma, m : \mathbb{N}, n : \mathbb{N} \vdash e : A]\!]^e\ \theta\ (\gamma, k, [\![\Theta;\ \Gamma, m : \mathbb{N} \vdash \mathsf{s}(m) : \mathbb{N}]\!]^e\ \theta\ (\gamma, k)) & \text{Semantics} \\
= & [\![\Theta;\ \Gamma, m : \mathbb{N} \vdash [\mathsf{s}(m)/n]e : A]\!]^e\ \theta\ (\gamma, k) & \text{Substitution}
\end{array}
$$

These two are equal by appeal to the outer induction hypothesis, which we get via inversion on the original judgement.

- case EqAllBeta:

1	$\Theta;\ \Gamma \vdash (\Lambda\alpha : \kappa.\ e)\ \tau \equiv [\tau/\alpha]e : [\tau/\alpha]A$	Hypothesis
2	$\Theta;\ \Gamma \vdash \Lambda\alpha : \kappa.\ e : \forall\alpha : \kappa.\ A$	Inversion on 1
3	$\Theta \vdash \tau : \kappa$	Inversion on 1
4	$\Theta, \alpha : \kappa;\ \Gamma \vdash e : A$	Inversion on 2
5	$\Theta;\ \Gamma \vdash [\tau/\alpha]e : [\tau/\alpha]A$	Substitute 3 into 4
6	$\Theta;\ \Gamma \vdash (\Lambda\alpha : \kappa.\ e)\ \tau : [\tau/\alpha]A$	Rule ETApp on 2, 3

$[\![\Theta;\ \Gamma \vdash (\Lambda\alpha : \kappa.\ e)\ \tau : [\tau/\alpha]A]\!]^e\ \theta\ \gamma =$

$$
\begin{array}{lll}
= & [\![\Theta;\ \Gamma \vdash \Lambda\alpha : \kappa.\ e : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma\ \tau(\theta) & \text{Semantics} \\
= & (\lambda\sigma.\ [\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : A]\!]^e\ (\theta, \sigma)\ \gamma)\tau(\theta) & \text{Semantics} \\
= & [\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : A]\!]^e\ (\theta, \tau(\theta))\ \gamma & \\
= & [\![\Theta;\ \Gamma \vdash [\tau/\alpha]e : [\tau/\alpha]A]\!]^e\ \theta\ \gamma & \text{Type Substitution}
\end{array}
$$

- case EqAllEta:

1	$\Theta;\ \Gamma \vdash e \equiv e' : \forall\alpha : \kappa.\ A$	Hypothesis
2	$\Theta;\ \Gamma \vdash e : \forall\alpha : \kappa.\ A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash e' : \forall\alpha : \kappa.\ A$	Inversion on 1

$$
\begin{array}{lcll}
[\![\Theta, \alpha : \kappa;\ \Gamma \vdash e\ \alpha : A]\!]^e\ (\theta, \sigma)\ \gamma & = & [\![\Theta, \alpha : \kappa;\ \Gamma \vdash e'\ \alpha : A]\!]^e\ (\theta, \sigma)\ \gamma & \text{Induction} \\
[\![\Theta, \alpha : \kappa;\ \Gamma \vdash e\ \alpha : A]\!]^e\ (\theta, \sigma)\ \gamma & = & ([\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : \forall\alpha : \kappa.\ A]\!]^e\ (\theta, \sigma)\ \gamma)\ \sigma & \text{Semantics} \\
 & = & ([\![\Theta;\ \Gamma \vdash e : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma)\ \sigma & \text{Strengthening} \\
[\![\Theta, \alpha : \kappa;\ \Gamma \vdash e'\ \alpha : A]\!]^e\ (\theta, \sigma)\ \gamma & = & ([\![\Theta, \alpha : \kappa;\ \Gamma \vdash e' : \forall\alpha : \kappa.\ A]\!]^e\ (\theta, \sigma)\ \gamma)\ \sigma & \text{Semantics} \\
 & = & ([\![\Theta;\ \Gamma \vdash e' : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma)\ \sigma & \text{Strengthening} \\
([\![\Theta;\ \Gamma \vdash e : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma)\ \sigma & = & ([\![\Theta;\ \Gamma \vdash e' : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma)\ \sigma & \text{Transitivity} \\
[\![\Theta;\ \Gamma \vdash e : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash e' : \forall\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma & \text{Extensionality}
\end{array}
$$

- case EqExistsBeta:

1	$\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = \mathsf{pack}(\tau, e)\ \mathsf{in}\ e' \equiv [\tau/\alpha, e/x]e' : C$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{pack}(\tau, e) : \exists\alpha : \kappa.\ A$	Inversion on 1
3	$\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash e' : C$	Inversion on 1
4	$\Theta \vdash \tau : \kappa$	Inversion on 2
5	$\Theta;\ \Gamma \vdash e : [\tau/\alpha]A$	Inversion on 2
6	$\Theta;\ \Gamma \vdash [\tau/\alpha, e/x]e' : C$	Substitution of 4,5 into 3
7	$\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = \mathsf{pack}(\tau, e)\ \mathsf{in}\ e' : C$	By rule EUnpack on 2, 3

$[\![\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = \mathsf{pack}(\tau, e)\ \mathsf{in}\ e' : C]\!]^e\ \theta\ \gamma =$

$$
\begin{array}{lll}
= & (\lambda(\sigma, v).\ [\![\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash e' : C]\!]^e\ (\theta, \sigma)\ (\gamma, v))\ [\![\Theta;\ \Gamma \vdash \mathsf{pack}(\tau, e) : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma & \text{Semantics} \\
= & (\lambda(\sigma, v).\ [\![\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash e' : C]\!]^e\ (\theta, \sigma)\ (\gamma, v))\ (\tau(\theta), [\![\Theta;\ \Gamma \vdash e : [\tau/\alpha]A]\!]^e\ \theta\ \gamma) & \text{Semantics} \\
= & [\![\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash e' : C]\!]^e\ (\theta, \tau(\theta))\ (\gamma, [\![\Theta;\ \Gamma \vdash e : [\tau/\alpha]A]\!]^e\ \theta\ \gamma) & \text{Simplify} \\
= & [\![\Theta;\ \Gamma, x : A \vdash [\tau/\alpha, e/x]e' : C]\!]^e\theta\ \gamma & \text{Substitution}
\end{array}
$$

- case EqExistsEta:

1	$\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = e \ \mathsf{in}\ [\mathsf{pack}(\alpha, x)/z]e' \equiv [e/z]e' : B$	Hypothesis
2	$\Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A$	Inversion on 1
3	$\Theta;\ \Gamma, z : \exists\alpha : \kappa.\ A \vdash e' : B$	Inversion on 1
4	$\Theta;\ \Gamma \vdash [e/z]e' : B$	Substitution of 2 into 3
5	$\Theta, \alpha : \kappa;\ \Gamma, x : A, z : \exists\alpha : \kappa.\ A \vdash e' : B$	Weakening on 3
6	$\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash \mathsf{pack}(\alpha, x) : \exists\alpha : \kappa.\ A$	Rule EPack
7	$\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash [\mathsf{pack}(\alpha, x)/z]e' : B$	Substitution of 6 into 5
8	$\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = e \ \mathsf{in}\ [\mathsf{pack}(\alpha, x)/z]e' : B$	Rule EUnpack on 2, 7

$[\![\Theta;\ \Gamma \vdash \mathsf{unpack}(\alpha, x) = e \ \mathsf{in}\ [\mathsf{pack}(\alpha, x)/z]e' : B]\!]^e\ \theta\ \gamma =$

$=$	$(\lambda(\sigma, v).\ [\![\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash [\mathsf{pack}(\alpha, x)/z]e' : B]\!]^e\ (\theta, \sigma)\ (\gamma, v))$ $[\![\Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma$	Semantics
$=$	$(\lambda(\sigma, v).\ [\![\Theta, \alpha : \kappa;\ \Gamma, x : A, z : \exists\alpha : \kappa.\ A \vdash e' : B]\!]^e$ (θ, σ) $(\gamma, v, [\![\Theta, \alpha : \kappa;\ \Gamma, x : A \vdash \mathsf{pack}(\alpha, x) : \exists\alpha : \kappa.\ A]\!]^e\ (\theta, \sigma)\ (\gamma, v))$ $[\![\Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma$	Substitution
$=$	$(\lambda(\sigma, v).\ [\![\Theta, \alpha : \kappa;\ \Gamma, x : A, z : \exists\alpha : \kappa.\ A \vdash e' : B]\!]^e\ (\theta, \sigma)\ (\gamma, v, (\sigma, v)))$ $[\![\Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma$	Semantics
$=$	$(\lambda(\sigma, v).\ [\![\Theta;\ \Gamma, z : \exists\alpha : \kappa.\ A \vdash e' : B]\!]^e\ \theta\ (\gamma, (\sigma, v)))$ $[\![\Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma$	Strengthening
$=$	$[\![\Theta;\ \Gamma, z : \exists\alpha : \kappa.\ A \vdash e' : B]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : \exists\alpha : \kappa.\ A]\!]^e\ \theta\ \gamma)$	Semantics
$=$	$[\![\Theta;\ \Gamma \vdash [e/z]e' : B]\!]^e\ \theta\ \gamma$	Semantics

- case EqCommandEta:

1	$\Theta; \Gamma \vdash c \equiv \mathsf{letv}\ x = [c]\ \mathsf{in}\ x \div A$	Hypothesis
2	$\Theta;\ \Gamma \vdash c \div A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash [c] : \bigcirc A$	Rule EMonad on 2
4	$\Theta;\ \Gamma, x : A \vdash x : A$	Rule EHyp
5	$\Theta;\ \Gamma, x : A \vdash x \div A$	Rule CReturn on 4
6	$\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [c]\ \mathsf{in}\ x \div A$	By Rule CLet on 3,5

$[\![\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [c]\ \mathsf{in}\ x \div A]\!]^c\ \theta\ \gamma =$

$=$	$(\lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash x \div A]\!]^c\ \theta\ (\gamma, v))^*$ $[\![\Theta;\ \Gamma \vdash [c] : \bigcirc A]\!]^e\ \theta\ \gamma$	Semantics
$=$	$(\lambda v.\ \eta([\![\Theta;\ \Gamma, x : A \vdash x : A]\!]^e\ \theta\ (\gamma, v)))^*$ $[\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma$	Semantics
$=$	$(\lambda v.\ (\eta(v)))^*\ ([\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma)$	Simplify
$=$	$id([\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma)$	Monad law
$=$	$[\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma$	Simplify

- EqCommandBeta:

1	$\Theta; \Gamma \vdash \mathsf{letv}\ x = [e]\ \mathsf{in}\ c \equiv [e/x]c \div B$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [e]\ \mathsf{in}\ c \div B$	Inversion on 1
3	$\Theta;\ \Gamma \vdash [e] : \bigcirc A$	Inversion on 2
4	$\Theta;\ \Gamma, x : A \vdash c \div B$	Inversion on 2
5	$\Theta;\ \Gamma \vdash e \div A$	Inversion on 3
6	$\Theta;\ \Gamma \vdash e : A$	Inversion on 5
7	$\Theta;\ \Gamma \vdash [e/x]c \div B$	Substitution of 6 into 4

$[\![\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [e]\ \mathsf{in}\ c \div B]\!]^c\ \theta\ \gamma =$

$=$	$(\lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash c \div B]\!]^c\ \theta\ (\gamma, v))^*$ $([\![\Theta;\ \Gamma \vdash [e] : \bigcirc A]\!]^e\ \theta\ \gamma)$	Semantics
$=$	$(\lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash c \div B]\!]^c\ \theta\ (\gamma, v))^*$ $([\![\Theta;\ \Gamma \vdash e \div A]\!]^c\ \theta\ \gamma)$	Semantics
$=$	$(\lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash c \div B]\!]^c\ \theta\ (\gamma, v))^*$ $\eta([\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma)$	Semantics
$=$	$(\lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash c \div B]\!]^c\ \theta\ (\gamma, v))$ $([\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma)$	Monad laws
$=$	$[\![\Theta;\ \Gamma, x : A \vdash c \div B]\!]^c\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma)$	Simplify
$=$	$[\![\Theta;\ \Gamma \vdash [e/x]c \div B]\!]^c\ \theta\ \gamma$	Substitution

- EqCommandAssoc:

1	$\Theta; \Gamma \vdash \mathsf{letv}\ x = [\mathsf{letv}\ y = e\ \mathsf{in}\ c_1]\ \mathsf{in}\ c_2 \equiv \mathsf{letv}\ y = e\ \mathsf{in}\ \mathsf{letv}\ x = [c_1]\ \mathsf{in}\ c_2 \div C$	Hypothesis
2	$\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [\mathsf{letv}\ y = e\ \mathsf{in}\ c_1]\ \mathsf{in}\ c_2 \div C$	Inversion on 1
3	$\Theta;\ \Gamma \vdash [\mathsf{letv}\ y = e\ \mathsf{in}\ c_1] : \bigcirc B$	Inversion on 2
4	$\Theta;\ \Gamma, x : B \vdash c_2 \div C$	Inversion on 2
5	$\Theta;\ \Gamma \vdash \mathsf{letv}\ y = e\ \mathsf{in}\ c_1 \div B$	Inversion on 3
6	$\Theta;\ \Gamma \vdash e : \bigcirc A$	Inversion on 5
7	$\Theta;\ \Gamma, y : A \vdash c_1 \div B$	Inversion on 5
8	$\Theta;\ \Gamma, y : A, x : B \vdash c_2 \div C$	Weakening on 4
9	$\Theta;\ \Gamma, y : A \vdash [c_1] : \bigcirc B$	Rule EMonad on 7
10	$\Theta;\ \Gamma, y : A \vdash \mathsf{letv}\ x = [c_1]\ \mathsf{in}\ c_2 \div C$	Rule CLet on 9, 8
11	$\Theta;\ \Gamma \vdash \mathsf{letv}\ y = e\ \mathsf{in}\ \mathsf{letv}\ x = [c_1]\ \mathsf{in}\ c_2 \div C$	Rule CLet on 6, 10

$[\![\Theta;\ \Gamma \vdash \mathsf{letv}\ x = [\mathsf{letv}\ y = e\ \mathsf{in}\ c_1]\ \mathsf{in}\ c_2 \div C]\!]^c\ \theta\ \gamma =$

$$
\begin{array}{lll}
= & (\lambda v_2.\ [\![\Theta;\ \Gamma, x : B \vdash c_2 \div C]\!]^c\ \theta\ (\gamma, v_2))^* & \\
& [\![\Theta;\ \Gamma \vdash [\mathsf{letv}\ y = e\ \mathsf{in}\ c_1] : \bigcirc B]\!]^e\ \theta\ \gamma & \text{Semantics} \\
= & (\lambda v_2.\ [\![\Theta;\ \Gamma, x : B \vdash c_2 \div C]\!]^c\ \theta\ (\gamma, v_2))^* & \\
& [\![\Theta;\ \Gamma \vdash \mathsf{letv}\ y = e\ \mathsf{in}\ c_1 \div \bigcirc B]\!]^c\ \theta\ \gamma & \text{Semantics} \\
= & (\lambda v_2.\ [\![\Theta;\ \Gamma, x : B \vdash c_2 \div C]\!]^c\ \theta\ (\gamma, v_2))^* & \\
& ((\lambda v_1.\ [\![\Theta;\ \Gamma, y : A \vdash c_1 \div B]\!]^c\ \theta\ (\gamma, v_1))^* & \text{Semantics} \\
& [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]^e\ \theta\ \gamma) & \\
= & (\lambda v_1.\ (\lambda v_2.\ [\![\Theta;\ \Gamma, x : B \vdash c_2 \div C]\!]^c\ \theta\ (\gamma, v_2))^* & \\
& ([\![\Theta;\ \Gamma, y : A \vdash c_1 \div B]\!]^c\ \theta\ (\gamma, v_1)))^* & \text{Monad Laws} \\
& [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]^e\ \theta\ \gamma) & \\
= & (\lambda v_1.\ (\lambda v_2.\ [\![\Theta;\ \Gamma, y : A, x : B \vdash c_2 \div C]\!]^c\ \theta\ (\gamma, v_1, v_2))^* & \\
& ([\![\Theta;\ \Gamma, y : A \vdash c_1 \div B]\!]^c\ \theta\ (\gamma, v_1)))^* & \text{Weakening} \\
& [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]^e\ \theta\ \gamma) & \\
= & (\lambda v_1.\ (\lambda v_2.\ [\![\Theta;\ \Gamma, y : A, x : B \vdash c_2 \div C]\!]^c\ \theta\ (\gamma, v_1, v_2))^* & \\
& ([\![\Theta;\ \Gamma, y : A \vdash [c_1] : \bigcirc B]\!]^e\ \theta\ (\gamma, v_1)))^* & \text{Semantics} \\
& [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]^e\ \theta\ \gamma) & \\
= & (\lambda v_1.\ [\![\Theta;\ \Gamma, y : A \vdash \mathsf{letv}\ x = [c_1]\ \mathsf{in}\ c_2 \div C]\!]^c\ \theta\ (\gamma, v_1))^* & \text{Semantics} \\
& [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]^e\ \theta\ \gamma & \\
= & [\![\Theta;\ \Gamma \vdash \mathsf{letv}\ y = e\ \mathsf{in}\ \mathsf{letv}\ x = c_1\ \mathsf{in}\ c_2 \div C]\!]^c\ \theta\ \gamma & \text{Semantics}
\end{array}
$$

- EqRefl

1	$\Theta;\ \Gamma \vdash e \equiv e : A$	Hypothesis
2	$\Theta;\ \Gamma \vdash e : A$	Inversion on 1

$$[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma \quad = \quad [\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma \quad \text{Reflexivity}$$

- EqSymm

1	$\Theta;\ \Gamma \vdash e \equiv e' : A$	Hypothesis
2	$\Theta;\ \Gamma \vdash e' \equiv e : A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash e : A$	Induction on 2
4	$\Theta;\ \Gamma \vdash e' : A$	Induction on 2

$$
\begin{array}{lll}
[\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma \quad \text{Induction on 2, above} \\
[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma \quad \text{Symmetry on prev step}
\end{array}
$$

- case EqTrans

1	$\Theta;\ \Gamma \vdash e \equiv e'' : A$	Hypothesis
2	$\Theta;\ \Gamma \vdash e : e'A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash e' : e''A$	Inversion on 1
4	$\Theta;\ \Gamma \vdash e : A$	Induction on 2
5	$\Theta;\ \Gamma \vdash e'' : A$	Induction on 3

$[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma$	$=$	$[\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma$	Induction
$[\![\Theta;\ \Gamma \vdash e' : A]\!]^e\ \theta\ \gamma$	$=$	$[\![\Theta;\ \Gamma \vdash e'' : A]\!]^e\ \theta\ \gamma$	Induction
$[\![\Theta;\ \Gamma \vdash e : A]\!]^e\ \theta\ \gamma$	$=$	$[\![\Theta;\ \Gamma \vdash e'' : A]\!]^e\ \theta\ \gamma$	Transitivity

- case EqSubst:

1	$\Theta;\ \Gamma \vdash [e_2/x]e_1 \equiv [e_2'/x]e_1' : B$	Hypothesis
2	$\Theta;\ \Gamma, x : A \vdash e_1 \equiv e_1' : B$	Inversion on 1
3	$\Theta;\ \Gamma \vdash e_2 \equiv e_2' : A$	Inversion on 1
4	$\Theta;\ \Gamma, x : A \vdash e_1 : B$	Induction on 2
5	$\Theta;\ \Gamma, x : A \vdash e_1' : B$	Induction on 2
6	$\Theta;\ \Gamma \vdash e_2 : A$	Induction on 3
7	$\Theta;\ \Gamma \vdash e_2' : A$	Induction on 3
8	$\Theta;\ \Gamma \vdash [e_2/x]e_1 : B$	Substitution of 6 into 4
9	$\Theta;\ \Gamma \vdash [e_2'/x]e_1' : B$	Substitution of 7 into 5

$[\![\Theta;\ \Gamma \vdash [e_2/x]e_1 : B]\!]^e\ \theta\ \gamma =$

$=$	$[\![\Theta;\ \Gamma, x : A \vdash e_1 : B]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e_2 : A]\!]^e\ \theta\ \gamma)$	Substitution
$=$	$[\![\Theta;\ \Gamma, x : A \vdash e_1 : B]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e_2' : A]\!]^e\ \theta\ \gamma)$	Induction
$=$	$[\![\Theta;\ \Gamma, x : A \vdash e_1' : B]\!]^e\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e_2' : A]\!]^e\ \theta\ \gamma)$	Induction
$=$	$[\![\Theta;\ \Gamma \vdash [e_2'/x]e_1' : B]\!]^e\ \theta\ \gamma$	Substitution

- EqCommandRefl

1	$\Theta; \Gamma \vdash c \equiv c \div A$	Hypothesis
2	$\Theta;\ \Gamma \vdash c \div A$	Inversion on 1

$$[\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma \quad = \quad [\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma \quad \text{Reflexivity}$$

- EqCommandSymm

1	$\Theta; \Gamma \vdash c \equiv c' \div A$	Hypothesis
2	$\Theta; \Gamma \vdash c' \equiv c \div A$	Inversion on 1
3	$\Theta;\ \Gamma \vdash c \div A$	Induction on 2
4	$\Theta;\ \Gamma \vdash c' \div A$	Induction on 2

$$\begin{array}{lcll} [\![\Theta;\ \Gamma \vdash c' \div A]\!]^c\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma & \text{Induction on 2, above} \\ [\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash c' \div A]\!]^c\ \theta\ \gamma & \text{Symmetry on prev step} \end{array}$$

- case EqCommandTrans

$$\begin{array}{lll} 1 & \Theta; \Gamma \vdash c \equiv c'' \div A & \text{Hypothesis} \\ 2 & \Theta;\ \Gamma \vdash c \div c' A & \text{Inversion on 1} \\ 3 & \Theta;\ \Gamma \vdash c' \div c'' A & \text{Inversion on 1} \\ 4 & \Theta;\ \Gamma \vdash c \div A & \text{Induction on 2} \\ 5 & \Theta;\ \Gamma \vdash c'' \div A & \text{Induction on 3} \end{array}$$

$$\begin{array}{lcll} [\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash c' \div A]\!]^c\ \theta\ \gamma & \text{Induction} \\ [\![\Theta;\ \Gamma \vdash c' \div A]\!]^c\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash c'' \div A]\!]^c\ \theta\ \gamma & \text{Induction} \\ [\![\Theta;\ \Gamma \vdash c \div A]\!]^c\ \theta\ \gamma & = & [\![\Theta;\ \Gamma \vdash c'' \div A]\!]^c\ \theta\ \gamma & \text{Transitivity} \end{array}$$

- case EqCommandSubst:

$$\begin{array}{lll} 1 & \Theta; \Gamma \vdash [e_2/x]c_1 \equiv [e_2'/x]c_1' \div B & \text{Hypothesis} \\ 2 & \Theta; \Gamma, x : A \vdash c_1 \equiv c_1' \div B & \text{Inversion on 1} \\ 3 & \Theta;\ \Gamma \vdash e_2 \equiv e_2' : A & \text{Inversion on 1} \\ 4 & \Theta;\ \Gamma, x : A \vdash c_1 \div B & \text{Induction on 2} \\ 5 & \Theta;\ \Gamma, x : A \vdash c_1' \div B & \text{Induction on 2} \\ 6 & \Theta;\ \Gamma \vdash e_2 : A & \text{Induction on 3} \\ 7 & \Theta;\ \Gamma \vdash e_2' : A & \text{Induction on 3} \\ 8 & \Theta;\ \Gamma \vdash [e_2/x]c_1 \div B & \text{Substitution of 6 into 4} \\ 9 & \Theta;\ \Gamma \vdash [e_2'/x]c_1' \div B & \text{Substitution of 7 into 5} \end{array}$$

$[\![\Theta;\ \Gamma \vdash [e_2/x]c_1 \div B]\!]^c\ \theta\ \gamma =$

$$\begin{array}{lcll} & [\![\Theta;\ \Gamma, x : A \vdash c_1 \div B]\!]^c\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e_2 : A]\!]^e\ \theta\ \gamma) & \text{Substitution} \\ = & [\![\Theta;\ \Gamma, x : A \vdash c_1 \div B]\!]^c\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e_2' : A]\!]^e\ \theta\ \gamma) & \text{Induction} \\ = & [\![\Theta;\ \Gamma, x : A \vdash c_1' \div B]\!]^c\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e_2' : A]\!]^e\ \theta\ \gamma) & \text{Induction} \\ = & [\![\Theta;\ \Gamma \vdash [e_2'/x]c_1' \div B]\!]^c\ \theta\ \gamma & \text{Substitution} \end{array}$$

□

Chapter 3

The Semantics of Separation Logic

In this chapter, I will describe the semantics of our separation logic. Rather than working directly with the heap model of separation logic, we will approach the semantics in a somewhat more modular style.

First, we will define what we mean by "semantics of separation logic" in terms of *BI algebras*, which give an algebraic semantics of separation logic in the same way that the Heyting algebras give semantics to intuitionistic logics. Then, we will look at how we can construct a BI-algebra from sets of elements of an arbitrary partial commutative monoid, proving that we can satisfy each of the axioms of a BI algebra.

Then, we will show that our predomain of heaps actually forms a partial commutative monoid, which means that we can now apply the theorems and definitions of the previous sections to immediately get the heap model we want.

With a semantic definition of assertions in hand, we will then move on to the semantics of specifications. We will again play the algebraic game, and give a Kripke semantics for specifications. We will also give a semantic interpretation of Hoare triples which validates the frame rule and fixed point induction.

After defining the semantics of assertions and specifications, I will give their syntax.

3.1 BI Algebras

A *BI algebra* is a Heyting algebra with additional residuated monoidal structure to model the separating conjunction and wand. This means that a BI algebra is a partial order $(B, \leq)$ with operations $(\top, \wedge, \supset, \bot, \vee, I, *, \mathrel{-\!\!*})$ satisfying:

1. $\forall p \in B.\ p \leq \top$
2. $\forall p \in B.\ \bot \leq p$
3. $\forall p, q, r \in B.$ if $r \leq p$ and $r \leq q$, then $r \leq p \wedge q$ and $p \wedge q \leq p$ and $p \wedge q \leq q$
4. $\forall p, q, r \in B.$ if $p \leq r$ and $q \leq r$, then $p \vee q \leq r$ and $p \leq p \vee q$ and $q \leq p \vee q$.
5. $\forall p, q, r.\ p \wedge q \leq r \iff p \leq q \supset r$
6. $\forall p.\ p * I = p$

7. $\forall p, q.\ p * q = q * p$
8. $\forall p, q, r.\ (p * q) * r = p * (q * r)$
9. $\forall p, q, r.\ p * q \leq r \iff p \leq q \mathrel{-\!\!*} r$

The first five conditions are just the usual conditions for a Heyting algebra (that it have greatest and least elements, greatest lower bounds and least upper bounds, and that it have an implication). The next three are the monoid structure axioms, that say I is the unit, and that $*$ is commutative and associative. The last axiom asserts the existence of a wand that is adjoint to the separating conjunction the same way that the implication is adjoint to the ordinary conjunction.

In addition, we will also ask that this algebra be *complete*, which means that meets and joins of arbitrary sets of elements be well-defined.

10. $\forall r \in B, P \subseteq B$. if $(\forall p \in P.\ r \leq p)$, then $r \leq \bigwedge P$ and $\forall p \in P.\ \bigwedge P \leq p$
11. $\forall r \in B, P \subseteq B$. if $(\forall p \in P.\ p \leq r)$, then $\bigvee P \leq r$ and $\forall p \in P.\ p \leq \bigvee P$

We will eventually use completeness in order to interpret quantifiers as possibly-infinitary conjunctions or disjunctions.

3.1.1 Partial Commutative Monoids

A partial commutative monoid is a triple $(M, e, \cdot)$, where $e \in M$, and $(\cdot)$ is a partial operation from $M \times M$ to M. We will write $m \# m'$ to mean that $m \cdot m'$ is defined.

Furthermore, the following properties must hold:

- e is a unit, so that $e \# m$ and $e \cdot m = m$.
- $(\cdot)$ is commutative. If $m_1 \# m_2$, then $m_2 \# m_1$ and $m_1 \cdot m_2 = m_2 \cdot m_1$.
- $(\cdot)$ is associative.
  - If $m_1 \# m_2$ and $(m_1 \cdot m_2) \# m_3$, then $m_2 \# m_3$ and $m_1 \# (m_2 \cdot m_3)$ and $(m_1 \cdot m_2) \cdot m_3 = m_1 \cdot (m_2 \cdot m_3)$.
  - If $m_2 \# m_3$ and $m_1 \# (m_2 \cdot m_3)$, then $m_1 \# m_2$ and $(m_1 \cdot m_2) \# m_3$ and $(m_1 \cdot m_2) \cdot m_3 = m_1 \cdot (m_2 \cdot m_3)$.

3.1.2 BI-Algebras over Partial Commutative Monoids

Given a partial commutative monoid, we can show that the powerset $\mathcal{P}(M)$ forms a BI-algebra.

Lemma 12. *(Powersets of Partial Commutative Monoids) Given a partial commutative monoid $(M, e, \cdot)$, its powerset $(\mathcal{P}(M), \subseteq)$ forms a BI-algebra with the following operations:*

- $\top = M$
- $p \wedge q = p \cap q$
- $p \supset q = \{m \in M \mid \textit{if } m \in p \textit{ then } m \in q\}$
- $\bot = \emptyset$
- $p \vee q = p \cup q$
- $I = \{e\}$
- $p * q = \{m \in M \mid \exists m_1, m_2.\ m_1 \# m_2 \textit{ and } m_1 \in p \textit{ and } m_2 \in q \textit{ and } m_1 \cdot m_2 = m\}$

- $p \mathrel{-\!\!*} q = \{m \in M \mid \forall m' \in p.\ \textit{if}\ m \# m'\ \textit{then}\ m \cdot m' \in q\}$
- $\bigwedge P = \bigcap P$
- $\bigvee P = \bigcup P$

Proof.

1. We want to show $\forall p \in \mathcal{P}(M).\ p \leq \top$

   Assume $p \in \mathcal{P}(M)$
   1 By definition of powerset, we have $p \subseteq M$
   2 By definition of $\top$, we have $p \subseteq \top$
   3 By definition of $\leq$, we have $p \leq \top$

2. We want to show $\forall p \in B.\ \bot \leq p$

   Assume $p \in \mathcal{P}(M)$
   1 By definition of $\emptyset$, we have $\emptyset \subseteq p$
   2 By definition of $\bot$, $\leq$, we have $\bot \leq p$

3. We want to show $\forall p, q, r \in \mathcal{P}(M)$. if $r \leq p$ and $r \leq q$, then $r \leq p \wedge q$ and $p \wedge q \leq p$ and $p \wedge q \leq q$

   Assume $p, q, r \in \mathcal{P}(M)$
   Assume $r \leq p$, $r \leq q$

   1 Expanding definition of $\leq$, we have $r \subseteq p$ and $r \subseteq q$
   2 By properties of $\cap$, we have $r \cap r \subseteq p \cap q$
   3 Hence $r \subseteq p \cap q$
   4 By definition of $\leq$ and $\wedge$, we have $r \leq p \wedge q$

   5 By properties of $\cap$, we have $p \cap q \subseteq p$
   6 By definition of $\leq$ and $\wedge$, we have $p \wedge q \leq p$

   7 By properties of $\cap$, we have $p \cap q \subseteq q$
   8 By definition of $\leq$ and $\wedge$, we have $p \wedge q \leq q$

4. We want to show $\forall p, q, r \in \mathcal{P}(M)$. if $p \leq r$ and $q \leq r$, then $p \vee q \leq r$ and $p \leq p \vee q$ and $q \leq p \vee q$.

   Assume $p, q, r \in \mathcal{P}(M)$, $p \leq r$, $q \leq$

   1 By definition of $\leq$, we have $p \subseteq r$, $q \subseteq r$
   2 By set properties, we have $p \cup q \subseteq r$
   3 By definition of $\leq$, we have $p \vee q \leq r$

4 By set properties we have $p \subseteq p \cup q$
5 By definitions of $\leq$ and $\vee$, we have $p \leq p \vee q$

6 By set properties, we have $q \subseteq p \cup q$
7 By definitions of $\leq$ and $\vee$, we have $q \leq p \vee q$

5. We want to show $\forall p, q, r \in \mathcal{P}(M).\ p \wedge q \leq r \iff p \leq q \supset r$

Assume $p, q, r \in \mathcal{P}(M)$

$\Rightarrow$ direction:
1 Assume $p \wedge q \leq r$
2 By definitions of $\leq$ and $\wedge$, we have $p \cap q \subseteq r$
3 We want to show $p \leq q \supset r$
4 So we want to show $p \subseteq (q \supset r)$
5 So we want to show $\forall m$, if $m \in p$ then $m \in (q \supset r)$
6 Assume m and $m \in p$
7 Want to show $m \in q \supset r$
8 Equivalent to showing if $m \in q$, then $m \in r$
9 Assume $m \in q$
10 Since $m \in p$ and $m \in q$, we know $m \in p \cap q$.
11 Since $p \cap q \subseteq r$, we know $m \in r$
12 Therefore $p \subseteq q \supset r$
13 By definition of $\leq$, we see $p \leq q \supset r$

$\Leftarrow$ direction:
14 Assume $p \leq q \supset r$
15 By definition of $\leq$, we know $p \subseteq q \supset r$
16 We want to show $p \wedge q \leq r$
17 So we want to show $p \cap q \subseteq r$
18 So we want to show $\forall m$, if $m \in p \cap q$ then $m \in r$
19 Assume $m \in p \cap q$
20 Hence $m \in p$ and $m \in q$
21 Since $p \subseteq q \supset r$, we know for all m, if $m \in p$, then if $m \in q$, then $m \in r$
22 Hence $m \in r$
23 Hence $p \cap q \subseteq r$
24 By definition of $\leq$ and $\wedge$, we conclude $p \wedge q \leq r$

6. We want to show $\forall p \in \mathcal{P}(M).\ p * I = p$

1 Assume $p \in \mathcal{P}(M)$
2 We want to show $p * I = p$
3 This is equivalent to showing for all $m \in M$, that $m \in p * I$ if and only if $m \in p$

4 Assume $m \in M$
5 $\Rightarrow$ direction:
6 Assume $m \in p * I$
7 Therefore $\exists m_1, m_2 \in M$ such that
$m_1 \# m_2$ and $m_1 \in p$ and $m_2 \in I$ and $m = m_1 \cdot m_2$
8 Let m_1, m_2 be witnesses, so that
9 $m_1 \# m_2$ and $m_1 \in p$ and $m_2 \in I$ and $m = m_1 \cdot m_2$
10 Since $I = \{e\}$, we know $m_2 = e$
11 By unit property, $m = m_1 \cdot e = m_1$
12 Since $m_1 \in p$, we know $m \in p$
13 $\Leftarrow$ direction:
14 Assume $m \in p$
15 We want to show $m \in p * I$
16 So we want to show there are m_1, m_2
such that $m_1 \# m_2$ and $m_1 \in p$ and $m_2 \in I$ and $m = m_1 \cdot m_2$
17 Choose m_1 to be m, and m_2 to be e
18 So we want to show $m \# e$ and $m \in p$ and $e \in I$ and $m = m \cdot e$
19 By properties of unit, $m \# e$ and $m = m \cdot e$
20 By definition of I, we know $e \in I$
21 We know $m \in p$ by hypothesis
22 Therefore goal in line 16 met.

7. We want to show $\forall p, q \in \mathcal{P}(M).\ p * q = q * r$
Assume $p, q \in \mathcal{P}(M)$

$$
\begin{array}{rcll}
p * q & = & \{m \in M \mid \exists m_1 \in p, m_2 \in q.\ m_1 \# m_2 \wedge m_1 \cdot m_2 = m\} & \text{Definition} \\
& = & \{m \in M \mid \exists m_2 \in q, m_1 \in p.\ m_1 \# m_2 \wedge m_1 \cdot m_2 = m\} & \text{Logical manipulation} \\
& = & \{m \in M \mid \exists m_2 \in q, m_1 \in p.\ m_2 \# m_1 \wedge m_2 \cdot m_1 = m\} & \text{Commutativity} \\
& = & q * p & \text{Definition}
\end{array}
$$

8. We want to show $\forall p, q, r \in \mathcal{P}(M).\ (p * q) * r = p * (q * r)$

1 Assume $p, q, r \in \mathcal{P}(M)$
2 We want to show $(p * q) * r = p * (q * r)$
3 This is equivalent to showing for all $m \in M$, that
$m \in (p * q) * r$ if and only if $m \in p * (q * r)$
4 Assume $m \in M$
5 $\Rightarrow$ direction:
6 Assume $m \in (p * q) * r$
7 Therefore there are m_{pq}, m_r such that
$m_{pq} \# m_r$ and $m = m_{pq} \cdot m_r$ and $m_{pq} \in p * q$ and $m_r \in r$
8 From $m_{pq} \in p * q$, we know there are m_p, m_q such that
$m_p \# m_q$ and $m_{pq} = m_p \cdot m_q$ and $m_p \in p$ and m_q in r

9 By $m_{pq} = m_p \cdot m_q$, we know $m = (m_p \cdot m_q) \cdot m_r$ and $(m_p \cdot m_q)\#m_r$
10 Since $m_p\#m_q$ and $(m_p \cdot m_q)\#m_r$, by associativity we know
$m_q\#m_r$ and $m_p\#(m_q \cdot m_r)$ and $m = (m_p \cdot m_q) \cdot m_r = m_p \cdot (m_q \cdot m_r)$
11 Since $m_q\#m_r$ and $m_q \cdot m_r = m_q \cdot m_r$ and $m_q \in q$ and $m_r \in r$,
we know $m_q \cdot m_r \in q * r$
12 Since $m_p\#(m_q \cdot m_r)$ and $m_p \cdot (m_q \cdot m_r) = m_p \cdot (m_q \cdot m_r)$
and $m_p \in p$ and $m_q \cdot m_r \in q * r$, we know $m_p \cdot (m_q \cdot m_r) \in p * (q * r)$
13 Therefore $m \in p * (q * r)$
14 $\Leftarrow$ direction:
15 Assume we have m_p, m_{qr} such that
$m_p\#m_{qr}$ and $m = m_p \cdot m_{qr}$ and $m_p \in p$ and $m_{qr} \in q * r$
16 Therefore we have m_q, m_r such that
$m_q\#m_r$ and $m_{qr} = m_q \cdot m_r$ and $m_q \in q$ and $m_r \in r$
17 From $m_{qr} = m_q \cdot m_r$ we have
$m = m_p \cdot (m_q \cdot m_r)$ and $m_p\#(m_q \cdot m_r)$
18 By associativity with $m_q\#m_r$ and $m_p\#(m_q \cdot m_r)$, we have
$m_p\#m_q$ and $(m_p \cdot m_q)\#m_r$ and $m = m_p \cdot (m_q \cdot m_r) = (m_p \cdot m_q) \cdot m_r$
19 Since $m_p\#m_q$ and $m_p \in p$ and $m_q \in q$, we know
$(m_p \cdot m_q) \in p * q$
20 Since $(m_p \cdot m_q)\#m_r$ and $(m_p \cdot m_q) in p * q$ and $m_r \in r$, we have
$(m_p \cdot m_q) \cdot m_r \in (p * q) * r$
21 Therefore $m \in (p * q) * r$

9. We want to show that for all $p, q, r \in \mathcal{P}(M)$, $p * q \leq r$ if and only if $p \leq q \mathrel{-\!\!*} r$.

Assume $p, q, r \in \mathcal{P}(M)$.
1 First, we will give the $\Rightarrow$ direction.
2 Assume $p * q \leq r$
3 So we know $\{m \mid \exists m_1, m_2.\ m_1\#m_2$ and $m_1 \in p$ and $m_2 \in q$ and $m_1 \cdot m_2 = m\} \subseteq r$
4 Thus, $\forall m.\ (\exists m_1, m_2.\ m_1\#m_2$ and $m_1 \in p$ and $m_2 \in q$ and $m_1 \cdot m_2 = m)$ implies $m \in r$.
5 By turning existentials on the left into universals, and instantiating m,
6 $\forall m_1, m_2.\ (m_1\#m_2$ and $m_1 \in p$ and $m_2 \in q)$ implies $(m_1 \cdot m_2) \in r$ [HYP1]
7 Now, to show $p \leq q \mathrel{-\!\!*} r$, we must show for all m, if $m \in p$, that $m \in q \mathrel{-\!\!*} r$.
8 Assume m, $m \in p$. [HYP2]
9 Now, we want to show $m \in \{m \mid \forall m' \in q.\ m\#m'$ and m' implies $(m \cdot m') \in r\}$
10 This is equivalent to showing $\forall m' \in q.\ m\#m'$ and m' implies $(m \cdot m') \in r$
11 Assume m', $m' \in q$, $m\#m'$. [HYP3]
12 Instantiating [HYP1] with m and m' and the hypotheses in [HYP2] and [HYP3],
13 we can conclude $(m \cdot m') \in r$.

14 Now, we will show the $\Leftarrow$ direction.
15 Assume $p \leq q \mathrel{-\!\!*} r$.
16 So we know, $p \subseteq \{m \mid \forall m' \in q.m\#m'$ and $m' \in q$ implies $(m \cdot m') \in r\}$

17 Therefore $\forall m.m \in p$ implies $\forall m' \in q.\ m\#m'$ and $m' \in q$ implies $(m \cdot m') \in r$.
18 Therefore $\forall m, m'.\ m \in p$ and $m' \in q$ and $m\#m'$ implies $(m \cdot m') \in r$
19 Therefore $\forall m_o, m, m'.\ m \in p$ and $m' \in q$ and $m\#m'$ and $m_o = m \cdot m'$ implies $m_o \in r$
20 Therefore $\forall m_o, (\exists m, m'.\ m \in p$ and $m' \in q$ and $m\#m'$ and $m_o = m \cdot m')$ implies $m_o \in r$
21 Therefore $\forall m_o, m_o \in (p * q)$ implies $m_o \in r$
22 Therefore $p * q \subseteq r$
23 Therefore $p * q \leq r$

10. We want to show $\forall r, P \subseteq B$, if $(\forall p \in P.\ r \leq p)$, then $r \leq \bigwedge P$ and $\forall p \in P.\ \bigwedge P \leq p$.

1 Assume r, P, $P \subseteq B$, and $(\forall p \in P.\ r \leq p)$

2 First, we want to show $r \leq \bigwedge P$.
3 This is equivalent to showing $r \subseteq \bigcap P$
4 This is equivalent to showing that for all $m \in r$, $m \in \bigcap P$.
5 Assume m, $m \in r$.
6 Showing $m \in \bigcap P$ is equivalent to $\forall p \in P.m \in p$
7 Assume $p \in P$.
8 Instantiating hypothesis with p, $r \subseteq p$.
9 This means $\forall m'.m' \in r \supset m' \in p$.
10 Instantiating m' with m, we learn $m \in p$.
11 Therefore, $r \leq \bigwedge P$.

12 Second, we want to show that $\forall p \in P.\ \bigwedge P \leq p$.
13 Assume p, $p \in P$.
14 Now, we want to show $\bigwedge P \leq p$.
15 This is equivalent to showing $\bigcap P \subseteq p$.
16 This is equivalent to showing $\forall m.\ m \in \bigcap P \supset m \in p$
17 Assume m, $m \in \bigcap P$.
18 Therefore, $\forall p' \in P.\ m \in p'$.
19 Instantiating p' with p, we get $m \in p$.

11. We want to show $\forall r, P \subseteq B$, if $(\forall p \in P.\ p \leq r)$, then $\bigvee P \leq r$ and $\forall p \in P.\ p \leq \bigvee P$.

1 Assume $r, P \subseteq B$, and $(\forall p \in P.\ p \leq r)$

2 First, we want to show $\bigvee P \leq r$
3 This is equivalent to showing $\bigcup P \subseteq r$
4 This is equivalent to showing $\forall m.\ m \in \bigcup P \supset m \in r$
5 Assume m, $m \in \bigcup P$.
6 $m \in \bigcup P$ is equivalent to $\exists p \in P.\ m \in p$
7 Suppose $p' \in P$ is the witness, and that $m \in p'$
8 Instantiating the quantifier p in the hypothesis, we get $p' \leq r$

9 This means $\forall m', m' \in p' \supset m' \in r$
10 Instantiating the quantifier m' with m, we conclude $m \in r$.
11 Therefore, $\forall m.\ m \in \bigcup P \supset m \in r$

12 Second, we want to show $\forall p \in P.\ p \leq \bigvee P$.
13 Assume p, $p \in P$
14 We want to show $p \leq \bigvee P$
15 This is equivalent to showing $p \subseteq \bigcup P$
16 This is equivalent to showing $\forall m.\ m \in p \supset m \in \bigcup P$
17 This is equivalent to showing $\forall m.\ m \in p \supset \exists p' \in P.\ m \in p'$
18 Assume m, $m \in p$
19 Take p' to be p, since $p \in P$.
20 Thus, $m \in p$ by hypothesis
21 Therefore, $p \leq \bigvee P$

□

3.1.3 Sets of Heaps Form the BI Algebra of Heap Assertions

Now, we will take our predomain H and form a partial commutative monoid from it, from which we can build a complete BI algebra. This algebra will serve as the domain of interpretation of heap assertions. To construct it, we will first apply the forgetful functor U to H, to forget the partial order structure and leaving us with $U(H)$, the ordinary set of heaps.

$$U(H) = \sum L \in \mathcal{P}^{\text{fin}}(Loc).\ (\prod (n, A) \in L.\ [\![\cdot \vdash A : \bigstar]\!]\ \langle\rangle\ (K, K))$$

However, in what follows we will suppress the U, in order to reduce clutter. Now, we can define the operations on it as follows:

- The unit element $e \triangleq (\emptyset, \emptyset)$
- The operation $(L, f) \cdot (L', g)$ is defined when $L \cap L' = \emptyset$, and is equal to

$$\left(L \cup L', \lambda x.\ \begin{cases} f(x) & \text{when } x \in L \\ g(x) & \text{when } x \in L' \end{cases}\right)$$

The lambda-expression in the operation definition actually defines a function. Since we know that since L and L' are disjoint, this means that any element of $x \in L \cup L'$ is exclusively either in L or L', which means that the definition is unambiguous. Since f and g are well-typed with respect to the index sets L and L' respectively, our new function must be as well.

Now, we can check the properties.

- First, we will check that e is a unit. Suppose we have $m = (L, f) \in H$.

$$\begin{array}{rcll} (\emptyset, \emptyset) \cdot (L, f) & = & \left(\emptyset \cup L, \lambda x.\ \begin{cases} \emptyset(x) & \text{when } x \in \emptyset \\ f(x) & \text{when } x \in L \end{cases}\right) & \text{Definition} \\ & = & \left(L, \lambda x.\ \ f(x) \ \ \text{when } x \in L\ \right) & \text{Simplification} \\ & = & (L, f) & \text{Simplification} \end{array}$$

- Second, we will check commutativity. Suppose we have $(L, f) \in H$ and $(L', g) \in H$. First, it's obviously the case that if $L \cap L' = \emptyset$, then $L' \cap L = \emptyset$.

$$\begin{array}{rcll}
(L,f)\cdot(L',g) & = & \left(L \cup L', \lambda x. \left\{ \begin{array}{ll} f(x) & \mathsf{when}\ x \in L \\ g(x) & \mathsf{when}\ x \in L' \end{array} \right.\right) & \text{Definition} \\
& = & \left(L' \cup L, \lambda x. \left\{ \begin{array}{ll} f(x) & \mathsf{when}\ x \in L \\ g(x) & \mathsf{when}\ x \in L' \end{array} \right.\right) & \text{Commutativity of } \cup \\
& = & \left(L' \cup L, \lambda x. \left\{ \begin{array}{ll} g(x) & \mathsf{when}\ x \in L' \\ f(x) & \mathsf{when}\ x \in L \end{array} \right.\right) & \text{Reordering Cases} \\
& = & (L',g)\cdot(L,f) & \text{Definition}
\end{array}$$

- Now, we will check associativity. Suppose (L, f), (L', g), and (L'', h) are in H.
  First, assume that $L \cap L' = \emptyset$ and that $(L \cup L') \cap L'' = \emptyset$. Then we know that $L \cap L'' = \emptyset$ and $L' \cap L'' = \emptyset$, so we can conclude that $L \cap (L' \cup L'') = \emptyset$.
  Second, assume that $L' \cap L'' = \emptyset$ and that $L \cap (L' \cup L'') = \emptyset$. Then we know that $L \cap L' = \emptyset$ and $L \cap L'' = \emptyset$, so we can conclude that $(L \cup L') \cap L'' = \emptyset$.

$$\begin{array}{rcll}
(L,f)\cdot((L',g)\cdot(L'',h)) & = & (L,f)\cdot\left(L' \cup L'', \lambda x. \left\{ \begin{array}{ll} g(x) & \mathsf{when}\ x \in L' \\ h(x) & \mathsf{when}\ x \in L'' \end{array} \right.\right) & \text{Definition} \\
& = & \left(L \cup (L' \cup L''), \lambda x. \left\{ \begin{array}{ll} f(x) & \mathsf{when}\ x \in L \\ g(x) & \mathsf{when}\ x \in L' \\ h(x) & \mathsf{when}\ x \in L'' \end{array} \right.\right) & \text{Definition} \\
& = & \left((L \cup L') \cup L'', \lambda x. \left\{ \begin{array}{ll} f(x) & \mathsf{when}\ x \in L \\ g(x) & \mathsf{when}\ x \in L' \\ h(x) & \mathsf{when}\ x \in L'' \end{array} \right.\right) & \text{Associativity} \\
& = & \left(L \cup L', \lambda x. \left\{ \begin{array}{ll} f(x) & \mathsf{when}\ x \in L \\ g(x) & \mathsf{when}\ x \in L' \end{array} \right.\right)\cdot(L'',h) & \text{Definition} \\
& = & ((L,f)\cdot(L',g))\cdot(L'',h) & \text{Definition}
\end{array}$$

Therefore, we can use the construction of the previous subsection to equip sets of heaps with the structure of a complete BI algebra.

3.2 Challenges in Interpreting Specifications

In this section, we will give the semantics of specifications. There are two main technical challenges to overcome. First, we will need to handle the problem of supporting fixed point induction, and second, we will need to support the frame rule of separation logic. We will first describe these two problems, and then give each problem's solution.

3.2.1 Admissibility and Fixed Point Induction

We are designing a partial correctness program logic, and so to prove the correctness of recursive definitions, we would like an LCF-style fixed point induction rule. Suppose we wish to state a property P of the fixed point of a functional $f : \bigcirc A \to \bigcirc A$. (For example, P may be a Hoare triple). Ideally, we want to give an inference rule for recursion taking the following form:

$$\frac{P(\bot) \qquad \forall x.\ P(x) \supset P(f(x))}{P(\mathit{fix}(f))}\ \textsc{Fixed Point Induction (Almost)}$$

This rule asserts that we prove a property P by showing it holds of $\bot$, and that applying f preserved the truth of P. However, this rule is not sound for all P — it is only sound for *admissible* P. That is, we need the additional condition that if P holds for every element of a chain, it also holds for the limit of the chain. Not all predicates are admissible; consider for example the f on the (pointed natural numbers):

$$\begin{array}{lcl} f & = & \lambda g.\ \lambda x.\ \mathsf{if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ g(x-1) \\ P(h) & = & \exists n : \mathbb{N}.\ h(n) = \bot \end{array}$$

Here, $P(f^k(\bot))$ holds for any k-approximation — $f^k(\bot)$ will loop on any input larger than k. However, the fixed point will never loop on any natural number input.

Often, admissibility is checked by syntactic conditions which pick out a class of predicates known to be admissible. We cannot take this approach, since we will use a higher-order assertion logic. As a result, we do not know what the shape of a proposition is, since it may refer to proposition variables.

To handle this problem, we will use a technique sometimes called continuation closure or $\top\top$-closure, which will force our Hoare triples to be admissible (indeed, continuous) predicates. This will let us make free use of arbitrary assertions in Hoare triples, with no restrictions on the forms of pre- or post-conditions.

3.2.2 The Frame Rule

We also face a second sticky issue: our denotational semantics does not validate the frame property. The interpretation of the $\mathsf{new}_A(e)$ command allocates a new reference by finding the largest numeric id of any reference in the heap's domain, and then allocating a reference whose numeric id is one greater than that.

This means that the behavior of the memory allocator is deterministic, which means that our semantic domain can include awkward programs which crash if the heap is larger than a certain size. Concretely, consider the following basic continuation:

$$k_{bad}\ (L, h) \quad = \quad \mathsf{if}\ |L| < 50\ \mathsf{then}\ \bot\ \mathsf{else}\ \top$$

This continuation is safe (i.e., returns $\bot$) if the heap contains fewer than 50 elements, and will crash (i.e., return $\top$) otherwise. Obviously, the safety monotonicity property cannot hold for such programs, because extending the heap enough can cause programs to switch from running

safely to crashing. On the other hand, we do not actually want to prove the correctness of any of these pathological programs: all the programs we actually want to write and prove correct are actually well-behaved, and will not pay attention to state they have no ownership of.

To make use of this fact, we will adapt an idea of Birkedal and Yang [7]. They proposed changing the interpretation of program specifications from a boolean semantics (in which each specification is either true or false) into a Kripke interpretation.

The modal frame they proposed was one in which worlds are sets of assertions of separation logic (i.e., elements of a BI algebra). Intuitively, we can think of each world as "the set of assertions that can be safely framed onto this specification". A specification is then true when all assertions can be framed onto it, which is how we will end up justifying the frame rule. As we did for assertions, we will proceed in a modular way. We will first define a "world preorder" on elements of a BI algebra — the extension ordering — and then use this ordering to give the truth values as upwards-closed sets of assertions.

3.3 Basic Hoare Triples

Since we have a continuation semantics, it is natural to define a continuation style interpretation of basic Hoare triples, as well. Our initial attempt, however, will not succeed:

$$[P]\ C\ [a : A.\ Q(a)] \triangleq \forall h \in P.\ (C\ Q\ h = \bot)$$

The idea is that we can view a postcondition $Q(v)$ as a sort of continuation, which returns $\bot$ if $h \in Q(v)$, and $\top$ otherwise. So we say that for every heap in P, $C\ Q\ h$ must be bottom.

Thinking of $\top$ as a sort of crash, this definition says that C satisfies the precondition P and post-condition Q when, if it is applied to a continuation which does not crash on any Q-heap, yields a result which will not crash when given any P-heap.

While this idea is elegant, it does not quite work.

The problem is that Q is an assertion, an arbitrary set-theoretic function from A to sets of to heaps, and C requires its continuation argument to be *continuous*, a property which not met by arbitrary Q. This idea can be repaired if we define a "best continuous approximation" to each Q.

3.3.1 Approximating Postconditions

Given a predomain A, and a $Q \in U(A) \to \mathcal{P}(H)$, we define $Approx(Q)$ as the set:

$$Approx(Q) \triangleq \{k \in A \to H \to O \mid \forall v \in A, h \in Q(v).\ k\ v\ h = \bot\}$$

These define a set of continuations which "continuously approximate" the postcondition Q – they are the set of continuations which run forever when given a value and heap in Q. Using this set we will define the function $Best(Q)$, which will be the "best continuous approximation" to Q.

Intuitively, think of this as being like a closure operator from topology, which finds the smallest open set containing the given set. In our case, we want to find the

$$Best(Q) \triangleq \lambda v \in A.\ \lambda h \in H.\ \begin{cases} \top & \text{when } \exists k \in Approx(Q).\ k\ v\ h = \top \\ \bot & \text{otherwise} \end{cases}$$

Of course, we have to verify that $Best(Q)$ is actually a continuous function.

- First, we need to check that $Best(Q)$ is a monotone function.

Suppose we have $v \sqsubseteq v'$ and $h \sqsubseteq h'$.
  We know $Best(Q)\ v\ h \in O$. Analyzing this by cases, we see
  Suppose $Best(Q)\ v\ h = \bot$
    Since $\forall o \in O.\ \bot \sqsubseteq o$, it follows that $\bot \sqsubseteq Best(Q)\ v'\ h'$
    So $Best(Q)\ v\ h \sqsubseteq Best(Q)\ v'\ h'$
  Suppose $Best(Q)\ v\ h = \top$
    By definition of $Best(Q)$, $\exists k \in Approx(Q).\ k\ v\ h = \top$
    Let $k \in Approx(Q)$ be the witness such that $k\ v\ h = \top$
      Since k is monotone, $k\ v\ h \sqsubseteq k\ v'\ h'$
      So $\top \sqsubseteq k\ v'\ h'$
      Since $\top$ is maximal in O, $k\ v'\ h' = \top$
      So we can take k to be the witness such that $\exists k.\ k\ v'\ h' = \top$
    Therefore $Best(Q)\ v'\ h' = \top$
    Therefore $Best(Q)\ v\ h \sqsubseteq Best(Q)\ v'\ h'$

- Second, we need to show that $Best(Q)$ preserves limits.

Suppose we have two chains v_i and h_i such that $i \leq j$ implies $v_i \sqsubseteq v_j$ and $h_i \sqsubseteq h_j$.
  We want to show that $\bigsqcup_i Best(Q)\ v_i\ h_i = Best(Q)\ (\sqcup v_i)\ (\sqcup h_i)$
  By excluded middle, either some k in $Approx(Q)$ such that $k\ (\sqcup v_i)\ (\sqcup h_i) = \top$, or not.
  Suppose that $\exists k \in Approx(Q).\ k\ (\sqcup v_i)\ (\sqcup h_i) = \top$
    Therefore $Best(Q)\ (\sqcup v_i)\ (\sqcup h_i) = \top$
    By continuity of k, $\bigsqcup_i k\ v_i\ h_i = \top$
    Since O is discrete, there is an n such that $k\ v_n\ h_n = \top$
    Therefore, for all $j \geq n$, $Best(Q)\ v_n\ h_n = \top$
    This means $\bigsqcup_i Best(Q)\ v_i\ h_n = \top$
    Therefore $\bigsqcup_i Best(Q)\ v_i\ h_n = Best(Q)\ (\sqcup v_i)\ (\sqcup h_i)$
  Suppose that $\neg(\exists k \in Approx(Q).\ k\ (\sqcup v_i)\ (\sqcup h_i) = \top$
    This is equivalent to $\forall k \in Approx(Q).\ k\ (\sqcup v_i)\ (\sqcup h_i) = \bot$
    This means $Best(Q)\ (\sqcup v_i)\ (\sqcup h_i) = \bot$
    Now, assume $k \in Approx(Q)$
      So $k\ (\sqcup v_i)\ (\sqcup h_i) = \bot$
      By continuity, $\bigsqcup_i k\ v_i\ h_i = \bot$
      Therefore for all i, $k\ v_i\ h_i = \bot$
    So for all $k \in Approx(Q)$ and i, we know $k\ v_i\ h_i = \bot$
    This is equivalent to $\forall i.\ \neg(\exists k \in Approx(Q).\ k\ v_i\ h_i = \top)$

20 Therefore, for all i, we know $Best(Q)\ v_i\ h_i = \bot$
21 Therefore, we know $\bigsqcup_i Best(Q)\ v_i\ h_i = \bot$
22 So we conclude $\bigsqcup_i Best(Q)\ v_i\ h_i = Best(Q)\ (\sqcup v_i)\ (\sqcup h_i)$

This establishes that $Best(Q)$ is a continuous function.

Now we will prove a minor lemma about this function, which shows that we are interpreting the two-point Sierpinksi lattice $O = \{\bot \sqsubseteq \top\}$ such that the bottom element is truth, and the top element is falsehood.

Lemma 13. *(Inclusion Order Reverses Approximation Order) Suppose Q and Q' are assertions in $[\![A]\!] \to \mathcal{P}(H)$, and that for all $a \in [\![A]\!]$, $Q(a) \subseteq Q'(a)$. Then $Best(Q') \sqsubseteq Best(Q)$.*

1 We want to show $Best(Q') \sqsubseteq Best(Q)$
2 First, we will observe that $Approx(Q') \subseteq Approx(Q)$
3 So, suppose that $k \in Approx(Q')$. We want to show $k \in Approx(Q)$.
4 We want to show that for all $a \in [\![A]\!]$ and $h \in Q(a)$, $k\ a\ h = \bot$.
5 Assume that $a \in [\![A]\!]$ and $h \in Q(a)$.
6 We know that since $Q(a) \subseteq Q'(a)$, we have $h \in Q'(a)$.
7 Therefore since $k \in Approx(Q')$, we know $k\ a\ h = \bot$
8 Now, we want to show that $Approx(Q') \subseteq Approx(Q)$
9 So we want to show that for all $a \in [\![A]\!]$ and $h \in H$, $Approx(Q')\ a\ h \subseteq Approx(Q)\ a\ h$
10 Assume we have $a \in [\![A]\!]$ and $h \in H$
11 Suppose $Approx(Q')\ a\ h = \top$:
12 Therefore there is a $k \in Approx(Q')$ such that $k\ a\ h = \top$
13 Therefore $k \in Approx(Q)$, and so $Approx(Q)\ a\ h = \top$
14 Suppose $Approx(Q')\ a\ h = \bot$:
15 Therefore for all $k \in Approx(Q')$, we have $k\ a\ h = \top$
16 Therefore for all $k \in Approx(Q)$, we have $Approx(Q)\ a\ h = \bot$

3.3.2 Defining the Basic Hoare Triples

Supposing that P is an element of the BI algebra $\mathcal{P}(H)$, C is an element of the domain of commands $(A \to K) \to K$, and Q is an A-indexed assertion, of type $A \to \mathcal{P}(H)$, then we can define the basic boolean Hoare triple $[P]\ C\ [a : A.\ Q(a)]$:

$$[P]\ C\ [a : A.\ Q(a)] \triangleq \forall h \in P.\ (C\ (Best\ Q)\ h = \bot)$$

If C is given a continuation which will run forever (i.e., yields $\bot$) whenever it receives a heap in Q, then given a heap $h \in P$, C applied to that continuation and that heap will also run forever.

The reason that we interpret triples this way is to make the fixed-point induction rule a sound rule of inference. Intuitively, we are defining our triples only in terms of continuous things, and so the limit of a chains should agree with the chain.

We will not prove this fact here, deferring that proof for the Kripke Hoare triples we will actually use. The reason is that these basic triples are too basic — they do not validate the frame

property — and so we will not need to use basic Hoare triples except as a tool used to define our Kripke Hoare triples.

3.4 Kripke Hoare Triples

As with the development of the assertion logic, we will proceed in a modular style, by first giving algebraic conditions we want our

3.4.1 World Preorders

We can define a world preorder $W(B, \preceq)$ over a BI algebra B as follows. The elements of $W(B, \leq)$ are the elements of B, and for any two elements p and q, the ordering $p \preceq q$ is defined as follows:

$$p \preceq q \iff \exists r.\ p * r = q$$

To verify the relation $\preceq$ is a preorder, we need to show it is reflexive and transitive.
$p \preceq p$ holds because we can take r to be I.
To show transitivity, we must show that $p_1 \preceq p_3$, given that $p_1 \preceq p_2$ and $p_2 \preceq p_3$.

1 Assume $p_1 \preceq p_2$
2 Assume $p_2 \preceq p_3$
3 By definition of $\preceq$, $\exists r.\ p_1 * r = p_2$
4 By definition of $\preceq$, $\exists r'.\ p_2 * r' = p_3$
5 Let r and r' be the witnesses in lines 3 and 4, so we have
6 $p_1 * r = p_2$
7 $p_2 * r' = p_3$
8 Substituting for p_2, we get $p_1 * r * r' = p_3$
9 Taking as witness $r'' = r * r'$, we show $\exists r''.\ p_1 * r'' = p_3$
10 By definition of $\preceq$, $p_1 \preceq p_3$

3.4.2 Heyting Algebras over Preorders

Given any preorder $(P, \preceq)$, we can construct a complete Heyting algebra by considering the set of its upward-closed subsets $(\mathcal{P}^{\uparrow}(P), \subseteq)$:

$$\mathcal{P}^{\uparrow}(P) = \{S \in \mathcal{P}(P) \mid \forall p \in S, \forall q \in P. \text{ if } p \preceq q \text{ then } q \in S\}$$

The ordering relation for the Heyting algebra is set inclusion, and the operations are:

- $\top = P$
- $\bot = \emptyset$
- $S \wedge S' = S \cap S'$

- $S \vee S' = S \cup S'$
- $\bigwedge_{i \in I} S_i = \bigcap_{i \in I} S_i$
- $\bigvee_{i \in I} S_i = \bigcup_{i \in I} S_i$
- $S \supset S' = \{r \in P \mid \forall r' \succeq r.\ \text{if } r' \in S \text{ then } r' \in S'\}$

The idea here is that we will take the world preorder over assertions (ordered by the extension order $\preceq$) and use it to define a Heyting algebra, whose elements will become the interpretations of specifications.

To show that these operations actually form a Heyting algebra, we need to show that they satisfy the Heyting algebra axioms. First, we need to show that the meet and join are the greatest lower bounds and least upper bounds respectively. To do this, we will just show that arbitrary meets and joins exist, and then the nullary and binary meets and joins will fall out as a special case.

Lemma 14. *(Meets in the algebra of specifications) If $X \subseteq \mathcal{P}^{\uparrow}(P)$, then $\bigwedge X$ defines a meet.*

Proof. We need to show that if for all $i \in I$, $S_i \in \mathcal{P}^{\uparrow}(P)$, then $\bigwedge_{i \in I} S_i \in \mathcal{P}^{\uparrow}(P)$. First, we will verify that the intersection of a family of upwards-closed subsets is itself an upwards-closed subset.

1 Assume $\forall i \in I.\ S_i \in \mathcal{P}^{\uparrow}(P)$
2 We want to show $\bigwedge_{i \in I} S_i \in \mathcal{P}^{\uparrow}(P)$
3 So we want to show for all $x \in \bigwedge_{i \in I} S_i$ and for all $y \in P$, if $x \preceq y$ then $y \in \bigwedge_{i \in I} S_i$
4 Assume x, $x \in \bigwedge_{i \in I} S_i$, y, $y \in P$, $x \preceq y$
5 Since $\bigwedge_{i \in I} S_i = \bigcap_{i \in I} S_i$, we know $\forall i \in I.\ x \in S_i$
6 Assume $i \in I$
7 Since $x \in S_i$, $x \preceq y$, and S_i is upwards-closed, $y \in S_i$
8 Therefore, $\forall i \in I$, $y \in S_i$
9 Therefore for all $x \in \bigwedge_{i \in I} S_i$ and for all $y \in P$, if $x \preceq y$ then $y \in \bigwedge_{i \in I} S_i$
10 Which means $\bigwedge_{i \in I} S_i \in \mathcal{P}^{\uparrow}(P)$

Next, we need to show the Heyting algebra axiom for meets. Stated formally, this is $\forall S \in \mathcal{P}^{\uparrow}(P)$, if $X \subseteq \mathcal{P}^{\uparrow}(P)$ and $(\forall S' \in X.\ S \leq S')$, then $S \leq \bigwedge X$ and $\forall S' \in X, \bigwedge X \leq S'$.

1 Assume $S \in \mathcal{P}^{\uparrow}(P), X \subseteq \mathcal{P}^{\uparrow}(P)$, and $(\forall S' \in X.\ S \leq S')$
2 First, we want to show $S \leq \bigwedge X$
3 This is equivalent to showing $\forall p.\ p \in S \supset p \in \bigwedge X$
4 Assume $p \in S$
5 We want to show $p \in \bigwedge X$, so we want to show $\forall S' \in X.\ p \in S'$.
6 Assume $S' \in X$
7 From the hypothesis in 1, we know $S \leq S'$
8 This means $\forall p.\ p \in S \supset p \in S'$
9 Instantiate the quantifier with p and use hypothesis 4 to conclude $p \in S'$
10 Therefore, $\forall p.\ p \in S \supset p \in \bigwedge X$, so $S \leq \bigwedge X$

Second, we want to show $\forall S' \in X.\ \bigwedge X \leq S'$
Assume $S' \in X$
We want to show $\bigwedge X \leq S'$, so we must show $\forall p.\ p \in \bigwedge X \supset p \in S'$
Assume $p \in \bigwedge X$
Therefore, we know $\forall S' \in X.\ p \in S'$
Instantiate the quantifier with S' to conclude $p \in S'$
Therefore $\bigwedge X \leq S'$
Therefore $\forall S' \in X.\ \bigwedge X \leq S'$

□

Lemma 15. *(Joins in the algebra of specifications) If $X \subseteq \mathcal{P}^{\uparrow}(P)$, then $\bigvee X$ defines an arbitrary join.*

Proof. First, we need to verify the join we defined actually gives us an upward closed set — that is, if $X \subseteq \mathcal{P}^{\uparrow}(P)$, then $\bigvee X \in \mathcal{P}^{\uparrow}(P)$

Assume $X \subseteq \mathcal{P}^{\uparrow}(P)$
We want to show $\bigvee X \in \mathcal{P}^{\uparrow}(P)$
This means for all $x \in \bigvee X, y \in P$, if $x \preceq y$ then $y \in \bigvee X$
Assume $x \in \bigvee X$, $y \in P$, $x \preceq y$
Since $x \in \bigvee X$, we know $\exists S \in X.\ x \in S$
Let S be the witness of the existential, so $S \in X$ and $x \in S$
Since S is upward closed, $x \in S$, and $x \preceq y$, we know $y \in S$
Therefore we can conclude $\exists S \in X.\ y \in S$
Therefore $y \in \bigvee X$

Now, we need to show the Heyting axioms for disjunction. Formally stated, it is $\forall S \in \mathcal{P}^{\uparrow}(P), X \subseteq \mathcal{P}^{\uparrow}(P)$, if $(\forall S' \in X.\ S' \leq S)$, then $\bigvee X \leq S$ and $\forall S' \in X.\ S' \leq \bigvee X$.

Assume $S \in \mathcal{P}^{\uparrow}(P), X \subseteq \mathcal{P}^{\uparrow}(P)$, and $\forall S' \in X.\ S' \leq S$
First, we want to show $\bigvee X \leq S$
This is the same as $\forall p, p \in \bigvee X \supset p \in S$
Assume $p \in \bigvee X$
This means $\exists S' \in X.\ p \in S'$
Let S' be the witness to the existential, so $S' \in X$ and $p \in S'$
Instantiating the quantifier in the hypothesis with S', we get $S' \leq S$
This means $\forall p \in S', p \in S$
Instantiating the quantifier with p, we get $p \in S$
Therefore $\forall p, p \in \bigvee X \supset p \in S$
This is equivalent to $\bigvee X \leq S$

Second, we want to show $\forall S' \in X.\ S' \leq \bigvee X$
Assume $S' \in X$
We want to show $S' \leq \bigvee X$

15 This means $\forall p \in S', p \in \bigvee X$
16 Assume $p \in S'$
17 We want to show $p \in \bigvee X$
18 This means we must show $\exists S' \in X.\ p \in S'$
19 Witness the existential with S', so we can show $p \in S'$ by hypothesis
20 Therefore $\forall S' \in X.\ S' \leq \bigvee X$

□

Lemma 16. *(Implication) If $S_1, S_2 \in \mathcal{P}^{\uparrow}(P)$, then $S_1 \supset S_2 \in \mathcal{P}^{\uparrow}(P)$.*
Proof. First, we will check that the definition gives us an upward closed set.

1 Assume $x \in S_1 \supset S_2$, and that $y \succeq x$
2 From this, we know $\forall r' \succeq x$, if $r' \in S_1$ then $r' \in S_2$
3 We want to show $\forall r' \succeq y$, if $r' \in S_1$ then $r' \in S_2$
4 Assume $r' \succeq y$ and $r' \in S_1$
5 Since $r' \succeq y$ and $y \succeq x$, we know $r' \succeq x$
6 From this and $r' \in S_1$, we can use the hypothesis in line 2 to get $r' \in S_2$
7 Therefore $\forall r' \succeq y$, if $r' \in S_1$ then $r' \in S_2$
8 Therefore $y \in S_1 \supset S_2$

Now that we know that $\supset$ has the correct codomain, we need to verify that it satisfies the adjoint relationship between conjunction and implication:

$$S_1 \wedge S_2 \subseteq R \iff S_1 \subseteq S_2 \supset R$$

This is equivalent to showing that

$$(\forall x.\ x \in S_1 \wedge S_2 \Rightarrow x \in R) \iff (\forall x.\ x \in S_1 \Rightarrow x \in S_2 \supset R)$$

First, let's show the $\Rightarrow$ direction.

Assume for all $x.\ x \in S_1 \wedge S_2 \Rightarrow x \in R$	(1)
Assume $x \in S_1$	(2)
Assume $r' \succeq x$	(3)
Assume $r' \in S_2$	(4)
$r' \in S_1$	Since $x \in S_1$ and $r' \succeq x$
$r' \in S_1 \cap S_2$	Since $r' \in S_1$ and $r' \in S_2$
$r' \in R$	By assumption (1)
$r' \in S_2 \Rightarrow r' \in R$	Implication intro (4)
$\forall r' \succeq x.\ r' \in S_2 \Rightarrow r' \in R$	Universal intro (3)
$x \in \{r \in P \mid \forall r' \succeq r.\ r' \in S_2 \Rightarrow r' \in R$	Comprehension intro
$x \in S_2 \supset R$	Definition of $\supset$
$\forall x.\ x \in S_1. \Rightarrow x \in S_2 \supset R$	Universal intro (2)
$(\forall x.\ x \in S_1 \wedge S_2 \Rightarrow x \in R) \Rightarrow (\forall x.\ x \in S_1 \Rightarrow x \in S_2 \supset R)$	Implication intro (1)

Next, let's show the $\Leftarrow$ direction.

Assume $\forall x.\ x \in S_1 \Rightarrow x \in S_2 \supset R$	(1) Assumption
Assume $x \in S_1 \wedge S_2$	(-) Assumption
$x \in S_1$	(2) Since $x \in S_1 \wedge S_2$
$x \in S_2$	(3) Since $x \in S_1 \wedge S_2$
$x \in S_2 \supset R$	(-) By (2) and (1)
$x \in \{r \in P \mid \forall r' \succeq r.$ if $r' \in S_2$ then $r' \in R\}$	(-) Definition of $\supset$
$\forall r' \succeq x.$ if $r' \in S_2$ then $r' \in R$	(4) Comprehension instantiation
$x \succeq x$	(5) Reflexivity
if $x \in S_2$ then $x \in R$	(-) Instantiation of (4) with (5)
$x \in R$	(-) Implication elim via (3)
$\forall x.\ x \in S_1 \wedge S_2 \Rightarrow x \in R$	(-) Universal/Implication intro (2)
$(\forall x.\ x \in S_1 \Rightarrow x \in S_2 \supset R) \Rightarrow (\forall x.\ x \in S_1 \wedge S_2 \Rightarrow x \in R)$	(-) Implication intro (1)

□

These lemmas establish that $\mathcal{P}^{\uparrow}(P)$ forms a complete Heyting algebra.

3.4.3 Defining Kripke Hoare Triples

In this section, we will define *Kripke Hoare triples*, which will be the basic elements we will use to define our specification logic. As always, the definition will come in a piece-wise fashion. Having already defined "basic Hoare triples", we can then use these to define our true Kripke Hoare triples.

First, note that our separation logic assertions form a BI-algebra, and hence form a world preorder. This then gives rise to a Heyting algebra $\mathcal{P}^{\uparrow}(\mathcal{P}(H))$, which we will take to be our domain of specifications.

Given an assertion $p \in \mathcal{P}(H)$, an element c of the domain of A-commands $(A \to K) \to K$, and an assertion $q \in U(A) \to \mathcal{P}(H)$, we define the meaning of a Kripke triple as:

$$\{p\}c\{a : A.\ q(a)\} \triangleq \{r \in \mathcal{P}(H) \mid \forall s \succeq r.\ [p * s]\ c\ [a : A.\ q(a) * s]\}$$

The intuition behind this definition is that the meaning of a specification is the set of assertions which can be framed onto it, and hence a true specification allows anything to be framed onto it, since the topmost element of the specification lattice is the set of all assertions. We confirm that Kripke triples are indeed elements of the specification lattice below.

Lemma 17. *(Kripke Triples are Specifications) For suitable p, c, A, and q, we have that*

$$\{p\}c\{a : A.\ q(a)\} \in \mathcal{P}^{\uparrow}(\mathcal{P}(H))$$

Proof.

1 We want to show $\{p\}c\{a : A.\ q(a)\} \in \mathcal{P}^{\uparrow}(\mathcal{P}(H))$
2 This is equivalent to $\forall r, s$ if $r \in \{p\}c\{a : A.\ q(a)\}$ and $s \succeq r$, then $s \in \{p\}c\{a : A.\ q\}$
3 Assume $r, s, r \in \{p\}c\{a : A.\ q(a)\}$, and $s \succeq r$
4 $\quad r \in \{p\}c\{a : A.\ q(a)\}$ is equivalent to $\forall s \succeq r.\ [p * s]\ c\ [a : A.\ q(a) * s]$

5 We want to show $s \in \{p\}c\{a : A.\ q\}$
6 This is equivalent to showing $\forall t \succeq s.\ [p * t]\ c\ [a : A.\ q(a) * t]$
7 Assume t, $t \succeq s$
8 By transitivity with $t \succeq s$ and $s \succeq r$, we know $t \succeq r$
9 Instantiating quantifier in line 4 with t, $[p * t]\ c\ [a : A.\ q(a) * t]$
10 Therefore $\forall t \succeq s.\ [p * t]\ c\ [a : A.\ q(a) * t]$
11 Therefore $s \in \{p\}c\{a : A.\ q\}$
12 Therefore $\forall r, s$ if $r \in \{p\}c\{a : A.\ q(a)\}$ and $s \succeq r$, then $s \in \{p\}c\{a : A.\ q\}$
13 We have shown $\{p\}c\{a : A.\ q(a)\} \in \mathcal{P}^{\uparrow}(\mathcal{P}(H))$

□

Fixed Point Induction

The reason we have gone to the trouble of using continuous approximations to the postcondition is to create an admissibility property which will allow us to justify a fixed point induction rule. We will now cash in that work by giving a proof that the fixed point induction rule is sound.

Lemma 18. *(Bottom Satisfies All Specifications) We have that* $\{p\}\bot\{a : A.\ q(a)\} = \mathcal{P}(H)$.

Proof.

1 We want to show $\{p\}\bot\{a : A.\ q(a)\} = \mathcal{P}(H)$
2 It suffices to show $\forall r \in \mathcal{P}(H), r \in \{p\}\bot\{a : A.\ q(a)\}$
3 Assume $r \in \mathcal{P}(H)$
4 We want to show $r \in \{p\}\bot\{a : A.\ q(a)\}$, which is equivalent to $\forall s \succeq r.\ [p * s] \bot [a : A.\ q(a) * s]$
5 Assume $s \succeq r$
6 We want to show $[p * s] \bot [a : A.\ q(a) * s]$
7 This is equivalent to $\forall h \in p.\bot\ Best(q)\ h = \bot$
8 Assume $h \in p$
9 By definition of least element, $\bot\ Best(q)\ h = \bot$
10 Therefore $\forall h \in p.\bot\ Best(q)\ h = \bot$
11 Therefore $[p * s] \bot [a : A.\ q(a) * s]$
12 Therefore $\forall s \succeq r.\ [p * s] \bot [a : A.\ q(a) * s]$
13 Therefore $r \in \{p\}\bot\{a : A.\ q(a)\}$
14 Therefore $\forall r \in \mathcal{P}(H), r \in \{p\}\bot\{a : A.\ q(a)\}$
15 Therefore $\{p\}\bot\{a : A.\ q(a)\} = \mathcal{P}(H)$

□

Lemma 19. *(Admissibility of Triple Subsets) Define* $\{p\}-\{a : A.\ q(a)\}$ *to be*

$$\{p\}-\{a : A.\ q(a)\} \triangleq \{c \in (A \to K) \to K \mid \{p\}c\{a : A.\ q(a)\} = \mathcal{P}(H)\}$$

Then, $\{p\}-\{a : A.\ q(a)\}$ *forms an admissible subset of* $(A \to K) \to K$*. That is, given a chain* $c_i \in \{p\}-\{a : A.\ q(a)\}$*, we know that* $\{p\}\sqcup_i c_i\{a : A.\ q(a)\}$.

Proof.

Suppose we have a chain $c_i \in \{p\}-\{a : A.\ q(a)\}$.
  We want to show that $\sqcup_i c_i \in \{p\}-\{a : A.\ q(a)\}$
  This is equivalent to $\{p\}\sqcup c_i\{a : A.\ q(a)\} = \mathcal{P}(H)$
  This is equivalent to $\forall r \in \mathcal{P}(H), s \succeq r.\ [p * s] \sqcup c_i\ [a : A.\ q(a) * s]$
  Assume $r \in \mathcal{P}(H), s \succeq r$
    We want to show $[p * s] \sqcup c_i\ [a : A.\ q(a) * s]$
    This is equivalent to $\forall h \in p * s.\ (\sqcup c_i)\ Best(\lambda a.\ q(a) * s)\ h = \bot$
    Assume $h \in p * s$
      By continuity, we know $(\sqcup c_i)\ Best(\lambda a.\ q(a) * s)\ h = \bigsqcup(c_i\ Best(\lambda a.\ q(a) * s)\ h)$
      Suppose c is an element of the chain of c_i
        Then we know $c \in \{p\}-\{a : A.\ q\}$
        This is equivalent to $\{p\}c\{a : A.\ q\} = \mathcal{P}(H)$
        This is equivalent to $\forall r, s \succeq r.\ [p * s]\ c\ [a : A.\ q(a) * s]$
        This is equivalent to $\forall r, s \succeq r, h \in p * s, c\ Best(\lambda a.\ q(a) * s)\ h = \bot$
        Instantiating quantifiers with r, s, and h, we get $c\ Best(\lambda a.\ q(a) * s)\ h = \bot$
      Therefore $\forall c \in \{c_i \mid i \in \mathbb{N}\}, c\ Best(\lambda a.\ q(a) * s)\ h = \bot$
      Therefore $\bigsqcup(c_i\ Best(\lambda a.\ q(a) * s)\ h) = \bot$
      Therefore $(\sqcup c_i)\ Best(\lambda a.\ q(a) * s)\ h = \bot$
    Therefore $\forall h \in p * s.\ (\sqcup c_i)\ Best(\lambda a.\ q(a) * s)\ h = \bot$
    Therefore $[p * s] \sqcup c_i\ [a : A.\ q(a) * s]$
  Therefore $\forall r \in \mathcal{P}(H), s \succeq r.\ [p * s] \sqcup c_i\ [a : A.\ q(a) * s]$
  Therefore $\{p\}\sqcup c_i\{a : A.\ q(a)\} = \mathcal{P}(H)$
  Therefore $\sqcup_i c_i \in \{p\}\sqcup_i c_i\{a : A.\ q(a)\}$

□

Lemma 20. *(Fixed Point Induction) If we know that for all x, $\{p\}x\{a : A.\ q(a)\} = \mathcal{P}(H)$ implies $\{p\}f(x)\{a : A.\ q(a)\}) = \mathcal{P}(H)$, then we know that $\{p\}fix(f)\{a : A.\ q(a)\} = \mathcal{P}(H)$*

Proof. First, observe that $f^n(\bot)$ forms a chain – that is, for all i, $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$.

We want to show $\forall i,\ f^i(\bot) \sqsubseteq f^{i+1}(\bot)$
We proceed by induction on i
  Case $i = 0$:
    We want to show $\bot \sqsubseteq f(\bot)$
    This follows immediately from the fact that $\bot$ is the least element of a domain.
  Case $i = j + 1$
    We want to show $f^j(\bot) \sqsubseteq f^{j+1}(\bot) \supset f^i(\bot) \sqsubseteq f^{i+1}(\bot)$
    Assume $f^j(\bot) \sqsubseteq f^{j+1}(\bot)$
      By monotonicity of f, $f(f^j(\bot)) \sqsubseteq f(f^{j+1}(\bot))$
      Therefore $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$
Therefore $\forall i,\ f^i(\bot) \sqsubseteq f^{i+1}(\bot)$

Now, observe that for every n, $f^n(\bot) \in \{p\}-\{a : A.\ q\}$.

1 Assume for all x, $\{p\}x\{a : A.\ q(a)\} = \mathcal{P}(H)$ implies $\{p\}f(x)\{a : A.\ q(a)\}) = \mathcal{P}(H)$
2 We want to show $\forall n, f^n(\bot) \in \{p\}{-}\{a : A.\ q(a)\}$
3 We proceed by induction on n:
4 Case $n = 0$
5 We want to show $\bot \in \{p\}{-}\{a : A.\ q(a)\}$
6 This follows from the fact that bottom satisfies all specifications.
7 Case $n = m + 1$
8 We want to show $f^m(\bot) \in \{p\}{-}\{a : A.\ q\} \supset f^{m+1}(\bot) \in \{p\}{-}\{a : A.\ q\}$
9 Assume $f^m(\bot) \in \{p\}{-}\{a : A.\ q(a)\}$
10 This means $\{p\}f^m(\bot)\{a : A.\ q(a)\} = \mathcal{P}(H)$
11 Instantiate line 1 with $f^m(\bot)$, to conclude $\{p\}f(f^m(\bot))\{a : A.\ q(a)\}) = \mathcal{P}(H)$
12 This means $\{p\}f^{m+1}(\bot)\{a : A.\ q(a)\} = \mathcal{P}(H)$
13 This means $f^{m+1}(\bot) \in \{p\}{-}\{a : A.\ q(a)\}$
14 Therefore $\forall n, f^n(\bot) \in \{p\}{-}\{a : A.\ q(a)\}$

Finally, by the admissibility of $\{p\}{-}\{a : A.\ q(a)\}$, we know that $\sqcup f^n(\bot) \in \{p\}{-}\{a : A.\ q(a)\}$. Since $\sqcup f^n(\bot) = fix(f)$, we know that $\{p\}fix(f)\{a : A.\ q(a)\} = \mathcal{P}(H)$. $\square$

3.4.4 The Framing Operator

One nice feature of the Kripke-style interpretation of specifications is that it naturally validates higher-order frame rules. We define the operation $S \otimes p$, where $S \in \mathcal{P}^{\uparrow}(W(\mathcal{P}(H)))$ and $p \in W(\mathcal{P}(H))$:

$$S \otimes p = \{r \in W(\mathcal{P}(H)) \mid r * p \in S\}$$

The way to understand this is that $S \otimes p$ restricts S to only those elements which can be extended by p. That is, if we think of S as the set of propositions that can be framed on to a basic triple, then we want $S \otimes p$ to be only the frames in S, such that if we added p to them we continue to have a frame in S. So this operation gives a semantic interpretation of the frame rule.

So we need to show that first, $S \otimes p$ actually is an element of our Heyting algebra of specification truth values, and second, we want the frame rule to be sound – we want S to always imply $S \otimes p$.

Lemma 21. *(Framing is a Lattice Operation on Specifications) For all S and p, we have that $S \otimes p$ is in $\mathcal{P}^{\uparrow}(W(\mathcal{P}(H)))$, and that $S \subseteq S \otimes p$.*

Proof. To show that $S \otimes p \in \mathcal{P}^{\uparrow}(W(\mathcal{P}(H)))$, we need to show that for all x, y, if $x \in S \otimes p$ and $y \succeq x$, then $y \in S \otimes p$.

1 We want to show for all x, y, if $x \in S \otimes p$ and $y \succeq x$, then $y \in S \otimes p$
2 Assume $x, y, x \in S \otimes p, y \succeq x$
3 From $x \in S \otimes p$, we know $x * p \in S$
4 From $y \succeq x$, we know $\exists r.\ y = x * r$
5 Let r be the witness so that $y = x * r$
6 Since S is upward closed, $x * p * r \in S$

7 Since $x * p * r = (x * r) * p$, we know $y * p \in S$
8 Therefore $y \in S \otimes p$

To show $S \subseteq S \otimes p$, we need to show that if $x \in S$, then $x \in S \otimes p$.

1 Assume $x \in S$
2 Since $\preceq$ is the extension ordering, $x \preceq x * p$
3 Since S is upward closed, $x * p \in S$
4 Therefore $x \in S \otimes p$
□

Framing Commutes With Logical Operators

Lemma 22. *(Framing onto Kripke Triples) We have that*

$$\{p\}c\{a : A.\ q(a)\} \otimes r = \{p * r\}c\{a : A.\ q(a) * r\}$$

Proof.

1 We want to show $\{p\}c\{a : A.\ q(a)\} \otimes r = \{p * r\}c\{a : A.\ q(a) * r\}$.
2 This means $\forall s \in W(\mathcal{P}(H)).\ s \in (\{p\}c\{a : A.\ q(a)\} \otimes r)$ if and only if $s \in \{p * r\}c\{a : A.\ q(a) * r\}$.
3 Assume $s \in W(\mathcal{P}(H))$
4 $\Rightarrow$ direction:
5 Assume $s \in (\{p\}c\{a : A.\ q(a)\} \otimes r)$
6 This means $s * r \in \{p\}c\{a : A.\ q(a)\}$
7 This means $\forall t \succeq s * r.\ [p * t]\ c\ [a : A.\ q * t]$
8 We want to show $s \in \{p * r\}c\{a : A.\ q(a) * r\}$
9 So we want $\forall t' \succeq s.\ [p * r * t']\ c\ [a : A.\ q(a) * r * t']$
10 Assume $t' \succeq s$
11 Clearly, $t' * r \succeq s * r$
12 Instantiate quantifier in 7 with $t' * r$ to conclude $[p * t' * r]\ c\ [a : A.\ q * t' * r]$
13 Rearranging, we get $[p * r * t']\ c\ [a : A.\ q * r * t']$
14 Therefore $\forall t' \succeq s.\ [p * r * t']\ c\ [a : A.\ q(a) * r * t']$
15 Therefore $s \in \{p * r\}c\{a : A.\ q(a) * r\}$
16 $\Leftarrow$ direction:
17 Assume $s \in \{p * r\}c\{a : A.\ q(a) * r\}$.
18 This means $\forall t \succeq s.\ [p * r * t]\ c\ [a : A.\ q(a) * r * t]$
19 We want to show $s \in (\{p\}c\{a : A.\ q(a)\} \otimes r)$
20 So we want $s * r \in \{p\}c\{a : A.\ q(a)\}$
21 So we want $\forall t \succeq s * r.\ [p * t]\ c\ [a : A.\ q * t]$
22 Assume $t \succeq s * r$
23 Since $t \succeq s * r$, we know $\exists u.\ t = s * r * u$
24 Let u be the witness such that $t = s * r * u$
25 Now, note that $s * u \succeq s$

26 Instantiate quantifier in 18 with $s * u$, so $[p * r * s * u]\ c\ [a : A.\ q(a) * r * s * u]$
27 Rearranging, $[p * s * r * u]\ c\ [a : A.\ q(a) * s * r * u]$
28 By equality in 24, $[p * t]\ c\ [a : A.\ q * t]$
29 Therefore $\forall t \succeq s * r.\ [p * t]\ c\ [a : A.\ q * t]$
30 Therefore $s * r \in \{p\}c\{a : A.\ q(a)\}$
31 Therefore $s \in (\{p\}c\{a : A.\ q(a)\} \otimes r)$
□

Lemma 23. *(Framing Commutes with Meets) We have that*

$$\left(\bigwedge_{i\in I} S_i\right) \otimes p = \bigwedge_{i\in I}(S_i \otimes p)$$

Proof. To show $\left(\bigwedge_{i\in I} S_i\right) \otimes p = \bigwedge_{i\in I}(S_i \otimes p)$, we need to show $\forall r.\ r \in \left(\bigwedge_{i\in I} S_i\right) \otimes p$ if and only if $r \in \bigwedge_{i\in I}(S_i \otimes p)$.

1 Assume $r \in W(\mathcal{P}(H))$
2 $\Rightarrow$ direction:
3 Assume $r \in \left(\bigwedge_{i\in I} S_i\right) \otimes p$
4 From assumption, we know $r * p \in \bigwedge_{i\in I} S_i$
5 This means that $\forall i \in I.\ r * p \in S_i$
6 We want to show $r \in \bigwedge_{i\in I}(S_i \otimes p)$
7 So we want $\forall i \in I.\ r \in (S_i \otimes p)$
8 So we want $\forall i \in I.\ r * p \in S_i$
9 Assume $i \in I$
10 We want to show $r * p \in S_i$
11 Instantiating line 5 with i, we get $r * p \in S_i$
12 $\Leftarrow$ direction:
13 Assume $r \in \bigwedge_{i\in I}(S_i \otimes p)$
14 From this, we know that $\forall i \in I.\ r \in (S_i \otimes p)$
15 We want to show $r \in \left(\bigwedge_{i\in I} S_i\right) \otimes p$
16 So we want to show $r * p \in \bigwedge_{i\in I} S_i$
17 So we want to show $\forall i \in I.\ r * p \in S_i$
18 Assume $i \in I$
19 Instantiating line 14 with i, we get $r \in (S_i \otimes p)$
20 This means $r * p \in S_i$
21 Therefore $\forall i \in I.\ r * p \in S_i$
22 Therefore $r * p \in \bigwedge_{i\in I} S_i$
23 Therefore $r \in \left(\bigwedge_{i\in I} S_i\right) \otimes p$
□

Lemma 24. *(Framing Commutes with Joins) We have that*

$$\left(\bigvee_{i\in I} S_i\right) \otimes p = \bigvee_{i\in I}(S_i \otimes p)$$

Proof. Showing this is equivalent to showing $\forall r.\ r \in \left(\bigvee_{i \in I} S_i\right) \otimes p$ if and only if $r \in \bigvee_{i \in I}(S_i \otimes p)$.

Assume r
  $\Rightarrow$ direction:
    Assume $r \in \left(\bigvee_{i \in I} S_i\right) \otimes p$
    This means $r * p \in \bigvee_{i \in I} S_i$
    This means $\exists i \in I.\ r * p \in S_i$
    We want to show $r \in \bigvee_{i \in I}(S_i \otimes p)$
    So we want to show $\exists i \in I.\ r \in (S_i \otimes p)$
    Let i be the witness in 5, such that $r * p \in S_i$
      From this, we see $r \in (S_i \otimes p)$
      From this and i, we conclude $\exists i \in I.\ r \in (S \otimes p)$
    Therefore $r \in \bigvee_{i \in I}(S_i \otimes p)$
  $\Leftarrow$ direction:
    Assume $r \in \bigvee_{i \in I}(S_i \otimes p)$.
      From this, $\exists i \in I.\ r \in (S_i \otimes p)$
      We want to show $r \in \left(\bigvee_{i \in I} S_i\right) \otimes p$
      So we want $r * p \in \bigvee_{i \in I} S_i$
      So we want $\exists i \in I.\ r * p \in S_i$
      Let $i \in I$ be the witness in 14, so that $r \in (S_i \otimes p)$
        From this, $r * p \in S_i$
        With this and $i \in I$, we know $\exists i \in I.\ r * p \in S_i$
      Therefore $r * p \in \bigvee_{i \in I} S_i$
      Therefore $r \in \left(\bigvee_{i \in I} S_i\right) \otimes p$

□

Lemma 25. *(Framing Commutes Through Implication) We have that*

$$(S_1 \supset S_2) \otimes p = (S_1 \otimes p) \supset (S_2 \otimes p)$$

Proof. This is equivalent to showing that $\forall r, r \in [(S_1 \supset S_2) \otimes p]$ if and only if $r \in [(S_1 \otimes p) \supset (S_2 \otimes p)]$.

Assume r
  $\Rightarrow$ direction:
    Assume $r \in [(S_1 \supset S_2) \otimes p]$
      This means $r * p \in (S_1 \supset S_2)$
      This means $\forall s \succeq r * p$, if $s \in S_1$ then $s \in S_2$
      We want to show $r \in [(S_1 \otimes p) \supset (S_2 \otimes p)]$
      So we want $\forall s \succeq r$, if $s \in S_1 \otimes p$ then $s \in S_2 \otimes p$
      Assume $s \succeq r$ and $s \in S_1 \otimes p$
        From this $s * p \in S_1$

10 Since $s \succeq r$, we have $s * p \succeq r * p$
11 Instantiating line 5 with $s * p$, we have if $s * p \in S_1$ then $s * p \in S_2$
12 Using this and line 9, we have $s * p \in S_2$
13 From this, we have $s \in S_2 \otimes p$
14 Therefore $\forall s \succeq r$, if $s \in S_1 \otimes p$ then $s \in S_2 \otimes p$
15 Therefore $r \in [(S_1 \otimes p) \supset (S_2 \otimes p)]$
16 $\Leftarrow$ direction:
17 Assume $r \in [(S_1 \otimes p) \supset (S_2 \otimes p)]$
18 From this, $\forall s \succeq r$, if $s \in (S_1 \otimes p)$, then $s \in (S_2 \otimes p)$
19 We want to show $r \in [(S_1 \supset S_2) \otimes p]$
20 So we want $r * p \in (S_1 \supset S_2)$
21 So we want $\forall s \succeq r * p$, if $s \in S_1$ then $s \in S_2$
22 Assume $s \succeq r * p$ and $s \in S_1$
23 From this, $\exists t.\ s = t * r * p$
24 Let t be the witness such that $s = t * r * p$
25 Note $t * r \succeq r$
26 Instantiating 18 with $t * r$, we get if $t * r \in (S_1 \otimes p)$, then $t * r \in (S_2 \otimes p)$
27 From this, we have if $t * r * p \in S_1$ then $t * r * p \in S_2$
28 So we have if $s \in S_1$, then $s \in S_2$
29 From this and 22, we have $s \in S_2$
30 Therefore $\forall s \succeq r * p$, if $s \in S_1$ then $s \in S_2$
31 Therefore $r * p \in (S_1 \supset S_2)$
32 Therefore $r \in [(S_1 \supset S_2) \otimes p]$

□

3.5 Syntax of Assertions and Specifications

In this section, we will give the syntax of specifications and assertions, and then we will give their interpretations. The syntactic categories are given in Figure 3.1. The sorts of our logic are ranged over by ω, and include the kinds κ, the polymorphic types A, and the propositional sorts υ. The propositional sorts ω include prop, the sort $\omega \Rightarrow \upsilon$, which are the sort of propositional functions, and the sort $\Pi\alpha : \kappa.\ \upsilon$, which are type-constructor-indexed families.

Note that we syntactically identify a family of propositional sorts υ. These sorts all end in prop, and by distinguishing them from kinds κ and types A, we forbid the formation of sorts like $\mathsf{prop} \Rightarrow A$. This will ensure that program terms will never depend on purely logical facts, though the converse (logical terms depending on program terms) is allowed. This restriction is a slight variation of the usual convention in higher-order logic, where sorts must bottom out in the assertion type.

However, general sorts ω include both types and kinds. This lets us define the assertions we will need for asserting facts about polymorphic programs. For example, the sort of the list predicate for polymorphic lists can be given as $\mathsf{listprop} : \Pi\alpha : \star.\ \mathsf{list}\ \alpha \Rightarrow \mathsf{seq}\ \alpha \Rightarrow \mathsf{prop}$.

The terms (for which we will use $p\ q$ as metavariables) which are categorized by our sorts are also given in Figure 3.1. They include lambda-abstraction and application for the two function

space sorts $\omega \Rightarrow \upsilon$ and $\Pi\alpha : \kappa.\ \upsilon$, terms e for the sorts A, type expressions τ for the sorts κ.

Finally, we also have the assertions of separation logic for the sort prop. These include the usual propositional logical connectives, $\top, p \wedge q, p \supset q, \bot, p \vee q$, as well as the spatial connectives $\mathsf{emp}, p * q, p \mathrel{-\!\!*} q$, and $e \mapsto_A e'$. Note that the points-to proposition is typed; it indicates that e is a reference of type $\mathsf{ref}\ A$ with contents e' of type A.

The quantifiers $\forall u : \omega.\ p$ and $\exists u : \omega.\ p$ are higher-order quantifiers. They can range over all sorts, including the sort of assertions prop, and so we have the full power of higher-order separation logic available. One way in which we will use this expressive power is by taking advantage of the definability of many mathematical types (such as numbers, sequences, trees, finite sets, subsets, etc.) in higher-order logic, to augment our sorts with these types on an as-needed basis.

Finally, we have the *specification embedding assertion* $S\ \mathsf{spec}$. This is an *assertion* that the specification S is true, and is useful for writing assertions that include facts about the behavior of code.

The specifications S begin with the basic Hoare triple $\{p\}c\{a : A.\ q\}$, which says that the computation c, when run from a pre-state in p, will end in a post-state in q, with its return value named by a. Similarly, we have the monadic Hoare triple form $\langle p \rangle e \langle a : A.\ q \rangle$ which says that the suspended monadic computation e (of type $\bigcirc A$), will take a pre-state p to a post-state q if it were to be run. The specification $\{p\}$ is the assertion-embedding specification, which says that p is a truth of separation logic.

This does mean that assertions and specifications are mutually recursive, which means that we will have to give the semantics of these two syntaxes simultaneously. This is one of the reasons we spent the first half of this chapter developing the semantics with no reference to the intended syntax at all — I wanted to ensure the semantic domains were well-defined before giving the mutually-recursive semantics of the program logic.

We also can form conjunctions $S\ \&\ S'$, disjunctions $S\ ||\ S'$ and implications $S \Rrightarrow S'$ over specifications, as well as universal $\forall u : \omega.\ S$ and existential $\exists u : \omega.\ S$ quantification over specifications, with the legitimate domains of quantifications being the same sorts as for the assertion language.

A propositional context Δ is a sequence of sorted variables of the form $u : \upsilon$. However, due to the fact that the sorts contain variables, we need a judgment to establish whether a sort is well-formed with respect to a context of type constructor variables.

In Figure 3.2, we give the judgment $\Theta \rhd \omega : \mathsf{sort}$ used to decide whether or not a given sort is well-formed or not. The judgment $\Theta \rhd \Delta$, also given in Figure 3.2, then uses the well-sorting judgment to establish whether a particular context is well-formed or not.

Finally, we need an equality judgment for sorts, since we have an equality theory for types. The judgment $\Theta \rhd \omega \equiv \omega' : \mathsf{sort}$, defined in Figure 3.2, judges whether two sorts are equal, by means of an almost-congruence. That is, the rules of this judgment are all congruence rules, except for the single case SORTEQTYPE, which inherits the equality judgment for polytypes defined in the previous chapter.

Now that we have defined what the sorts are, we define what it means for a term to be well-sorted in context with the judgment $\Theta; \Gamma; \Delta \rhd p : \omega$, defined in Figure 3.3.

In the rules TTYPE and TEXPR, we simply inherit well-kindedness and well-typedness from the corresponding judgments for types and terms, defined in the previous chapter. The rule THYP

Propositional Sorts	υ	$::=$	$\mathsf{prop} \mid \omega \Rightarrow \upsilon \mid \Pi\alpha : \kappa.\ \upsilon$
Sorts	ω	$::=$	$\upsilon \mid \kappa \mid A$
Terms	p, q	$::=$	$u \mid \hat{\lambda} u : \omega.\ p \mid \hat{\underline{\lambda}} \alpha : \kappa.\ p \mid p\ q \mid p\ [\tau] \mid \tau \mid e$
		$\mid$	$\top \mid p \wedge q \mid p \supset q \mid \bot \mid p \vee q$
		$\mid$	$\mathsf{emp} \mid p * q \mid p \mathrel{-\!\!*} q \mid p \mapsto_A q$
		$\mid$	$\forall u : \omega.\ p \mid \exists u : \omega.\ p \mid p =_\omega q \mid S\ \mathsf{spec}$
Specifications	S	$::=$	$\{p\} c \{a : A.\ q\} \mid \langle p \rangle e \langle a : A.\ q \rangle \mid \{p\}$
		$\mid$	$S\ \&\ S' \mid S \Rrightarrow S' \mid S \mid\mid S'$
		$\mid$	$\forall u : \omega.\ S \mid \exists u : \omega.\ S$
Propositional Contexts	Δ	$::=$	$\cdot \mid \Delta, u : \upsilon$

Figure 3.1: Syntax of Assertions and Specifications

is the hypothesis rule for propositional variables. (Type and program expression variables can only be referenced through the TTYPE and TEXPR rules.)

lambda-abstraction and application rules for the function sort $\omega \Rightarrow \upsilon$ – there are no surprises here. Likewise, the TABSALL and TAPPALL rules allow abstracting applying kind-indexed products.

Finally, there are all the rules giving the sorting of propositions. The nullary propositions $\top, \bot$, and emp are typed with the TCONST rule, and the binary propositions $\wedge, \vee, \supset, *$, and $\mathrel{-\!\!*}$ are typed with the TBINARY rule, requiring their two arguments to both be of sort prop.

The two quantifiers $\forall u : \omega.\ p$ and $\exists u : \omega.\ p$ are sorted with the TQUANTIFY rules. We have three variants of this rule, putting a new variable into different contexts depending on whether the variable is a type, term or propositional variable. The points-to $e \mapsto_A e'$ and equality $p =_\omega q$ each require that their arguments be of the correct sort. Note that $e \mapsto_A e'$ is restricted to program types, as expected, whereas equality is permitted at any sort.

Finally, we have the rule TSPEC for the specification-embedding assertion $S\ \mathsf{spec}$, which recursively invokes the well-sorted specification $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$. This judgment is defined in Figure 3.4, and consists of a handful of rules. The SPECTRIPLE rule asserts that $\{p\} c \{a : A.\ q\}$ is well-kinded when A is a type, p is an assertion, c is a computation yielding an A, and q is an assertion with a as an extra free variable. Likewise the SPECMTRIPLE rule does the same job for monadic expressions, saying that $\langle p \rangle e \langle a : A.\ q \rangle$, saying that e must be a term of monadic type $\bigcirc A$, but otherwise as in the SPECTRIPLE rule. Finally, the remaining atomic proposition SPECASSERT rule recursively calls back into the assertion well-kinding judgment.

The SPECBINARY rules gives well-formedness conditions for the conjunction ($S\ \&\ S'$), disjunction ($S \mid\mid S'$), and implication ($S \Rrightarrow S'$) over specifications, in each case asking the subterms to be well-formed specifications. The quantifier rules SPECQUANTIFY simply extend the context with the newly quantified variable. As with assertions, we have three versions of this

$$\boxed{\Theta \rhd \omega : \mathsf{sort}}$$

$$\frac{}{\Theta \rhd \kappa : \mathsf{sort}}\ \textsc{SortKind} \qquad \frac{\Theta \vdash A : \bigstar}{\Theta \rhd A : \mathsf{sort}}\ \textsc{SortA} \qquad \frac{}{\Theta \rhd \mathsf{prop} : \mathsf{sort}}\ \textsc{SortProp}$$

$$\frac{\Theta \rhd \omega : \mathsf{sort} \qquad \Theta \rhd \upsilon : \mathsf{sort}}{\Theta \rhd \omega \Rightarrow \upsilon : \mathsf{sort}}\ \textsc{SortImp} \qquad \frac{\Theta, \alpha : \kappa \rhd \upsilon : \mathsf{sort}}{\Theta \rhd \Pi\alpha : \kappa.\ \upsilon : \mathsf{sort}}\ \textsc{SortImp}$$

$$\boxed{\Theta \rhd \Delta}$$

$$\frac{}{\Theta \rhd \cdot}\ \textsc{SortCtxNil} \qquad \frac{\Theta \rhd \Delta \qquad \Theta \rhd \upsilon : \mathsf{sort}}{\Theta \rhd \Delta, u : \upsilon}\ \textsc{SortCtxCons}$$

$$\boxed{\Theta \rhd \omega \equiv \omega' : \mathsf{sort}}$$

$$\frac{}{\Theta \rhd \mathsf{prop} \equiv \mathsf{prop} : \mathsf{sort}}\ \textsc{SortEqProp} \qquad \frac{}{\Theta \rhd \kappa \equiv \kappa : \mathsf{sort}}\ \textsc{SortEqKind}$$

$$\frac{\Theta \rhd \omega \equiv \omega' : \mathsf{sort} \qquad \Theta \rhd \upsilon \equiv \upsilon' : \mathsf{sort}}{\Theta \rhd \omega \Rightarrow \upsilon \equiv \omega' \Rightarrow \upsilon' : \mathsf{sort}}\ \textsc{SortEqImp} \qquad \frac{\Theta \vdash A \equiv B : \bigstar}{\Theta \rhd A \equiv B : \mathsf{sort}}\ \textsc{SortEqType}$$

$$\frac{\Theta, \alpha : \kappa \rhd \upsilon \equiv \upsilon' : \mathsf{sort}}{\Theta \rhd \Pi\alpha : \kappa.\ \upsilon \equiv \Pi\alpha : \kappa.\ \upsilon' : \mathsf{sort}}\ \textsc{SortEqAll}$$

Figure 3.2: Well-sorting of Sorts and Contexts

$$\boxed{\Theta; \Gamma; \Delta \rhd p : \omega}$$

$$\frac{\Theta \rhd \Delta \qquad \Theta \vdash \Gamma \qquad \Theta \vdash \tau : \kappa}{\Theta; \Gamma; \Delta \rhd \tau : \kappa}\ \textsc{TType} \qquad \frac{\Theta \rhd \Delta \qquad \Theta;\ \Gamma \vdash e : A}{\Theta; \Gamma; \Delta \rhd e : A}\ \textsc{TExpr}$$

$$\frac{\Theta \rhd \Delta \qquad \Theta \vdash \Gamma \qquad u : \omega \in \Delta}{\Theta; \Gamma; \Delta \rhd u : \omega}\ \textsc{THyp} \qquad \frac{\Theta; \Gamma; \Delta, u : \upsilon' \rhd p : \upsilon}{\Theta; \Gamma; \Delta \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon}\ \textsc{TAbs1}$$

$$\frac{\Theta; \Gamma, x : A; \Delta \rhd p : \upsilon}{\Theta; \Gamma; \Delta \rhd \hat{\lambda} x : A.\ p : A \Rightarrow \upsilon}\ \textsc{TAbs2} \qquad \frac{\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \upsilon \qquad \alpha \notin \mathrm{FV}(\Gamma, \Delta, \upsilon)}{\Theta; \Gamma; \Delta \rhd \hat{\lambda} \alpha : \kappa.\ p : \kappa \Rightarrow \upsilon}\ \textsc{TAbs3}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p : \omega \Rightarrow \upsilon \qquad \Theta; \Gamma; \Delta \rhd q : \omega}{\Theta; \Gamma; \Delta \rhd p\ q : \upsilon}\ \textsc{TApp}$$

$$\frac{\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \upsilon \qquad \alpha \notin \mathrm{FV}(\Gamma, \Delta)}{\Theta; \Gamma; \Delta \rhd \underline{\hat{\lambda}} \alpha : \kappa.\ p : \Pi \alpha : \kappa.\ \upsilon}\ \textsc{TAbsAll}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p : \Pi \alpha : \kappa.\ \upsilon \qquad \Theta; \Gamma; \Delta \rhd \tau : \kappa}{\Theta; \Gamma; \Delta \rhd p\ [\tau] : [\tau/\alpha]\upsilon}\ \textsc{TAppAll}$$

$$\frac{\Theta \rhd \Delta \qquad \Theta \vdash \Gamma \qquad p \in \{\top, \bot, \mathsf{emp}\}}{\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}}\ \textsc{TConst}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p : \mathsf{prop} \qquad \Theta; \Gamma; \Delta \rhd q : \mathsf{prop} \qquad \oplus \in \{\wedge, \vee, \supset, *, \mathrel{-\!\!*}\}}{\Theta; \Gamma; \Delta \rhd p \oplus q : \mathsf{prop}}\ \textsc{TBinary}$$

$$\frac{\Theta; \Gamma; \Delta, u : \omega \rhd p : \mathsf{prop} \qquad Q \in \{\forall, \exists\}}{\Theta; \Gamma; \Delta \rhd Q u : \upsilon.\ p : \mathsf{prop}}\ \textsc{TQuantify1}$$

$$\frac{\Theta; \Gamma, x : A; \Delta \rhd p : \mathsf{prop} \qquad Q \in \{\forall, \exists\}}{\Theta; \Gamma; \Delta \rhd Q x : A.\ p : \mathsf{prop}}\ \textsc{TQuantify2}$$

$$\frac{\Theta, \alpha : \kappa; \Gamma; \Delta, u : \omega \rhd p : \mathsf{prop} \qquad Q \in \{\forall, \exists\} \qquad \alpha \notin \mathrm{FV}(\Gamma, \Delta)}{\Theta; \Gamma; \Delta \rhd Q \alpha : \kappa.\ p : \mathsf{prop}}\ \textsc{TQuantify3}$$

$$\frac{\Theta; \Gamma; \Delta \rhd e : \mathsf{ref}\ A \qquad \Theta; \Gamma; \Delta \rhd e' : A}{\Theta; \Gamma; \Delta \rhd e \mapsto_A e' : \mathsf{prop}}\ \textsc{TPointsTo}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p : \omega \qquad \Theta; \Gamma; \Delta \rhd q : \omega \qquad \Theta \rhd \omega : \mathsf{sort}}{\Theta; \Gamma; \Delta \rhd p =_\omega q : \mathsf{prop}}\ \textsc{TEqual}$$

$$\frac{\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}}{\Theta; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}}\ \textsc{TSpec} \qquad \frac{\Theta \rhd \omega \equiv \omega' : \mathsf{sort} \qquad \Theta; \Gamma; \Delta \rhd p : \omega'}{\Theta; \Gamma; \Delta \rhd p : \omega}\ \textsc{TEqSort}$$

Figure 3.3: Well-sorting of Assertions

$$\boxed{\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p : \mathsf{prop} \qquad \Theta; \Gamma; \Delta \rhd [c] : \bigcirc A \qquad \Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}}{\Theta; \Gamma; \Delta \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}}\ \textsc{SpecTriple}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p : \mathsf{prop} \qquad \Theta; \Gamma; \Delta \rhd e : \bigcirc A \qquad \Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}}{\Theta; \Gamma; \Delta \rhd \langle p \rangle e \langle a : A.\ q \rangle : \mathsf{spec}}\ \textsc{SpecMTriple}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}}{\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}}\ \textsc{SpecAssert}$$

$$\frac{\Theta; \Gamma; \Delta, u : \upsilon \rhd S : \mathsf{spec} \qquad u \in \{\forall, \exists\}}{\Theta; \Gamma; \Delta \rhd Q u : \upsilon.\ S : \mathsf{spec}}\ \textsc{SpecQuantify1}$$

$$\frac{\Theta, \alpha : \kappa, \Gamma, \Delta \rhd S : \mathsf{spec} \qquad u \in \{\forall, \exists\} \qquad \alpha \notin \mathrm{FV}(\Gamma, \Delta)}{\Theta; \Gamma; \Delta \rhd Q \alpha : \kappa.\ S : \mathsf{spec}}\ \textsc{SpecQuantify2}$$

$$\frac{\Theta; \Gamma, x : A; \Delta \rhd S : \mathsf{spec} \qquad u \in \{\forall, \exists\}}{\Theta; \Gamma; \Delta \rhd Q x : A.\ S : \mathsf{spec}}\ \textsc{SpecQuantify3}$$

$$\frac{\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec} \qquad \Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec} \qquad \oplus \in \{\ \&\ ,\ ||\ , \Rrightarrow\}}{\Theta; \Gamma; \Delta \rhd S_1 \oplus S_2 : \mathsf{spec}}\ \textsc{SpecBinary}$$

Figure 3.4: Well-sorting of Specifications

rule, one for each context.

Since spec is not a sort, this means that the language of specifications is a multi-sorted first-order logic, rather than a higher-order logic. There are no technical obstacles to extending it in this fashion, but we have not felt any strong need to do so.

3.5.1 Substitution Properties

In the syntax for the program logic, we have systematically made the decision to forbid the appearance of logical expressions within program expressions. We accomplish this by prohibiting variables from having sorts like $P : \mathsf{prop} \Rightarrow \mathbb{N}$. As a result, it is not possible to form program expressions which depend on subexpressions of logical type, such as writing an expression of monadic type $\bigcirc\mathbb{N}$ like $[P(x \mapsto y)]$.

This restriction ensures that program expressions never use variables of logical type, and so any term expression can be typed using only type and term variables, without any dependence on variables drawn from the logical sorts. By requiring program expressions to only contain

programs as subexpressions, we prevent the use of "ghost expressions" in our programs. That is, there are no commands or expressions in the programming language which manipulate purely logical values.

For first-order programs, it is relatively straightforward to distinguish between ghost variables and actual variables, but this is a distinction which breaks down in the presence of higher-order expressions. As a result, prohibiting these programs greatly simplifies the semantics of the program logic. I intend the sorts of our logic to be interpreted in Set, but the terms of our programming language are interpreted in CPO. Since there are no logical subexpressions of programs, then the forgetful functor U supplies sufficient tools to perform this embedding.

However, this does not restrict the expressiveness of our program logic reasoning, since our program logic is a full specification logic. We have no need of ghost state, since we can place quantifiers outside of Hoare triples. This lets us relate variables across a pre- and post-condition without any dubious manipulations of binding structure or extensions of the heap semantics.

The price we must pay for this simplicity is an increase in the complexity of the substitution theorems of the logic: since we have three contexts, we need six substitution theorems — three each for assertions and specifications. We will give these theorems after we have given the proofs, since it will be convenient to state the syntactic and semantic substitution properties together.

3.6 Semantics

Now, we will consider the interpretation of the syntax. To do this, we will proceed in stages. First, we will show how to interpret sorts and contexts by sets. Then, we will give a mutually-recursive interpretation of terms and specifications (since they each embed in the other), and finally we will prove that the semantics satisfies a substitution theorem.

Then, in the next section, we will be in a position to give the actual axioms of the program logic, and show them sound.

3.6.1 Interpretation of Sorts

We will start by explaining how to interpret the sorts. We interpret sorts as sets, but since sorts can have free variables, we need to give an interpretation of sorts-in-context, as is usual in categorical logic. We give the definitions in Figure 3.5.

A well-sortedness derivation is interpreted as a function from a tuple representing the type constructor environment θ (under the syntactic semantics) into a set. The interpretation of a kind κ is just its set-theoretic semantics, defined in the previous chapter. The interpretation of a type A is the domain-theoretic interpretation at the environment θ, then hit with the forgetful functor U to get its underlying set of points.

Propositions prop are interpreted as the powerset of heaps, and the function space $\omega \Rightarrow \upsilon$ is just the set-theoretic function space between the two sorts. The type-indexed sort $\Pi\alpha : \kappa.\ \upsilon$ is interpreted by an indexed family of sets, with the index extending the context.

Finally, we use this interpretation of sorts to give an interpretation of propositional contexts as a function from the type constructor environment into a nested tuple of sorts.

$$
\begin{array}{lcl}
[\![\Theta \rhd \omega : \mathsf{sort}]\!] & \in & [\![\Theta]\!]^s \to \mathrm{Set} \\
[\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta & = & [\![\kappa]\!] \\
[\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta & = & U([\![\Theta \vdash A : \bigstar]\!]\ \theta) \\
[\![\Theta \rhd \mathsf{prop} : \mathsf{sort}]\!]\ \theta & = & \mathcal{P}(H) \\
[\![\Theta \rhd \omega \Rightarrow \upsilon : \mathsf{sort}]\!]\ \theta & = & [\![\Theta \rhd \omega : \mathsf{sort}]\!]\ \theta \to [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta \\
[\![\Theta \rhd \Pi\alpha : \kappa.\ \upsilon : \mathsf{sort}]\!]\ \theta & = & \Pi\tau \in [\![\kappa]\!].\ [\![\Theta, \alpha : \kappa \rhd \upsilon : \mathsf{sort}]\!]\ (\theta, \tau)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\Theta \rhd \Delta]\!] & \in & [\![\Theta]\!]^s \to \mathrm{Set} \\
[\![\Theta \rhd \cdot]\!]\ \theta & = & 1 \\
[\![\Theta \rhd \Delta, u : \upsilon]\!]\ \theta & = & [\![\Theta \rhd \Delta]\!]\ \theta \times [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta
\end{array}
$$

Figure 3.5: Interpretation of Sorts and Contexts

3.6.2 Interpretation of Terms and Specifications

The term judgment $\Theta; \Gamma; \Delta \rhd p : \omega$ is mutually recursively defined with the specification judgment $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$. Therefore, when we give the semantics of these judgments, we need to define them together. The definition of the interpretation of these two judgments is given in Figures 3.6 and 3.7.

There are no surprises in the interpretations – everything is interpreted straightforwardly. The only unusual feature is that lambda-abstractions and quantifiers have three cases, corresponding to their three typing rules. However, the interpretation is still syntax-directed, since types, kinds, and propositional sorts are syntactically distinct.

3.6.3 Substitution Properties

We state the main soundness theorems below.

Lemma 26. *(Substitution for Sorts) Suppose* $\Theta \vdash \tau : \kappa$.

1. *If* $\Theta, \alpha : \kappa \rhd \omega : \mathsf{sort}$, *then* $\Theta \rhd [\tau/\alpha]\omega : \mathsf{sort}$ *and* $[\![\Theta \rhd [\tau/\alpha]\omega : \mathsf{sort}]\!]\ \theta$ *equals* $[\![\Theta, \alpha : \kappa \rhd \omega : \mathsf{sort}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)$
2. *If* $\Theta, \alpha : \kappa \rhd \Delta$, *then* $\Theta \rhd [\tau/\alpha]\Delta$ *and* $[\![\Theta \rhd [\tau/\alpha]\Delta]\!]\ \theta$ *equals* $[\![\Theta, \alpha : \kappa \rhd \Delta]\!](\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)$
3. *If* $\Theta, \alpha : \kappa \rhd \omega \equiv \omega' : \mathsf{sort}$, *then* $\Theta \rhd [\tau/\alpha]\omega \equiv [\tau/\alpha]\omega' : \mathsf{sort}$.
4. *If* $\Theta \rhd \omega \equiv \omega' : \mathsf{sort}$ *is derivable, then* $[\![\Theta \rhd \omega : \mathsf{sort}]\!] = [\![\Theta \rhd \omega' : \mathsf{sort}]\!]$.
5. *If* $\Theta \rhd \omega : \mathsf{sort}$, *then* $\Theta, \alpha : \kappa \rhd \omega : \mathsf{sort}$
6. *If* $\Theta \rhd \omega : \mathsf{sort}$ *and* $\Theta, \alpha : \kappa \rhd \omega : \mathsf{sort}$, *then* $[\![\Theta \rhd \omega : \mathsf{sort}]\!]\ \theta = [\![\Theta, \alpha : \kappa \rhd \omega : \mathsf{sort}]\!]\ (theta, \tau)$

Proof. These lemmas follow from routine inductions. □

$$
\begin{array}{lcl}
[\![\Theta;\Gamma;\Delta \rhd p : \omega]\!] & \in & \prod \theta \in [\![\theta]\!], \gamma \in [\![\Theta \vdash \Gamma]\!]\ \theta, \delta \in [\![\Theta \rhd \Delta]\!]\ \theta. \\
& & \quad \to [\![\Theta \rhd \omega : \mathsf{sort}]\!]\ \theta\gamma\delta \\
[\![\Theta;\Gamma;\Delta \rhd S : \mathsf{spec}]\!] & \in & \prod \theta \in [\![\theta]\!], \gamma \in [\![\Theta \vdash \Gamma]\!]\ \theta, \delta \in [\![\Theta \rhd \Delta]\!]\ \theta. \\
& & \quad \to \mathcal{P}^{\uparrow}(W(\mathcal{P}(H)))
\end{array}
$$

$$
\begin{array}{lcl}
[\![\Theta;\Gamma;\Delta \rhd \tau : \kappa]\!]\ \theta\,\gamma\,\delta & = & [\![\Theta \vdash \tau : \kappa]\!]\ \theta \\
[\![\Theta;\Gamma;\Delta \rhd e : A]\!]\ \theta\,\gamma\,\delta & = & U([\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta)\ \gamma \\
[\![\Theta;\Gamma;\Delta \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon]\!]\ \theta\,\gamma\,\delta & = & \lambda v \in [\![\Theta \rhd \upsilon' : \mathsf{sort}]\!]\ \theta.\ [\![\Theta;\Gamma;\Delta, u : \upsilon' \rhd p : \upsilon]\!]\ \theta\,\gamma\,(\delta, v) \\
[\![\Theta;\Gamma;\Delta \rhd \hat{\lambda} x : A.\ p : A \Rightarrow \upsilon]\!]\ \theta\,\gamma\,\delta & = & \lambda v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta.\ [\![\Theta;\Gamma, x : A;\Delta \rhd p : \upsilon]\!]\ \theta\,(\gamma, v)\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd \hat{\lambda} \alpha : \kappa.\ p : \kappa \Rightarrow \upsilon]\!]\ \theta\,\gamma\,\delta & = & \lambda \tau \in [\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta.\ [\![\Theta, \alpha : \kappa;\Gamma;\Delta \rhd p : \upsilon]\!]\ (\theta, \tau)\,\gamma\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd p\ q : \upsilon]\!]\ \theta\ \gamma\ \delta & = & ([\![\Theta;\Gamma;\Delta \rhd p : \omega \Rightarrow \upsilon]\!]\ \delta)\ ([\![\Theta;\Gamma;\Delta \rhd q : \omega]\!]\ \delta) \\
[\![\Theta;\Gamma;\Delta \rhd \underline{\hat{\lambda}} \alpha : \kappa.\ p : \Pi \alpha : \kappa.\ \upsilon]\!]\ \theta\,\gamma\,\delta & = & \lambda \tau \in [\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta.\ [\![\Theta, \alpha : \kappa;\Gamma;\Delta \rhd p : \upsilon]\!]\ (\theta, \tau)\,\gamma\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd p\ [\tau] : [\tau/\alpha]\upsilon]\!]\ \theta\,\gamma\,\delta & = & ([\![\Theta;\Gamma;\Delta \rhd p : \Pi \alpha : \kappa.\ \upsilon]\!]\ \theta\,\gamma\,\delta)\ ([\![\Theta \vdash \tau : \kappa]\!]\ \theta) \\
[\![\Theta;\Gamma;\Delta \rhd u : \omega]\!]\ \theta\,\gamma\,\delta & = & \pi_u(\delta) \\
[\![\Theta;\Gamma;\Delta \rhd c : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & [\![c]\!]^0 \\
[\![\Theta;\Gamma;\Delta \rhd p \oplus q : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & [\![\Theta;\Gamma;\Delta \rhd p : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta \\
& & [\![\oplus]\!]^2\ \ [\![\Theta;\Gamma;\Delta \rhd q : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd e \mapsto_A e' : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & \text{let } l = [\![\Theta;\ \Gamma \vdash e : \mathsf{ref}\ A]\!]\ \theta\,\gamma \text{ in} \\
& & \text{let } v = [\![\Theta;\ \Gamma \vdash e' : A]\!]\ \theta\,\gamma \text{ in} \\
& & l \mapsto v \\
[\![\Theta;\Gamma;\Delta \rhd p =_\omega q : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & \text{if } [\![\Theta;\Gamma;\Delta \rhd p : \omega]\!]\ \theta\,\gamma\,\delta = [\![\Theta;\Gamma;\Delta \rhd q : \omega]\!]\ \theta\,\gamma\,\delta \\
& & \text{then } \top \text{ else} \bot \\
[\![\Theta;\Gamma;\Delta \rhd Q u : \upsilon.\ p : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & [\![Q]\!]^{\infty}_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta}[\![\Theta;\Gamma;\Delta, u : \omega \rhd p : \mathsf{prop}]\!]\ \theta\,\gamma\,(\delta, v) \\
[\![\Theta;\Gamma;\Delta \rhd Q x : A.\ p : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & [\![Q]\!]^{\infty}_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta}[\![\Theta;\Gamma, x : A;\Delta \rhd p : \mathsf{prop}]\!]\ \theta\,(\gamma, v)\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd Q \alpha : \kappa.\ p : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & [\![Q]\!]^{\infty}_{\tau \in [\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta}[\![\Theta, \alpha : \kappa;\Gamma;\Delta \rhd p : \mathsf{prop}]\!]\ (\theta, \tau)\,\gamma\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd S\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta & = & \text{if } [\![\Theta;\Gamma;\Delta \rhd S : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta = \top_{\mathcal{P}^{\uparrow}(W(\mathcal{P}(H)))} \text{ then } \top \text{ else } \bot \\
[\![\Theta;\Gamma;\Delta \rhd p : \omega]\!]\ \theta\,\gamma\,\delta & = & [\![\Theta;\Gamma;\Delta \rhd p : \omega']\!]\ \delta \text{ when } \Theta \rhd \omega \equiv \omega' : \mathsf{sort}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\top]\!]^0 & = & \top \\
[\![\bot]\!]^0 & = & \bot \\
[\![\mathsf{emp}]\!]^0 & = & I
\end{array}
$$

$$
\begin{array}{lcl}
[\![\wedge]\!]^2 & = & \wedge \\
[\![\supset]\!]^2 & = & \supset \\
[\![\vee]\!]^2 & = & \vee \\
[\![*]\!]^2 & = & * \\
[\![\mathrel{-\!\!*}]\!]^2 & = & \mathrel{-\!\!*}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\forall]\!]^{\infty} & = & \bigwedge \\
[\![\exists]\!]^{\infty} & = & \bigvee
\end{array}
$$

$$
l \mapsto v \quad = \quad \{(\{l\}, \lambda loc \in \{l\}.\ v)\}
$$

Figure 3.6: Interpretation of Terms

$$
\begin{array}{lcl}
[\![\Theta;\Gamma;\Delta \rhd \{p\}c\{a:A.\ q\} : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta & = & \{[\![\Theta;\Gamma;\Delta \rhd p : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta\} \\
 & & [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma\ \delta \\
 & & \{v.\ [\![\Delta, a : A \rhd q : \mathsf{prop}]\!]\ \theta\,(\gamma, v)\,\delta\} \\
[\![\Theta;\Gamma;\Delta \rhd \langle p\rangle e\langle a:A.\ q\rangle : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta & = & \{[\![\Theta;\Gamma;\Delta \rhd p : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta\} \\
 & & [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]\ \theta\ \gamma\ \delta \\
 & & \{v.\ [\![\Delta, a : A \rhd q : \mathsf{prop}]\!]\ \theta\,(\gamma, v)\,\delta\} \\
[\![\Theta;\Gamma;\Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta & = & \text{if } [\![\Theta;\Gamma;\Delta \rhd p : \mathsf{prop}]\!]\ \theta\,\gamma\,\delta = \top_{\mathcal{P}(H)} \text{ then } \top \text{ else } \bot \\
[\![\Theta;\Gamma;\Delta \rhd S_1 \oplus S_2 : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta & = & [\![\Theta;\Gamma;\Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta\ \ [\![\oplus]\!] \\
 & & [\![\Theta;\Gamma;\Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd Qu : v.\ S : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta & = & [\![Q]\!]_{v \in [\![\Theta \rhd v:\mathsf{sort}]\!]\ \theta}[\![\Theta;\Gamma;\Delta, u : v \rhd S : \mathsf{spec}]\!]\ \theta\,\gamma\,(\delta, v) \\
[\![\Theta;\Gamma;\Delta \rhd Qx : A.\ S : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta & = & [\![Q]\!]_{v \in [\![\Theta \rhd A:\mathsf{sort}]\!]\ \theta}[\![\Theta;\Gamma, x : A;\Delta \rhd S : \mathsf{spec}]\!]\ \theta\,(\gamma, v)\,\delta \\
[\![\Theta;\Gamma;\Delta \rhd Q\alpha : \kappa.\ S : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta & = & [\![Q]\!]_{\tau \in [\![\Theta \rhd \kappa:\mathsf{sort}]\!]\ \theta}[\![\Theta, \alpha : \kappa;\Gamma;\Delta \rhd S : \mathsf{spec}]\!]\ (\theta, \tau)\,\gamma\,\delta \\
 & & \\
[\![\ \&\ ]\!] & = & \wedge \\
[\![\ ||\ ]\!] & = & \vee \\
[\![\Rrightarrow]\!] & = & \supset \\
 & & \\
[\![\forall]\!] & = & \bigwedge \\
[\![\exists]\!] & = & \bigvee
\end{array}
$$

Figure 3.7: Interpretation of Specifications

Lemma 27. *(Weakening and Strengthening)*

1. *If* $\Theta; \Gamma; \Delta \rhd p : \omega$*, then* $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega$.
2. *If* $\Theta; \Gamma; \Delta \rhd p : \omega$ *and* $\Theta \rhd A : \mathsf{sort}$*, then* $\Theta; \Gamma, x : A; \Delta \rhd p : \omega$.
3. *If* $\Theta; \Gamma; \Delta \rhd p : \omega$ *and* $\Theta \rhd \upsilon : \mathsf{sort}$*, then* $\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \omega$.
4. *If* $\Theta; \Gamma; \Delta \rhd p : \omega$ *and* $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega'$ *and* $\Theta \rhd \omega \equiv \omega' : \mathsf{sort}$*, then* $[\![\Theta; \Gamma; \Delta \rhd p : \omega]\!]\ \theta\ \gamma\ \delta = [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega']\!]\ (\theta, \tau)\ \gamma\ \delta$.
5. *If* $\Theta; \Gamma; \Delta \rhd p : \omega$ *and* $\Theta; \Gamma, x : A; \Delta \rhd p : \omega'$ *and* $\Theta \rhd \omega \equiv \omega' : \mathsf{sort}$*, then* $[\![\Theta; \Gamma; \Delta \rhd p : \omega]\!]\ \theta\ \gamma\ \delta = [\![\Theta; \Gamma, x : A; \Delta \rhd p : \omega']\!]\ \theta\ (\gamma, v)\ \delta$.
6. *If* $\Theta; \Gamma; \Delta \rhd p : \omega$ *and* $\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \omega'$ *and* $\Theta \rhd \omega \equiv \omega' : \mathsf{sort}$*, then* $[\![\Theta; \Gamma; \Delta \rhd p : \omega]\!]\ \theta\ \gamma\ \delta = [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \omega']\!]\ (\theta, \tau)\ \gamma\ (\delta, v)$.

In the last three cases, we assume that the arguments to the semantic functions are suitably typed.

Proof. These proofs all follow from a routine induction. The extra premise in the final three cases handles the slight non-syntax-directedness induced by the TEQSORT rule. □

Lemma 28. *(Substitution for Terms and Specifications)*

1. *Suppose* $\Theta \vdash \tau : \kappa$.
   (a) *If* $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega$*, then*
      i. $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega$
      ii. $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega]\!]\ \theta\ \gamma\ \delta$ *equals* $[\![\Theta, \alpha : \kappa; \Gamma, \Delta \rhd p : \omega]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta$
   (b) *If* $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}$*, then*
      i. $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}$
      ii. $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}]\!]\ \theta$ *equals* $[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}]\!](\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)$
2. *Suppose* $\Theta;\ \Gamma \vdash e : A$.
   (a) *If* $\Theta; \Gamma, x : A; \Delta \rhd p : \omega$*, then*
      i. $\Theta; \Gamma; \Delta \rhd [e/x]p : \omega$,
      ii. $[\![\Theta; \Gamma; \Delta \rhd [e/x]p : \omega]\!]\ \theta\ \gamma\ \delta$ *equals* $[\![\Theta; \Gamma, x : A; \Delta \rhd p : \omega]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ \gamma)\ \delta$
   (b) *If* $\Theta; \Gamma, x : A; \Delta \rhd S : \mathsf{spec}$*, then*
      i. $\Theta; \Gamma; \Delta \rhd [e/x]S : \mathsf{spec}$.
      ii. $[\![\Theta; \Gamma; \Delta \rhd [e/x]S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ *equals* $[\![\Theta; \Gamma, x : A; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ \gamma)\ \delta$
3. *Suppose* $\Theta; \Gamma; \Delta \rhd q : \upsilon$.
   (a) *If* $\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \omega$*, then*
      i. $\Theta; \Gamma; \Delta \rhd [q/u]p : \omega$
      ii. $[\![\Theta; \Gamma; \Delta \rhd [q/u]p : \omega]\!]\ \theta\ \gamma\ \delta$ *equals* $[\![\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \omega]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd q : \upsilon]\!]\ \theta\ \gamma\ \delta)$
   (b) *If* $\Theta; \Gamma : A; \Delta, u : \upsilon \rhd S : \mathsf{spec}$*, then*
      i. $\Theta; \Gamma; \Delta \rhd [q/u]S : \mathsf{spec}$

$$\frac{\Theta;\Gamma;\Delta \rhd \{p\}c\{a : A.\ q\} \Rrightarrow \langle p\rangle[c]\langle a : A.\ q\rangle : \mathsf{spec}}{\Theta;\Gamma;\Delta \rhd \{p\}c\{a : A.\ q\} \Rrightarrow \langle p\rangle[c]\langle a : A.\ q\rangle : \mathsf{spec\ valid}}\ \textsc{AxEquiv1}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \langle p\rangle[c]\langle a : A.\ q\rangle \Rrightarrow \{p\}c\{a : A.\ q\} : \mathsf{spec}}{\Theta;\Gamma;\Delta \rhd \langle p\rangle[c]\langle a : A.\ q\rangle \Rrightarrow \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid}}\ \textsc{AxEquiv2}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{P\}e\{a : A.\ P \wedge a = e\} : \mathsf{spec}}{\Theta;\Gamma;\Delta \rhd \{P\}e\{a : A.\ P \wedge a = e\} : \mathsf{spec\ valid}}\ \textsc{AxReturn}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{e \mapsto_A -\}e := e'\{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec}}{\Theta;\Gamma;\Delta \rhd \{e \mapsto_A -\}e := e'\{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec\ valid}}\ \textsc{AxAssign}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec}}{\Theta;\Gamma;\Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec\ valid}}\ \textsc{AxAlloc}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto_A e' \wedge a = e'\} : \mathsf{spec}}{\Theta;\Gamma;\Delta \rhd \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto_A e' \wedge a = e'\} : \mathsf{spec\ valid}}\ \textsc{AxDeref}$$

$$\frac{\begin{array}{c}\Theta;\Gamma;\Delta \rhd \langle p\rangle e\langle x : A.\ q\rangle : \mathsf{spec\ valid} \\ \Theta;\Gamma, x : A;\Delta \rhd \{q\}c\{a : B.\ r\} : \mathsf{spec\ valid} \qquad \Theta;\Gamma, a : B;\Delta \rhd r : \mathsf{prop}\end{array}}{\Theta;\Gamma;\Delta \rhd \{p\}\mathsf{letv}\ x = e\ \mathsf{in}\ c\{a : B.\ r\} : \mathsf{spec\ valid}}\ \textsc{AxBind}$$

$$\frac{\Theta;\Gamma;\Delta \rhd (\forall x : \bigcirc A.\ \langle p\rangle x\langle a : A.\ q\rangle) \Rrightarrow \langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec\ valid}}{\Theta;\Gamma;\Delta \rhd \langle p\rangle \mathsf{fix}\ x : \bigcirc A.\ e\langle a : A.\ q\rangle : \mathsf{spec\ valid}}\ \textsc{AxFix}$$

Figure 3.8: Basic Axioms of Specification Logic

ii. $[\![\Theta;\Gamma;\Delta \rhd [q/u]S : \mathsf{spec}]\!]\ \theta\,\gamma\,\delta$ *equals*
$[\![\Theta;\Gamma : A;\Delta, u : \upsilon \rhd S : \mathsf{spec}]\!]\ \theta\,\gamma\,(\delta, [\![\Theta;\Gamma;\Delta \rhd q : \upsilon]\!]\ \theta\,\gamma\,\delta)$

Proof. The proof is at the end of the chapter. □

3.7 The Program Logic

My program logic consists of three judgments:

1. $\Theta;\Gamma;\Delta \rhd p : \mathsf{prop\ valid}$, which asserts that a proposition p is valid.
2. $\Theta;\Gamma;\Delta \rhd S : \mathsf{spec\ valid}$, which asserts that a specification S is valid.
3. $\Theta;\Gamma;\Delta \rhd p \equiv q : \omega$, which asserts that p and q are validly equal.

The semantics of these three judgments is given as follows:

$$\frac{\Theta;\Gamma;\Delta \rhd \{r\} \Rrightarrow \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid} \quad r \text{ is a pure formula} \quad \Theta;\Gamma;\Delta \rhd p \supset r : \mathsf{prop\ valid}}{\Theta;\Gamma;\Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid}}\ \textsc{AxExtract}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{r\} \Rrightarrow \{p \wedge r\}c\{a : A.\ q\} : \mathsf{spec\ valid}}{\Theta;\Gamma;\Delta \rhd \{r\} \Rrightarrow \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid}}\ \textsc{AxEmbed}$$

$$\frac{}{\Theta;\Gamma;\Delta \rhd \{S\ \mathsf{spec}\} \Rrightarrow S : \mathsf{spec\ valid}}\ \textsc{AxUseValid}$$

$$\frac{\Theta;\Gamma;\Delta, u : \upsilon \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid} \quad u \notin \mathrm{FV}(q)}{\Theta;\Gamma;\Delta \rhd \{\exists u : \upsilon.\ p\}c\{a : A.\ q\} : \mathsf{spec\ valid}}\ \textsc{AxForgetEx1}$$

$$\frac{\Theta;\Gamma, y : B;\Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid} \quad y \notin \mathrm{FV}(c) \quad y \notin \mathrm{FV}(q)}{\Theta;\Gamma;\Delta \rhd \{\exists y : B.\ p\}c\{a : A.\ q\} : \mathsf{spec\ valid}}\ \textsc{AxForgetEx2}$$

$$\frac{\Theta, \alpha : \kappa;\Gamma;\Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid} \quad \alpha \notin \mathrm{FV}(c) \quad u \notin \mathrm{FV}(\alpha)}{\Theta;\Gamma;\Delta \rhd \{\exists \alpha : \kappa.\ p\}c\{a : A.\ q\} : \mathsf{spec\ valid}}\ \textsc{AxForgetEx3}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{r\} \Rrightarrow \{p\}[e/x]c\{a : A.\ q\} : \mathsf{spec\ valid} \quad \Theta;\Gamma;\Delta \rhd r \supset e =_A e' : \mathsf{prop\ valid}}{\Theta;\Gamma;\Delta \rhd \{r\} \Rrightarrow \{p\}[e'/x]c\{a : A.\ q\} : \mathsf{spec}}\ \textsc{AxEquality}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{p'\}c\{a : A.\ q'\} : \mathsf{spec\ valid} \quad \Theta;\Gamma;\Delta \rhd p \supset p' : \mathsf{prop\ valid} \quad \Theta;\Gamma, a : A;\Delta \rhd q' \supset q : \mathsf{prop\ valid}}{\Theta;\Gamma;\Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid}}\ \textsc{AxConsequence}$$

$$\frac{\Theta;\Gamma;\Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec\ valid} \quad \Theta;\Gamma;\Delta \rhd \{p'\}c\{a : A.\ q'\} : \mathsf{spec\ valid}}{\Theta;\Gamma;\Delta \rhd \{p \vee p'\}c\{a : A.\ q \vee q'\} : \mathsf{spec\ valid}}\ \textsc{AxDisjunction}$$

$$\frac{\Theta;\Gamma;\Delta \rhd p : \mathsf{prop\ valid}}{\Theta;\Gamma;\Delta \rhd \{p\} : \mathsf{spec\ valid}}\ \textsc{AxEmbedAssert}$$

$$\frac{\Theta;\Gamma;\Delta \rhd S : \mathsf{spec} \quad \Theta;\Gamma;\Delta \rhd r : \mathsf{prop}}{\Theta;\Gamma;\Delta \rhd S \Rrightarrow S \otimes r : \mathsf{spec\ valid}}\ \textsc{AxFrame}$$

(+ all of the axioms of intuitionistic logic)

Figure 3.9: Structural Axioms of the Program Logic

$$\frac{\Theta; \Gamma; \Delta \rhd S : \mathsf{spec\ valid}}{\Theta; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop\ valid}} \qquad \frac{\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}}{\Theta; \Gamma; \Delta \rhd \{p\}\ \mathsf{spec} \supset p : \mathsf{prop\ valid}}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p \equiv q : \omega}{\Theta; \Gamma; \Delta \rhd p =_\omega q : \mathsf{prop\ valid}}$$

(plus axioms of higher-order separation logic)

Figure 3.10: Axioms of Assertion Logic

$$\frac{\Theta; \Gamma; \Delta \rhd p : \omega}{\Theta; \Gamma; \Delta \rhd p \equiv p : \omega} \qquad \frac{\Theta; \Gamma; \Delta \rhd p \equiv q : \omega}{\Theta; \Gamma; \Delta \rhd q \equiv p : \omega}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p \equiv q : \omega \qquad \Theta; \Gamma; \Delta \rhd q \equiv r : \omega}{\Theta; \Gamma; \Delta \rhd p \equiv r : \omega} \qquad \frac{\Theta; \Gamma; \Delta \rhd (\lambda x : \omega.\ p)\ q : \omega'}{\Theta; \Gamma; \Delta \rhd (\lambda x : \omega.\ p)\ q \equiv [q/x]p : \omega'}$$

$$\frac{\Delta, x : \omega \rhd p\ x \equiv p'\ x : \omega'}{\Theta; \Gamma; \Delta \rhd \lambda x : \omega.\ p \equiv \lambda x : \omega.\ p' : \omega \Rightarrow \omega'}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p \equiv p' : \omega \Rightarrow \omega' \qquad \Theta; \Gamma; \Delta \rhd q \equiv q' : \omega}{\Theta; \Gamma; \Delta \rhd p\ q \equiv p'\ q' : \omega'} \qquad \frac{\Theta \vdash \tau \equiv \tau' : \kappa}{\Theta; \Gamma; \Delta \rhd \tau \equiv \tau' : \kappa}$$

$$\frac{\Theta;\ \Gamma \vdash e \equiv e' : A}{\Theta; \Gamma; \Delta \rhd e \equiv e' : A} \qquad \frac{\Theta; \Gamma; \Delta \rhd p \iff q : \mathsf{prop\ valid}}{\Theta; \Gamma; \Delta \rhd p \equiv q : \mathsf{prop}}$$

$$\frac{\Theta; \Gamma; \Delta \rhd p \equiv q : \omega \qquad \Theta \rhd \omega \equiv \omega' : \mathsf{sort}}{\Theta; \Gamma; \Delta \rhd p \equiv q : \omega'}$$

Figure 3.11: Equality Judgment for Assertions

1. An assertion $\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}$ is valid, if and only if for all $\theta \in [\![\theta]\!], \gamma \in [\![\Theta \vdash \Gamma]\!]\theta$, and $\delta \in [\![\Theta \rhd \Delta]\!]\theta$, we have $[\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = \top_{\mathcal{P}(H)}$.
2. Similarly, a specification $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$ is valid if and only if, for all $\theta \in [\![\theta]\!], \gamma \in [\![\Theta \vdash \Gamma]\!]\theta$, and $\delta \in [\![\Theta \rhd \Delta]\!]\theta$, we have $[\![\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top_{\mathcal{P}^\uparrow(W(\mathcal{P}(H)))}$.
3. Finally, two assertion terms $\Theta; \Gamma; \Delta \rhd p : \omega$ and $\Theta; \Gamma; \Delta \rhd q : \omega$ are validly equal, if and only if for all $\theta \in [\![\theta]\!], \gamma \in [\![\Theta \vdash \Gamma]\!]\theta$, and $\delta \in [\![\Theta \rhd \Delta]\!]\theta$, we have that $[\![\Delta \vdash p : \omega]\!]\ \theta\ \gamma\ \delta = [\![\Delta \vdash p : \omega]\!]\ \theta\ \gamma\ \delta$

So this says that for any assignment of the free variables, a valid assertion p must be true, and similarly a valid specification S must be true. Likewise, two syntactic terms p and q are validly equal if their interpretations are equal under all environments.

The program logic consists of a set of three mutually-recursive judgments for deriving valid assertions, specifications, and equalities. In Figure 3.8, I give the primitive rules for deriving valid specifications of commands, and Figure 3.9, I give a collection of structural axioms. In Figure 3.10, I give some of the rules for deriving valid assertions, and in Figure 3.11, I give rules for deriving equalities between terms.

In Figure 3.8, the AXEQUIV1 and AXEQUIV2 axioms assert the equivalence of Hoare triples over commands and monadic expressions. The AXRETURN axiom asserts that when we return a value e, we can strengthen the postcondition with the assertion that the return value is e. The AXASSIGN axiom asserts that if e points to something in the precondition, the command $e := e'$ ensures that in the postcondition $e \mapsto e'$. The AXALLOC axiom asserts that allocating a new reference $\mathsf{new}_A(e)$ will allocate a new pointer which is the return value a, and that $a \mapsto e$ will be star'd onto the postcondition. The AXDEREF rule asserts that if $e \mapsto e'$ in the precondition, then in the postcondition $e \mapsto e'$ will continue to hold, and that the return value $a = e'$.

The AXBIND rule is the sequential composition rule of this logic. Because of the monadic nature of our programming language, we can name and bind intermediate results, but of course there is a side-condition to ensure that variables do not escape their scopes.

Finally the AXFIX rule is a fixed point induction rule for this calculus. As in LCF, this rule looks like an induction without a base case, which makes sense since this program logic is a partial correctness calculus. This rule generalizes to all the pointed types, but the syntactic overhead for stating the rules gets higher as the types get larger.

In Figure 3.9, the AXEXTRACT rule takes a pure consequence of a precondition, and lifts it to hypothetical assertion of the validity of an assertion. The intuition for this rule is that to show the Hoare triple, we must assume p, and so this means we might as well assume r generally while proving this triple. Conversely, The AXEMBED rule says that if we have assumed the validity of an assertion, it does no harm to assume it while proving a Hoare triple. The AXUSEVALID axiom states that if we know that the assertion S spec is valid, then we may as well assume that S is valid, and the AXEMBEDASSERT says that valid assertions can be embedded in specifications.

The AXEQUALITY rule lets us rewrite programs using equational reasoning, and the AXFORGETEX axiom lets us drop existentials from preconditions. The AXCONSEQUENCE rule is just the rule of consequence, and the AXDISJUNCTION rule is just the rule of disjunction, both familiar from Hoare logic. However, it is worth pointing out that my language does *not* validate the rule of conjunction, as a consequence of having a language with a continuation semantics.

Finally, the AXFRAME schema gives the frame rule. My logic supports the higher-order frame rule, and so we need to define a syntactic frame operator $S \otimes r$ to act on arbitrary specifications.

$$\begin{array}{lcl}
\{p\}c\{a : A.\ q\} \otimes r & = & \{p * r\}c\{a : A.\ q * r\} \\
\langle p\rangle c\langle a : A.\ q\rangle \otimes r & = & \langle p * r\rangle c\langle a : A.\ q * r\rangle \\
\{p\} \otimes r & = & \{p\} \\
(S_1 \ \&\ S_2) \otimes r & = & S_1 \otimes r \ \&\ S_2 \otimes r \\
(S_1 \mid\mid S_2) \otimes r & = & S_1 \otimes r \mid\mid S_2 \otimes r \\
(S_1 \Rrightarrow S_2) \otimes r & = & S_1 \otimes r \Rrightarrow S_2 \otimes r \\
(\forall x : \omega.\ S) \otimes r & = & \forall x : \omega.\ (S \otimes r) \\
(\exists x : \omega.\ S) \otimes r & = & \exists x : \omega.\ (S \otimes r)
\end{array}$$

This is just the familiar action of the frame rule on Hoare triples. It does nothing to the validity assertion $\{p\}$, and distributes over all the other connectives.

Framing is well-defined, and corresponds precisely to the semantic frame rule we considered earlier in the chapter.

Proposition 8. *(Syntactic Well-Formedness of Frame Operator) If* $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$ *and* $\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}$*, then we define* $\Theta; \Gamma; \Delta \rhd S \otimes p : \mathsf{spec}$.

Proof. By structural induction on specifications. □

More interesting than the syntactic well-formedness of the frame operator is its semantic well-formedness.

Lemma 29. *(Syntactic Framing is Semantic Framing) If* $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$ *and* $\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}$*, then for suitably typed* θ, γ *and* δ*,* $[\![\Theta; \Gamma; \Delta \rhd S \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ *equals* $[\![\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$.

Proof. The proof is at the end of the chapter. □

The distinctive axioms of the assertion logic are far fewer. In Figure 3.10, I give only three rules tailored to this particular program logic. One for embedding specifications in assertions, another for extracting assertions out of specifications, and the last says that equality derivations entail equality assertions.

All the other rules are just the axioms of higher-order separation logic [6]. These include all the axioms of separation logic, and all the axioms of higher-order logic. In addition, many modal properties such as purity can be expressed internally using higher-order predicates. (For example, we can define purity as the second-order predicate $Pure(P) \equiv \forall Q : \mathsf{prop}.\ P * Q \iff P \wedge Q$.) The only caveat worth mentioning is that when forming sets by comprehension, the comprehending predicate must be a pure one.

Likewise, the valid equality judgment in Figure 3.11 is fairly minimal: it contains the β and η rules for functions, as well as inheriting the equality judgments for types and terms.

With all the buildup so far, it is easy to prove the following soundness theorem:

Theorem 1. *(Soundness of the Program Logic)*

1. *If* $\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}\ \mathsf{valid}$*, then* $\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}$ *is valid.*
2. *If* $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}\ \mathsf{valid}$*, the* $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$ *is valid.*
3. *If* $\Theta; \Gamma; \Delta \rhd p \equiv q : \omega$*, then* p *and* q *are validly equal.*

Proof. The proof of these three theorems follows from a mutual structural induction. The soundness of each atomic rule for the two validity judgments is proven in the next section. The soundness of the rules for equality are a consequence of the equality rules we have already proven, and the cartesian closure of Set. □

3.8 Correctness Proofs

3.8.1 Substitution Theorems

Lemma. *(28, page 28: Substitution for Terms and Specifications)*

1. *Suppose* $\Theta \vdash \tau : \kappa$.
   (a) *If* $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega$, *then*
      i. $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega$
      ii. $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega]\!]\ \theta\, \gamma\, \delta$ *equals*
      $[\![\Theta, \alpha : \kappa; \Gamma, \Delta \rhd p : \omega]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\, \gamma\, \delta$
   (b) *If* $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}$, *then*
      i. $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}$
      ii. $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}]\!]\ \theta$ *equals*
      $[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}]\!](\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)$
2. *Suppose* $\Theta;\ \Gamma \vdash e : A$.
   (a) *If* $\Theta; \Gamma, x : A; \Delta \rhd p : \omega$, *then*
      i. $\Theta; \Gamma; \Delta \rhd [e/x]p : \omega$,
      ii. $[\![\Theta; \Gamma; \Delta \rhd [e/x]p : \omega]\!]\ \theta\, \gamma\, \delta$ *equals*
      $[\![\Theta; \Gamma, x : A; \Delta \rhd p : \omega]\!]\ \theta\, (\gamma, [\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ \gamma)\, \delta$
   (b) *If* $\Theta; \Gamma, x : A; \Delta \rhd S : \mathsf{spec}$, *then*
      i. $\Theta; \Gamma; \Delta \rhd [e/x]S : \mathsf{spec}$.
      ii. $[\![\Theta; \Gamma; \Delta \rhd [e/x]S : \mathsf{spec}]\!]\ \theta\, \gamma\, \delta$ *equals*
      $[\![\Theta; \Gamma, x : A; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\, (\gamma, [\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ \gamma)\, \delta$
3. *Suppose* $\Theta; \Gamma; \Delta \rhd q : \upsilon$.
   (a) *If* $\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \omega$, *then*
      i. $\Theta; \Gamma; \Delta \rhd [q/u]p : \omega$
      ii. $[\![\Theta; \Gamma; \Delta \rhd [q/u]p : \omega]\!]\ \theta\, \gamma\, \delta$ *equals*
      $[\![\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \omega]\!]\ \theta\, \gamma\, (\delta, [\![\Theta; \Gamma; \Delta \rhd q : \upsilon]\!]\ \theta\, \gamma\, \delta)$
   (b) *If* $\Theta; \Gamma : A; \Delta, u : \upsilon \rhd S : \mathsf{spec}$, *then*
      i. $\Theta; \Gamma; \Delta \rhd [q/u]S : \mathsf{spec}$
      ii. $[\![\Theta; \Gamma; \Delta \rhd [q/u]S : \mathsf{spec}]\!]\ \theta\, \gamma\, \delta$ *equals*
      $[\![\Theta; \Gamma : A; \Delta, u : \upsilon \rhd S : \mathsf{spec}]\!]\ \theta\, \gamma\, (\delta, [\![\Theta; \Gamma; \Delta \rhd q : \upsilon]\!]\ \theta\, \gamma\, \delta)$

Proof. First, assume $\Theta \vdash \tau : \kappa$, and $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega$, and $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}$.

Now, we proceed by mutual induction on the derivation of p and S:

1. Case TTYPE: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \tau' : \kappa'$

   First, the syntax:

1 By inversion, we know $\Theta, \alpha : \kappa \vdash \Delta$
2 By inversion, we know $\Theta, \alpha : \kappa \vdash \Gamma$
3 By inversion, we know $\Theta, \alpha : \kappa \vdash \tau' : \kappa'$
4 By substitution, $\Theta \vdash [\tau/\alpha]\Delta$
5 By substitution, $\Theta \vdash [\tau/\alpha]\Gamma$
6 By substitution, $\Theta \vdash [\tau/\alpha]\tau' : \kappa'$
7 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]\tau' : \kappa'$

For semantics, $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]\tau' : \kappa']\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & [\![\Theta \vdash [\tau/\alpha]\tau' : \kappa']\!]\ \theta & \text{Semantics} \\ = & [\![\Theta, \alpha : \kappa \vdash \tau' : \kappa']\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta) & \text{Substitution thm} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Theta \rhd \tau' : \kappa']\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

2. Case TEXPR: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd e : A$

First, the syntax:

1 By inversion, we know $\Theta, \alpha : \kappa \vdash \Delta$
2 By inversion, we know $\Theta, \alpha : \kappa;\ \Gamma \vdash e : A$
3 By substitution, we know $\Theta \vdash [\tau/\alpha]\Delta$
4 By substitution, we know $\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]e : [\tau/\alpha]A$
5 By rule, we know $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]e : [\tau/\alpha]A$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]e : [\tau/\alpha]A]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & [\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]e : [\tau/\alpha]A]\!]\ \theta\ \gamma & \text{Semantics} \\ = & [\![\Theta;\ \Gamma \vdash e : A]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma & \text{Substitution} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd e : A]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

3. Case THYP: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd u : \upsilon$

First, the syntax:

1 By inversion, we know $\Theta, \alpha : \kappa \vdash \Gamma$
2 By inversion, we know $\Theta, \alpha : \kappa \vdash \Delta$
3 By substitution, we know $\Theta \vdash [\tau/\alpha]\Gamma$
4 By substitution, we know $\Theta \vdash [\tau/\alpha]\Delta$
5 By rule, we know $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd u : [\tau/\alpha]A$

Next, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd u : [\tau/\alpha]A]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \pi_u(\delta) & \text{Semantics} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd u : A]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

4. Case TABS1: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon$

   First, the syntax:

   1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta, u : \upsilon' \rhd p : \upsilon$
   2 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, u : [\tau/\alpha]\upsilon' \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon$
   3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd \hat{\lambda} u : [\tau/\alpha]\upsilon'.\ [\tau/\alpha]p : [\tau/\alpha]\upsilon' \Rightarrow [\tau/\alpha]\upsilon$
   4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\lambda} u : \upsilon'.\ p) : [\tau/\alpha](\upsilon' \Rightarrow \upsilon)$

   For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\lambda} u : \upsilon'.\ p) : [\tau/\alpha](\upsilon' \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & \lambda v.\ [\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, u : [\tau/\alpha]\upsilon' \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon]\!]\ \theta\ \gamma\ (\delta, v) & \text{Semantics} \\ = & \lambda v.\ [\![\Theta, \alpha : \kappa; \Gamma; \Delta, u : \upsilon' \rhd p : \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ (\delta, v) & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

   The correctness of the application of γ and δ follows from the equations for contexts under substitution.

5. Case TABS2: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\lambda} x : A.\ p : A \Rightarrow \upsilon$

   First, the syntax:

   1 By inversion, $\Theta, \alpha : \kappa; \Gamma, x : A; \Delta \rhd p : \upsilon$
   2 By induction, $\Theta; [\tau/\alpha]\Gamma, x : [\tau/\alpha]A; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon$
   3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd \hat{\lambda} x : [\tau/\alpha]A.\ [\tau/\alpha]p : [\tau/\alpha]A \Rightarrow [\tau/\alpha]\upsilon$
   4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\lambda} x : A.\ p) : [\tau/\alpha](A \Rightarrow \upsilon)$

   For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\lambda} x : A.\ p) : [\tau/\alpha](A \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & \lambda v.\ [\![\Theta; [\tau/\alpha]\Gamma, x : [\tau/\alpha]A; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon]\!]\ \theta\ (\gamma, v)\ \delta & \text{Semantics} \\ = & \lambda v.\ [\![\Theta, \alpha : \kappa; \Gamma, x : A; \Delta \rhd p : \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ (\gamma, v)\ \delta & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\lambda} x : A.\ p : A \Rightarrow \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

   The correctness of the application of γ and δ follows from the equations for contexts under substitution.

6. Case TABS3: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\lambda} \beta : \kappa'.\ p : \kappa' \Rightarrow \upsilon$

   First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd p : \upsilon$
2 By induction, $\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon$
3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd \hat{\lambda}\beta : \kappa'.\ [\tau/\alpha]p : \kappa' \Rightarrow [\tau/\alpha]\upsilon$
4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\lambda}\beta : \kappa'.\ p) : [\tau/\alpha](\kappa' \Rightarrow \upsilon)$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\lambda}\beta : \kappa'.\ p) : [\tau/\alpha](\kappa' \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \lambda\tau'.\ [\![\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon]\!]\ (\theta, \tau')\ \gamma\ \delta & \text{Semantics} \\ = & \lambda\tau'.\ [\![\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd p : \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta, \tau')\ \gamma\ \delta & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\lambda}\beta : \kappa'.\ p : \kappa' \Rightarrow \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution. We also silently permuted the context at the second step, and made use of the fact that β is not free in Γ, Δ or τ.

7. Case TABSALL: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\underline{\lambda}}\beta : \kappa'.\ p : \Pi\beta : \kappa'.\ \upsilon$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd p : \upsilon$
2 By induction, $\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon$
3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd \hat{\underline{\lambda}}\beta : \kappa'.\ [\tau/\alpha]p : \Pi\beta : \kappa'.\ [\tau/\alpha]\upsilon$
4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\underline{\lambda}}\beta : \kappa'.\ p) : [\tau/\alpha]\Pi\beta : \kappa'.\ \upsilon$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\hat{\lambda}\beta : \kappa'.\ p) : [\tau/\alpha](\Pi\beta : \kappa'.\ \upsilon)]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \lambda\tau'.\ [\![\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\upsilon]\!]\ (\theta, \tau')\ \gamma\ \delta & \text{Semantics} \\ = & \lambda\tau'.\ [\![\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd p : \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta, \tau')\ \gamma\ \delta & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \hat{\lambda}\beta : \kappa'.\ p : \Pi\beta : \kappa'.\ \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution. We also silently permuted the context at the second step, and made use of the fact that β is not free in Γ or Δ.

8. Case TAPP: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p\ q : \upsilon$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega \Rightarrow \upsilon$
2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd q : \omega$
3 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha](\omega \Rightarrow \upsilon)$
4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]q : [\tau/\alpha]\omega$
5 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p\ [\tau/\alpha]q : [\tau/\alpha]\upsilon$
6 By subst def, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p\ q) : [\tau/\alpha]\upsilon$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p\ q) : [\tau/\alpha]\upsilon]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{c}([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha](\omega \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta) \\ ([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]q : [\tau/\alpha]\omega]\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\ = & \begin{array}{c}([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega \Rightarrow \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta) \\ ([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd q : \omega]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p\ q : \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

9. Case TAPPALL: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p\ [\tau'] : [\tau'/\beta]\upsilon$
   First, the syntax:

   1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \Pi\beta : \kappa'.\ \upsilon$
   2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \tau' : \kappa'$
   3 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha](\Pi\beta : \kappa'.\ \upsilon)$
   4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]\tau' : \kappa'$
   5 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p\ [\tau/\alpha][\tau'] : [\tau/\alpha, [\tau/\alpha]\tau'/\beta]\upsilon$
   6 By subst def, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p\ [\tau']) : [\tau/\alpha, [\tau/\alpha]\tau'/\beta]\upsilon$

   For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p\ [\tau']) : [\tau/\alpha][\tau'/\beta]\upsilon]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{c}([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha](\Pi\beta : \kappa'.\ \upsilon)]\!]\ \theta\ \gamma\ \delta) \\ ([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]\tau' : \kappa']\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\ = & \begin{array}{c}([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \Pi\beta : \kappa'.\ \upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta) \\ ([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \tau' : \kappa']\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p\ [\tau'] : [\tau'/\beta]\upsilon]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

10. Case TCONST:
    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa \vdash \Gamma$
    2 By inversion, $\Theta, \alpha : \kappa \vdash \Delta$
    3 By substitution, $\Theta \vdash [\tau/\alpha]\Gamma$
    4 By substitution, $\Theta \vdash [\tau/\alpha]\Delta$
    5 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd c : \mathsf{prop}$

    For semantics consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd c : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & [\![c]\!]^0 & \text{Semantics} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd c : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

    The correctness of the application of γ and δ follows from the equations for contexts under substitution.

11. Case TBINARY: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p \oplus q : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}$
    2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd q : \mathsf{prop}$
    3 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}$
    4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]q : \mathsf{prop}$
    5 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p \oplus [\tau/\alpha]q : [\tau/\alpha]\mathsf{prop}$
    6 By subst def, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p \oplus q) : \mathsf{prop}$

    For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p \oplus q) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & \begin{array}{c}([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)\ [\![\oplus]\!]^2 \\ ([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]q : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\ = & \begin{array}{c}([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta)[\![\oplus]\!]^2 \\ ([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd q : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p \oplus q : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

12. Case TQUANTIFY1: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qu : \upsilon.\ p : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta, u : \upsilon \rhd p : \mathsf{prop}$
    2 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, u : [\tau/\alpha]\upsilon \rhd [\tau/\alpha]p : \mathsf{prop}$
    3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd Qu : [\tau/\alpha]\upsilon.\ [\tau/\alpha]p : \mathsf{prop}$
    4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qu : \upsilon.\ p) : \mathsf{prop}$

    For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qu : \upsilon.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & \begin{array}{c}[\![Q]\!]_{v \in [\![\Theta \rhd [\tau/\alpha]\upsilon : \mathsf{sort}]\!]\ \theta} \\ [\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, u : [\tau/\alpha]\upsilon \rhd [\tau/\alpha]p : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, v)\end{array} & \text{Semantics} \\ = & \begin{array}{c}[\![Q]\!]_{v \in [\![\Theta, \alpha:\kappa \rhd \upsilon : \mathsf{sort}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)} \\ [\![\Theta, \alpha : \kappa; \Gamma; \Delta, u : \upsilon \rhd p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ (\delta, v)\end{array} & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qu : \upsilon.\ p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

    The correctness of the application of γ and δ follows from the equations for contexts under substitution.

13. Case TQUANTIFY2: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qx : A.\ p : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa; \Gamma, x : A; \Delta \rhd p : \mathsf{prop}$
    2 By induction, $\Theta; [\tau/\alpha]\Gamma, x : [\tau/\alpha]A; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}$
    3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd Qx : [\tau/\alpha]A.\ [\tau/\alpha]p : \mathsf{prop}$
    4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qx : A.\ p) : \mathsf{prop}$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qx : A.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd [\tau/\alpha]A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; [\tau/\alpha]\Gamma, x : [\tau/\alpha]A; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta, \alpha:\kappa \rhd A : \mathsf{sort}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)} \\ \quad [\![\Theta, \alpha : \kappa; \Gamma, x : A; \Delta \rhd p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ (\gamma, v)\ \delta\end{array} & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qx : A.\ p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

14. Case TQUANTIFY3: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Q\beta : \kappa'.\ p : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd p : \mathsf{prop}$
    2 By induction, $\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}$
    3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd Q\beta : \kappa'.\ [\tau/\alpha]p : \mathsf{prop}$
    4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Q\beta : \kappa'.\ p) : \mathsf{prop}$

    For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Q\beta : \kappa'.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}]\!]\ (\theta, \tau')\ \gamma\ \delta\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta, \alpha:\kappa \rhd \kappa' : \mathsf{sort}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)} \\ \quad [\![\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta, \tau')\ \gamma\ (\delta, v)\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta \rhd Q\beta : \kappa'.\ p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

    The correctness of the application of γ and δ follows from the equations for contexts under substitution. We also silently permute the context in the second line.

15. Case TEQUAL: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p =_\omega q : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega$
    2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd q : \omega$
    3 By inversion, $\Theta, \alpha : \kappa \rhd \omega : \mathsf{sort}$
    4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega$
    5 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]q : [\tau/\alpha]\omega$
    6 By substitution, $\Theta \rhd [\tau/\alpha]\omega : \mathsf{sort}$
    7 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p =_\omega q) : \mathsf{prop}$

    For the semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](p =_\omega q) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \text{if } [\![[\tau/\alpha]p]\!]\theta\ \gamma\ \delta = [\![[\tau/\alpha]q]\!]\theta\ \gamma\ \delta \text{ then } \top \text{ else } \bot & \text{Semantics} \\ = & \text{if } [\![p]\!](\theta, [\![\Theta \vdash \tau : \kappa]\!]\theta)\ \gamma\ \delta = [\![q]\!](\theta, [\![\Theta \vdash \tau : \kappa]\!]\theta)\ \gamma\ \delta \text{ then } \top \text{ else } \bot & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd (p =_\omega q) : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution. We also need to use the equality of sorts under substitution to justify forming the equality in the third line.

16. Case TPOINTSTO: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd e \mapsto_A e' : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd e : \mathsf{ref}\ A$
    2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd e' : A$
    3 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]e : [\tau/\alpha]\mathsf{ref}\ A$
    4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]e' : [\tau/\alpha]A$
    5 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](e \mapsto_A e') : \mathsf{prop}$

    For the semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](e \mapsto_A e') : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} & [\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]e : [\tau/\alpha]\mathsf{ref}\ A]\!]\ \theta\ \gamma & \\ = & \mapsto & \text{Semantics} \\ & [\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]e' : [\tau/\alpha]A]\!]\ \theta\ \gamma & \\ & [\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : \mathsf{ref}\ A]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma & \\ = & \mapsto & \text{Induction} \\ & [\![\Theta, \alpha : \kappa;\ \Gamma \vdash e' : A]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma & \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd e \mapsto_A e' : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma & \text{Semantics} \end{array}$$

    The correctness of the application of γ and δ follows from the equations for contexts under substitution.

17. Case TEQSORT: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega$

    First, the syntax:

    1 By inversion, $\Theta, \alpha : \kappa \rhd \omega \equiv \omega' : \mathsf{sort}$
    2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega'$
    3 By substitution, $\Theta \rhd [\tau/\alpha]\omega \equiv [\tau/\alpha]\omega' : \mathsf{sort}$
    4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega'$
    5 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega$

    For the semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & [\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : [\tau/\alpha]\omega']\!]\ \theta\ \gamma\ \delta & \text{Semantics} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega']\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \omega]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

    The correctness of the application of γ and δ follows from the equations for contexts under substitution.

18. Case TSPEC: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}$:

    First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}$
2 By mutual induction $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}$
3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S\ \mathsf{spec} : \mathsf{prop}$

For the semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \text{if } [\![[\tau/\alpha]S]\!]\ \theta\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Semantics} \\ = & \text{if } [\![S]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

19. Case SPECTRIPLE: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}$
2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd [c] : \bigcirc A$
3 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta, a : A \rhd q : \mathsf{prop}$
4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}$
5 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha][c] : [\tau/\alpha] \bigcirc A$
6 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, [\tau/\alpha]a : A \rhd [\tau/\alpha]q : \mathsf{prop}$
7 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\{p\}c\{a : A.\ q\}) : \mathsf{spec}$

For the semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\{p\}c\{a : A.\ q\}) : \mathsf{spec}]\!]\ \theta; \gamma\ \delta$

$$\begin{array}{lll} & \{[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta\} & \\ = & [\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]c \div A]\!]\ \theta\ \gamma\ \delta & \text{Semantics} \\ & \{v.\ [\![\Theta; [\tau/\alpha]\Gamma, a : A; [\tau/\alpha]\Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\} & \\ & \{[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta\} & \\ = & [\![\Theta, \alpha : \kappa;\ \Gamma \vdash c \div A]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Induction} \\ & \{v.\ [\![\Theta, \alpha : \kappa; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ (\gamma, v)\ \delta\} & \\ = & [\![\Theta; \Gamma; \Delta \rhd (\{p\}c\{a : A.\ q\}) : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

20. Case SPECMTRIPLE: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \langle p\rangle e\langle a : A.\ q\rangle : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}$
2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd e : \bigcirc A$
3 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta, a : A \rhd q : \mathsf{prop}$
4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}$
5 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]e : [\tau/\alpha] \bigcirc A$
6 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, [\tau/\alpha]a : A \rhd [\tau/\alpha]q : \mathsf{prop}$
7 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec}$

For the semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](\langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec}]\!]\ \theta; \gamma\ \delta$

$$
\begin{array}{lll}
 & \{[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta\} & \\
= & [\![\Theta;\ [\tau/\alpha]\Gamma \vdash [\tau/\alpha]e : \bigcirc A]\!]\ \theta\ \gamma\ \delta & \text{Semantics} \\
 & \{v.\ [\![\Theta; [\tau/\alpha]\Gamma, a : A; [\tau/\alpha]\Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\} & \\
 & \{[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta\} & \\
= & [\![\Theta, \alpha : \kappa;\ \Gamma \vdash e : \bigcirc A]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Induction} \\
 & \{v.\ [\![\Theta, \alpha : \kappa; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ (\gamma, v)\ \delta\} & \\
= & [\![\Theta; \Gamma; \Delta \rhd (\langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta); \gamma\ \delta & \text{Semantics}
\end{array}
$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

21. Case SPECQUANTIFY1: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qu : v.\ S : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta, u : v \rhd S : \mathsf{spec}$
2 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, u : [\tau/\alpha]v \rhd [\tau/\alpha]S : \mathsf{spec}$
3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd Qu : [\tau/\alpha]v.\ [\tau/\alpha]S : \mathsf{spec}$
4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qu : v.\ S) : \mathsf{spec}$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qu : v.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd [\tau/\alpha]v : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta, u : [\tau/\alpha]v \rhd [\tau/\alpha]S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, v)\end{array} & \text{Semantics} \\
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta, \alpha : \kappa \rhd v : \mathsf{sort}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)} \\ \quad [\![\Theta, \alpha : \kappa; \Gamma; \Delta, u : v \rhd S : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ (\delta, v)\end{array} & \text{Induction} \\
= & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qu : v.\ S : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics}
\end{array}
$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

22. Case SPECQUANTIFY2: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qx : A.\ S : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma, x : A; \Delta \rhd S : \mathsf{spec}$
2 By induction, $\Theta; [\tau/\alpha]\Gamma, x : [\tau/\alpha]A; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}$
3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd Qx : [\tau/\alpha]A.\ [\tau/\alpha]S : \mathsf{spec}$
4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qx : A.\ S) : \mathsf{spec}$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Qx : A.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd [\tau/\alpha]A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; [\tau/\alpha]\Gamma, x : [\tau/\alpha]A; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}]\!]\ \theta\ (\gamma, v)\ \delta\end{array} & \text{Semantics} \\
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta, \alpha : \kappa \rhd A : \mathsf{sort}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)} \\ \quad [\![\Theta, \alpha : \kappa; \Gamma, x : A; \Delta \rhd S : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ (\gamma, v)\ \delta\end{array} & \text{Induction} \\
= & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qx : A.\ S : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics}
\end{array}
$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

23. Case SPECQUANTIFY3: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Q\beta : \kappa'.\ S : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd S : \mathsf{spec}$
2 By induction, $\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}$
3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd Q\beta : \kappa'.\ [\tau/\alpha]S : \mathsf{spec}$
4 By def of subst, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Q\beta : \kappa'.\ S) : \mathsf{spec}$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](Q\beta : \kappa'.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}]\!]\ (\theta, \tau')\ \gamma\ \delta\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta, \alpha : \kappa \rhd \kappa' : \mathsf{sort}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)} \\ \quad [\![\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta, \tau')\ \gamma\ (\delta, v)\end{array} & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Q\beta : \kappa'.\ S : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution. We also silently permute the context in the second line.

24. Case SPECBINARY: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S \oplus S' : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}$
2 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S' : \mathsf{spec}$
3 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}$
4 By induction, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S' : \mathsf{spec}$
5 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S \oplus [\tau/\alpha]S' : \mathsf{spec}$
6 By subst def, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](S \oplus S') : \mathsf{spec}$

For semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha](S \oplus S') : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta)\ [\![\oplus]\!] \\ \quad ([\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]S' : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\ = & \begin{array}{l}([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta)[\![\oplus]\!] \\ \quad ([\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S' : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S \oplus S' : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

25. Case TSPEC: $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}$:

First, the syntax:

1 By inversion, $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}$
2 By mutual induction $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd [\tau/\alpha]p : \mathsf{prop}$
3 By rule, $\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd \{[\tau/\alpha]p\} : \mathsf{spec}$

For the semantics, consider $[\![\Theta; [\tau/\alpha]\Gamma; [\tau/\alpha]\Delta \rhd \{[\tau/\alpha]p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \text{if } [\![[\tau/\alpha]p]\!]\ \theta\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Semantics} \\ = & \text{if } [\![p]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Induction} \\ = & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ (\theta, [\![\Theta \vdash \tau : \kappa]\!]\ \theta)\ \gamma\ \delta & \text{Semantics} \end{array}$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

First, assume $\Theta;\ \Gamma \vdash e'' : B$, and $\Theta; \Gamma, y : B; \Delta \rhd p : \omega$, and $\Theta; \Gamma, y : B; \Delta \rhd S : \mathsf{spec}$.
Now, we proceed by mutual induction on the derivation of p and S:

1. Case TTYPE: $\Theta; \Gamma, y : B; \Delta \rhd \tau' : \kappa'$

   First, the syntax:

   1 By inversion, we know $\Theta \vdash \Delta$
   2 By inversion, we know $\Theta \vdash \Gamma, y : B$
   3 By inversion, we know $\Theta \vdash \tau' : \kappa'$
   4 By inversion, we know $\Theta \vdash \Gamma$
   5 By rule, $\Theta; \Gamma; \Delta \rhd \tau' : \kappa'$

   For semantics, $[\![\Theta; \Gamma; \Delta \rhd [e''/y]\tau' : \kappa']\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & [\![\Theta \vdash \tau' : \kappa']\!]\ \theta & \text{Semantics} \\ = & [\![\Theta; \Gamma, y : B; \Theta \rhd \tau' : \kappa']\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

2. Case TEXPR: $\Theta; \Gamma, y : B; \Delta \rhd e : A$

   First, the syntax:

   1 By inversion, we know $\Theta \vdash \Delta$
   2 By inversion, we know $\Theta;\ \Gamma, y : B \vdash e : A$
   3 By substitution, we know $\Theta;\ \Gamma \vdash [e''/y]e : A$
   4 By rule, we know $\Theta; \Gamma; \Delta \rhd [e''/y]e : A$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y]e : A]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & [\![\Theta;\ \Gamma \vdash [e''/y]e : A]\!]\ \theta\ \gamma & \text{Semantics} \\ = & [\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma) & \text{Substitution} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd e : A]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

3. Case THYP: $\Theta; \Gamma, y : B; \Delta \rhd u : \upsilon$

   First, the syntax:

   1 By inversion, we know $\Theta \vdash \Gamma, y : B$
   2 By inversion, we know $\Theta \vdash \Delta$
   3 By inversion, we know $\Theta \vdash \Gamma$
   4 By rule, we know $\Theta; \Gamma; \Delta \rhd u : A$
   5 Hence, we know $\Theta; \Gamma; \Delta \rhd [e''/y]u : A$

   Next, consider $[\![\Theta; \Delta; \Gamma \rhd u : A]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & \pi_u(\delta) & \text{Semantics} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd u : A]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

4. Case TABS1: $\Theta; \Gamma, y : B; \Delta \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon$

   First, the syntax:

   1 By inversion, $\Theta; \Gamma, y : B; \Delta, u : \upsilon' \rhd p : \upsilon$
   2 By induction, $\Theta; \Gamma; \Delta, u : \upsilon' \rhd [e''/y]p : \upsilon$
   3 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\lambda} u : \upsilon'.\ [e''/y]p : \upsilon' \Rightarrow \upsilon$
   4 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](\hat{\lambda} u : \upsilon'.\ p) : \upsilon' \Rightarrow \upsilon$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd (\hat{\lambda} u : \upsilon'.\ [e''/y]p) : \upsilon' \Rightarrow \upsilon]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & \lambda v.\ [\![\Theta; \Gamma; \Delta, u : \upsilon' \rhd [e''/y]p : \upsilon]\!]\ \theta\ \gamma\ (\delta, v) & \text{Semantics} \\ = & \lambda v.\ [\![\Theta; \Gamma, y : B; \Delta, u : \upsilon' \rhd p : \upsilon]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ (\delta, v) & \text{Induction} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

5. Case TABS2: $\Theta; \Gamma, y : B; \Delta \rhd \hat{\lambda} x : A.\ p : A \Rightarrow \upsilon$

   First, the syntax:

   1 By inversion, $\Theta; \Gamma, y : B, x : A; \Delta \rhd p : \upsilon$
   2 By induction, $\Theta; \Gamma, x : A; \Delta \rhd [e''/y]p : \upsilon$
   3 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\lambda} x : A.\ [e''/y]p : A \Rightarrow \upsilon$
   4 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](\hat{\lambda} x : A.\ p) : (A \Rightarrow \upsilon)$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](\hat{\lambda} x : A.\ p) : (A \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \lambda v.\ [\![\Theta; \Gamma, x : A; \Delta \rhd [e''/y]p : v]\!]\ \theta\ (\gamma, v)\ \delta & \text{Semantics} \\
= & \lambda v.\ [\![\Theta; \Gamma, y : B, x : A; \Delta \rhd p : v]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma, v)\ \delta & \text{Induction} \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd \hat{\lambda} x : A.\ p : A \Rightarrow v]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

We silently permute arguments in the second line.

6. Case TABS3: $\Theta; \Gamma, y : B; \Delta \rhd \hat{\lambda}\beta : \kappa'.\ p : \kappa' \Rightarrow v$

   First, the syntax:

   1 By inversion, $\Theta, \beta : \kappa'; \Gamma, y : B; \Delta \rhd p : v$
   2 By weakening, $\Theta, \beta : \kappa';\ \Gamma \vdash e'' : B$
   3 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]p : v$
   4 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\lambda}\beta : \kappa'.\ [e''/y]p : \kappa' \Rightarrow v$
   5 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](\hat{\lambda}\beta : \kappa'.\ p) : \kappa' \Rightarrow v$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](\hat{\lambda}\beta : \kappa'.\ p) : \kappa' \Rightarrow v]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]p : v]\!]\ (\theta, \tau)\ \gamma\ \delta & \text{Semantics} \\
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma, y : B; \Delta \rhd p : v]\!]\ (\theta, \tau)\ (\gamma, [\![\Theta, \beta : \kappa';\ \Gamma \vdash e'' : B]\!]\ (\theta, \tau)\ \gamma)\ \delta & \text{Induction} \\
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma, y : B; \Delta \rhd p : v]\!]\ (\theta, \tau)\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Strengthening} \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd \hat{\lambda}\beta : \kappa'.\ p : \kappa' \Rightarrow v]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

   This case relies upon the fact that Γ and Δ do not have β free and the equality of sorts under substitution.

7. Case TABSALL: $\Theta; \Gamma, y : B; \Delta \rhd \hat{\underline{\lambda}}\beta : \kappa'.\ p : \Pi\beta : \kappa'.\ v$

   First, the syntax:

   1 By inversion, $\Theta, \beta : \kappa'; \Gamma, y : B; \Delta \rhd p : v$
   2 By weakening, $\Theta, \beta : \kappa';\ \Gamma \vdash e'' : B$
   3 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]p : v$
   4 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\underline{\lambda}}\beta : \kappa'.\ [e''/y]p : \Pi\beta : \kappa'.\ v$
   5 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](\hat{\underline{\lambda}}\beta : \kappa'.\ p) : \Pi\beta : \kappa'.\ v$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](\hat{\lambda}\beta : \kappa'.\ p) : \Pi\beta : \kappa'.\ v]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]p : v]\!]\ (\theta, \tau)\ \gamma\ \delta & \text{Semantics} \\
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma, y : B; \Delta, \beta : \kappa' \rhd p : v]\!]\ (\theta, \tau)\ (\gamma, [\![\Theta, \beta : \kappa';\ \Gamma \vdash e'' : B]\!]\ (\theta, \tau)\ \gamma)\ \delta & \text{Induction} \\
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma, y : B; \Delta, \beta : \kappa' \rhd p : v]\!]\ (\theta, \tau)\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Strengthen} \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd \hat{\lambda}\beta : \kappa'.\ p : \Pi\beta : \kappa'.\ v]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

   This case relies upon the fact that Γ and Δ do not have β free and the equality of sorts under substitution.

8. Case TAPP: $\Theta; \Gamma, y : B; \Delta \rhd p\ q : \upsilon$

   First, the syntax:

   1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \omega \Rightarrow \upsilon$
   2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd q : \omega$
   3 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]p : (\omega \Rightarrow \upsilon)$
   4 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]q : \omega$
   5 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y](p\ q) : \omega \Rightarrow \upsilon$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](p\ q) : \upsilon]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & ([\![\Theta; \Gamma; \Delta \rhd [e''/y]p : (\omega \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \\ & \quad ([\![\Theta; \Gamma; \Delta \rhd [e''/y]q : \omega]\!]\ \theta\ \gamma\ \delta) & \\ = & ([\![\Theta; \Gamma, y : B; \Delta \rhd p : \omega \Rightarrow \upsilon]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta) & \text{Induction} \\ & \quad ([\![\Theta; \Gamma, y : B; \Delta \rhd q : \omega]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta) & \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd p\ q : \upsilon]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

9. Case TAPPALL: $\Theta; \Gamma, y : B; \Delta \rhd p\ [\tau'] : [\tau'/\beta]\upsilon$

   First, the syntax:

   1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \Pi\beta : \kappa'.\ \upsilon$
   2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd \tau' : \kappa'$
   3 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]p : (\Pi\beta : \kappa'.\ \upsilon)$
   4 By induction, $\Theta; \Gamma; \Delta \rhd \tau' : \kappa'$
   5 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y](p\ [\tau']) : [\tau/\alpha, [\tau/\alpha]\tau'/\beta]\upsilon$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](p\ [\tau']) : \upsilon]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & ([\![\Theta; \Gamma; \Delta \rhd [e''/y]p : (\Pi\beta : \kappa'.\ \upsilon)]\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \\ & \quad ([\![\Theta; \Gamma; \Delta \rhd \tau' : \kappa']\!]\ \theta\ \gamma\ \delta) & \\ = & ([\![\Theta; \Gamma, y : B; \Delta \rhd p : \Pi\beta : \kappa'.\ \upsilon]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta) & \text{Induction} \\ & \quad ([\![\Theta; \Gamma, y : B; \Delta \rhd \tau' : \kappa']\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta) & \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd p\ [\tau'] : \upsilon]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

10. Case TCONST:

    First, the syntax:

    1 By inversion, $\Theta \vdash \Gamma, y : B$
    2 By inversion, $\Theta \vdash \Delta$
    3 By inversion, $\Theta \vdash \Gamma$
    4 By rule, $\Theta; \Gamma; \Delta \rhd c : \mathsf{prop}$

For semantics consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y]c : \mathsf{prop}]\!]\ \theta; \gamma\ \delta$

$$\begin{array}{lll} = & [\![c]\!]^0 & \text{Semantics} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd c : \mathsf{prop}]\!]\ \theta; (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

11. Case TBINARY: $\Theta; \Gamma, y : B; \Delta \rhd p \oplus q : \mathsf{prop}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}$
2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd q : \mathsf{prop}$
3 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]q : \mathsf{prop}$
5 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y]p \oplus [e''/y]q : \mathsf{prop}$
6 By subst def, $\Theta; \Gamma; \Delta \rhd [e''/y](p \oplus q) : \mathsf{prop}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](p \oplus q) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}([\![\Theta; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)\ [\![\oplus]\!]^2 \\ \quad ([\![\Theta; \Gamma; \Delta \rhd [e''/y]q : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\ = & \begin{array}{l}([\![\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta)[\![\oplus]\!]^2 \\ \quad ([\![\Theta; \Gamma, y : B; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd p \oplus q : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

12. Case TQUANTIFY1: $\Theta; \Gamma, y : B; \Delta \rhd Qu : \upsilon.\ p : \mathsf{prop}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta, u : \upsilon \rhd p : \mathsf{prop}$
2 By induction, $\Theta; \Gamma; \Delta, u : \upsilon \rhd [e''/y]p : \mathsf{prop}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ [e''/y]p : \mathsf{prop}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](Qu : \upsilon.\ p) : \mathsf{prop}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](Qu : \upsilon.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd [e''/y]p : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, v)\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma, y : B; \Delta, u : \upsilon \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ (\delta, v)\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

13. Case TQUANTIFY2: $\Theta; \Gamma, y : B; \Delta \rhd Qx : A.\ p : \mathsf{prop}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B, x : A; \Delta \rhd p : \mathsf{prop}$
2 By induction, $\Theta; \Gamma, x : A; \Delta \rhd [e''/y]p : \mathsf{prop}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Qx : A.\ [e''/y]p : \mathsf{prop}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](Qx : A.\ p) : \mathsf{prop}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](Qx : A.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma, x : A; \Delta \rhd [e''/y]p : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\end{array} & \text{Semantics} \\
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta, \alpha : \kappa \rhd A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma, y : B, x : A; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma, v)\ \delta\end{array} & \text{Induction} \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd Qx : A.\ p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

Here, we make use of the fact that x is not free in e'', and we silently permute the context as needed.

14. Case TQUANTIFY3: $\Theta; \Gamma, y : B; \Delta \rhd Q\beta : \kappa'.\ p : \mathsf{prop}$

First, the syntax:

1 By inversion, $\Theta, \beta : \kappa'; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}$
2 By weakening, $\Theta, \beta : \kappa';\ \Gamma \vdash e'' : B$
3 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}$
4 By rule, $\Theta; \Gamma; \Delta \rhd Q\beta : \kappa'.\ [e''/y]p : \mathsf{prop}$
5 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](Q\beta : \kappa'.\ p) : \mathsf{prop}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](Q\beta : \kappa'.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}]\!]\ (\theta, \tau')\ \gamma\ \delta\end{array} & \text{Semantics} \\
= & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}]\!]\ (\theta, \tau)\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta\end{array} & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta \rhd Q\beta : \kappa'.\ p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

In this case we silently use the fact that β does not occur free in e'' or B.

15. Case TEQUAL: $\Theta; \Gamma, y : B; \Delta \rhd p =_\omega q : \mathsf{prop}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \omega$
2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd q : \omega$
3 By inversion, $\Theta \rhd \omega : \mathsf{sort}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]p : \omega$
5 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]q : \omega$
6 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y](p =_\omega q) : \mathsf{prop}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](p =_\omega q) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \text{if } [\![[e''/y]p]\!]\theta\ \gamma\ \delta = [\![[e''/y]q]\!]\theta\ \gamma\ \delta \text{ then } \top \text{ else } \bot & \text{Semantics} \\
= & \text{if } [\![p]\!]\ \theta\ (\gamma, [\![e'']\!]\theta\ \gamma)\ \delta = [\![q]\!]\ \theta\ (\gamma, [\![e'']\!]\theta\ \gamma)\ \delta \text{ then } \top \text{ else } \bot & \text{Induction} \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd (p =_\omega q) : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

16. Case TPOINTSTO: $\Theta; \Gamma, y : B; \Delta \rhd e \mapsto_A e' : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd e : \mathsf{ref}\ A$
    2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd e' : A$
    3 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]e : \mathsf{ref}\ A$
    4 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]e' : A$
    5 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y](e \mapsto_A e') : \mathsf{prop}$

    For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](e \mapsto_A e') : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} & [\![\Theta;\ \Gamma \vdash [e''/y]e : \mathsf{ref}\ A]\!]\ \theta\ \gamma & \\ = & \mapsto & \text{Semantics} \\ & [\![\Theta;\ \Gamma \vdash [e''/y]e' : A]\!]\ \theta\ \gamma & \\ & [\![\Theta;\ \Gamma, y : B \vdash e : \mathsf{ref}\ A]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma) & \\ = & \mapsto & \text{Induction} \\ & [\![\Theta;\ \Gamma, y : B \vdash e' : A]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma) & \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd e \mapsto_A e' : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

17. Case TEQSORT: $\Theta; \Gamma, y : B; \Delta \rhd p : \omega$

    First, the syntax:

    1 By inversion, $\Theta \rhd \omega \equiv \omega' : \mathsf{sort}$
    2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \omega'$
    3 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]p : \omega'$
    4 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y]p : \omega$

    For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y]p : \omega]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & [\![\Theta; \Gamma; \Delta \rhd [e''/y]p : \omega']\!]\ \theta\ \gamma\ \delta & \text{Semantics} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd p : \omega']\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Induction} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd p : \omega]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

18. Case TSPEC: $\Theta; \Gamma, y : B; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}$:

    First, the syntax:

    1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd S : \mathsf{spec}$
    2 By mutual induction $\Theta; \Gamma; \Delta \rhd [e''/y]S : \mathsf{spec}$
    3 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y]S\ \mathsf{spec} : \mathsf{prop}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y]S \ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \text{if } [\![[e''/y]S]\!]\ \theta\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Semantics} \\
= & \text{if } [\![S]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Induction} \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd S \ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

19. Case SPECTRIPLE: $\Theta; \Gamma, y : B; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}$
2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd [c] : \bigcirc A$
3 By inversion, $\Theta; \Gamma, y : B; \Delta, a : A \rhd q : \mathsf{prop}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}$
5 By induction, $\Theta; \Gamma; \Delta \rhd [[e''/y]c] : \bigcirc A$
6 By induction, $\Theta; \Gamma; \Delta, a : A \rhd [e''/y]q : \mathsf{prop}$
7 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y](\{p\}c\{a : A.\ q\}) : \mathsf{spec}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](\{p\}c\{a : A.\ q\}) : \mathsf{spec}]\!]\ \theta; \gamma\ \delta$

$$
\begin{array}{lll}
 & \{[\![\Theta; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta\} & \\
= & [\![\Theta;\ \Gamma \vdash [e''/y]c \div A]\!]\ \theta\ \gamma & \text{Semantics} \\
 & \{v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd [e''/y]q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\} & \\
 & \{[\![\Theta, \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta\} & \\
= & [\![\Theta;\ \Gamma, y : B \vdash c \div A]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma) & \text{Induction} \\
 & \{v.\ [\![\Theta; \Gamma, y : B, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma, v)\ \delta\} & \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd (\{p\}c\{a : A.\ q\}) : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

The correctness of the application of γ and δ follows from the equations for contexts under substitution.

20. Case SPECMTRIPLE: $\Theta; \Gamma, y : B; \Delta \rhd \langle p\rangle e\langle a : A.\ q\rangle : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}$
2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd e : \bigcirc A$
3 By inversion, $\Theta; \Gamma, y : B; \Delta, a : A \rhd q : \mathsf{prop}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}$
5 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]e : \bigcirc A$
6 By induction, $\Theta; \Gamma; \Delta, a : A \rhd [e''/y]q : \mathsf{prop}$
7 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y](\langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](\langle p \rangle e \langle a : A.\ q \rangle) : \mathsf{spec}]\!]\ \theta; \gamma\ \delta$

$$
\begin{array}{lll}
 & \{[\![\Theta; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta\} & \\
= & [\![\Theta;\ \Gamma \vdash [e''/y]e : \bigcirc A]\!]\ \theta\ \gamma & \text{Semantics} \\
 & \{v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd [e''/y]q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\} & \\
 & \{[\![\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta\} & \\
= & [\![\Theta;\ \Gamma, y : B \vdash e : \bigcirc A]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma) & \text{Induction} \\
 & \{v.\ [\![\Theta; \Gamma, y : B, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma, v)\ \delta\} & \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd (\langle p \rangle e \langle a : A.\ q \rangle) : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

21. Case SPECQUANTIFY1: $\Theta; \Gamma, y : B; \Delta \rhd Qu : \upsilon.\ S : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta, u : \upsilon \rhd S : \mathsf{spec}$
2 By induction, $\Theta; \Gamma; \Delta, u : \upsilon \rhd [e''/y]S : \mathsf{spec}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ [e''/y]S : \mathsf{spec}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](Qu : \upsilon.\ S) : \mathsf{spec}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](Qu : \upsilon.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & [\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} & \\
 & [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd [e''/y]S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, v) & \text{Semantics} \\
= & [\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} & \\
 & [\![\Theta; \Gamma, y : B; \Delta, u : \upsilon \rhd S : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ (\delta, v) & \text{Induction} \\
= & [\![\Theta; \Gamma, y : B; \Delta \rhd Qu : \upsilon.\ S : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

22. Case SPECQUANTIFY2: $\Theta; \Gamma, y : B; \Delta \rhd Qx : A.\ S : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B, x : A; \Delta \rhd S : \mathsf{spec}$
2 By induction, $\Theta; \Gamma, x : A; \Delta \rhd [e''/y]S : \mathsf{spec}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Qx : A.\ [e''/y]S : \mathsf{spec}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](Qx : A.\ S) : \mathsf{spec}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](Qx : A.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & [\![Q]\!]_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta} & \\
 & [\![\Theta; \Gamma, x : A; \Delta \rhd [e''/y]S : \mathsf{spec}]\!]\ \theta\ (\gamma, v)\ \delta & \text{Semantics} \\
= & [\![Q]\!]_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta} & \\
 & [\![\Theta; \Gamma, y : A, x : A; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma, v)\ \delta & \text{Induction} \\
= & [\![\Theta; \Gamma, y : A; \Delta \rhd Qx : A.\ S : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics}
\end{array}
$$

23. Case SPECQUANTIFY3: $\Theta; \Gamma, y : A; \Delta \rhd Q\beta : \kappa'.\ S : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta, \beta : \kappa'; \Gamma, y : A; \Delta \rhd S : \mathsf{spec}$
2 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]S : \mathsf{spec}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Q\beta : \kappa'.\ [e''/y]S : \mathsf{spec}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [e''/y](Q\beta : \kappa'.\ S) : \mathsf{spec}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](Q\beta : \kappa'.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [e''/y]S : \mathsf{spec}]\!]\ (\theta, \tau')\ \gamma\ \delta\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \alpha : \kappa, \beta : \kappa'; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ (\theta, \tau')\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta \rhd Q\beta : \kappa'.\ S : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

24. Case SPECBINARY: $\Theta; \Gamma, y : B; \Delta \rhd S \oplus S' : \mathsf{spec}$
First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd S : \mathsf{spec}$
2 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd S' : \mathsf{spec}$
3 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]S : \mathsf{spec}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [e''/y]S' : \mathsf{spec}$
5 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y]S \oplus [e''/y]S' : \mathsf{spec}$
6 By subst def, $\Theta; \Gamma; \Delta \rhd [e''/y](S \oplus S') : \mathsf{spec}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y](S \oplus S') : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \begin{array}{l}([\![\Theta; \Gamma; \Delta \rhd [e''/y]S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta)\ [\![\oplus]\!] \\ \quad ([\![\Theta; \Gamma; \Delta \rhd [e''/y]S' : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\ = & \begin{array}{l}([\![\Theta; \Gamma, y : B; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta)[\![\oplus]\!] \\ \quad ([\![\Theta; \Gamma, y : B; \Delta \rhd S' : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd S \oplus S' : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

25. Case TSPEC: $\Theta; \Gamma, y : B; \Delta \rhd \{p\} : \mathsf{spec}$:
First, the syntax:

1 By inversion, $\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}$
2 By mutual induction $\Theta; \Gamma; \Delta \rhd [e''/y]p : \mathsf{prop}$
3 By rule, $\Theta; \Gamma; \Delta \rhd [e''/y]\{p\} : \mathsf{spec}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [e''/y]\{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \text{if } [\![[e''/y]p]\!]\ \theta\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Semantics} \\ = & \text{if } [\![p]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Induction} \\ = & [\![\Theta; \Gamma, y : B; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ (\gamma, [\![\Theta;\ \Gamma \vdash e'' : B]\!]\ \theta\ \gamma)\ \delta & \text{Semantics} \end{array}$$

Finally, assume $\Theta; \Gamma; \Delta \rhd r : \upsilon''$, and $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega$, and $\Theta; \Gamma, y : B; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}$.

Now, we proceed by mutual induction on the derivation of p and S:

1. Case TTYPE: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \tau' : \kappa'$

   First, the syntax:

   1 By inversion, we know $\Theta \vdash \Delta, b : \upsilon''$
   2 By inversion, we know $\Theta \vdash \Gamma$
   3 By inversion, we know $\Theta \vdash \Delta$
   4 By rule, $\Theta; \Gamma; \Delta \rhd \tau' : \kappa'$

   For semantics, $[\![\Theta; \Gamma; \Delta \rhd [r/b]\tau' : \kappa']\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & [\![\Theta \vdash \tau' : \kappa']\!]\ \theta & \text{Semantics} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \tau' : \kappa']\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

2. Case TEXPR: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd e : A$

   First, the syntax:

   1 By inversion, we know $\Theta \vdash \Delta, b : \upsilon''$
   2 By inversion, we know $\Theta \vdash \Delta$
   3 By inversion, we know $\Theta;\ \Gamma \vdash e : A$
   4 By substitution, we know $\Theta;\ \Gamma \vdash [r/b]e : A$
   5 By rule, we know $\Theta; \Gamma; \Delta \rhd [r/b]e : A$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b]e : A]\!]\ \theta\ \gamma\ \delta$

   $$\begin{array}{lll} = & [\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ \gamma & \text{Semantics} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd e : A]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

3. Case THYP: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd u : \upsilon$

   There are two cases.

   - Case $u = b$: (hence $\upsilon'' = \upsilon$)

     In this case, the syntax follows since $\Theta; \Gamma; \Delta \rhd r : \upsilon''$

     For semantics, consider that $[\![\Theta; \Gamma; \Delta \rhd r : \upsilon]\!]\ \theta; \gamma\ \delta$

     $$\begin{array}{lll} = & [\![\Theta; \Gamma; \Delta, b : \upsilon \rhd b : \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon]\!]\ \theta\ \gamma\ \delta) & \text{Definition} \end{array}$$

   - Case $u \neq b$:

     First, the syntax:

1 By strengthening, $\Theta; \Gamma; \Delta \rhd [r/b]u : \upsilon$

For semantics, consider that $[\![\Theta; \Gamma; \Delta \rhd [r/b]u : \upsilon]\!]\ \theta; \gamma\ \delta$

$$\begin{array}{lll} = & \pi_u(\delta) & \text{Semantics} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd u : \upsilon]\!]\ \theta; \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

4. Case TABS1: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon$
First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'', u : \upsilon' \rhd p : \upsilon$
2 By induction, $\Theta; \Gamma; \Delta, u : \upsilon' \rhd [r/b]p : \upsilon$
3 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\lambda} u : \upsilon'.\ [r/b]p : \upsilon' \Rightarrow \upsilon$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](\hat{\lambda} u : \upsilon'.\ p) : \upsilon' \Rightarrow \upsilon$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd (\hat{\lambda} u : \upsilon'.\ [r/b]p) : \upsilon' \Rightarrow \upsilon]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \lambda v.\ [\![\Theta; \Gamma; \Delta, u : \upsilon' \rhd [r/b]p : \upsilon]\!]\ \theta\ \gamma\ (\delta, v) & \text{Semantics} \\ = & \lambda v.\ [\![\Theta; \Gamma; \Delta, b : \upsilon'', u : \upsilon' \rhd p : \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta, v) & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\lambda} u : \upsilon'.\ p : \upsilon' \Rightarrow \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

5. Case TABS2: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\lambda} x : A.\ p : A \Rightarrow \upsilon$
First, the syntax:

1 By inversion, $\Theta; \Gamma, x : A; \Delta, b : \upsilon'' \rhd p : \upsilon$
2 By induction, $\Theta; \Gamma, x : A; \Delta \rhd [r/b]p : \upsilon$
3 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\lambda} x : A.\ [r/b]p : A \Rightarrow \upsilon$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](\hat{\lambda} x : A.\ p) : (A \Rightarrow \upsilon)$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](\hat{\lambda} x : A.\ p) : (A \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \lambda v.\ [\![\Theta; \Gamma, x : A; \Delta \rhd [r/b]p : \upsilon]\!]\ \theta\ (\gamma, v)\ \delta & \text{Semantics} \\ = & \lambda v.\ [\![\Theta; \Gamma, x : A; \Delta, b : \upsilon'' \rhd p : \upsilon]\!]\ \theta\ (\gamma, v)\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\lambda} x : A.\ p : A \Rightarrow \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

We silently permute arguments in the second line.

6. Case TABS3: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\lambda} \beta : \kappa'.\ p : \kappa' \Rightarrow \upsilon$
First, the syntax:

1 By inversion, $\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'' \rhd p : \upsilon$
2 By weakening, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd r : \upsilon''$
3 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]p : \upsilon$
4 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\lambda}\beta : \kappa'.\ [r/b]p : \kappa' \Rightarrow \upsilon$
5 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](\hat{\lambda}\beta : \kappa'.\ p) : \kappa' \Rightarrow \upsilon$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](\hat{\lambda}\beta : \kappa'.\ p) : \kappa' \Rightarrow \upsilon]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]p : \upsilon]\!]\ (\theta, \tau)\ \gamma\ \delta & \text{Semantics} \\
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'' \rhd p : \upsilon]\!]\ (\theta, \tau)\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\lambda}\beta : \kappa'.\ p : \kappa' \Rightarrow \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

This case relies upon the fact that Γ and Δ do not have β free and the equality of sorts under substitution.

7. Case TABSALL: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\underline{\lambda}}\beta : \kappa'.\ p : \Pi\beta : \kappa'.\ \upsilon$
First, the syntax:

1 By inversion, $\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'' \rhd p : \upsilon$
2 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]p : \upsilon$
3 By rule, $\Theta; \Gamma; \Delta \rhd \hat{\underline{\lambda}}\beta : \kappa'.\ [r/b]p : \Pi\beta : \kappa'.\ \upsilon$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](\hat{\underline{\lambda}}\beta : \kappa'.\ p) : \Pi\beta : \kappa'.\ \upsilon$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](\hat{\lambda}\beta : \kappa'.\ p) : \Pi\beta : \kappa'.\ \upsilon]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]p : \upsilon]\!]\ (\theta, \tau)\ \gamma\ \delta & \text{Semantics} \\
= & \lambda\tau.\ [\![\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'', \beta : \kappa' \rhd p : \upsilon]\!]\ (\theta, \tau)\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \hat{\lambda}\beta : \kappa'.\ p : \Pi\beta : \kappa'.\ \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

This case relies upon the fact that Γ and Δ do not have β free and the equality of sorts under substitution.

8. Case TAPP: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p\ q : \upsilon$
First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega \Rightarrow \upsilon$
2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd q : \omega$
3 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]p : (\omega \Rightarrow \upsilon)$
4 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]q : \omega$
5 By rule, $\Theta; \Gamma; \Delta \rhd [r/b](p\ q) : \omega \Rightarrow \upsilon$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](p\ q) : \upsilon]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} & ([\![\Theta; \Gamma; \Delta \rhd [r/b]p : (\omega \Rightarrow \upsilon)]\!]\ \theta\ \gamma\ \delta) & \\ = & \quad ([\![\Theta; \Gamma; \Delta \rhd [r/b]q : \omega]\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \\ & ([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega \Rightarrow \upsilon]\!]\ \theta\ \gamma\ \delta) & \\ = & \quad ([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd q : \omega]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)) & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p\ q : \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

9. Case TAPPALL: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p\ [\tau'] : [\tau'/\beta]\upsilon$
   First, the syntax:

   1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \Pi\beta : \kappa'.\ \upsilon$
   2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \tau' : \kappa'$
   3 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]p : (\Pi\beta : \kappa'.\ \upsilon)$
   4 By induction, $\Theta; \Gamma; \Delta \rhd \tau' : \kappa'$
   5 By rule, $\Theta; \Gamma; \Delta \rhd [r/b](p\ [\tau']) : [\tau/\alpha, [\tau/\alpha]\tau'/\beta]\upsilon$

   For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](p\ [\tau']) : \upsilon]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} & ([\![\Theta; \Gamma; \Delta \rhd [r/b]p : (\Pi\beta : \kappa'.\ \upsilon)]\!]\ \theta\ \gamma\ \delta) & \\ = & \quad ([\![\Theta; \Gamma; \Delta \rhd \tau' : \kappa']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \\ & ([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \Pi\beta : \kappa'.\ \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)) & \\ = & \quad ([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \tau' : \kappa']\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)) & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p\ [\tau'] : \upsilon]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

10. Case TCONST:
    First, the syntax:

    1 By inversion, $\Theta \vdash \Gamma$
    2 By inversion, $\Theta \vdash \Delta, b : \upsilon''$
    3 By inversion, $\Theta \vdash \Delta$
    4 By rule, $\Theta; \Gamma; \Delta \rhd c : \mathsf{prop}$

    For semantics consider $[\![\Theta; \Gamma; \Delta \rhd [r/b]c : \mathsf{prop}]\!]\ \theta; \gamma\ \delta$

$$\begin{array}{lll} = & [\![c]\!]^0 & \text{Semantics} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd c : \mathsf{prop}]\!]\ \theta; \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

11. Case TBINARY: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p \oplus q : \mathsf{prop}$
    First, the syntax:

    1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}$

2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd q : \mathsf{prop}$
3 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]q : \mathsf{prop}$
5 By rule, $\Theta; \Gamma; \Delta \rhd [r/b]p \oplus [r/b]q : \mathsf{prop}$
6 By subst def, $\Theta; \Gamma; \Delta \rhd [r/b](p \oplus q) : \mathsf{prop}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](p \oplus q) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{l}([\![\Theta; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)\ [\![\oplus]\!]^2 \\ \quad ([\![\Theta; \Gamma; \Delta \rhd [r/b]q : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\
= & \begin{array}{l}([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta))[\![\oplus]\!]^2 \\ \quad ([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd q : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta))\end{array} & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p \oplus q : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

12. Case TQUANTIFY1: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Qu : \upsilon.\ p : \mathsf{prop}$
First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'', u : \upsilon \rhd p : \mathsf{prop}$
2 By induction, $\Theta; \Gamma; \Delta, u : \upsilon \rhd [r/b]p : \mathsf{prop}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ [r/b]p : \mathsf{prop}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](Qu : \upsilon.\ p) : \mathsf{prop}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](Qu : \upsilon.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd [r/b]p : \mathsf{prop}]\!]\ \theta\, \gamma\, (\delta, v)\end{array} & \text{Semantics} \\
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma; \Delta, b : \upsilon'', u : \upsilon \rhd p : \mathsf{prop}]\!]\ \theta\, \gamma\, (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta, v)\end{array} & \text{Induction} \\
= & [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd Qu : \upsilon.\ p : \mathsf{prop}]\!]\ \theta\, \gamma\, (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

13. Case TQUANTIFY2: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Qx : A.\ p : \mathsf{prop}$
First, the syntax:

1 By inversion, $\Theta; \Gamma, x : A; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}$
2 By induction, $\Theta; \Gamma, x : A; \Delta \rhd [r/b]p : \mathsf{prop}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Qx : A.\ [r/b]p : \mathsf{prop}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](Qx : A.\ p) : \mathsf{prop}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](Qx : A.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma, x : A; \Delta \rhd [r/b]p : \mathsf{prop}]\!]\ \theta\, (\gamma, v)\, \delta\end{array} & \text{Semantics} \\
= & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma, x : A; \Delta, u : \upsilon'' \rhd p : \mathsf{prop}]\!]\ \theta\, (\gamma, v)\, (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Qx : A.\ p : \mathsf{prop}]\!]\ \theta\, \gamma\, (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

Here, we make use of the fact that x is not free in r, and we silently permute the context as needed.

14. Case TQUANTIFY3: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Q\beta : \kappa'.\ p : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}$
    2 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}$
    3 By rule, $\Theta; \Gamma; \Delta \rhd Q\beta : \kappa'.\ [r/b]p : \mathsf{prop}$
    4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](Q\beta : \kappa'.\ p) : \mathsf{prop}$

    For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](Q\beta : \kappa'.\ p) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}]\!]\ (\theta, \tau')\ \gamma\ \delta\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta, \alpha : \kappa \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}]\!]\ (\theta, \tau)\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Q\beta : \kappa'.\ p : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

    In this case we silently use the fact that β does not occur free in e'' or B.

15. Case TEQUAL: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p =_\omega q : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega$
    2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd q : \omega$
    3 By inversion, $\Theta \rhd \omega : \mathsf{sort}$
    4 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]p : \omega$
    5 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]q : \omega$
    6 By rule, $\Theta; \Gamma; \Delta \rhd [r/b](p =_\omega q) : \mathsf{prop}$

    For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](p =_\omega q) : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & \text{if } [\![[r/b]p]\!]\theta\ \gamma\ \delta = [\![[r/b]q]\!]\theta\ \gamma\ \delta \text{ then } \top \text{ else } \bot & \text{Semantics} \\ = & \text{if } [\![p]\!]\ \theta\ (\gamma, [\![e'']\!]\theta\ \gamma)\ \delta = [\![q]\!]\ \theta\ \gamma\ (\delta, [\![r]\!]\theta\ \gamma\ \delta) \text{ then } \top \text{ else } \bot & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd (p =_\omega q) : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

16. Case TPOINTSTO: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd e \mapsto_A e' : \mathsf{prop}$

    First, the syntax:

    1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd e : \mathsf{ref}\ A$
    2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd e' : A$
    3 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]e : \mathsf{ref}\ A$
    4 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]e' : A$
    5 By rule, $\Theta; \Gamma; \Delta \rhd [r/b](e \mapsto_A e') : \mathsf{prop}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](e \mapsto_A e') : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} & [\![\Theta;\ \Gamma \vdash [r/b]e : \mathsf{ref}\ A]\!]\ \theta\ \gamma & \\ = & \mapsto & \text{Semantics} \\ & [\![\Theta;\ \Gamma \vdash [r/b]e' : A]\!]\ \theta\ \gamma & \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd e \mapsto_A e' : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

17. Case TEqSort: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega$

First, the syntax:

1 By inversion, $\Theta \rhd \omega \equiv \omega' : \mathsf{sort}$
2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega'$
3 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]p : \omega'$
4 By rule, $\Theta; \Gamma; \Delta \rhd [r/b]p : \omega$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b]p : \omega]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & [\![\Theta; \Gamma; \Delta \rhd [r/b]p : \omega']\!]\ \theta\ \gamma\ \delta & \text{Semantics} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega']\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \omega]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

18. Case TSpec: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S\ \mathsf{spec} : \mathsf{prop}$:

First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}$
2 By mutual induction $\Theta; \Gamma; \Delta \rhd [r/b]S : \mathsf{spec}$
3 By rule, $\Theta; \Gamma; \Delta \rhd [r/b]S\ \mathsf{spec} : \mathsf{prop}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b]S\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

$$\begin{array}{lll} = & \text{if } [\![[r/b]S]\!]\ \theta\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Semantics} \\ = & \text{if } [\![S]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) = \top \text{ then } \top \text{ else } \bot & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

19. Case SpecTriple: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}$
2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd [c] : \bigcirc A$
3 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'', a : A \rhd q : \mathsf{prop}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}$
5 By induction, $\Theta; \Gamma; \Delta \rhd [[r/b]c] : \bigcirc A$
6 By induction, $\Theta; \Gamma; \Delta, a : A \rhd [r/b]q : \mathsf{prop}$
7 By rule, $\Theta; \Gamma; \Delta \rhd [r/b](\{p\}c\{a : A.\ q\}) : \mathsf{spec}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](\{p\}c\{a : A.\ q\}) : \mathsf{spec}]\!]\ \theta; \gamma\ \delta$

$$
\begin{array}{lll}
 & \{[\![\Theta; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta\} & \\
= & [\![\Theta;\ \Gamma \vdash [r/b]c \div A]\!]\ \theta\ \gamma & \text{Semantics} \\
 & \{v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd [r/b]q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\} & \\
 & \{[\![\Theta, \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)\} & \\
= & [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma & \text{Induction} \\
 & \{v.\ [\![\Theta; \Gamma, a : A; \Delta, b : \upsilon'' \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)\} & \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd (\{p\}c\{a : A.\ q\}) : \mathsf{spec}]\!]\ \theta; \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

20. Case SPECMTRIPLE: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \langle p\rangle e\langle a : A.\ q\rangle : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}$
2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd e : \bigcirc A$
3 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'', a : A \rhd q : \mathsf{prop}$
4 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}$
5 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]e : \bigcirc A$
6 By induction, $\Theta; \Gamma; \Delta, a : A \rhd [r/b]q : \mathsf{prop}$
7 By rule, $\Theta; \Gamma; \Delta \rhd [r/b](\langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](\langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec}]\!]\ \theta; \gamma\ \delta$

$$
\begin{array}{lll}
 & \{[\![\Theta; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta\} & \\
= & [\![\Theta;\ \Gamma \vdash [r/b]e : \bigcirc A]\!]\ \theta\ \gamma & \text{Semantics} \\
 & \{v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd [r/b]q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta\} & \\
 & \{[\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)\} & \\
= & [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]\ \theta\ \gamma & \text{Induction} \\
 & \{v.\ [\![\Theta; \Gamma, a : A; \Delta, b : \upsilon'' \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)\} & \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd (\langle p\rangle e\langle a : A.\ q\rangle) : \mathsf{spec}]\!]\ \theta; \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

21. Case SPECQUANTIFY1: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Qu : \upsilon.\ S : \mathsf{spec}$

First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'', u : \upsilon \rhd S : \mathsf{spec}$
2 By induction, $\Theta; \Gamma; \Delta, u : \upsilon \rhd [r/b]S : \mathsf{spec}$
3 By rule, $\Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ [r/b]S : \mathsf{spec}$
4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](Qu : \upsilon.\ S) : \mathsf{spec}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](Qu : \upsilon.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & [\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} & \\
 & \quad [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd [r/b]S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, v) & \text{Semantics} \\
= & [\![Q]\!]_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} & \\
 & \quad [\![\Theta; \Gamma; \Delta, b : \upsilon'', u : \upsilon \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta, v) & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Qu : \upsilon.\ S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

22. Case SPECQUANTIFY2: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Qx : A.\ S : \mathsf{spec}$
    First, the syntax:

    1 By inversion, $\Theta; \Gamma, x : A; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}$
    2 By induction, $\Theta; \Gamma, x : A; \Delta \rhd [r/b]S : \mathsf{spec}$
    3 By rule, $\Theta; \Gamma; \Delta \rhd Qx : A.\ [r/b]S : \mathsf{spec}$
    4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](Qx : A.\ S) : \mathsf{spec}$

    For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](Qx : A.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma, x : A; \Delta \rhd [r/b]S : \mathsf{spec}]\!]\ \theta\ (\gamma, v)\ \delta\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{v \in [\![\Theta \rhd A : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta; \Gamma, x : A; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}]\!]\ \theta\ (\gamma, v)\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Qx : A.\ S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

23. Case SPECQUANTIFY3: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Q\beta : \kappa'.\ S : \mathsf{spec}$
    First, the syntax:

    1 By inversion, $\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}$
    2 By induction, $\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]S : \mathsf{spec}$
    3 By rule, $\Theta; \Gamma; \Delta \rhd Q\beta : \kappa'.\ [r/b]S : \mathsf{spec}$
    4 By def of subst, $\Theta; \Gamma; \Delta \rhd [r/b](Q\beta : \kappa'.\ S) : \mathsf{spec}$

    For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](Q\beta : \kappa'.\ S) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

    $$\begin{array}{lll} = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; \Gamma; \Delta \rhd [r/b]S : \mathsf{spec}]\!]\ (\theta, \tau')\ \gamma\ \delta\end{array} & \text{Semantics} \\ = & \begin{array}{l}[\![Q]\!]_{\tau' \in [\![\Theta \rhd \kappa' : \mathsf{sort}]\!]\ \theta} \\ \quad [\![\Theta, \beta : \kappa'; \Gamma; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}]\!]\ (\theta, \tau')\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta, v)\end{array} & \text{Induction} \\ = & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd Q\beta : \kappa'.\ S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics} \end{array}$$

24. Case SPECBINARY: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S \oplus S' : \mathsf{spec}$
    First, the syntax:

    1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}$
    2 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S' : \mathsf{spec}$
    3 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]S : \mathsf{spec}$
    4 By induction, $\Theta; \Gamma; \Delta \rhd [r/b]S' : \mathsf{spec}$
    5 By rule, $\Theta; \Gamma; \Delta \rhd [r/b]S \oplus [r/b]S' : \mathsf{spec}$
    6 By subst def, $\Theta; \Gamma; \Delta \rhd [r/b](S \oplus S') : \mathsf{spec}$

For semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b](S \oplus S') : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \begin{array}{c}([\![\Theta; \Gamma; \Delta \rhd [r/b]S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta)\ [\![\oplus]\!] \\ ([\![\Theta; \Gamma; \Delta \rhd [r/b]S' : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta)\end{array} & \text{Semantics} \\
= & \begin{array}{c}([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta))[\![\oplus]\!] \\ ([\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S' : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta))\end{array} & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd S \oplus S' : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

25. Case TSPEC: $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \{p\} : \mathsf{spec}$:

First, the syntax:

1 By inversion, $\Theta; \Gamma; \Delta, b : \upsilon'' \rhd p : \mathsf{prop}$
2 By mutual induction $\Theta; \Gamma; \Delta \rhd [r/b]p : \mathsf{prop}$
3 By rule, $\Theta; \Gamma; \Delta \rhd [r/b]\{p\} : \mathsf{spec}$

For the semantics, consider $[\![\Theta; \Gamma; \Delta \rhd [r/b]\{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

$$
\begin{array}{lll}
= & \text{if } [\![[r/b]p]\!]\ \theta\ \gamma\ \delta = \top \text{ then } \top \text{ else } \bot & \text{Semantics} \\
= & \text{if } [\![p]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) = \top \text{ then } \top \text{ else } \bot & \text{Induction} \\
= & [\![\Theta; \Gamma; \Delta, b : \upsilon'' \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, [\![\Theta; \Gamma; \Delta \rhd r : \upsilon'']\!]\ \theta\ \gamma\ \delta) & \text{Semantics}
\end{array}
$$

□

3.8.2 Soundness of Assertion Logic Axioms

Lemma 30. *(Getting Specs Into Assertions) If* $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$ *is valid, then* $\Theta; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}$ *is valid.*

Proof. Assume we have a suitable θ, γ, and δ. By hypothesis we know $[\![\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$ in the specification lattice. However, we know the interpretation of $[\![\Theta; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$ is equal to if $[\![\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$ then $\top$ else $\bot$, so it follows that $[\![\Theta; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = \top$ in the assertion lattice. Hence it is true for all substitutions, and is therefore valid. □

Lemma 31. *(Getting Assertions out of Specs) If* $\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}$*, then* $\Theta; \Gamma; \Delta \rhd \{p\}\ \mathsf{spec} \supset p : \mathsf{prop}$ *is valid.*

Proof.

1 Assume we have $\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}$ and a suitable θ, γ, and δ.
2 We want to show that $[\![\Theta; \Gamma; \Delta \rhd \{p\}\ \mathsf{spec} \supset p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$ is equal to $\top$.
3 This is equivalent to every $h \in H$ being in $[\![\Theta; \Gamma; \Delta \rhd \{p\}\ \mathsf{spec} \supset p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
4 From the semantics of implication, we want to show that
5 if $h \in [\![\Theta; \Gamma; \Delta \rhd \{p\}\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$, then $h \in [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$.
6 Assume we have an $h \in [\![\Theta; \Gamma; \Delta \rhd \{p\}\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$.
7 We know if $[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$ then $\top$ else $\bot$

8 Since we have an h in this set, $[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$.
9 However, $[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \mathsf{if}\ [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = \top\ \mathsf{then}\ \top\ \mathsf{else}\ \bot$
10 So $[\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = \top$.
11 Since this is the full set of heaps H, it follows that $h \in [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$.
□

Lemma 32. *(Soundness of Equality) If* $\Theta; \Gamma; \Delta \rhd p \equiv q : \omega$ *is a valid equality, then* $\Theta; \Gamma; \Delta \rhd p =_\omega q : \mathsf{prop}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd p \equiv q : \omega$ is a valid equality
2 Assume we have a suitable θ, γ, and δ.
3 Then we know that $[\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = [\![\Theta; \Gamma; \Delta \rhd q : \omega]\!]\ \theta\ \gamma\ \delta$.
4 By semantics $[\![\Theta; \Gamma; \Delta \rhd p =_\omega q : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = \mathsf{if}\ [\![\Theta; \Gamma; \Delta \rhd q : \omega]\!]\ \theta\ \gamma\ \delta\ \mathsf{then}\ \top\ \mathsf{else}\ \bot$
5 Therefore $[\![\Theta; \Gamma; \Delta \rhd p =_\omega q : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = \top$
6 Therefore $\Theta; \Gamma; \Delta \rhd p =_\omega q : \mathsf{prop}$ is valid.
□

3.8.3 Soundness of Program Logic Axioms

Lemma 33. *(Validity and Classical and Intuitionistic Implication) The statement that* $\Theta; \Gamma; \Delta \rhd S_1 \Rrightarrow S_2 : \mathsf{spec}$ *is valid is logically equivalent to: if for all* θ, γ, δ *and* r*, if* $r \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ *then* $r \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

Proof.

1 $\Rightarrow$ direction:
2 Assume $\Theta; \Gamma; \Delta \rhd S_1 \Rrightarrow S_2 : \mathsf{spec}$ is valid
3 We want to show for all θ, γ, δ and r, if $r \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \delta$, then $r \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
4 Assume appropriate $\theta\ \gamma\ \delta$, and r such that $r \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
5 We want to show $r \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
6 From the hypothesis we know that $[\![\Theta; \Gamma; \Delta \rhd S_1 \Rrightarrow S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
7 So we know for all r and $s \succeq r$, that
if $s \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ then $s \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
8 Since $r \succeq r$, we know
if $r \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ then $s \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
9 Since $r \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$, we know that $r \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
10 $\Leftarrow$ direction:
11 Assume for all $\theta\ \gamma\ \delta$ and r,
if $r \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$, then $r \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
12 We want to show for all $\theta\ \gamma\ \delta$ that $[\![\Theta; \Gamma; \Delta \rhd S_1 \Rrightarrow S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
13 Assume $\theta\ \gamma\ \delta$ are suitable substitutions
14 So we want to show $[\![\Theta; \Gamma; \Delta \rhd S_1 \Rrightarrow S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
15 So we want to show for all r and $s \succeq r$, that

if $s \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ then $s \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

16 Assume r and s such that $s \succeq r$ and $s \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

17 Instantiate the quantifier in the hypothesis with $\theta\ \gamma\ \delta$ and s, so we learn
if $s \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ then $s \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

18 Since $s \in [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ we see that $s \in [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

□

Lemma 34. *(Equivalence of the Two Forms of Triples) We have that* $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$ *is valid if and only if* $\Theta; \Gamma; \Delta \rhd \langle p\rangle[c]\langle a : A.\ q\rangle : \mathsf{spec}$ *is valid.*

Proof.

1 We want to show that for all suitable θ, γ, and δ, $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec} = \top$ iff $\Theta; \Gamma; \Delta \rhd \langle p\rangle[c]\langle a : A.\ q\rangle : \mathsf{spec} = \top$

2 Now, let
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$E = [\![\Theta;\ \Gamma \vdash [c] : A]\!]\ \theta\ \gamma$
$C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
$Q = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ \gamma\ \theta\ (\gamma, v)\ \delta$

3 Observe that $E = C$, from the semantics of $[c]$

4 $\Rightarrow$: Assume $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$ is valid

5 Hence the semantic spec $\{P\}C\{a.\ Q(a)\} = \top$

6 Hence $\{P\}E\{a.\ Q(a)\} = \top$

7 $\Leftarrow$: Assume $\Theta; \Gamma; \Delta \rhd \langle p\rangle[c]\langle a : A.\ q\rangle : \mathsf{spec}$ is valid

8 Hence the semantic spec $\{P\}E\{a.\ Q(a)\} = \top$

9 Hence $\{P\}C\{a.\ Q(a)\} = \top$

□

Lemma 35. *(Return Value Axiom) The schema* $\Theta; \Gamma; \Delta \rhd \{P\}e\{a : A.\ P \wedge a = e\} : \mathsf{spec}$ *is valid.*

Proof.

1 We want to show that for all suitable θ, γ, and δ, $[\![\{P\}e\{a : A.\ P \wedge a = e\}]\!]\ \theta\ \gamma\ \delta = \top$

2 Now let
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$C = [\![\Theta;\ \Gamma \vdash e \div A]\!]\ \theta\ \gamma$
$E = [\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ \gamma$
$Q = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd P \wedge a = e : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$

3 We want to show that $\{P\}C\{a.\ Q(a)\} = \top$

4 So we want to show that for all R, we have $R \in \{P\}C\{a.\ Q(a)\}$

5 Hence it suffices to show that $[P * R]\ C\ [v.\ P \wedge [\![a = e]\!]\ \theta\ (\gamma, v)\ \delta * R]$

6 So we want to show that for all $h \in P * R$,
we have $C\ Best(\lambda v.\ P \wedge [\![a = e * R]\!]\ \theta\ (\gamma, v)\ \delta)\ h = \bot$

7 Assume we have $h \in P * R$

8 Next, observe that $C\ k\ h = k\ E\ h$

9 To show that $Best(\lambda a.\ P \wedge a = e * R)\ E\ h = \bot$,
we need to show that $k\ \ E\ h = \bot$, for every $k \in Approx(\lambda a.\ P \wedge a = e * R)$
10 So assume $k \in Approx(\lambda v.\ P \wedge [\![a = e]\!]\ \theta\ (\gamma, v)\ \delta * R)$
11 Therefore for all $v \in [\![A]\!]$, and $h \in Q(v) * R$, we know $k\ v\ h = \bot$
12 We know $E \in [\![A]\!]$
13 So we need to show that given $h \in P * R$, we have $h \in Q(E) * R$
14 Assume we have $h \in P * R$
15 So we need to show $h \in (P \wedge [\![a = e]\!]\ (\delta, E)) * R$
16 Note that $[\![a = e]\!]\ (\delta, E) = \top$ if $E = E$, which is true
17 Hence $P = (P \wedge [\![a = e]\!]\ (\delta, E))$
18 Hence we need to show $h \in P * R$
19 This is a hypothesis, so we are done

□

Lemma 36. *(Assignment Axiom)* $\Theta; \Gamma; \Delta \rhd \{e \mapsto_A -\}e := e'\{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec}$ *is valid*

Proof.

1 Assume suitable θ, γ, and δ
2 We want to show $[\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A -\}e := e'\{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
3 So we want to show $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A -\}e := e'\{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
4 Assume r
5 We want to show $r \in [\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A -\}e := e'\{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
6 Let $P = [\![\Theta; \Gamma; \Delta \rhd e \mapsto_A - : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
7 Let $C = [\![\Theta;\ \Gamma \vdash [e := e'] : \bigcirc A]\!]\ \theta\ \gamma$
8 Let $E = [\![\Theta;\ \Gamma \vdash e : \mathsf{ref}\ A]\!]\ \theta\ \gamma$
9 Let $E' = [\![\Theta;\ \Gamma \vdash e' : A]\!]\ \theta\ \gamma$
10 let $Q = \lambda v.\ [\![\Theta; \Gamma; \Delta \rhd e \mapsto_A e' : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
11 We want to show $r \in \{P\}C\{a : A.\ Q(a)\}$
12 We want to show $\forall s \succeq r.\ [P * s]\ C\ [a : A.\ Q(a) * s]$
13 Assume $s \succeq r$
14 We want to show $[P * s]\ C\ [a : A.\ Q(a) * s]$
15 We want to show $\forall h \in (P * s).\ C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
16 Assume $h \in P * s$
17 By semantics $h \in P * s$ means $\exists h_1, h_2.\ h_1 \in P$, $h_2 \in s$, and $h_1 \# h_2$
18 By semantics of assertions, $h_1 \in P$ means $\exists v \in [\![\Theta \vdash A : \bigstar]\!]\ \theta.\ h_1 = [E : v]$
19 Expanding definitions, $C = \lambda k.\ \lambda h.\ k\ \langle\rangle$ if $E \in \mathrm{dom}(h)$ then $[h|E : E']$ else $\top$
20 Therefore $C\ Best(\lambda a.\ Q(a) * s)\ h =$
if $E \in \mathrm{dom}(h_1 \cdot h_2)$ then $Best(\lambda a.\ Q(a) * s)\ \langle\rangle\ [h_1 \cdot h_2|E : E']$ else $\top$
21 Since $E \in \mathrm{dom}(h_1)$, we know $C\ Best(\lambda a.\ Q(a) * s)\ h =$
$Best(\lambda a.\ Q(a) * s)\ \langle\rangle\ [h_1 \cdot h_2|E : E']$
22 Therefore we want to show $Best(\lambda a.\ Q(a) * s)\ \langle\rangle\ [h_1 \cdot h_2|E : E'] = \bot$
23 So we must show $\neg\exists k \in Approx(\lambda a.\ Q(a) * s).\ k\ \langle\rangle\ [h_1 \cdot h_2|E'] = \top$
24 So we must show $\forall k \in Approx(\lambda a.\ Q(a) * s).\ k\ \langle\rangle\ [h_1 \cdot h_2|E : E'] = \bot$

25 Assume $k \in Approx(\lambda a.\ Q(a) * s)$
26 Since $[h_1 \cdot h_2 | E : E'] = [E : E'] \cdot h_2$, we want $k \langle\rangle\ ([E : E'] \cdot h_2) = \bot$
27 Now, $k \in Approx(\lambda a.\ Q(a) * s)$ means $\forall v, h \in (Q(v) * s)\ k\ v\ h = \bot$
28 Now, instantiate the quantifier with $\langle\rangle$ and $[E : E'] \cdot h_2$
29 Now we must check $[E : E'] \cdot h_2 \in Q(\langle\rangle) * s$
30 We will check that $[E : E'] \in Q(\langle\rangle)$ and $h_2 \in s$
31 By semantics of assertions $Q(\langle\rangle) = \{[E : E']\}$, so $[E : E'] \in Q(\langle\rangle)$
32 From line 19, $h_2 \in s$
33 Since $[E : E']$ has the same domain as h_1, we know $[E : E'] \# h_2$
34 Therefore $[E : E'] \cdot h_2 \in Q(\langle\rangle) * s$
35 Therefore $k \langle\rangle\ [h_1 \cdot h_2 | E : E'] = \bot$
36 Therefore $\forall k \in Approx(\lambda a.\ Q(a) * s).\ k \langle\rangle\ [h_1 \cdot h_2 | E : E'] = \bot$
37 Therefore $\neg \exists k \in Approx(\lambda a.\ Q(a) * s).\ k\ \langle\rangle\ [h_1 \cdot h_2 | E'] = \top$
38 Therefore $Best(\lambda a.\ Q(a) * s)\ \langle\rangle\ [h_1 \cdot h_2 | E : E'] = \bot$
39 Therefore $\forall h \in (P * s).\ C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
40 Therefore $[P * s]\ C\ [a : A.\ Q(a) * s]$
41 Therefore $\forall s \succeq r.\ [P * s]\ C\ [a : A.\ Q(a) * s]$
42 Therefore $r \in \{P\} C \{a : A.\ Q(a)\}$
43 Therefore $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A -\} e := e' \{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
44 Therefore $[\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A -\} e := e' \{a : \mathbf{1}.\ e \mapsto_A e'\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$

□

Lemma 37. *(Allocation Axiom) If* $\Theta; \Gamma; \Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec}$ *is valid*

Proof.

1 Assume suitable θ, γ, and δ
2 We want to show $[\![\Theta; \Gamma; \Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta = \top$
3 So we want to show $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
4 Assume r,
5 So we want to show $r \in [\![\Theta; \Gamma; \Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
6 Let $C = [\![\Theta;\ \Gamma \vdash [\mathsf{new}_A(e)] : \bigcirc \mathsf{ref}\ A\ \theta]\!]\ \gamma$
7 Let $E = [\![\Theta;\ \Gamma \vdash e : A\ \theta]\!]\ \gamma$
8 Let $Q = \lambda v.\ [\![\Delta, a : \mathsf{ref}\ A \rhd a \mapsto_A e : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
9 So we want to show $r \in \{I\} C \{a : A.\ Q(a)\}$
10 So we want to show $\forall s \succeq r.\ [s]\ C\ [a : A.\ Q(a) * s]$
11 Assume $s \succeq r$
12 We want to show $[s]\ C\ [a : A.\ Q(a) * s]$
13 We want to show $\forall h \in s.\ C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
14 Assume $h \in s$
15 Expanding definitions, $C\ Best((\lambda a.\ Q(a) * s)\ h =$
let $l = newloc(h, A), [\theta(A)])$ in $Best(\lambda a.\ Q(a) * s)\ l\ [h | l : E]$
16 So let $l = newloc(h, A)$
17 We want to show $Best(\lambda a.\ Q(a) * s)\ l\ [h | l : E] = \bot$

18 To show this, we must show $\neg(\exists k \in Approx(\lambda a.\ Q(a) * s).\ k\ l\ [h|l : E] = \top)$
19 So we must show $\forall k \in Approx(\lambda a.\ Q(l) * s).\ k\ l\ [h|l : E] = \bot)$
20 Assume $k \in Approx(\lambda a.\ Q(a) * s)$
21 This means $\forall v, h \in Q(v) * s.\ k\ v\ h = \bot$
22 Instantiate v with l, and h with $[l : E] \cdot h$
23 Now we must check $[l : E] \cdot h \in Q(l) * s$
24 So we will check $[l : E] \in Q(l)$ and $h \in s$
25 Expanding definitions, $Q(l) = \{[l : E]\}$, so $[l : E] \in Q(l)$
26 From line 18, $h \in s$
27 Since l is bigger than anything in dom(h), it follows $[l : E]\#h$
28 Therefore $[l : E] \cdot h \in Q(l) * s$
29 Therefore $k\ l\ ([l : E] \cdot h) = \bot$
30 Therefore $\forall k \in Approx(\lambda a.\ Q(l) * s).\ k\ l\ [h|l : E] = \bot)$
31 Therefore $\forall h \in s.\ C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
32 Therefore $[s]\ C\ [a : A.\ Q(a) * s]$
33 Therefore $\forall s \succeq r.\ [s]\ C\ [a : A.\ Q(a) * s]$
34 Therefore $r \in \{I\}C\{a : A.\ Q(a)\}$
35 Therefore $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
36 Therefore $[\![\Theta; \Gamma; \Delta \rhd \{\mathsf{emp}\}\mathsf{new}_A(e)\{a : \mathsf{ref}\ A.\ a \mapsto e\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta = \top$

□

Lemma 38. *(Read Axiom) We have that* $\Theta; \Gamma; \Delta\ \rhd\ \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto_A e' \wedge a = e'\}$: spec *is valid.*

Proof.

1 Assume suitable θ, γ, and δ
2 We want to show $[\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto e' \wedge a = e'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta = \top$
3 So we want to show $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto e' \wedge a = e'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
4 Assume r,
5 So we want to show $r \in r \in [\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto e' \wedge a = e'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
6 Let $P = [\![\Theta; \Gamma; \Delta \rhd e \mapsto_A e' : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
7 Let $C = [\![\Theta;\ \Gamma \vdash [!e] : \bigcirc A\ \theta]\!]\ \gamma$
8 Let $E = [\![\Theta;\ \Gamma \vdash e : \mathsf{ref}\ A\ \theta]\!]\ \gamma$
9 Let $E' = [\![\Theta;\ \Gamma \vdash e' : A\ \theta]\!]\ \gamma$
10 Let $Q = \lambda v.\ [\![\Theta; \Gamma, a : \mathsf{ref}\ A; \Delta \rhd e \mapsto_A e' \wedge a = e' : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
11 So we want to show $r \in \{P\}C\{a : A.\ Q(a)\}$
12 So we want to show $\forall s \succeq r.\ [P * s]\ C\ [a : A.\ Q(a) * s]$
13 Assume $s \succeq r$
14 We want to show $[P * s]\ C\ [a : A.\ Q(a) * s]$
15 We want to show $\forall h \in P * s.\ C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
16 Assume $h \in P * s$
17 There are h_1 and h_2 such that $h_1 \in P$, and $h_2 \in s$, and $h_1\#h_2$
18 Expanding definitions, $C\ Best(\lambda a.\ Q(a) * s)\ (h_1 \cdot h_2) =$

if $E \in \mathrm{dom}(h)$ then $Best(\ (h\ E)\ h$ else $\top$
From definition of P, $h_1 = [E : E']$
Hence $E \in \mathrm{dom}(h_1)$, so $E \in \mathrm{dom}(h_1 \cdot h_2)$
Hence $C\ Best((\lambda a.\ Q(a) * s)\ (h_1 \cdot h_2) = Best(\lambda a.\ Q(a) * s)\ (h\ E)\ h$
So we want to show $Best(\lambda a.\ Q(a) * s)\ (h\ E)\ h = \bot$
To show this, we must show $\neg(\exists k \in Approx(\lambda a.\ Q(a) * s).\ (h\ E)\ h = \top)$
So we want $\forall k \in Approx(\lambda a.\ Q(a) * s).\ (h\ E)\ h = \bot)$
Assume $k \in Approx(\lambda a.\ Q(a) * s)$
So we know $\forall v, h \in Q(v) * s.\ k\ v\ h = \bot$
Instantiate with v with $(h\ E)$, and h with h
So we must show $h \in Q(h\ E) * s$
So we will check $h_1 \in Q(h\ E)$ and $h_2 \in s$, since $h_1 \cdot h_2 = h$
So we want to check $h_1 \in [\![\Theta; \Gamma; \Delta \rhd e \mapsto_A e' : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = P$
and also $h_1 \in [\![\Gamma, a : A \rhd a = e' : \mathsf{prop}]\!](\delta, h\ E)$
We know $h_1 \in P$ by assumption
Since $h\ E = E' = [\![\Theta; \Gamma; \Delta \rhd e' : A]\!]\theta\ \gamma\ \delta$, we know $[\![\Theta; \Gamma, a : A; \Delta \rhd a = e' : \mathsf{prop}]\!]\ \theta\ (\gamma, h$
Therefore $h_1 \in [\![\Theta; \Gamma, a : A; \Delta \rhd a = e' : \mathsf{prop}]\!]\ \theta\ (\gamma, h\ E)\ \delta$
We know $h_2 \in s$ by assumption
Therefore $h \in Q(h\ E) * s$
Therefore we know $k\ (h\ E)\ h = \bot$
Therefore $\forall k \in Approx((\lambda a.\ Q(a) * s).\ (h\ E)\ h = \bot$
Therefore $Best(\lambda a.\ Q(a) * s)\ (h\ E)\ h = \bot$
Therefore $\forall h \in P * s.\ C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
Therefore $[P * s]\ C\ [a : A.\ Q(a) * s]$
Therefore $\forall s \succeq r.\ [P * s]\ C\ [a : A.\ Q(a) * s]$
Therefore $r \in \{P\}C\{a : A.\ Q(a)\}$
Therefore $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto e' \wedge a = e'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
Therefore $[\![\Theta; \Gamma; \Delta \rhd \{e \mapsto_A e'\}!e\{a : A.\ e \mapsto e' \wedge a = e'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta = \top$

□

Lemma 39. *(Sequential Composition Axiom) Suppose* $\Theta; \Gamma; \Delta \rhd \langle p\rangle e\langle x : A.\ q\rangle : \mathsf{spec}$ *is valid, and* $\Delta, x : A \rhd \{q\}c\{a : B.\ r\} : \mathsf{spec}$ *is valid, and* $x \notin \mathrm{FV}(r)$. *Then* $\Theta; \Gamma; \Delta \rhd \{p\}\mathsf{letv}\ x = e\ \mathsf{in}\ c\{a : B.\ r\} : \mathsf{spec}$ *is valid.*

Proof.

Assume $\Theta; \Gamma; \Delta \rhd \langle p\rangle e\langle x : A.\ q\rangle : \mathsf{spec}$ is valid
Assume $\Delta, x : A \rhd \{q\}c\{a : B.\ r\} : \mathsf{spec}$ is valid
Assume suitable θ, γ, and δ
We want to show $[\![\Theta; \Gamma; \Delta \rhd \{p\}\mathsf{letv}\ x = e\ \mathsf{in}\ c\{a : B.\ r\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta = \top$
So we want $\forall t.\ t \in [\![\Theta; \Gamma; \Delta \rhd \{p\}\mathsf{letv}\ x = e\ \mathsf{in}\ c\{a : B.\ r\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
Assume t
Let $E = [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]\ \theta\ \gamma$
Let $F = \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash [c] : \bigcirc B]\!]\ \theta\ (\gamma, v)$

Let $P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\theta\ \gamma\ \delta$
Let $Q = \lambda v.\ [\![\Theta; \Gamma, x : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
Let $R = \lambda v.\ \lambda v'.\ [\![\Theta; \Gamma, x : A, a : B; \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v, v')\ \delta$
Let $R' = \lambda v'.\ [\![\Theta; \Gamma, a : B; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v')\ \delta$
Note $\forall v.\ R\ v = R'$ since $x \notin \mathrm{FV}(r)$
Let $C = [\![\Theta;\ \Gamma \vdash [\mathsf{letv}\ x = e\ \mathsf{in}\ c] : \bigcirc B]\!]\ \theta\ \gamma$
By semantics, $C = F^*(E)$
By definition of monadic lift, $C = \lambda k.\ E\ (\lambda v.\ F\ v\ k)$
So we want to show $t \in \{P\}C\{a : A.\ R'(a)\}$
So we want $\forall u \succeq t.\ [P * u]\ C\ [a : A.\ (R'(a) * u)]$
Assume $u \succeq t$
So we want to show $\forall h \in P * u.\ C\ Best(\lambda a.\ R'(a) * u)\ h = \bot$
Assume $h \in P * u$
So we want to show $E\ (\lambda v.\ F\ v\ Best(\lambda a.\ R'(a) * u))\ h = \bot$
We know $\Theta; \Gamma; \Delta \rhd \langle p \rangle e \langle x : A.\ q \rangle : \mathsf{spec}$ is valid
Instantiating with the environment $\theta\ \gamma\ \delta$, we get $\forall t.\ t \in \{P\}E\{a : A.\ Q(a)\}$
Thus $\forall t, u \succeq t.\ [P * u]\ E\ [a : A.\ Q(a) * u]$
Thus $\forall t, u \succeq t, h \in P * u, E\ Best(\lambda a.\ Q(a) * u)\ h = \bot$
Instantiating, we get $E\ Best(\lambda a.\ Q(a) * u)\ h = \bot$
So it suffices to show $(\lambda v.\ F\ v\ Best(\lambda a.\ R'(a) * u)) \sqsubseteq Best(\lambda a.\ Q(a) * u)$
To do this, we can show $(\lambda v.\ F\ v\ Best(\lambda a.\ R'(a) * u)) \in Approx(\lambda a.\ Q(a) * u)$
To do this, we must show $\forall v, h \in Q(v) * u, F\ v\ Best(\lambda a.\ R'(a) * u)\ h = \bot$
Assume $v, h \in Q(v) * u$
We know $\Delta, x : A \rhd \{q\}c\{a : B.\ r\} : \mathsf{spec}$ is valid
Instantiating with the environment $\theta\ (\gamma, v)\ \delta$, we get
$\forall v, t, u \succeq t, h \in Q(v) * u, (F\ v)\ Best(\lambda a.\ R\ v\ a * u)\ h = \bot$
Instantiating, we get $(F\ v)\ Best(\lambda a.\ R\ v\ a * u)\ h = \bot$
By equality, we get $(F\ v)\ Best(\lambda a.\ R'(a) * u)\ h = \bot$
Therefore $\forall v, h \in Q(v) * u, F\ v\ Best(\lambda a.\ R'(a) * u)\ h = \bot$
Therefore $(\lambda v.\ F\ v\ Best(\lambda a.\ R'(a) * u)) \sqsubseteq Best(\lambda a.\ Q(a) * u)$
Therefore $(\lambda v.\ F\ v\ Best(\lambda a.\ R'(a) * u)) \in Approx(\lambda a.\ Q(a) * u)$
Therefore $E\ (\lambda v.\ F\ v\ Best(\lambda a.\ R'(a) * u))\ h = \bot$
Therefore $\forall h \in P * u.\ C\ Best(\lambda a.\ R'(a) * u)\ h = \bot$
Therefore $\forall u \succeq t.\ [P * u]\ C\ [a : A.\ (R'(a) * u)]$
Therefore $t \in \{P\}C\{a : A.\ R'(a)\}$
Therefore $\forall t.\ t \in [\![\Theta; \Gamma; \Delta \rhd \{p\}\mathsf{letv}\ x = e\ \mathsf{in}\ c\{a : B.\ r\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
Therefore $[\![\Theta; \Gamma; \Delta \rhd \{p\}\mathsf{letv}\ x = e\ \mathsf{in}\ c\{a : B.\ r\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta = \top$

□

Lemma 40. *(Fixed Point Induction) We have that if*

$$S \triangleq \Theta; \Gamma; \Delta \rhd (\forall x : \bigcirc A.\ \langle p \rangle x \langle a : A.\ q(a) \rangle \Rrightarrow \langle p \rangle e \langle a : A.\ q \rangle) : \mathsf{spec}$$

is valid, then

$$S' \triangleq \Theta; \Gamma; \Delta \rhd \langle p \rangle \mathsf{fix}\ x : \bigcirc A.\ e \langle a : A.\ q \rangle : \mathsf{spec}$$

is valid.

Proof.

1 Let $S = \forall x : \bigcirc A.\ \langle p \rangle x \langle a : A.\ q(a) \rangle \Rrightarrow \langle p \rangle e \langle a : A.\ q \rangle$
2 Let $S' = \langle p \rangle \mathsf{fix}\ x : \bigcirc A.\ e \langle a : A.\ q \rangle$
3 Assume suitable θ, γ, and δ
4 So we want to show $[\![S']\!]\ \theta\ \gamma\ \delta = \top$
5 Let $P = [\![\Theta; \Gamma; \Delta \rhd P : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
6 Let $C = [\![\Theta;\ \Gamma \vdash \mathsf{fix}\ x : \bigcirc A.\ e : \bigcirc A]\!]\ \theta\ \gamma$
7 Let $Q = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
8 Let $F(v) = [\![\Theta;\ \Gamma, a : A \vdash e : \bigcirc A]\!]\ \theta\ (\gamma, v)$
9 So we want to show $\{P\} C \{a : A.\ Q(a)\} = \top$
10 We know $[\![S]\!]\ \theta\ \gamma\ \delta = \top$
11 Thus for all v, we know $[\![\langle p \rangle x \langle a : A.\ q(a) \rangle \Rrightarrow \langle p \rangle e \langle a : A.\ q \rangle]\!]\ \theta\ (\gamma, v)\ \delta = \top$
12 Thus we know for all v, r, and $s \succeq r$,
if $s \in [\![\langle p \rangle x \langle a : A.\ q(a) \rangle]\!]\ \theta\ (\gamma, v)\ \delta$ then $s \in [\![\langle p \rangle e \langle a : A.\ q \rangle]\!]\ \theta\ (\gamma, v)\ \delta$
13 Since $x \notin \mathrm{FV}(p)$, we know for all v, $[\![\Theta; \Gamma, x : \bigcirc A; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta = P$
14 Since $x \notin \mathrm{FV}(q)$, we know for all v, $\lambda v'.\ [\![\Theta; \Gamma, x : \bigcirc A, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v, v')\ \delta = Q$
15 Thus we know for all v, r, and $s \succeq r$,
if $s \in \{P\} v \{a : A.\ Q(a)\}$ then $s \in \{P\} F(v) \{a : A.\ Q(a)\}$
16 This implies that for all v, if $\{P\} F(v) \{a : A.\ Q(a)\} = \top$ then $\{P\} F(v) \{a : A.\ Q(a)\} = \top$
17 Then by the semantic fixed point theorem, $\{P\} fix(F) \{a : A.\ Q(a)\} = \top$
18 So $\{P\} C \{a : A.\ Q(a)\} = \top$
19 Therefore $[\![S']\!]\ \theta\ \gamma\ \delta = \top$

□

Lemma 41. *(Assumptions From Preconditions) If we know that* $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\} c \{a : A.\ q\} : \mathsf{spec}$ *is valid,* r *is a pure formula of separation logic, and we know that* $p \supset r$ *is a valid truth of separation logic, then* $\Theta; \Gamma; \Delta \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\} c \{a : A.\ q\} : \mathsf{spec}$ is valid
2 Assume $p \supset r$ is a valid truth of separation logic
3 We know that all suitable θ, γ, and θ, $[\![\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\} c \{a : A.\ q\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ is $\top$
4 We want for all suitable θ, γ, and θ, that $[\![\Theta; \Gamma; \Delta \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
5 Assume $\theta\ \gamma\ \delta$ is a suitable environment
6 We want to show for all r, and all $s \succeq r$, that $[P * s]\ C\ [a.\ Q(a) * s]$ holds
where
$R = [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
$Q = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$

7 To show $[P * s]\ C\ [a.\ Q(a) * s]$, we must show $\forall h \in P * s.\ c\ (Best(\lambda a.\ Q(a) * s))\ h = \bot$
8 Assume $h \in P * s$
9 Since $p \supset r$ is a valid truth, we know $P \supset R$ and so $h \in R * s$
10 Since R is pure, we know $h \in R \wedge s$
11 So we know $h \in R$
12 Since R is pure it is either $\emptyset$ or H, and since we know $h \in R$, $R = H$
13 Therefore we know that $[\![\Theta; \Gamma; \Delta \rhd \{r\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
14 Therefore we know that $[\![\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$

□

Lemma 42. *(Assumptions Into Preconditions) If we know that* $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p \wedge r\}c\{a : A.\ q\} : \mathsf{spec}$ *is valid, then* $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\}c\{a : A.\ q\} : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p \wedge r\}c\{a : A.\ q\} : \mathsf{spec}$ is valid.
2 This is equivalent to for all suitable θ, γ, θ and s,
if $s \in [\![\{r\}]\!]\ \theta\ \gamma\ \delta$, then $s \in [\![\{p \wedge r\}c\{a : A.\ q\}]\!]\ \theta\ \gamma\ \delta$.
3 We want to show that $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\}c\{a : A.\ q\} : \mathsf{spec}$ is valid
4 This is equivalent to showing for all suitable θ, γ, and θ and s,
if $s \in [\![\{r\}]\!]\ \theta\ \gamma\ \delta$, then $s \in [\![\{p\}c\{a : A.\ q\}]\!]\ \theta\ \gamma\ \delta$.
5 Assume we have $\theta\ \gamma\ \delta$ and s such that $s \in [\![\{r\}]\!]\ \theta\ \gamma\ \delta$.
6 We want to show $s \in [\![\{p\}c\{a : A.\ q\}]\!]\ \theta\ \gamma\ \delta$ is valid.
7 So we want to show for all $t \succeq s$, that $[P * t]\ C\ [a.\ Q(a) * t]$ holds
where
$R = [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
$Q = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
8 Assume $t \succeq s$, and $h \in P * t$.
9 We want to show that $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
10 Since $s \in [\![\{p \wedge r\}c\{a : A.\ q\}]\!]\ \theta\ \gamma\ \delta$, we know $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$ if $h \in (P \wedge R) * t$.
11 Since we assumed that $[\![\{r\}]\!]\ \theta\ \gamma\ \delta$ was non-empty, this means that $[\![r]\!]\ \theta\ \gamma\ \delta = \top = H$.
12 Hence $h \in (P \wedge R) * t$.
13 Hence $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$

□

Lemma 43. *(Getting Specs Out of Assertions) We have that* $\Theta; \Gamma; \Delta \rhd \{S\ \mathsf{spec}\} \Rrightarrow S : \mathsf{spec}$ *is valid.*

Proof.

1 We want to show $\Theta; \Gamma; \Delta \rhd \{S\ \mathsf{spec}\} \Rrightarrow S : \mathsf{spec}$ is valid
2 Equivalently, we need for all suitable θ, γ, and θ and r, if $r \in [\![\Theta; \Gamma; \Delta \rhd \{S\ \mathsf{spec}\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

then $r \in [\![\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
3 Assume we have a suitable θ, γ, and δ and r such that $r \in [\![\Theta; \Gamma; \Delta \rhd \{S\ \mathsf{spec}\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
4 From the semantics, we know $[\![\Theta; \Gamma; \Delta \rhd \{S\ \mathsf{spec}\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ is either $\top$ or $\bot$
5 Since $r \in [\![\Theta; \Gamma; \Delta \rhd \{S\ \mathsf{spec}\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$, we know it is non-empty, hence $[\![\Theta; \Gamma; \Delta \rhd \{S\ \mathsf{spec}\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
6 Therefore, we know that $[\![\Theta; \Gamma; \Delta \rhd S\ \mathsf{spec} : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = H$
7 Therefore, we know that $[\![\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$

□

Lemma 44. *(Existential Dropping 1) If* $\Theta; \Gamma; \Delta \rhd \{\exists u : \upsilon.\ p\} c \{a : A.\ q\} : \mathsf{spec}$ *is valid, then* $\Theta; \Gamma; \Delta, u : \upsilon \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd \{\exists y : \upsilon.\ p\} c \{a : A.\ q\} : \mathsf{spec}$ is valid
2 We want to show $\Theta; \Gamma; \Delta, u : \upsilon \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}$ is valid
3 So we want to show for all $\theta, \gamma, v, \delta$, that $[\![\Theta; \Gamma; \Delta, u : \upsilon \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}]\!]\ \theta\ \gamma\ (\delta, v) = \top$
4 Assume we have a suitable $(\theta, \gamma, \delta, v)$ and $\delta' = (\delta, v)$
5 We want to show for all r, that $r \in \{P\} C \{a.\ Q(a)\}$
6 where
$P = [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta'$
$C = [\![\Theta;\ \Gamma, y : B \vdash c \div A]\!]\ \theta\ \gamma$
$Q = \lambda v'.\ [\![\Theta; \Gamma, a : A; \Delta, u : \upsilon \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v')\ \delta'$
7 We also know that $C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
8 and that $Q = \lambda v'.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v')\ \delta'$
9 So now we must show for all $s \succeq r$, that $[P * s]\ C\ [a.\ Q(a) * s]$ holds
10 Now assume s such that $s \succeq r$
11 We need for all $h \in [P * s]\ C\ [a.\ Q(a) * s]$, that $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
12 We know $\{\exists u : \upsilon.\ p\} c \{a : A.\ q\}$ is valid
13 So we know for all r, that $r \in \{P'\} C \{a.\ Q(a)\}$ where $P' = \bigvee_{v \in [\![\Theta \rhd \upsilon : \mathsf{sort}]\!]\ \theta} [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ (\delta, v)$
14 Since $P = [\![\Theta; \Gamma; \Delta, u : \upsilon \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta'$, we know $P \subseteq P'$
15 Therefore $h \in P' * s$, and so $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$

□

Lemma 45. *(Existential Dropping 2) If* $\Theta; \Gamma; \Delta \rhd \{\exists y : B.\ p\} c \{a : A.\ q\} : \mathsf{spec}$ *is valid, then* $\Theta; \Gamma, y : B; \Delta \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd \{\exists y : B.\ p\} c \{a : A.\ q\} : \mathsf{spec}$ is valid
2 We want to show $\Theta; \Gamma, y : B; \Delta \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}$ is valid
3 So we want to show for all $\theta, \gamma, v, \delta$, that $[\![\Theta; \Gamma, y : B; \Delta \rhd \{p\} c \{a : A.\ q\} : \mathsf{spec}]\!]\ \theta\ (\gamma, v)\ \delta = \top$

4 Assume we have a suitable $(\theta, \gamma, \delta, v)$ and $\gamma' = (\gamma, v)$
5 We want to show for all r, that $r \in \{P\}C\{a.\ Q(a)\}$
6 where
$P = [\![\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma'\ \delta$
$C = [\![\Theta;\ \Gamma, y : B \vdash c \div A]\!]\ \theta\ \gamma'$
$Q = \lambda v'.\ [\![\Theta; \Gamma, y : B, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma', v')\ \delta$
7 We also know that $C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
8 and that $Q = \lambda v'.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v')\ \delta$
9 So now we must show for all $s \succeq r$, that $[P * s]\ C\ [a.\ Q(a) * s]$ holds
10 Now assume s such that $s \succeq r$
11 We need for all $h \in [P * s]\ C\ [a.\ Q(a) * s]$, that $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
12 We know $\{\exists y : B.\ p\}c\{a : A.\ q\}$ is valid
13 So we know for all r, that $r \in \{P'\}C\{a.\ Q(a)\}$
where $P' = \bigvee_{v \in [\![\Theta \rhd B : \mathsf{sort}]\!]\ \theta} [\![\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
14 Since $P = [\![\Theta; \Gamma, y : B; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$, we know $P \subseteq P'$
15 Therefore $h \in P' * s$, and so $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$

□

Lemma 46. *(Existential Dropping 3) If* $\Theta; \Gamma; \Delta \rhd \{\exists \alpha : \kappa.\ p\}c\{a : A.\ q\} : \mathsf{spec}$ *is valid, then* $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd \{\exists \alpha : \kappa.\ p\}c\{a : A.\ q\} : \mathsf{spec}$ is valid
2 We want to show $\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$ is valid
3 So we want to show for all $\theta, \tau, \gamma, \delta$
that $[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}]\!]\ (\theta, \tau)\ \gamma\ \delta = \top$
4 Assume we have a suitable $(\theta, \gamma, \delta, \tau)$ and $\theta' = (\theta, v)$
5 We want to show for all r, that $r \in \{P\}C\{a.\ Q(a)\}$
6 where
$P = [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta'\ \gamma\ \delta$
$C = [\![\Theta, \alpha : \kappa;\ \Gamma, y : B \vdash c \div A]\!]\ \theta'\ \gamma$
$Q = \lambda v'.\ [\![\Theta, \alpha : \kappa; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta'\ (\gamma, v')\ \delta$
7 We also know that $C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
8 and that $Q = \lambda v'.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v')\ \delta$
9 So now we must show for all $s \succeq r$, that $[P * s]\ C\ [a.\ Q(a) * s]$ holds
10 Now assume s such that $s \succeq r$
11 We need for all $h \in [P * s]\ C\ [a.\ Q(a) * s]$, that $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
12 We know $\{\exists \alpha : \kappa.\ p\}c\{a : A.\ q\}$ is valid
13 So we know for all r, that $r \in \{P'\}C\{a.\ Q(a)\}$
where $P' = \bigvee_{\tau \in [\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta} [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ (\theta, \tau)\ \gamma\ \delta$
14 Since $P = [\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta'\ \gamma\ \delta$, we know $P \subseteq P'$
15 Therefore $h \in P' * s$, and so $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$

□

Lemma 47. *(Disjunction Rule) If* $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$ *is valid and* $\Theta; \Gamma; \Delta \rhd \{p'\}c\{a : A.\ q'\} : \mathsf{spec}$ *is valid, then* $\Theta; \Gamma; \Delta \rhd \{p \vee p'\}c\{a : A.\ q \vee q'\} : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$ is valid
2 Assume $\Theta; \Gamma; \Delta \rhd \{p'\}c\{a : A.\ q'\} : \mathsf{spec}$ is valid
3 Assume suitable θ, γ, and δ
4 We want to show $[\![\Theta; \Gamma; \Delta \rhd \{p \vee p'\}c\{a : A.\ q \vee q'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta = \top$
5 So we want to show $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{p \vee p'\}c\{a : A.\ q \vee q'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
6 Assume r,
7 So we want to show $r \in [\![\Theta; \Gamma; \Delta \rhd \{p \vee p'\}c\{a : A.\ q \vee q'\} : \mathsf{spec}]\!]\theta\ \gamma\ \delta$
8 Let $P'' = [\![\Theta; \Gamma; \Delta \rhd p \vee p' : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
9 Let $C = [\![\Theta;\ \Gamma \vdash [c] : \bigcirc A]\!]\ \theta\ \gamma$
10 Let $Q'' = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q \vee q' : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
11 So we want to show $r \in \{P''\}C\{a : A.\ Q''(a)\}$
12 So we want to show $\forall s \succeq r.\ [P'' * s]\ C\ [a : A.\ Q''(a) * s]$
13 Assume $s \succeq r$
14 We want to show $[P'' * s]\ C\ [a : A.\ Q''(a) * s]$
15 We want to show $\forall h \in P'' * s.\ C\ Best(\lambda a.\ Q''(a) * s)\ h = \bot$
16 Assume $h \in P'' * s$
17 Therefore there are $h_1 \in P''$ and $h_2 \in s$ such that $h = h_1 \cdot h_2$
18 We know $P'' = [\![\Theta; \Gamma; \Delta \rhd p \vee p' : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta \vee [\![\Theta; \Gamma; \Delta \rhd p' : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
19 Call $P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$ and $P' = [\![\Theta; \Gamma; \Delta \rhd p' : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
20 Call $Q(v) = [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
and $Q'(v) = [\![\Theta; \Gamma, a : A; \Delta \rhd q' : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
21 By the definition of $\vee$, we know that $h_1 \in P$ or $h_1 \in P'$
22 Suppose $h_1 \in P$:
23 We know by assumption that $[P * s]\ C\ [a : A.\ Q(a) * s]$
24 We know that $h = h_1 \cdot h_2 \in P * s$
25 Hence $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
26 For any a, $Q(a) \subseteq Q''(a)$,
27 Hence $Best(\lambda a.\ Q''(a) * s) \sqsubseteq Best(\lambda a.\ Q(a) * s)$
28 Since C is continuous, $C\ Best(\lambda a.\ Q''(a) * s)\ h = \bot$
29 Suppose $h_1 \in P'$:
30 We know by assumption that $[P' * s]\ C\ [a : A.\ Q(a) * s]$
31 We know that $h = h_1 \cdot h_2 \in P' * s$
32 Hence $C\ Best(\lambda a.\ Q'(a) * s)\ h = \bot$
33 For any a, $Q'(a) \subseteq Q''(a)$
34 Hence $Best(\lambda a.\ Q''(a) * s) \sqsubseteq Best(\lambda a.\ Q'(a) * s)$
35 Since C is continuous, $C\ Best(\lambda a.\ Q''(a) * s)\ h = \bot$

□

Lemma 48. *(Equality Substitution) If* $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\}c[e/x]\{a : A.\ q\} : \mathsf{spec}$ *is valid and* $\Theta; \Gamma; \Delta \rhd r \supset e =_A e' : \mathsf{prop}$ *valid is valid, then* $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\}c[e'/x]\{a : A.\ q\} : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Theta; \Gamma; \Delta \rhd \{r\} \Rrightarrow \{p\}c[e/x]\{a : A.\ q\} : \mathsf{spec}$ is valid.
2 This is equivalent to assuming for all suitable θ, γ, and θ and s,
3 if $s \in [\![\{r\}]\!]\ \theta\ \gamma\ \delta$, then $s \in [\![\{p\}c[e/x]\{a : A.\ q\}]\!]\ \theta\ \gamma\ \delta$.
4 Assume $\theta\ \gamma\ \delta$, r, and $s \in [\![\{r\}]\!]\ \theta\ \gamma\ \delta$
5 Let
$C = [\![c[e/x]]\!]\ \theta\ \gamma$
$C' = [\![c[e'/x]]\!]\ \theta\ \gamma$
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$Q = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
6 Since we have r, we know that $R = [\![r]\!]\ \theta\ \gamma\ \delta = \top = H$.
7 Therefore it follows that $[\![\Theta; \Gamma; \Delta \rhd e =_A e' : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = H$
8 Therefore $[\![\Theta;\ \Gamma \vdash e : A]\!]\ \theta\ \gamma = [\![\Theta;\ \Gamma \vdash e' : A]\!]\ \theta\ \gamma$
9 Since substitution is sound, $[\![c[e/x]]\!]\ \theta\ \gamma = [\![c[e'/x]]\!]\ \theta\ \gamma$, or $C = C'$
10 By hypothesis, $s \in [\![\{p\}c[e/x]\{a : A.\ q\}]\!]\ \theta\ \gamma\ \delta$.
11 So for all $t \succeq s, h \in P * t$, we have $C\ Best(\lambda v.\ Q(v) * t)\ h = \bot$
12 Since $C = C'$, we know for all $t \succeq s, h \in P * t$, we have $C'\ Best(\lambda v.\ Q(v) * t)\ h = \bot$
13 Therefore, $s \in [\![\{p\}c[e'/x]\{a : A.\ q\}]\!]\ \theta\ \gamma\ \delta$.

□

Lemma 49. *(Case Analysis) Suppose* $\Theta; \Gamma, x : A; \Delta \rhd \langle p \wedge e =_{A+B} \mathsf{inl}(x)\rangle e_A \langle a : C.\ q\rangle : \mathsf{spec}$ *is valid, and* $\Theta; \Gamma, y : B; \Delta \rhd \langle p \wedge e =_{A+B} \mathsf{inr}(y)\rangle e_B \langle a : C.\ q\rangle : \mathsf{spec}$ *is valid.*

Then $\Theta; \Gamma; \Delta \rhd \langle p\rangle \mathsf{case}(e,\ x.\ e_A,\ y.\ e_B)\langle a : C.\ q\rangle : \mathsf{spec}$ *is valid.*

Proof.

1 Assume $\Delta, x : A \rhd \langle p \wedge e =_{A+B} \mathsf{inl}(x)\rangle e_A \langle a : C.\ q\rangle : \mathsf{spec}$ is valid.
2 Assume $\Delta, y : B \rhd \langle p \wedge e =_{A+B} \mathsf{inr}(y)\rangle e_B \langle a : C.\ q\rangle : \mathsf{spec}$ is valid.
3 We want to show $\Theta; \Gamma; \Delta \rhd \langle p\rangle \mathsf{case}(e,\ x.\ e_A,\ y.\ e_B)\langle a : C.\ q\rangle : \mathsf{spec}$ is valid.
4 We want to show that for all $\theta\ \gamma\ \delta, s$,
$s \in [\![\Theta; \Gamma; \Delta \rhd \langle p\rangle \mathsf{case}(e,\ x.\ e_A,\ y.\ e_B)\langle a : C.\ q\rangle : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
5 Assume $\theta\ \gamma\ \delta$ and s, and $h \in P * s$, letting
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$Q = \lambda v.\ [\![\Theta; \Gamma, a : C; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
$C_A = \lambda v.\ [\![\Theta;\ \Gamma, x : A \vdash e_A : \bigcirc C]\!]\ \theta\ (\gamma, v)\ \delta$
$C_B = \lambda v.\ [\![\Theta;\ \Gamma, y : B \vdash e_B : \bigcirc C]\!]\ \theta\ (\gamma, v)\ \delta$
$E = [\![\Theta;\ \Gamma \vdash e : A + B]\!]\ \theta\ \gamma$
6 We want to show $[C_A, C_B](E)\ Best(\lambda v.\ Q(v) * s)\ h = \bot$
7 Suppose $E = \mathsf{inl}(u)$ for some u:

Then we want to show that $C_A\ u\ Best(\lambda v.\ Q(v) * s)\ h = \bot$
We know $\Delta, x : A \rhd \langle p \wedge e =_{A+B} \mathsf{inl}(x)\rangle e_A \langle a : C.\ q\rangle$: spec is valid.
So for all $h \in (P \wedge [\![e =_{A+B} \mathsf{inl}(x)]\!]\ \theta\ (\gamma, \mathsf{inl}(u))\ \delta) * s$, $C_A\ u\ Best(\lambda v.\ Q(v) * s)\ h = \bot$
Since $E = \mathsf{inl}(u)$, we know that $\top = [\![\Theta; \Gamma, x : A; \Delta \rhd e =_{A+B} \mathsf{inl}(x) : \mathsf{prop}]\!]\ \theta\ (\gamma, \mathsf{inl}(u)\ \delta)$
Since equality is pure, $h \in [\![p \wedge e =_{A+B} \mathsf{inl}(x)]\!]\ \theta\ (\gamma, \mathsf{inl}(u))\ \delta * s$
Therefore $C_A\ u\ Best(\lambda v.\ Q(v) * s)\ h = \bot$

Suppose $E = \mathsf{inr}(u)$ for some u:

Then we want to show that $C_B\ u\ Best(\lambda v.\ Q(v) * s)\ h = \bot$
We know $\Theta; \Gamma, y : B; \Delta \rhd \langle p \wedge e =_{A+B} \mathsf{inr}(x)\rangle e_A \langle a : C.\ q\rangle$: spec is valid.
So for all $h \in (P \wedge [\![e =_{A+B} \mathsf{inr}(x)]\!]\ \theta\ (\gamma, \mathsf{inr}(u))\ \delta) * s$, $C_B\ u\ Best(\lambda v.\ Q(v) * s)\ h = \bot$
Since $E = \mathsf{inr}(u)$, we know that $\top = [\![\Theta; \Gamma, y : B; \Delta \rhd e =_{A+B} \mathsf{inr}(x) : \mathsf{prop}]\!]\ \theta\ (\gamma, \mathsf{inr}(u)\ \delta)$
Since equality is pure, $h \in [\![p \wedge e =_{A+B} \mathsf{inr}(x)]\!]\ \theta(\gamma, \mathsf{inr}(u))\ \delta * s$
Therefore $C_B\ u\ Best(\lambda v.\ Q(v) * s)\ h = \bot$

□

Lemma 50. *(The Rule of Consequence) If* $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\}$: spec *is valid and* $\Theta; \Gamma; \Delta \rhd p' \supset p$: prop *is valid and* $\Theta; \Gamma, a : A; \Delta \rhd q \supset q'$: prop *is valid, then* $\Theta; \Gamma; \Delta \rhd \{p'\}c\{a : A.\ q'\}$: spec *is valid.*

Proof.

Assume $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\}$: spec is valid
Assume $\Theta; \Gamma; \Delta \rhd p' \supset p$: prop is valid
Assume $\Theta; \Gamma; \Delta \rhd q \supset q'$: prop is valid
Assume suitable θ, γ, and δ
We want to show $\forall r.\ r \in \Theta; \Gamma; \Delta \rhd \{p'\}c\{a : A.\ q'\}$: spec
Assume r, and let
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\theta\ \gamma\ \delta$
$P' = [\![\Theta; \Gamma; \Delta \rhd p' : \mathsf{prop}]\!]\theta\ \gamma\ \delta$
$C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
$Q = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
$Q' = \lambda v.\ [\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
We know $[\![\Theta; \Gamma; \Delta \rhd p' \supset p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta = P' \supset P$.
We know for all v, $[\![\Theta; \Gamma, a : A; \Delta \rhd q \supset q' : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta = Q(v) \supset Q'(v)$.
So we want to show that $\forall s \succeq r.\ [P' * s]\ C\ [a : A.\ Q'(a) * s]$
Assume $s \succeq r$, and $h \in P' * s$
Hence $\exists\ h_1$ and h_2 such that $h_1 \# h_2$ and $h = h_1 \cdot h_2$ and $h_1 \in P'$ and $h_2 \in s$
Since P' is a subset of P, $h_1 \in P$. So $h \in P * s$
Therefore $C\ Best(\lambda a.\ Q(a) * s)\ h = \bot$
For any a, we know that $Q(a) \subseteq Q'(a)$
Hence we know that $Best(\lambda a.\ Q'(a) * s) \sqsupseteq Best(\lambda a.\ Q(a) * s)$
Hence by continuity, $C\ Best(\lambda a.\ Q'(a) * s)\ h = \bot$
Hence $\forall s \succeq r.\ [P' * s]\ C\ [a : A.\ Q'(a) * s]$
Hence $\forall r.\ r \in [\![\Theta; \Gamma; \Delta \rhd \{p'\}c\{a : A.\ q'\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$

□

3.8.4 The Syntactic Frame Property

We have defined a *syntactic* frame operator $S \otimes r$ on specifications.

$$\begin{array}{lcl}
\{p\}c\{a : A.\ q\} \otimes r & = & \{p * r\}c\{a : A.\ q * r\} \\
\langle p\rangle c\langle a : A.\ q\rangle \otimes r & = & \langle p * r\rangle c\langle a : A.\ q * r\rangle \\
\{p\} \otimes r & = & \{p\} \\
(S_1\ \&\ S_2) \otimes r & = & S_1 \otimes r\ \&\ S_2 \otimes r \\
(S_1\ ||\ S_2) \otimes r & = & S_1 \otimes r\ ||\ S_2 \otimes r \\
(S_1 \Rrightarrow S_2) \otimes r & = & S_1 \otimes r \Rrightarrow S_2 \otimes r \\
(\forall x : \omega.\ S) \otimes r & = & \forall x : \omega.\ (S \otimes r) \\
(\exists x : \omega.\ S) \otimes r & = & \exists x : \omega.\ (S \otimes r)
\end{array}$$

Proposition. *(Syntactic Well-Formedness of Frame Operator) If* $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$ *and* $\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}$*, then we define* $\Theta; \Gamma; \Delta \rhd S \otimes p : \mathsf{spec}$.

Proof. By structural induction on specifications. □

More interesting than the syntactic well-formedness of the frame operator is its semantic well-formedness.

Lemma. *(Syntactic Framing is Semantic Framing) If* $\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}$ *and* $\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}$*, then for all suitable* θ, γ*, and* δ*,* $[\![\Theta; \Gamma; \Delta \rhd S \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = [\![\Theta; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

Proof. This proof proceeds by induction on the derivation of S.

- Case SPECTRIPLE:

  1 We know $\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}$
  2 By the semantics of triples, we know that
  3 $[\![\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \{P\}C\{a.\ Q(a)\}$
  where
  $P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
  $C = [\![\Theta;\ \Gamma \vdash c \div A]\!]\ \theta\ \gamma$
  $Q = \lambda v.\ ([\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
  4 Now from the definition of syntactic framing, we know that $S \otimes r = \{p * r\}c\{a : A.\ q(a) * r\}$
  5 By semantics of triples, we know that
  $[\![\Theta; \Gamma; \Delta \rhd \{p * r\}c\{a : A.\ q * r\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \{P * R\}C\{a.\ Q(a) * R\}$
  where $R = [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
  6 By lemma we know that $\{P * R\}C\{a.\ Q(a) * R\} = \{P\}C\{a.\ Q(a)\} \otimes R$ semantically
  7 Therefore $[\![\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
  $[\![\Theta; \Gamma; \Delta \rhd \{p\}c\{a : A.\ q\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

- Case SPECMTRIPLE

1 We know $\Theta; \Gamma; \Delta \rhd \langle p \rangle e \langle a : A.\ q \rangle : \mathsf{spec}$
2 By the semantics of triples, we know that
$[\![\Theta; \Gamma; \Delta \rhd \langle p \rangle e \langle a : A.\ q \rangle : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \langle P \rangle E \langle a.\ Q(a) \rangle$
where
$P = [\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
$E = [\![\Theta;\ \Gamma \vdash e : \bigcirc A]\!]\ \theta\ \gamma$
$Q = \lambda v.\ ([\![\Theta; \Gamma, a : A; \Delta \rhd q : \mathsf{prop}]\!]\ \theta\ (\gamma, v)\ \delta$
3 Now from the definition of syntactic framing, we know that $S \otimes r = \langle p * r \rangle e \langle a : A.\ q(a) * r \rangle$
4 By semantics of triples, we know that
$[\![\Theta; \Gamma; \Delta \rhd \langle p * r \rangle e \langle a : A.\ q * r \rangle : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \langle P * R \rangle e \langle a.\ Q(a) * R \rangle$
where $R = [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
5 By lemma we know that $\langle P * R \rangle E \langle a.\ Q(a) * R \rangle = \langle P \rangle E \langle a.\ Q(a) \rangle \otimes R$ semantically
6 Therefore $[\![\Theta; \Gamma; \Delta \rhd \langle p \rangle c \langle a : A.\ q \rangle \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd \langle p \rangle c \langle a : A.\ q \rangle : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

- Case SPECASSERT

1 We know $\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}$
2 So we also know that $\{p\} \otimes r = \{p\}$
3 Therefore, $[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$ is either $\top$ or $\bot$,
depending on whether $[\![\Theta; \Gamma; \Delta \rhd p : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$ is H or $\emptyset$
4 Now, we will do a case analysis on the meaning of $\{p\}$
5 If $[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \top$
6 Then, since $S \subseteq S \otimes R$ for all S and R, and $\top$ is maximal, we know that
$\top = \top \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$
7 So $[\![\Theta; \Gamma; \Delta \rhd \{p\} \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$
8 If $[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta = \bot$
9 Then, since $\bot$ is the empty set, and since $S \otimes r = \{p \mid p * r \in S\}$,
10 we know $\bot = \bot \otimes [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$
11 So we know $[\![\Theta; \Gamma; \Delta \rhd \{p\} \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd \{p\} : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

- Case SPECBINARY

1 We know $\Theta; \Gamma; \Delta \rhd S_1 \otimes S_2 : \mathsf{spec}$, where $\oplus \in \{\ \&\ ,\ ||\ , \gg\}$
2 We also know that $(S_1 \oplus S_2) \otimes r = (S_1 \otimes r) \oplus (S_2 \otimes r)$
3 By the semantics, we know
$[\![\Theta; \Gamma; \Delta \rhd S_1 \oplus S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
$= [\![\Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta\ [\![\oplus]\!]\ [\![\Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta$
4 By the semantics, we know that
$[\![\Theta; \Gamma; \Delta \rhd (S_1 \oplus S_2) \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$

$(\llbracket \Theta; \Gamma; \Delta \rhd S_1 \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta) \llbracket \oplus \rrbracket (\llbracket \Theta; \Gamma; \Delta \rhd S_2 \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta)$

5 By induction, we know
$\llbracket \Theta; \Gamma; \Delta \rhd S_1 \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta = \llbracket \Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

6 By induction, we know
$\llbracket \Theta; \Gamma; \Delta \rhd S_2 \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta = \llbracket \Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

7 Therefore, we know $\llbracket \Theta; \Gamma; \Delta \rhd (S_1 \oplus S_2) \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$(\llbracket \Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta))$
$\llbracket \oplus \rrbracket$
$(\llbracket \Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta))$

8 Therefore we know $\llbracket \Theta; \Gamma; \Delta \rhd (S_1 \oplus S_2) \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$(\llbracket \Theta; \Gamma; \Delta \rhd S_1 : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta\ \llbracket \oplus \rrbracket\ \llbracket \Theta; \Gamma; \Delta \rhd S_2 : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta) \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

9 Therefore we know $\llbracket \Theta; \Gamma; \Delta \rhd (S_1 \oplus S_2) \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$\llbracket \Theta; \Gamma; \Delta \rhd S_1 \oplus S_2 : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

- Case SPECQUANTIFY1:

1 We know $\Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ S : \mathsf{spec}$, where $Q \in \{\forall, \exists\}$

2 We also know that $(Qu : \upsilon.\ S) \otimes r = Qu : \upsilon.\ (S \otimes r)$

3 By the semantics, we know that $\llbracket \Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ (S \otimes r) : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$\llbracket Q \rrbracket_{v \in \llbracket \Theta \rhd \upsilon : \mathsf{sort}} \rrbracket\ \theta} \llbracket \Theta; \Gamma; \Delta, u : \upsilon \rhd S \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ (\delta, v)$

4 By induction, we know that for all appropriate $\theta\ \gamma\ \delta'$, $\llbracket \Theta; \Gamma; \Delta, u : \upsilon \rhd S \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta' =$
$\llbracket \Theta; \Gamma; \Delta, u : \upsilon \rhd S : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta' \otimes \llbracket \Delta, u : \upsilon \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta'$

5 Now, choosing θ, γ, and $\delta' = (\delta, v)$,
then $\llbracket \Theta; \Gamma; \Delta, u : \upsilon \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta' = \llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta$ since $u \notin \mathrm{FV}(r)$

6 So we know $\llbracket \Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ (S \otimes r) : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$\llbracket Q \rrbracket_{v \in \llbracket \Theta \rhd \upsilon : \mathsf{sort}} \rrbracket\ \theta} (\llbracket \Theta; \Gamma; \Delta, u : \upsilon \rhd S : \mathsf{spec} \rrbracket\ \theta\ \gamma\ (\delta, v) \otimes \llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

7 Since framing distributes through meets, we know $\llbracket \Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ (S \otimes r) : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$(\llbracket Q \rrbracket_{v \in \llbracket \Theta \rhd \upsilon : \mathsf{sort}} \rrbracket\ \delta} \llbracket \Theta; \Gamma; \Delta, u : \upsilon \rhd S : \mathsf{spec} \rrbracket\ \theta\ \gamma\ (\delta, v)) \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

8 So we know $\llbracket \Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ (S \otimes r) : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$\llbracket \Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ S : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

9 So we know $\llbracket \Theta; \Gamma; \Delta \rhd (Qu : \upsilon.\ S) \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$\llbracket \Theta; \Gamma; \Delta \rhd Qu : \upsilon.\ S : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta \otimes (\llbracket \Theta; \Gamma; \Delta \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma\ \delta)$

- Case SPECQUANTIFY3:

1 We know $\Theta; \Gamma; \Delta \rhd Qy : B.\ S : \mathsf{spec}$, where $Q \in \{\forall, \exists\}$

2 We also know that $(Qy : B.\ S) \otimes r = Qy : B.\ (S \otimes r)$

3 By the semantics, we know that $\llbracket \Theta; \Gamma; \Delta \rhd Qy : B.\ (S \otimes r) : \mathsf{spec} \rrbracket\ \theta\ \gamma\ \delta =$
$\llbracket Q \rrbracket_{v \in \llbracket \Theta \rhd B : \mathsf{sort}} \rrbracket\ \theta} \llbracket \Theta; \Gamma, y : B; \Delta \rhd S \otimes r : \mathsf{spec} \rrbracket\ \theta\ (\gamma, v)\ \delta$

4 By induction, we know that for all appropriate $\theta\ \gamma'\ \delta$, $\llbracket \Theta; \Gamma, y : B; \Delta \rhd S \otimes r : \mathsf{spec} \rrbracket\ \theta\ \gamma'\ \delta =$
$\llbracket \Theta; \Gamma, y : B; \Delta \rhd S : \mathsf{spec} \rrbracket\ \theta\ \gamma'\ \delta \otimes \llbracket \Delta, y : B \rhd r : \mathsf{prop} \rrbracket\ \theta\ \gamma'\ \delta$

5 Now, choosing θ, δ and $\gamma' = (\gamma, v)$,

then $[\![\Theta; \Gamma; \Delta, y : B \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma'\ \delta = [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$ since $u \notin \mathrm{FV}(r)$

6 So we know $[\![\Theta; \Gamma; \Delta \rhd Qy : B.\ (S \otimes r) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![Q]\!]_{v \in [\![\Theta \rhd B : \mathsf{sort}]\!]\ \theta}([\![\Theta; \Gamma; \Delta, y : B \rhd S : \mathsf{spec}]\!]\ \theta\ (\gamma, v)\ \gamma\ \delta \otimes [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

7 Since framing distributes through meets, we know $[\![\Theta; \Gamma; \Delta \rhd Qy : B.\ (S \otimes r) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$([\![Q]\!]_{v \in [\![\Theta \rhd B : \mathsf{sort}]\!]\ \delta}[\![\Theta; \Gamma, y : B; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ (\gamma, v)\ \gamma\ \delta) \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

8 So we know $[\![\Theta; \Gamma; \Delta \rhd Qy : B.\ (S \otimes r) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd Qy : B.\ S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

9 So we know $[\![\Theta; \Gamma; \Delta \rhd (Qy : B.\ S) \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd Qy : B.\ S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

- Case SPECQUANTIFY2:

1 We know $\Theta; \Gamma; \Delta \rhd Q\alpha : \kappa.\ S : \mathsf{spec}$, where $Q \in \{\forall, \exists\}$

2 We also know that $(Q\alpha : \kappa.\ S) \otimes r = Q\alpha : \kappa.\ (S \otimes r)$

3 By the semantics, we know that $[\![\Theta; \Gamma; \Delta \rhd Q\alpha : \kappa.\ (S \otimes r) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![Q]\!]_{\tau \in [\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta}[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S \otimes r : \mathsf{spec}]\!]\ (\theta, \tau)\ \gamma\ \delta$

4 By induction, we know that for all appropriate $\theta'\ \gamma\ \delta$, $[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes [\![\Delta, \alpha : \kappa \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$

5 Now, choosing $\theta' = (\theta, \tau)$ and γ, δ
then $[\![\Theta, \alpha : \kappa; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta'\ \gamma\ \delta = [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta$ since $\alpha \notin \mathrm{FV}(r, \Gamma, \Delta)$

6 So we know $[\![\Theta; \Gamma; \Delta \rhd Q\alpha : \kappa.\ (S \otimes r) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![Q]\!]_{\tau \in [\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta}([\![\Theta; \Gamma; \Delta, \alpha : \kappa \rhd S : \mathsf{spec}]\!]\ (\theta, \tau)\ \gamma\ \delta \otimes [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

7 Since framing distributes through meets, we know $[\![\Theta; \Gamma; \Delta \rhd Q\alpha : \kappa.\ (S \otimes r) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$([\![Q]\!]_{\tau \in [\![\Theta \rhd \kappa : \mathsf{sort}]\!]\ \theta}[\![\Theta; \alpha : \kappa; \Gamma; \Delta \rhd S : \mathsf{spec}]\!]\ (\theta, \tau)\ \gamma\ \delta) \otimes [\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

8 So we know $[\![\Theta; \Gamma; \Delta \rhd Q\alpha : \kappa.\ (S \otimes r) : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd Q\alpha : \kappa.\ S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

9 So we know $[\![\Theta; \Gamma; \Delta \rhd (Q\alpha : \kappa.\ S) \otimes r : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta =$
$[\![\Theta; \Gamma; \Delta \rhd Q\alpha : \kappa.\ S : \mathsf{spec}]\!]\ \theta\ \gamma\ \delta \otimes ([\![\Theta; \Gamma; \Delta \rhd r : \mathsf{prop}]\!]\ \theta\ \gamma\ \delta)$

□

Chapter 4

Proving the Correctness of Design Patterns

4.1 Introduction

The widespread use of object-oriented languages creates an opportunity for designers of formal verification systems, above and beyond a potential "target market". Object-oriented languages have been used for almost forty years, and in that time practitioners have developed a large body of informal techniques for structuring object-oriented programs called *design patterns*[12]. Design patterns were developed to both take best advantage of the flexibility object-oriented languages permit, and to control the potential complexities arising from the unstructured use of these features.

This pair of characteristics make design patterns an excellent set of benchmarks for a program logic. First, design patterns use higher order programs to manipulate aliased, mutable state. This is a difficult combination for program verification systems to handle, and attempting to verify these programs will readily reveal weaknesses or lacunae in the program logic. Second, the fact that patterns are intended to structure and modularize programs means that we can use them to evaluate whether the proofs in a program logic respect the conceptual structure of the program – we can check to see if we need to propagate conceptually irrelevant information out of program modules in order to meet our proof obligations. Third, we have the confidence that these programs, though small, actually reflect realistic patterns of usage.

In this chapter, we give good specifications for and verify the following programs:

- We prove a collection and iterator implementation, which builds the Java aliasing rules for iterators into its specification, and which allows the construction of new iterators from old ones via the composite and decorator patterns.
- We prove a general version of the flyweight pattern (also known as hash-consing in the functional programming community), which is a strategy for aggressively creating aliased objects to save memory and permit fast equality tests. This also illustrates the use of the factory pattern.
- We prove a general version of the subject-observer pattern in a way that supports a strong form of information hiding between the subject and the observers.

4.2 Iterators and Composites

The iterator pattern is a design pattern for uniformly enumerating the elements of a collection. The idea is that in addition to a collection, we have an auxiliary data structure called the iterator, which has an operation `next`. Each time `next` is called, it produces one more element of the collection, with some signal when all of the elements have been produced. The iterators are mutable data structures whose invariants depend on the collection, itself another mutable data structure. Therefore, most object oriented libraries state that while an iterator is active, a client is only permitted to call methods on a collection that do not change the collection state (for example, querying the size of a collection). If destructive methods are invoked (for example, adding or removing an element), it is no longer valid to query the iterator again.

We also support operations to create new iterators from old ones, and to aggregate them into composite iterators. For example, given an iterator and a predicate, we can construct a new iterator that only returns those elements for which the predicate returns true. This sort of decorator takes an iterator object, and *decorates* it to yield an iterator with different behavior. Likewise, we can take two iterators and a function, and combine them into a new, *composite* iterator that returns the result of a parallel iteration over them both. These sorts of synthesized iterators are found in the `itertools` library in the Python programming language, the Google Java collections library, or the C5 library [18] for C#.

Aliasing enters into the picture, above and beyond the restrictions on the underlying collections, because iterators are stateful objects. For example, if we create a filtering iterator, and advance the underlying iterator, then what the filtering iterator will return may change. Even more strikingly, we cannot pass the same iterator twice to a parallel iteration constructor – the iterators must be disjoint in order to correctly generate the two sequences of elements to combine.

Below, we give a specification of an iterator pattern. We'll begin by describing the interface informally, in English, and then move on to giving formal specifications and explaining them.

The interface consists of two types, one for collections, and one for iterators. The operations the collection type supports are 1) creating new mutable collections, 2) adding new elements to an existing collection, and 3) querying a collection for its size. Adding new elements to a collection is a destructive operation which modifies the existing collection, whereas getting a collection's size does not modify the collection.

The interface that the iterator type supports includes:

1. creating a new iterator on a collection,
2. destructively getting the next element from an iterator (returning an error value if the iterator is exhausted), and
3. operations producing new iterators from old. The operations we support are:
   (a) a filter operation, which takes an iterator along with a boolean predicate, and returns an iterator which enumerates the elements satisfying the predicate, and
   (b) a parallel map operation, which takes two iterators and a two-argument function, and returns an iterator which returns the result of enumerating the elements of the two iterators in parallel, and applying the function to each pair of elements.

The aliasing protocol that our iterator protocol will satisfy is essentially the same as the one

the Java standard libraries specify in their documentation.

- Any number of iterators can be created from a given collection. Each of these iterators depends on the state of the collection when the iterator is created, and is only valid as long as the collection state does not change.

  Each iterator also has its own traversal state, which tracks how many of the collection's elements it has yet to yield.

- Iterators can be constructed from other iterators, and these composite iterators depend on the traversal states of all of the iterators they are constructed from. As a result, they also depend on the state of each collection each constituent iterator depends upon.

- An iterator is valid only as long as none of the collections it depends on have been modified, and it is the only thing that has the right to modify the traversal state of the iterators it depends on.

  It is legal to call functions on an iterator only when it is in a valid state. Performing a destructive operation on any collection an iterator depends upon invalidates it. For example, adding an element to any collection an iterator depends on will invalidate the iterator, as will enumerating the elements from any of the iterators that it depends on.

  Likewise, only the iterator itself may modify the traversal state of any iterator it depends upon. Any other modification may invalidate it.

4.2.1 The Iterator Specification

Now, we will describe the specification, given in Figure 4.1, in detail. This whole specification follows the usual pattern of introducing abstract types, predicates, and operations with existential quantification, and then specifying the behavior of the operations with a conjunction of Hoare triples.

In lines 1 and 2, we introduce two abstract type constructors, $\mathsf{colltype}$ and $\mathsf{itertype}$. These are both type constructors of kind $\star \to \star$, which take an argument that describes their element type. So $\mathsf{colltype}(\mathbb{N})$ represents a collection of natural numbers, and $\mathsf{itertype}(\mathsf{bool})$ represents an iterator which will produce a sequence of boolean values.

In lines 3 and 4, we give two abstract predicates $coll$ and $iter$, to represent the state associated with a collection and iterator, respectively. The sort of the collection predicate is $\Pi\alpha : \star.\ \mathsf{colltype}(\alpha) \Rightarrow \mathsf{seq}\ \alpha \Rightarrow \mathsf{prop} \Rightarrow \mathsf{prop}$, which we will write using an expression like $coll_\alpha(c, xs, P)$.

The first argument is a type argument (e.g., α), indexing the whole predicate by the element type of the collection. (We will often suppress this type argument when it is obvious from context.) The second argument is an argument of collection type (in our example, c, of type $\mathsf{colltype}(\alpha)$) which indicates the value with which our state is associated. The third argument (here, xs) is the purely mathematical sequence (i.e., an element of the free monoid over α) the collection c represents. The fourth, and final, argument is a proposition-valued *abstract state* of the collection, which we use to track whether or not the collection has been modified or not.

The appearance of this argument might be a little bit surprising: naively, we might suppose the mathematical sequence that collection represents constitutes a sufficient description of the

collection, and so we might expect our predicates to take on the form $coll_\alpha(c, xs)$, with no state argument. However, this is *not* sufficient. The iterator contract forbids modifying the collection at all, while iterators are active upon it, and the mathematical sequence a collection represents is not enough information to decide whether a collection has been modified or not.

For example, suppose we have a collection c representing the mathematical sequence $\langle 2, 3, 4, 5\rangle$, which might have the predicate $coll_{\mathbb{N}}(c, \langle 2, 3, 4, 5\rangle, P)$. Now, suppose we first add the element 1 to the front of the sequence (so that the collection c now represents $\langle 1, 2, 3, 4, 5\rangle$), and then immediately remove the first element. Then, the collection c will still represent the sequence $\langle 2, 3, 4, 5\rangle$, but it will have suffered an intervening modification.

This kind of change can be catastrophic for iterator implementations. As a concrete example, suppose that we had represented our collection with a balanced binary tree, and represent the iterator as a pointer into the middle of that tree. Adding and removing elements from the collection can cause a tree rebalancing, which can potentially leave the iterator pointer with a stale or dangling pointer.

As a result, our specification must track whether a collection has been modified or not, and this is what the abstract state field on the predicate does. Operations on a collection which do not change its underlying state will leave the abstract state unchanged from pre- to post-condition, whereas destructive operations (such as adding or removing elements) do change the abstract state.

On line 4, we assert the existence of the iterator predicate $iter$. It is also a four-place predicate, and has sort $\Pi\alpha : \star.\ \mathsf{itertype}(\alpha) \Rightarrow \mathsf{seq}\ \alpha \Rightarrow \mathcal{P}(\Sigma\beta : \star.\ \mathsf{colltype}(\beta) \times \mathsf{seq}\ \beta \times\ \mathsf{prop}) \Rightarrow \mathsf{prop}$, which we will write using an expression like $iter_\alpha(i, xs, S)$.

The first argument (in our example, α) is a type argument describing the type of elements the iterator will produce. The second argument (here, i) is the concrete iterator value to which the predicate is associated. Then, we have a sequence argument (here, xs) which describes the elements yet to be produced from this iterator – if $xs = \langle 5, 7, 9\rangle$, then the next three elements the iterator will yield are 5, 7, and 9, in that order. Subsequently the iterator will be empty and unable to produce any more elements.

Finally, we have a state argument for iterators, as well. In contrast to the case for collections, this argument is a set of propositions, representing an entire set of collection states. Since our interface supports operations which allow building new iterators from old, an iterator may access many different collections to produce a single new element. As a result, we have to track the state of each collection the iterator depends on, so that we can verify that we do not ever need to read a modified collection.

The operation $\mathsf{newcoll}$ is declared on line 5, and specified on line 13. The specification asserts that the call $\mathsf{newcoll}_\alpha$ may happen from any precondition state, and that it creates and adds a new, empty collection to the program state. The postcondition assertion $\exists P.\ coll_\alpha(a, \epsilon, P)$ says that the return value a is a collection representing the empty sequence ϵ, and that the collection begins its life in some arbitrary abstract state P.

The $\mathsf{size}_\alpha(c)$ function, which is declared on line 6 and specified on line 14, takes a type argument α and a collection c, and returns the number of elements in c. To call this function, we must have access to the collection $coll_\alpha(c, xs, P)$ in our precondition, and it is returned to us unchanged in the postcondition, with the return value a equal to the length of xs, the sequence c represents. In particular, note that the abstract state P of the $coll(c, xs, P)$ predicate remains

$\exists \mathsf{colltype} : \star \to \star$
$\exists \mathsf{itertype} : \star \to \star$

$\exists coll : \Pi\alpha : \star.\ \mathsf{colltype}(\alpha) \Rightarrow \mathsf{seq}\ \alpha \Rightarrow \mathsf{prop} \Rightarrow \mathsf{prop}.$
$\exists iter : \Pi\alpha : \star.\ \mathsf{itertype}(\alpha) \Rightarrow \mathsf{seq}\ \alpha \Rightarrow \mathcal{P}(\Sigma\beta : \star.\ \mathsf{colltype}(\beta) \times \mathsf{seq}\ \beta \times \mathsf{prop}) \Rightarrow \mathsf{prop}.$

$\exists \mathsf{newcoll} : \forall\alpha : \star.\ \bigcirc\,(\mathsf{colltype}(\alpha)).$
$\exists \mathsf{size} : \forall\alpha : \star.\ \mathsf{colltype}(\alpha) \to \bigcirc\mathbb{N}.$
$\exists \mathsf{add} : \forall\alpha : \star.\ \mathsf{colltype}(\alpha) \times \alpha \to \bigcirc\mathbf{1}.$
$\exists \mathsf{remove} : \forall\alpha : \star.\ \mathsf{colltype}(\alpha) \to \bigcirc(\mathsf{option}(\alpha)).$

$\exists \mathsf{newiter} : \forall\alpha : \star.\ \mathsf{colltype}(\alpha) \to \bigcirc(\mathsf{itertype}(\alpha)).$
$\exists \mathsf{filter} : \forall\alpha : \star.\ (\alpha \to \mathsf{bool}) \times \mathsf{itertype}(\alpha) \to \bigcirc(\mathsf{itertype}(\alpha)).$
$\exists \mathsf{merge} : \forall\alpha, \beta, \gamma : \star.\ (\alpha \to \beta \to \gamma) \times \mathsf{itertype}(\alpha) \times \mathsf{itertype}(\beta) \to \bigcirc(\mathsf{itertype}(\gamma)).$
$\exists \mathsf{next} : \forall\alpha : \star.\ \mathsf{itertype}(\alpha) \to \bigcirc(\mathsf{option}(\alpha)).$

$\forall\alpha.\ \langle \mathsf{emp} \rangle \mathsf{newcoll}_\alpha \langle a : \mathsf{colltype}(\alpha).\ \exists P.\ coll_\alpha(a, \epsilon, P) \rangle\ \&$

$\forall\alpha, c, P, xs.\ \langle coll_\alpha(c, xs, P) \rangle$
$\quad \mathsf{size}_\alpha(c)$
$\quad \langle a : \mathbb{N}.\ coll_\alpha(c, xs, P) \wedge a = |xs| \rangle\ \&$

$\forall\alpha, c, P, x, xs.\ \langle coll_\alpha(c, xs, P) \rangle$
$\quad \mathsf{add}_\alpha(c, x)$
$\quad \langle a : 1.\ \exists Q.\ coll_\alpha(c, x \cdot xs, Q) \rangle\ \&$

$\forall\alpha, c, P.\ \langle coll_\alpha(c, \epsilon, P) \rangle \mathsf{remove}_\alpha(c) \langle a : \mathsf{option}(\alpha).\ coll_\alpha(c, \epsilon, P) \wedge a = \mathsf{None} \rangle\ \&$
$\forall\alpha, c, x, xs, P. \langle coll_\alpha(c, x \cdot xs, P) \rangle$
$\quad \mathsf{remove}_\alpha(c)$
$\quad \langle a : \mathsf{option}(\alpha).\ \exists Q.\ coll_\alpha(c, xs, Q) \wedge a = \mathsf{Some}(x) \rangle$

$\&\ \forall\alpha, c, P, xs.\ \langle coll_\alpha(c, xs, P) \rangle$
$\quad \mathsf{newiter}_\alpha(c)$
$\quad \langle a : \mathsf{itertype}(\alpha).\ coll(c, xs, P) * iter(a, xs, \{(\alpha, c, xs, P)\}) \rangle\ \&$

$\forall\alpha, p, i, S, xs.\ \langle iter_\alpha(i, xs, S) \rangle$
$\quad \mathsf{filter}_\alpha(p, i)$
$\quad \langle a : \mathsf{itertype}(\alpha).\ iter_\alpha(a, \mathit{filter}\ p\ xs, S) \rangle\ \&$

$\forall\alpha, \beta, \gamma, f, i, S, xs, i', S', xs'.$
$\quad \langle iter_\alpha(i, xs, S) * iter_\beta(i', xs', S') \rangle$
$\quad \mathsf{merge}_{\alpha\,\beta\,\gamma}(f, i, i')$
$\quad \langle a : \mathsf{itertype}(\gamma).\ iter_\gamma(a, \mathit{map2}\ f\ xs\ xs', S \cup S') \rangle\ \&$

$\forall\alpha, i, S.\ \langle colls(S) * iter_\alpha(i, \epsilon, S) \rangle$
$\quad \mathsf{next}_\alpha(i)$
$\quad \langle a : \mathsf{option}(\alpha).\ colls(S) * iter_\alpha(i, \epsilon, S) \wedge a = \mathsf{None} \rangle\ \&$

$\forall\alpha, i, S, x, xs.\ \langle iter_\alpha(i, x \cdot xs, S) * colls(S) \rangle$
$\quad \mathsf{next}(i)$
$\quad \langle a : \mathsf{option}(\alpha).\ iter_\alpha(i, xs, S) \wedge a = (\mathsf{Some}\ x) * colls(S) \rangle$

Figure 4.1: Interface to the Iterator Library

1 $colls(\emptyset) \equiv \mathsf{emp}$
2 $colls(\{(\alpha, c, xs, P)\} \uplus S) \equiv coll_\alpha(c, xs, P) * colls(S)$

3 $filter\ p\ \epsilon \equiv \epsilon$
4 $filter\ p\ (x \cdot xs) \equiv \mathsf{if}\ p\ x = \mathsf{true}\ \mathsf{then}\ x \cdot (filter\ p\ xs)\ \mathsf{else}\ filter\ p\ xs$

5 $map2\ f\ \epsilon\ ys = \epsilon$
6 $map2\ f\ xs\ \epsilon = \epsilon$
7 $map2\ f\ (x \cdot xs)\ (y \cdot ys) = (f\ x\ y) \cdot (map2\ f\ xs\ ys)$

Figure 4.2: Auxilliary Functions Used in the Iterator Specification

unchanged in the pre- and post-conditions, indicating that this function does not change the abstract state.

The function call $\mathsf{add}_\alpha(c, x)$, which adds an element x to a collection c, is declared on line 7 and is specified on line 17. We start with a precondition $coll_\alpha(c, xs, P)$ and move to a postcondition state $\exists Q.\ coll_\alpha(c, x \cdot xs, Q)$. Because we destructively modify the collection c when we add x to it, we also specify that the abstract state in the postcondition is existentially quantified. This ensures that clients cannot assume that the abstract state remains the same after a call to add has been made. In this way, the abstract state behaves a bit like a time stamp, changing to some new state whenever a modification is made to the collection.

Similarly, the function call $\mathsf{remove}_\alpha(c)$ (declared on line 8) removes an element from the collection c. We give this procedure two specifications, on lines 20 and 21, corresponding to when the collection is empty, or not. In the first specification (on line 20), we begin with the precondition $coll_\alpha(c, \epsilon, P)$, and end in the postcondition $coll_\alpha(c, \epsilon, P) \wedge a = \mathsf{None}$. The fact that the abstract state remains P means the collection is unchanged, and the return value a equals None, an element of option type, indicating that there was no element to remove. In the second specification (on line 21), we begin with the precondition $coll_\alpha(c, x \cdot xs, P)$, from which we can see that the collection is nonempty. Then, the postcondition is $\exists Q.\ coll_\alpha(c, xs, Q) \wedge a = \mathsf{Some}(x)$. The value $\mathsf{Some}(x)$ is returned as the return value of the function, and the state of the collection changes to reflect that the element x has been removed — including a change to the abstract state of the collection.

As an aside, in practice it is usually more convenient to specify a procedure with a single Hoare triple, rather than multiple Hoare triples. However, in this example, I choose to give multiple specifications of the same procedure in order to illustrate that it is indeed possible within specification logic.

The $\mathsf{newiter}_\alpha(c)$ function is declared on line 9. Its type is $\forall \alpha : \star.\ \mathsf{colltype}(\alpha) \to \bigcirc(\mathsf{itertype}(\alpha))$. This means that it is given a type and a collection of that type, and then it returns an iterator over that type, possibly creating auxilliary data structures.

A call $\mathsf{newiter}_\alpha(c)$ is specified on line 24, and beginning from a precondition state $coll_\alpha(c, xs, P)$, it goes to a postcondition state $coll_\alpha(c, xs, P) * iter_\alpha(a, xs, \{(c, xs, P)\})$. This means that given access to a collection c, our function will return an iterator object (bound to a), which will enumerate the elements of c (that is, it will produce the elements xs). Furthermore, the abstract state that it depends on is just the singleton set $\{P\}$, since this iterator will read only

c. Finally, the fact that $coll_\alpha(c, xs, P)$ occurs in both the pre- and the post-condition means that this function needs access to c's state, but does not modify its abstract state.

The $\mathsf{filter}_\alpha(p, i)$ (declared on line 10) takes a boolean function p and an iterator i, and returns a new iterator which will enumerate only those elements which for which p returns true. This function is specified on line 27, and it takes a precondition $iter_\alpha(i, xs, S)$ to a postcondition $iter_\alpha(a, \mathit{filter}\ p\ xs, S)$.

First, note that we use a mathematical function *filter* to explain the filtering behavior in terms of sequences. Second, note that the original iterator state $iter_\alpha(i, xs, S)$ vanishes from the postcondition – it is consumed by the call the filter. This reflects the fact the filtered iterator takes ownership of the underlying iterator, in order to prevent third parties from making calls to $\mathsf{next}(i)$ and possibly changing the state of the filtered iterator.

This is also why the support set S for an iterator only needs to track the abstract states of the collections, rather than tracking the state of both collections and iterators. When we take ownership of the argument's iterator state, we prevent third parties from being able to call functions on the argument after creating the new iterator. This takes advantage of the resource-conscious nature of separation logic: a specification must have access to its footprint, and so we can hide state inside a predicate to control which operations are allowed.

The merge function is declared on line 11, and has type $\forall \alpha, \beta, \gamma : \star.\ (\alpha \to \beta \to \gamma) \times \mathsf{itertype}(\alpha) \times \mathsf{itertype}(\beta) \to \bigcirc(\mathsf{itertype}(\gamma))$. Thus, a call $\mathsf{merge}_{\alpha\,\beta\,\gamma}(f, i, i')$ takes a function and two iterators, and constructs a new iterator which steps over the two inputs in parallel, returning the result of f applied to each pair of elements of i and i'.

We specify calls $\mathsf{merge}_{\alpha\,\beta\,\gamma}(f, i_1, i_2)$ on line 30, and it takes a precondition $iter_\alpha(i_1, xs, S_1) * iter_\beta(i_2, ys, S_2)$. This means that we have state associated with two separate iterators, which we take to the postcondition $iter_\gamma(a, \mathit{map2}\ f\ xs\ ys, S_1 \cup S_2)$. As with the filter function, we consume the two input iterators to produce the return value iterator. And also as with filter, we use a mathematical function $\mathit{map2}$ to specify the action on mathematical sequences.

One point worth noting is that it is important that the two argument iterators have separate state from one another. In a functional program, there is no difficulty with a program $\mathit{map2}\ f\ xs\ xs$, because we are free to re-traverse a list multiple times. However, since traversing an iterator is a destructive operation, a call like $\mathsf{merge}_{\alpha,\alpha,\beta}\ f\ i\ i$ could (if it were allowed) give the wrong answer, for example by pairing consecutive elements of the iterator.

The final operation in our interface is the next function, declared on line 12. The type of this function is $\forall \alpha : \star.\ \mathsf{itertype}(\alpha) \to \bigcirc(\mathsf{option}(\alpha))$. When invoked, it will return an option, with the None value if the iterator is exhausted, and Some of an element if the iterator still has elements to produce.

As with remove, we specify this procedure with two specifications, one for the case when the iterator is empty and another for when it is non-empty. On line 34, we give the specification for when the iterator is exhausted, and on line 37, we give the specification for when the iterator is not exhausted.

In either case, the precondition for the function contains as one part the predicate $colls(S)$. The assertion-level function $colls(S)$ is a function that iterates over a set of abstract states, and re-associates them with collection predicates (coming from the argument S) in the precondition, to form a predicate $coll_{\tau_1}(c_1, xs_1, S_1) * \ldots * coll_{\tau_n}(c_n, xs_n, S_n)$. This expresses the requirement that we need access to *all* of the collections i depends on, all in the correct abstract state, in order

to use it. This function is defined in Figure 4.2.[1]

In line 34, $colls(C, S)$ is joined with the specification of the iterator, $iter_\alpha(i, \epsilon, S)$. Note that the same S is used, so that we are referring only to the collections the iterator may need to read. As expected, $\mathsf{next}_\alpha(i)$ returns None. On the other hand, if the iterator still has elements (i.e., is in a state $iter_\alpha(i, x \cdot xs, S)$), we use the specification on line 37, and see it returns the first element as Some x, and sets the state to $iter(i, xs, S)$ in the postcondition (line 15).

Example Client

Below, we give an example use of this module in annotated program style. (Here, and in what follows, we suppress explicit type applications when they are obvious in context.)

1 $\{\mathsf{emp}\}$
2 letv $c_1 = \mathsf{newcoll}()$ in
3 $\{\exists P_1'.\ coll(c_1, \epsilon, P_1')\}$
4 $\{coll(c_1, \epsilon, P_1)\}$
5 letv $() = \mathsf{add}(c_1, 4)$ in
6 $\{\exists P_2.\ coll(c_1, 4 \cdot \epsilon, P_2)\}$
7 $\{coll(c_1, 4 \cdot \epsilon, P_2)\}$
8 letv $() = \mathsf{add}(c_1, 3)$ in
9 letv $() = \mathsf{add}(c_1, 2)$ in
10 $\{coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
11 letv $c_2 = \mathsf{newcoll}()$ in
12 letv $() = \mathsf{add}(c_2, 3)$ in
13 letv $() = \mathsf{add}(c_2, 5)$ in
14 $\{coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
15 letv $i_1 = \mathsf{newiter}(c_1)$ in
16 $\{iter(i_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\})$
17 $\quad * coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
18 letv $i_1' = \mathsf{filter}(even?, i_1)$ in
19 $\{iter(i_1', 2 \cdot 4 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\})$
20 $\quad * coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
21 letv $i_2 = \mathsf{newiter}(c_2)$ in
22 $\{iter(i_1', 2 \cdot 4 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\})$
23 $\quad * iter(i_2, 5 \cdot 3 \cdot \epsilon, \{(c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\})$
24 $\quad * coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
25 letv $i = \mathsf{merge}(plus, i_1', i_2)$ in
26 $\{iter(i, 7 \cdot 7 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\})$
27 $\quad * coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
28 letv $n = \mathsf{size}(c_2)$ in

[1]Technically, this is an abuse of notation, since primitive recursion is not well defined on sets. The proper way to do this would be to introduce a binary relation $colls(S, R)$ between state sets and predicates, and to put R in the precondition state. However, since the separating conjunction is commutative, no confusion is possible and I will retain the functional form.

29 $\{n = 2\ \wedge iter(i, 7 \cdot 7 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\})$
30 $\quad * coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
31 letv $x = \mathsf{next}(i)$ in
32 $\{n = 2 \wedge x = \mathsf{Some}\ 7 \wedge$
33 $\quad iter(i, 7 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\})$
34 $\quad * coll(c_2, 5 \cdot 3 \cdot \epsilon, Q_2) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
35 $\mathsf{add}(c_2, 17)$
36 $\{n = 2 \wedge x = \mathsf{Some}\ 7 \wedge$
37 $\quad iter(i, 7 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\})$
38 $\quad * (\exists Q_3.\ coll(c_2, 17 \cdot 5 \cdot 3 \cdot \epsilon, Q_3)) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$
39 $\{\exists Q_2, Q_3, P_4. n = 2 \wedge x = \mathsf{Some}\ 7 \wedge$
40 $\quad iter(i, 7 \cdot \epsilon, \{(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4), (c_2, 5 \cdot 3 \cdot \epsilon, Q_2)\})$
41 $\quad * coll(c_2, 17 \cdot 5 \cdot 3 \cdot \epsilon, Q_3) * coll(c_1, 2 \cdot 3 \cdot 4 \cdot \epsilon, P_4)\}$

In line 1 of this example, we begin in an empty heap. In line 2, we create a new collection c_1, which yields us the state $\exists P_1'.\ coll(c_1, \epsilon, P_1')$, with an existentially quantified abstract state.

Because P_1' is existentially quantified, we do not know what value it actually takes on. However, we can drop the existential using the AXFORGET axiom, which says that if we prove the rest of the program using a freshly-introduced variable P_1, then we know that the rest of the program will work for *any* value of P_1, because free variables are implicitly universally quantified. So it will work with whatever value P_1' had. So we drop the quantifier on line 4, and try to prove this program with the universally-quantified P_1.[2]

This permits us to add the element 4 to c_1 on line 5. Its specification puts the predicate $coll()$ on line 6 again into an existentially quantified state P_2. So we again replace P_2 with a fresh variable P_2 on line 7, and will elide these existential introductions and unpackings henceforth.

Starting on line 8, we add two more elements to c_1, and on lines 11-13, we create another collection c_2, and add 3 and 5 to it, as can be seen in the state predicate on line 14. On line 15, we create the iterator i_1 on the collection c_1. The $iter$ predicate on line 16 names i_1 as its value, and lists c_1 in state P_4 as its support, and promises to enumerate the elements 2, 3, and 4.

On line 18, $\mathsf{filter}(even?, i_1)$ creates the new iterator i_1'. This iterator yields only the even elements of i_1, and so will only yield 2 and 4. On line 19, i_1's iterator state has been consumed to make i_1''s state. We can no longer call $\mathsf{next}(i_1)$, since we do not have the resource invariant needed to prove anything about that call. Thus, we cannot write a program that would break i_1''s representation invariant.

On line 21, we create a second iterator i_2 enumerating the elements of c_2. The state on line 22 now has predicates for i_1', i_2, c_1 and c_2. On line 25, $\mathsf{merge}(plus, i_1', i_2)$ creates a new iterator i, which produces the pairwise sum of the elements of i_1' and i_2, and consumes the iterator states for i_1' and i_2 to yield the state for the new iterator i. Note that the invariant for i does not make any mention of what it was constructed from, naming only the collections it needs as support.

On line 28, the size call on c_2 illustrates that we can call non-destructive methods while iterators are active. The call to $\mathsf{next}(i)$ on line 31 binds $\mathsf{Some}\ 7$ to x, and the the iterator's sequence argument (line 33) shrinks by one element. On line 35, we call $\mathsf{add}(c_2, 17)$ the state

[2] A useful analogy is the existential elimination rule in the polymorphic lambda calculus: we prove that we can use an existential by showing that our program is well-typed no matter what the contents of the existential are.

of c_2 changes to $\exists Q_3.\ coll(c, 17 \cdot 5 \cdot 3 \cdot \epsilon, Q_3)$ (line 38). So we can no longer call $\mathsf{next}(i)$, since it needs c_2 to be in the state Q_2. (On the following line, we repack all of the existentials, in accordance with the AXFORGETEX rule of the program logic.)

Discussion. This example shows a pleasant synergy between higher-order quantification and separation logic. We can give a relatively simple specification to the clients of the collection library, even though the internal invariant is quite subtle (as we will see in the next section, it will use the magic wand). Higher-order logic also lets us freely define new data types, and so our specifications can take advantage of the pure, non-imperative nature of the mathematical world, as can be seen in the specifications of the filter and merge functions – we can use equational reasoning on purely functional lists in our specifications, even though our algorithms are imperative.

4.2.2 Example Implementation

In this subsection, we describe one particular implementation of iterators, based on a simple linked list implementation. The type and predicate definitions are given in Figure 4.3, and the implementation of the procedures is given in Figure 4.4.

Definitions of the Types and Predicates

We'll begin by giving an intuitive explanation of the predicates, before giving correctness proofs for the operations.

Starting on line 1 of Figure 4.3, we define the type of collections. Technically, these are recursive type definitions, which we did not define in our semantics in Chapter 1. Fortunately, there is no great difficulty in these definitions — we are giving polynomial data types, and we can justify these definition via the fact that for every polynomial functor F, the category of F-algebras over CPO has an initial object (which means that the data type and primitive iteration over it are well-defined).

The type of collections $\mathsf{colltype}(\tau)$ is a mutable linked list, consisting of a reference to a value of type $\mathsf{listcell}(\tau)$. A list content is either a Nil value, or a cons cell $\mathsf{Cons}(x, tl)$ consisting of a value of type τ and a tail list of type $\mathsf{colltype}(\tau)$. Unlike the typical definition of purely functional linked lists in ML, the tails of a list are mutable references, rather than list values.

The type of iterators, given in line 3, is an inductive datatype, with one clause for each of the possible ways to construct a new iterator from an old one. This type arises as follows. When we filter an iterator (or merge two iterators), we simply store a function together with the iterator (or two iterators) that are given as inputs. For an iterator into a single collection, we store an interior pointer into the argument list, giving us the type of a pointer to a list as the type of the One constructor.

Then, on line 6, we define the auxilliary predicate $list(\tau, c, xs)$, which asserts that c is a list value representing the mathematical sequence xs, with element type τ. This is defined by recursion over the sequence xs, with the empty sequence represented by a pointer to Nil, and a sequence $y \cdot ys$ represented by a pointer to a cons cell whose head is x and whose tail is a collection representing ys.

$$
\begin{array}{ll}
1 & \mathsf{listcell}(\alpha) = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha \times \mathsf{ref}\ (\mathsf{listcell}(\alpha)) \\
 & \mathsf{list}\ (\alpha) = \mathsf{ref}\ (\mathsf{listcell}(\alpha)) \\
2 & \mathsf{colltype}(\alpha) = \mathsf{list}\ \alpha \\
3 & \mathsf{itertype}(\alpha) = \mathsf{One\ ref}\ (\mathsf{colltype}(\alpha)) \\
4 & \qquad\qquad \mid \mathsf{Filter}\ (\alpha \to \mathit{bool}) \times \mathsf{itertype}(\alpha) \\
5 & \qquad\qquad \mid \mathsf{Merge}\ \exists \beta, \gamma : \star.\ (\beta \to \gamma \to \alpha) \times \mathsf{itertype}(\beta) \times \mathsf{itertype}(\gamma) \\
6 & \mathit{list}(\tau, c, xs) \qquad\qquad \equiv \exists v.\ c \mapsto_\tau v * \mathit{listcell}(\tau, v, xs) \\
7 & \mathit{listcell}(\tau, \mathsf{Nil}, xs) \qquad \equiv xs = \epsilon \\
8 & \mathit{listcell}(\tau, \mathsf{Cons}(y, c), xs) \quad \equiv \exists ys.\ xs = y \cdot ys \wedge \mathit{list}(\tau, c, ys) \\
9 & \mathit{coll}_\tau(c, xs, P) \equiv \mathit{list}(\tau, c, xs) \wedge P \wedge \mathsf{exact}(P) \\
10 & \mathit{iter}_\tau(\mathsf{One}\ i, xs, \{(\tau, c, ys, P)\}) \qquad \equiv \exists c'.\ i \mapsto c' * \\
11 & \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{coll}(c, ys, P) \mathrel{-\!\!*} (\mathit{coll}(c, ys, P) \wedge \\
12 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\top * \mathit{list}(c', xs)))) \\
13 & \mathit{iter}_\tau(\mathsf{One}\ i, xs, S) \qquad\qquad \equiv \bot \text{ when } |S| \neq 1 \\
14 & \mathit{iter}_\tau(\mathsf{Filter}(p, i), xs, S) \qquad \equiv \exists ys.\ \mathit{iter}_\tau(i, ys, S) \wedge xs = \mathit{filter}\ p\ ys \\
15 & \mathit{iter}_\tau(\mathsf{Merge}(\tau_1, \tau_2, f, i_1, i_2), xs, S) \qquad \equiv \exists S_1, ys, S_2, zs. \\
16 & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{iter}_{\tau_1}(i_1, ys, S_1) * \mathit{iter}_{\tau_2}(i_2, zs, S_2) \wedge \\
17 & \qquad\qquad\qquad\qquad\qquad\qquad S = S_1 \cup S_2 \wedge xs = \mathit{map2}\ f\ ys\ zs
\end{array}
$$

Figure 4.3: Type and Predicate Definitions of the Iterator Implementation

```
1   newcoll_α ≡ [new_α(Nil)]
2   size_α(c) ≡[letv p = [!c] in
3                run listcase(p,
4                             Nil → [0],
5                             Cons(_, tl) → [letv n = size_α(tl) in n + 1])]
6   add_α(c, x) ≡ [ letv p = [!c] in
7                   letv c' = new_listcell(α)(p) in
8                   c := Cons(x, c')]
9   remove_α(c) ≡ [letv p = [!c] in
10                  run listcase(p,
11                               Nil → [None],
12                               Cons(x, c') → [letv p' = [!c'] in
13                                              letv _ = [c := p'] in
14                                              Some(x)])]
15  newiter_α(c) ≡ [letv i = [new_colltype(α)(c)] in One(i)]
16  filter_α(p, i) ≡ [Filter(p, i)]
17  merge_α(f, i, i') ≡ [Merge(f, i, i')]
18  next_α(One i) ≡ [letv c = [!i] in
19                   letv p = [!c] in
20                   run listcase(p,
21                                Nil → [None],
22                                Cons(x, c') → [letv _ = [i := c'] in Some(x)])]
23  next_α(Filter(p, i)) ≡ [letv v = next_α(i) in
24                          run listcase(v,
25                                       None → [None],
26                                       Some x → if(p x, [v], next_α(Filter(p, i))))]
27  next_α(Merge(β, γ, f, i_1, i_2)) ≡ [letv x_1 = next_β(i_1) in
28                                      letv x_2 = next_γ(i_2) in
29                                      case(x_1,
30                                           None → None,
31                                           Some v_1 → case(x_2,
32                                                           None → None,
33                                                           Some v_2 → Some(f v_1 v_2)))]
```

Figure 4.4: Implementation of Collections and Iterators

The collection predicate $coll_\tau(c, xs, P)$, defined on line 9, makes use of the list predicate. In addition to asserting that the value c represents the sequence xs, it asserts two further things. First, it says that this program state is also described by the abstract predicate P, and that this predicate is an *exact* predicate.

Exact predicates are predicates that hold of exactly one heap: that is, they are the atomic elements of the lattice of assertions. This means that they uniquely identify a heap data structure. This property lets us track modifications to the collection: any change to the actual heap structure will result in the falsification of P.

As a modal operator on separation logic assertions, exactness can be *defined* in higher-order separation logic with the following definition:

$$\mathrm{Exact}(P) \triangleq \forall Q : \mathsf{prop}.\ (P \wedge Q) \to (P * (\mathsf{emp} \wedge (P \mathrel{-\!\!*} P \wedge Q))$$

This definition has the benefit of concisely demonstrating why we are interested in exactness: when P is exact, we can "subtract" it from any proposition P and Q, leaving a magic wand $P \mathrel{-\!\!*} P \wedge Q$ behind, which is ordinarily not legitimate. (There is possibly a connection to Parkinson's "septraction" operator [47].)

The iterator predicate, defined on line 10 is given as a recursive definition. The base case is when we have an iterator over a single collection, in the $\mathsf{One}(i)$ case. Here, we have the following assertion:

$$iter_\tau(\mathsf{One}(i), xs, \{(\tau, c, ys, P)\}) \equiv \exists c'.\ i \mapsto c' * (coll(c, ys, P) \mathrel{-\!\!*} (coll(c, ys, P) \wedge (\top * list(c', xs))))$$

The $i \mapsto c'$ clause says that i is a pointer to a linked list. The second clause of this invariant is more complex, and its purpose is to say that c is an interior pointer into a collection.

It says if ownership of the collection state $coll(c, ys, P)$ is transferred to us, then we will get it back, along with the additional information that $list(c', xs)$ is within the same state. In this way, we express the fact that the iterator's invariant depends on having controlled access to the state of the collection. However, we are also able to avoid giving the iterator state direct ownership of the collection; the magic wand lets us say that we don't own the collection currently, but rather that if we get it, then we can follow the pointer c when we are given the collection state of the collection.

The reason we go to this effort is to simplify the specification and proof of client programs — we could eliminate the use of the magic wand in the base case if iterators owned their collections, but this would complicate verifying programs that use multiple iterators over the same collection, or which want to call pure methods on the underlying collection. In those cases, the alternative would require us to explicitly transfer ownership of the collection in the proofs of client programs, which is quite cumbersome, and forces clients to reason using the magic wand. The current approach isolates that reasoning within the proof of the implementation.

Also, I should note that exactness plays a role in making this use of the magic wand work correctly. The semantics of the magic wand $p \mathrel{-\!\!*} q$ quantify over all heaps in p, but if p is exact, then there is at most one satisfying heap.

On the next line, we have a catch-all case saying that the predicate is false whenever a One iterator has other than a single collection it depends upon.

One the next line, we give the case for $iter_\tau(\mathsf{Filter}(p, i), xs, S)$. This is a very simple formula; we simply assert that i is an iterator yielding some other sequence ys, which when filtered with the predicate p is xs. There are no changes to the set of abstract states.

On the line after that, we give the case for $iter_\tau(\mathsf{Merge}(\tau_1, \tau_2, f, i_1, i_2), xs, S)$. This case says we can divide the abstract state into two parts (which may overlap), one of which is used by i_1, and the other of which is used by i_2, which yield sequences ys and zs respectively, and which can be merged using f to yield xs.

In both of these cases, we define the behavior of the imperative linked list in terms of purely functional sequences. This is a very common strategy in many verification efforts, but here we see that we can use it in a local way – in the base case, we are forced to consider issues of aliasing and ownership, but in the inductive cases we can largely avoid that effort.

Correctness Proofs of the Iterator Implementation

Now that we know the definitions of the types and predicates, we can give the correctness proofs for the operations defined in Figure 4.4.[3]

Lemma 51. *(Correctness of* newcoll*) We have that*

$$\forall\alpha.\ \langle\mathsf{emp}\rangle\mathsf{newcoll}_\alpha\langle a : \mathsf{colltype}(\alpha).\ \exists P.\ coll_\alpha(a, \epsilon, P)\rangle$$

is valid.

Proof. All $\mathsf{newcoll}_\alpha$ does is allocate a list. To prove the specification, we will assume that $\alpha : \star$, and then prove the program in annotated program style.

1 $\{\mathsf{emp}\}$
2 $\mathsf{new}_\alpha(\mathsf{Nil})$
3 $\{a \mapsto_\alpha \mathsf{Nil}\}$
4 $\{list(\alpha, a, \epsilon)\}$
5 $\{list(\alpha, a, \epsilon) \wedge \exists Q.\ Q \wedge \mathtt{exact}(Q)\}$
6 $\{\exists Q.\ list(\alpha, a, \epsilon) \wedge Q \wedge \mathtt{exact}(Q)\}$
7 $\{\exists Q.\ coll_\alpha(a, \epsilon, Q)\}$

Line 4 follows from 3, because of the definition of the list predicate. Line 5 follows from 4, because this is an axiomatic property of all predicates – if a predicate holds in a heap, there is an exact predicate describing that heap. Line 6 is a quantifier manipulation, and line 7 follows from the definition of the predicate. □

Lemma 52. *(Correctness of* size*) We have that*

$$\forall\alpha, c, P, xs.\ \langle coll_\alpha(c, xs, P)\rangle\mathsf{size}_\alpha(c)\langle a : \mathbb{N}.\ coll_\alpha(c, xs, P) \wedge a = |xs|\rangle$$

is valid.

Proof. This function, defined on line 2 of Figure 4.4, is a recursively defined function. So we will prove it using the fixed point rule, but with the altered specification:

$$\forall\alpha, c, P, Q, xs.\ \langle(P * list(\alpha, c, xs)) \wedge Q\rangle\mathsf{size}_\alpha(c)\langle a : \mathbb{N}.\ (P * list(\alpha, c, xs)) \wedge Q \wedge a = |xs|\rangle$$

[3]These definitions make use of the run abbreviation defined in Section 2.5.

The reason this specification works is that size never modifies its argument, and so will preserve any conjoined invariant. To show this, assume we have α, c, xs, and P, and proceed:

1 $\{(P * \mathit{list}(\alpha, c, xs)) \wedge Q\}$
2 $\{(P * \exists p.\ c \mapsto p \wedge \mathit{listcell}(\alpha, p, xs)) \wedge Q\}$
3 $\mathsf{letv}\ p = [!c]\ \mathsf{in}$
4 $\{(P * c \mapsto p \wedge \mathit{listcell}(\alpha, p, xs)) \wedge Q\}$
5 $\mathsf{run}\ \mathsf{listcase}(p,$
6 $\quad \mathsf{Nil} \rightarrow$
7 $\quad\quad \{(P * c \mapsto p \wedge p = \mathsf{Nil}) \wedge xs = \epsilon \wedge Q\}$
8 $\quad\quad [0]$
9 $\quad\quad \{(P * c \mapsto \mathsf{Nil} \wedge p = \mathsf{Nil}) \wedge xs = \epsilon \wedge a = 0 \wedge Q\}$
10 $\quad\quad \{(P * \mathit{list}(\alpha, c, xs)) \wedge Q \wedge a = |xs|\}$
11 $\quad \mathsf{Cons}(y, c') \rightarrow$
12 $\quad\quad \{(P * c \mapsto \mathsf{Cons}(y, c') * \mathit{list}(\alpha, c', ys)) \wedge xs = y \cdot ys \wedge Q\}$
13 $\quad\quad [\mathsf{letv}\ n = \mathsf{size}_\alpha(c')\ \mathsf{in}$
14 $\quad\quad \{(P * c \mapsto \mathsf{Cons}(y, c') * \mathit{list}(\alpha, c', ys)) \wedge xs = y \cdot ys \wedge Q \wedge n = |ys|\}$
15 $\quad\quad \{(P * \mathit{list}(\alpha, c, xs)) \wedge xs = y \cdot ys \wedge n = |ys| \wedge Q\}$
16 $\quad\quad n + 1]$
17 $\quad\quad \{(P * \mathit{list}(\alpha, c, xs)) \wedge xs = y \cdot ys \wedge a = |ys| + 1 \wedge Q\}$
18 $\quad\quad \{(P * \mathit{list}(\alpha, c, xs)) \wedge xs = y \cdot ys \wedge a = |y \cdot ys| \wedge Q\}$
19 $\quad\quad \{(P * \mathit{list}(\alpha, c, xs)) \wedge Q \wedge a = |xs|\}$
20 $\{(P * \mathit{list}(\alpha, c, xs)) \wedge Q \wedge a = |xs|\}$

Now, we can specialize this proof to get the specification we originally sought:

1 $\{\mathit{coll}_\alpha(c, xs, P)\}$
2 $\{\mathit{list}(\alpha, c, xs) \wedge P \wedge \mathtt{exact}(P)\}$
3 $\{(\mathsf{emp} * \mathit{list}(\alpha, c, xs)) \wedge P \wedge \mathtt{exact}(P)\}$
4 $\mathsf{size}_\alpha(c)$
5 $\{a : \mathbb{N}.\ (\mathsf{emp} * \mathit{list}(\alpha, c, xs)) \wedge P \wedge \mathtt{exact}(P) \wedge a = |xs|\}$
6 $\{a : \mathbb{N}.\ \mathit{coll}_\alpha(c, xs, P) \wedge a = |xs|\}$
□

Lemma 53. *(Specification of* add*) We have that for*

$$\forall \alpha, c, P, x, xs.\ \langle \mathit{coll}_\alpha(c, xs, P)\rangle \mathsf{add}_\alpha(c, x) \langle a : \mathbf{1}.\ \exists Q.\ \mathit{coll}_\alpha(c, x \cdot xs, Q)\rangle$$

Proof. This function, defined on line 6, just conses on an element. Assume we have α, x, c, xs, P, and then proceed with the following proof, in annotated specification style.

1 $\{\mathit{coll}_\alpha(c, xs, P)\}$
2 $\{\mathit{list}(\alpha, c, xs) \wedge P \wedge \mathtt{exact}(P)\}$
3 $\{\mathit{list}(\alpha, c, xs)\}$
4 $\{\exists p.\ c \mapsto p * \mathit{listcell}(\alpha, p, xs)\}$
5 $\mathsf{letv}\ p = [!c]\ \mathsf{in}$
6 $\{c \mapsto p * \mathit{listcell}(\alpha, p, xs)\}$

7 letv $c' = [\mathsf{new}_{\mathsf{listcell}(\alpha)}(p)]$ in
8 $\{c \mapsto p * c' \mapsto p * \mathit{listcell}(\alpha, p, xs)\}$
9 $\{c \mapsto p * \mathit{list}(\alpha, c', xs)\}$
10 $c := \mathsf{Cons}(x, c')$
11 $\{c \mapsto \mathsf{Cons}(x, c') * \mathit{list}(\alpha, c', xs)\}$
12 $\{\mathit{list}(\alpha, c, x \cdot xs)\}$
13 $\{\exists Q.\ \mathit{list}(\alpha, c, x \cdot xs) \wedge Q \wedge \mathtt{exact}(Q)\}$
14 $\{\exists Q.\ \mathit{coll}_\alpha(c, x \cdot xs, Q)\}$

□

Lemma 54. *(Correctness of* remove*, part 1) We have that*

$$\forall \alpha, c, P.\ \langle \mathit{coll}_\alpha(c, \epsilon, P)\rangle \mathsf{remove}_\alpha(c) \langle a : \mathsf{option}(\alpha).\ \mathit{coll}_\alpha(c, \epsilon, P) \wedge a = \mathsf{None}\rangle$$

Proof. This is one of the two specifications about remove, for the case when the list is empty. Assume we have α, c, P and give an annotated specification as follows:

1 $\{\mathit{coll}_\alpha(c, \epsilon, P)\}$
2 $\{\exists p.\ c \mapsto p \wedge p = \mathsf{Nil} \wedge P \wedge \mathtt{exact}(P)\}$
3 letv $p = [!c]$ in
4 $\{c \mapsto p \wedge p = \mathsf{Nil} \wedge P \wedge \mathtt{exact}(P)\}$
5 run listcase(p,
6 Nil $\rightarrow$[None]
7 $\{c \mapsto p \wedge p = \mathsf{Nil} \wedge P \wedge \mathtt{exact}(P) \wedge a = \mathsf{None}\}$
8 $\{\mathit{coll}_\alpha(c, \epsilon, P) \wedge a = \mathsf{None}\}$
9 $\mathsf{Cons}(y, c') \rightarrow$
10 $\{c \mapsto p \wedge p = \mathsf{Nil} \wedge p = \mathsf{Cons}(y, c') \wedge P \wedge \mathtt{exact}(P)\}$
11 $\{\bot\}$
12 [letv $p' = [!c']$ in
13 letv $\_ = [c := p']$ in
14 Some(y)])
15 $\{\mathit{coll}_\alpha(c, \epsilon, P) \wedge a = \mathsf{None}\}$
16 $\{\mathit{coll}_\alpha(c, \epsilon, P) \wedge a = \mathsf{None}\}$

□

Lemma 55. *(Correctness of* remove*, part 2) We have that*

$$\forall \alpha, c, x, xs, P.\ \langle \mathit{coll}_\alpha(c, x \cdot xs, P)\rangle \mathsf{remove}_\alpha(c) \langle a : \mathsf{option}(\alpha).\ \exists Q.\ \mathit{coll}_\alpha(c, xs, Q) \wedge a = \mathsf{Some}(x)\rangle$$

Proof. This is the other case of remove, for when the iterator is not yet exhausted. Assume that we have α, c, x, xs, and P of the appropriate type. We can give an annotated-specification style proof as follows:

1 $\{\mathit{coll}_\alpha(c, x \cdot xs, P)\}$
2 $\{\exists p.\ c \mapsto p \wedge \exists c'.\ p = \mathsf{Cons}(x, c') * \mathit{list}(\alpha, c', xs) \wedge P \wedge \mathtt{exact}(P)\}$
3 letv $p = [!c]$ in
4 $\{c \mapsto p \wedge \exists c'.\ p = \mathsf{Cons}(x, c') * \mathit{list}(\alpha, c', xs) \wedge P \wedge \mathtt{exact}(P)\}$

```
5   run listcase(p,
6                Nil →{p = Nil ∧ c ↦ p ∧ ∃c'. p = Cons(x, c') * list(α, c', xs) ∧ P ∧ exact(P)}
7                     {⊥}
8                     [None]
9                     {∃Q. coll_α(c, xs, Q) ∧ a = Some(x)}
10               Cons(y, c') →
11                    {p = Cons(y, c') ∧ c ↦ p ∧ ∃c'. p = Cons(x, c') * list(α, c', xs) ∧ P ∧ exact(P)}
12                    {x = y ∧ c ↦ p ∧ p = Cons(x, c') * list(α, c', xs) ∧ P ∧ exact(P)}
13                    {x = y ∧ c ↦ p ∧ p = Cons(x, c') * list(α, c', xs)}
14                    {x = y ∧ c ↦ p ∧ p = Cons(x, c') * ∃p'. c' ↦ p' * listcell(α, p', xs)}
15                    [letv p' = [!c'] in
16                    {x = y ∧ c ↦ p ∧ p = Cons(x, c') * c' ↦ p' * listcell(α, p', xs)}
17                    letv _ = [c := p'] in
18                    {x = y ∧ c ↦ p' * c' ↦ p' * listcell(α, p', xs)}
19                    {x = y ∧ c' ↦ p' * list(α, c, xs)}
20                    {x = y ∧ list(α, c, xs)}
21                    Some(y)])
22                    {list(α, c, xs) ∧ a = Some(y) ∧ x = y}
23                    {list(α, c, xs) ∧ a = Some(x)}
24                    {(∃Q. list(α, c, xs) ∧ Q ∧ exact(Q)) ∧ a = Some(x)}
25                    {∃Q. coll_α(c, xs, Q) ∧ a = Some(x)}
26  {∃Q. coll_α(c, xs, Q) ∧ a = Some(x)}
```

□

Lemma 56. *(Correctness of* newiter*) We have that*

$$\forall \alpha, c, P, xs.\ \langle coll_\alpha(c, xs, P)\rangle \mathsf{newiter}_\alpha(c)\langle a : \mathsf{itertype}(\alpha).\ coll_\alpha(c, xs, P) * iter_\alpha(a, xs, \{\alpha, c, xs, P\})\rangle$$

is valid.

Proof. The definition of newiter is given on line 15 of Figure 4.4. To prove the correctness of this implementation, we assume we have α, c, xs, and P, and then give the following proof:

```
1   {coll_α(c, xs, P)}
2   {list(c, xs, P) ∧ P ∧ exact(P)}
3   {list(c, xs, P) ∧ P ∧ exact(P) ∧ exact(list(c, xs, P) ∧ P ∧ exact(P))}
4   {coll(c, xs, P) ∧ exact(coll(c, xs, P))}
5   letv i = [new_colltype(α)(c)] in
6   {i ↦ c * coll(c, xs, P) ∧ exact(coll(c, xs, P))}
7   {i ↦ c * ((⊤ * coll(c, xs, P)) ∧ coll(c, xs, P)) ∧ exact(coll(c, xs, P))}
8   {i ↦ c * (coll(c, xs, P) −* (⊤ * coll(c, xs, P)) ∧ coll(c, xs, P)) * coll(c, xs, P)}
9   One(i)
10  {a. ∃i. a = One(i) ∧ iter(One(i), xs, {(α, c, xs, P)})}
11  {a. iter(a, xs, {(α, c, xs, P)})}
```

The key idea in this proof is that if P is exact, then $P \wedge Q$ is also exact, no matter what Q is. This lets us use the exactness of the abstract state to "subtract" the predicate $coll(c, xs, P)$ and keep the iterator and collection state separate.

□

Lemma 57. *(Correctness of* filter*) We have that*

$$\forall \alpha, p, i, S, xs.\ \langle iter_\alpha(i, xs, S)\rangle \mathsf{filter}_\alpha(p, i) \langle a : \mathsf{itertype}(\alpha).\ iter_\alpha(a, \mathit{filter}\ p\ xs, S)\rangle$$

Proof. The definition of this procedure is given on line 16. Assume α, p, i, xs, and S.

1 $\{iter_\alpha(i, xs, S)\}$
2 $\mathsf{Filter}(p, i)$
3 $\{iter_\alpha(i, xs, S) \wedge a = \mathsf{Filter}(p, i)\}$
4 $\{iter_\alpha(i, xs, S) \wedge \mathit{filter}\ p\ xs = \mathit{filter}\ p\ xs \wedge a = \mathsf{Filter}(p, i)\}$
5 $\{\exists i, ys.\ iter_\alpha(i, ys, S) \wedge \mathit{filter}\ p\ xs = \mathit{filter}\ p\ ys \wedge a = \mathsf{Filter}(p, i)\}$
6 $\{iter_\alpha(a, \mathit{filter}\ p\ xs, S)\}$

□

Lemma 58. *(Correctness of* merge*) We have that*

$$\forall \alpha, \beta, \gamma, f, i, S, xs, i', S', xs'.\ \begin{array}{l} \{iter_\alpha(i, xs, S) * iter_\beta(i', xs', S')\} \\ \mathsf{merge}_{\alpha\ \beta\ \gamma}(f, i, i') \\ \{a : \mathsf{itertype}(\gamma).\ iter_\gamma(a, map2\ f\ xs\ xs', S \cup S')\} \end{array}$$

Proof. The definition of merge is given on line 17 of Figure 4.4. Assume f, i, xs, S, i', xs', S' as hypotheses, and proceed with the proof as follows:

1 $\{iter_\alpha(i, xs, S) * iter_\beta(i', xs', S')\}$
2 $\mathsf{Merge}(\alpha, \beta, f, i, i')$
3 $\{iter_\alpha(i, xs, S) * iter_\beta(i', xs', S') \wedge a = \mathsf{Merge}(\alpha, \beta, f, i, i')\}$
4 $\{iter_\alpha(i, xs, S) * iter_\beta(i', xs', S') \wedge map2\ f\ xs\ xs' = map2\ f\ xs\ xs'$
5 $\quad \wedge\ a = \mathsf{Merge}(f, i, i')\}$
6 $\{\exists i, i', S_1, S_2, xs_1, xs_2.\ iter_\alpha(i, xs_1, S_1) * iter_\beta(i', xs_2, S_2) \wedge S \cup S' = S_1 \cup S_2 \wedge$
7 $\quad map2\ f\ xs\ xs' = map2\ f\ xs_1\ xs_2 \wedge a = \mathsf{Merge}(f, i, i')\}$
8 $\{\exists i, i'.\ iter_\gamma(\mathsf{Merge}(f, i, i'), map2\ f\ xs\ xs', S \cup S') \wedge a = Merge(f, i, i')\}$
9 $\{iter_\alpha(a, map2\ f\ xs\ xs', S \cup S')\}$

□

Lemma 59. *(Correctness of* next*) We have that*

$$\forall \alpha, i, S, xs.\ \begin{array}{l} \{iter_\alpha(i, xs, S) * colls(S)\} \\ \mathsf{next}_\alpha(i) \\ \{a : \mathsf{option}(\alpha). \\ \quad [(a = \mathsf{None} \wedge xs = \epsilon \wedge iter_\alpha(i, xs, S)) \vee \\ \quad (\exists y, ys.\ a = \mathsf{Some}(y) \wedge xs = y \cdot ys \wedge iter_\alpha(i, ys, S))] \\ \quad * colls(S)\} \end{array}$$

Proof. This function is defined on lines 18 to 33 of Figure 4.4. To prove this, we will use fixed point induction, and assume that the specification above holds for an identifier named next, and use it to prove the correctness for the body of the function.

Assuming α, i, S, xs, this proof will proceed by cases, on the structure of i. Then we can exploit the fact that we can unroll fixed points and beta-reduce expression to avoid proving the impossible branches (which in typical Hoare logic proofs are ruled out by getting false as a precondition).

- Suppose $i = \mathsf{One}(i')$. Then, we proceed with an annotated proof as follows.

1 $\{iter_\alpha(\mathsf{One}(i'), xs, S) * colls(S)\}$
2 When S has other than one element, the precondition is false, so we only need to
3 consider the one-element case:
4 $\{\exists c, ys, P.\ S = \{(c, ys, P)\} \wedge iter_\alpha(\mathsf{One}(i'), xs, \{(\alpha, c, ys, P)\}) * colls_\alpha(\{(\alpha, c, ys, P)\})\}$
5 We will use the existential rule and then frame off the definition of S:
6 $\{iter(\mathsf{One}(i'), xs, \{(c, ys, P)\}) * coll(c, ys, P)\}$
7 $\{\exists c'.\ i' \mapsto c' * (\top * list(c', xs)) \wedge coll(c, ys, P)\}$
8 letv $c' = [!i']$ in
9 $\{i' \mapsto c' * (\top * list(c', xs)) \wedge coll(c, ys, P)\}$
10 $\{\exists p.\ i' \mapsto c' * (\top * c' \mapsto p * listcell(p, xs)) \wedge coll(c, ys, P)\}$
11 letv $p = [!c']$ in
12 $\{i' \mapsto c' * (\top * c' \mapsto p * listcell(p, xs)) \wedge coll(c, ys, P)\}$
13 run listcase(p,
14 Nil$\rightarrow$
15 $\{p = \mathsf{Nil} \wedge i' \mapsto c' * (\top * c' \mapsto p * listcell(p, xs)) \wedge coll(c, ys, P)\}$
16 $\{xs = \epsilon \wedge i' \mapsto c' * (\top * c' \mapsto p * listcell(p, xs)) \wedge coll(c, ys, P)\}$
17 $\{xs = \epsilon \wedge iter(i, xs, \{(c, ys, P)\}) * coll(c, ys, P)\}$
18 Restoring the definition of S:
19 $\{xs = \epsilon \wedge iter(i, xs, S) * colls(S)\}$
20 [None]
21 $\{a.\ a = \mathsf{None} \wedge xs = \epsilon \wedge iter(i, xs, S) * colls(S)\}$
22 Cons$(z, c'') \rightarrow$
23 $\{p = \mathsf{Cons}(z, c'') \wedge i' \mapsto c' * (\top * c' \mapsto p * listcell(p, xs)) \wedge coll(c, ys, P)\}$
24 $\{\exists zs.\ xs = z \cdot zs \wedge p = \mathsf{Cons}(z, c'') \wedge i' \mapsto c' *$
25 $(\top * c' \mapsto \mathsf{Cons}(z, c'') * list(c'', zs)) \wedge coll(c, ys, P)\}$
26 $\{\exists zs.\ xs = z \cdot zs \wedge i' \mapsto c' \ * (\top * list(c'', zs)) \wedge coll(c, ys, P)\}$
27 [letv $\_ = i' := c''$ in
28 $\{\exists zs.\ xs = z \cdot zs \wedge i' \mapsto c'' \ * (\top * list(c'', zs)) \wedge coll(c, ys, P)\}$
29 $\{\exists zs.\ xs = z \cdot zs \wedge iter(\mathsf{One}(i'), zs, \{(c, ys, P)\}) * coll(c, ys, P)\}$
30 Restoring the definition of S:
31 $\{\exists zs.\ xs = z \cdot zs \wedge iter(\mathsf{One}(i'), zs, S) * colls(S)\}$
32 Some(z)])
33 $\{\exists z, zs.\ a = \mathsf{Some}(z) \wedge xs = z \cdot zs \wedge iter(i, zs, S) * colls(S)\}$
34 $\{a.\ ((a = \mathsf{None} \wedge xs = \epsilon \wedge iter(i, xs, S)) \vee$
35 $(\exists z, zs.\ a = \mathsf{Some}(z) \wedge xs = z \cdot zs \wedge iter(i, zs, S))) *$
36 $colls(S)\}$

- Suppose $i = \mathsf{Filter}(p, i')$. Then, we proceed with an annotated proof as follows:

$\{iter_\alpha(\mathsf{Filter}(p, i'), xs, S) * colls(S)\}$
$\{\exists ys.(iter_\alpha(i', ys, S) \wedge xs = filter\ p\ ys) * colls(S)\}$
$\{xs = filter\ p\ ys \wedge iter_\alpha(i', ys, S) * colls(S)\}$
Since it is pure, we can pull $xs = filter\ p\ ys$ out of the precondition.
That is, using the axiom AXEXTRACT we can pull pure consequences
of a precondition into the ambient specification context, and then
restore them whenever we need using the AXEMBED axiom.
$\{iter_\alpha(i', ys, S) * colls(S)\}$
$[\mathsf{letv}\ v = \mathsf{next}_\alpha(i')\ \mathsf{in}$
$\{[(ys = \epsilon \wedge v = \mathsf{None} \wedge iter_\alpha(i', ys, S)) \vee (\exists z, zs.\ ys = z \cdot zs \wedge v = \mathsf{Some}(z) \wedge iter_\alpha(i', zs, S))]$
$*colls(S)\}$
$\mathsf{run\ listcase}(v,$
$\mathsf{None} \rightarrow$
$\{[(ys = \epsilon \wedge v = \mathsf{None} \wedge iter_\alpha(i', ys, S)) \vee$
$(\exists z, zs.\ ys = z \cdot zs \wedge v = \mathsf{Some}(z) \wedge iter_\alpha(i', zs, S))] * colls(S) \wedge v = \mathsf{None}\}$
$\{ys = \epsilon \wedge iter_\alpha(i', ys, S) * colls(S)\}$
Since $xs = filter\ p\ ys = \epsilon$, we know
$\{xs = \epsilon \wedge iter_\alpha(i, xs, S) * colls(S)\}$
$[\mathsf{None}]$
$\{a.\ a = \mathsf{None} \wedge xs = \epsilon \wedge iter_\alpha(i, xs, S) * colls(S)\}$
$\mathsf{Some}(z) \rightarrow$
$\{[(ys = \epsilon \wedge v = \mathsf{None} \wedge iter_\alpha(i', ys, S)) \vee$
$(\exists z, zs.\ ys = z \cdot zs \wedge v = \mathsf{Some}(z) \wedge iter_\alpha(i', zs, S))] * colls(S) \wedge v = \mathsf{Some}(z)\}$
$\{\exists zs.\ ys = z \cdot zs \wedge v = \mathsf{Some}(z) \wedge iter_\alpha(i', zs, S) * colls(S)\}$
$\{ys = z \cdot zs \wedge v = \mathsf{Some}(z) \wedge iter_\alpha(i', zs, S) * colls(S)\}$
$\mathsf{run\ if}(p\ z,$
$\{p\ z = \mathsf{true} \wedge ys = z \cdot zs \wedge iter_\alpha(i', zs, S) * colls(S) \wedge v = \mathsf{Some}(z)\}$
Hence $xs = filter\ p\ ys = z \cdot (filter\ p\ zs)$. So
$\{xs = z \cdot (filter\ p\ zs) \wedge iter_\alpha(i', zs, S) * colls(S) \wedge v = Some(z)\}$
$\{\exists xs'.\ xs = z \cdot xs' \wedge iter_\alpha(\mathsf{Filter}(p, i'), xs', S) * colls(S) \wedge v = \mathsf{Some}(z)\}$
$\{\exists xs'.\ xs = z \cdot xs' \wedge iter_\alpha(i, xs', S) * colls(S) \wedge v = \mathsf{Some}(z)\}$
$[v],$
$\{a.\ \exists z, zs.\ xs = z \cdot zs \wedge a = \mathsf{Some}(z) \wedge iter_\alpha(i, zs, S) * colls(S)\}$
$\{a.\ ((a = \mathsf{None} \wedge xs = \epsilon \wedge iter(i, xs, S)) \vee$
$(\exists z, zs.\ a = \mathsf{Some}(z) \wedge xs = z \cdot zs \wedge iter(i, zs, S))) *$
$colls(S)\}$
Now, the else-branch:
$\{p\ z = \mathsf{false} \wedge ys = z \cdot zs \wedge iter_\alpha(i', zs, S) * colls(S)\}$
Hence $xs = filter\ p\ ys = filter\ p\ zs$. So
$\{xs = filter\ p\ zs \wedge iter_\alpha(i', zs, S) * colls(S)\}$
$\{iter_\alpha(\mathsf{Filter}(p, i'), xs, S) * colls(S)\}$
$\{iter_\alpha(i, xs, S) * colls(S)\}$
$\mathsf{next}_\alpha(i))))]$

$\{a.\ ((a = \mathsf{None} \wedge xs = \epsilon \wedge iter(i, xs, S)) \vee$
$(\exists z, zs.\ a = \mathsf{Some}(z) \wedge xs = z \cdot zs \wedge iter(i, zs, S))) *$
$colls(S)\}$
$\{a.\ ((a = \mathsf{None} \wedge xs = \epsilon \wedge iter(i, xs, S)) \vee$
$(\exists z, zs.\ a = \mathsf{Some}(z) \wedge xs = z \cdot zs \wedge iter(i, zs, S))) *$
$colls(S)\}$

Note that this is not a structural induction on i; we make a non-structural recursive call when the test of p fails. Here, we make use of fixed-point induction in our proof of next.

- Suppose $i = \mathsf{Merge}(f, i_1, i_2)$. Then, we proceed with an annotated proof as follows:

$\{iter_\alpha(\mathsf{Merge}(f, i_1, i_2), xs, S) * colls(S)\}$
$\{\exists \beta, \gamma, S_1, ys, S_2, zs.\ xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$iter_\beta(i_1, ys, S_1) * iter_\gamma(i_2, zs, S_2) * colls(S)\}$
$\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$iter_\beta(i_1, ys, S_1) * iter_\gamma(i_2, zs, S_2) * colls(S)\}$
$\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$iter_\beta(i_1, ys, S_1) * iter_\gamma(i_2, zs, S_2) * colls(S_1) * colls(S - S_1)\}$
$\mathsf{letv}\ v_1 = \mathsf{next}_\beta(i_1)\ \mathsf{in}$
$\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$[(ys = \epsilon \wedge v_1 = \mathsf{None} \wedge iter_\beta(i_1, ys, S_1))$
$\vee (\exists b, bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\alpha(i_1, bs, S_1))] *$
$iter_\gamma(i_2, zs, S_2) * colls(S_1) * colls(S - S_1)\}$
$\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$[(ys = \epsilon \wedge v_1 = \mathsf{None} \wedge iter_\beta(i_1, ys, S_1))$
$\vee (\exists b, bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\alpha(i_1, bs, S_1))] *$
$iter_\gamma(i_2, zs, S_2) * colls(S_2) * colls(S - S_2)\}$
$\mathsf{letv}\ v_2 = \mathsf{next}_\gamma(i_2)\ \mathsf{in}$
$\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$[(ys = \epsilon \wedge v_1 = \mathsf{None} \wedge iter_\beta(i_1, ys, S_1))$
$\vee (\exists b, bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\beta(i_1, bs, S_1))] *$
$[(zs = \epsilon \wedge v_2 = \mathsf{None} \wedge iter_\gamma(i_2, zs, S_2))$
$\vee (\exists c, cs.\ zs = c \cdot cs \wedge v_2 = \mathsf{Some}(c) \wedge iter_\gamma(i_2, cs, S_2))] *$
$colls(S_2) * colls(S - S_2)\}$
$\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$[(ys = \epsilon \wedge v_1 = \mathsf{None} \wedge iter_\beta(i_1, ys, S_1))$
$\vee (\exists b, bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\beta(i_1, bs, S_1))] *$
$[(zs = \epsilon \wedge v_2 = \mathsf{None} \wedge iter_\gamma(i_2, zs, S_2))$
$\vee (\exists c, cs.\ zs = c \cdot cs \wedge v_2 = \mathsf{Some}(c) \wedge iter_\gamma(i_2, cs, S_2))] *$
$colls(S)\}$
$\mathsf{case}(v_1$
$\mathsf{None} \rightarrow$
$\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$(ys = \epsilon \wedge v_1 = \mathsf{None} \wedge iter_\beta(i_1, ys, S_1)) *$

$[(zs = \epsilon \wedge v_2 = \mathsf{None} \wedge iter_\gamma(i_2, zs, S_2))$
$\vee (\exists c, cs.\ zs = c \cdot cs \wedge v_2 = \mathsf{Some}(c) \wedge iter_\gamma(i_2, cs, S_2))] *$
$colls(S)\}$

14 $\{xs = \epsilon \wedge xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$(ys = \epsilon \wedge v_1 = \mathsf{None} \wedge iter_\beta(i_1, ys, S_1)) *$
$[(zs = \epsilon \wedge v_2 = \mathsf{None} \wedge iter_\gamma(i_2, zs, S_2))$
$\vee (\exists c, cs.\ zs = c \cdot cs \wedge v_2 = \mathsf{Some}(c) \wedge iter_\gamma(i_2, cs, S_2))] *$
$colls(S)\}$

The next line follows because it holds in either branch of the disjunction

15 $\{xs = \epsilon \wedge iter_\alpha(\mathsf{Merge}(f, i_1, i_2), xs, S) *\ colls(S)\}$

16 None

17 $\{a = \mathsf{None} \wedge xs = \epsilon \wedge iter_\alpha(\mathsf{Merge}(f, i_1, i_2), xs, S) * colls(S)\}$

18 $\mathsf{Some}(b) \rightarrow$

19 $\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$(\exists bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\beta(i_1, bs, S_1)) *$
$[(zs = \epsilon \wedge v_2 = \mathsf{None} \wedge iter_\gamma(i_2, zs, S_2))$
$\vee (\exists c, cs.\ zs = c \cdot cs \wedge v_2 = \mathsf{Some}(c) \wedge iter_\gamma(i_2, cs, S_2))] *$
$colls(S)\}$

20 $\mathsf{case}(v_2,$

21 $\mathsf{None} \rightarrow$

22 $\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$(\exists bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\beta(i_1, bs, S_1)) *$
$(zs = \epsilon \wedge v_2 = \mathsf{None} \wedge iter_\gamma(i_2, zs, S_2)) *$
$colls(S)\}$

23 $\{xs = \epsilon \wedge xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge zs = \epsilon \wedge$
$(\exists bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\beta(i_1, bs, S_1)) *$
$(zs = \epsilon \wedge v_2 = \mathsf{None} \wedge iter_\gamma(i_2, zs, S_2))$
$colls(S)\}$

24 $\{xs = \epsilon \wedge iter_\alpha(\mathsf{Merge}(f, i_1, i_2), xs, S) * colls(S)\}$

25 None,

26 $\{a = \mathsf{None} \wedge xs = \epsilon \wedge iter_\alpha(\mathsf{Merge}(f, i_1, i_2), xs, S) * colls(S)\}$

27 $\mathsf{Some}(c) \rightarrow$

28 $\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$(\exists bs.\ ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\beta(i_1, bs, S_1)) *$
$(\exists cs.\ zs = c \cdot cs \wedge v_2 = \mathsf{Some}(c) \wedge iter_\gamma(i_2, cs, S_2)) *$
$colls(S)\}$

29 $\{xs = map2\ f\ ys\ zs \wedge S = S_1 \cup S_2 \wedge$
$(ys = b \cdot bs \wedge v_1 = \mathsf{Some}(b) \wedge iter_\beta(i_1, bs, S_1)) *$
$(zs = c \cdot cs \wedge v_2 = \mathsf{Some}(c) \wedge iter_\gamma(i_2, cs, S_2)) *$
$colls(S)\}$

30 $\{xs = (f\ b\ c) \cdot map2\ f\ bs\ cs \wedge S = S_1 \cup S_2 \wedge$
$iter_\beta(i_1, bs, S_1)) * iter_\gamma(i_2, cs, S_2)) *$
$colls(S)\}$

31 $\{xs = (f\ b\ c) \cdot map2\ f\ bs\ cs \wedge$

$iter_\alpha(\mathsf{Merge}(f, i_1, i_2), map2\ f\ bs\ cs, S) *$
$colls(S)\}$

32 $\mathsf{Some}(f\ b\ c)))$

33 $\{a = \mathsf{Some}(f\ b\ c) \wedge xs = (f\ b\ c) \cdot map2\ f\ bs\ cs \wedge$
$iter_\alpha(\mathsf{Merge}(f, i_1, i_2), map2\ f\ bs\ cs, S) *$
$colls(S)\}$

34 $\{\exists v, vs.\ a = \mathsf{Some}(v) \wedge xs = v \cdot vs \wedge$
$iter_\alpha(\mathsf{Merge}(f, i_1, i_2), vs, S) *$
$colls(S)\}$

□

next (lines 18-33 of Figure 4.4) recursively walks down the structure of the iterator tree, and combines the results from the leaves upwards. The base case is the $\mathsf{One}(i)$ case (lines 18-22). The iterator pointer is doubly-dereferenced, and then the contents examined. If the end of the list has been reached and the contents are Nil, then None is returned to indicate there are no more elements. Otherwise, the pointer r is advanced, and the head returned as the observed value. The $\mathsf{Filter}(\mathsf{p}, \mathsf{i})$ case (lines 23-26) will return None if i is exhausted, and if it is not, it will pull elements from i until it finds one that satisfies p, calling itself recursively until it succeeds or i is exhausted. Finally, in the $\mathsf{Map2}(f, i_1, i_2)$ case (lines 27-33), next will draw a value from both i_1 and i_2, and will return None if either is exhausted, and otherwise it will return f applied to the pair of values.

4.3 The Flyweight and Factory Patterns

The flyweight pattern is a style of cached object creation. Whenever a constructor method is called, it first consults a table to see if an object corresponding to those arguments has been created. If it has, then the preexisting object is returned. Otherwise, it allocates a new object, and updates the table to ensure that future calls with the same arguments will return this object. Because objects are re-used, they become pervasively aliased, and must be used in an immutable style to avoid surprising updates. (Functional programmers call this style of value creation "hash-consing".)

This is an interesting design pattern to verify, for two reasons. First, the constructor has a memo table to cache the result of constructor calls, which needs to be hidden from clients. Second, this pattern makes pervasive use of aliasing, in a programmer-visible way. In particular, programmers can test two references for identity in order to establish whether two values are equal or not. This allows constant-time equality testing, and is a common reason for using this pattern. Therefore, our specification has to be able to justify this reasoning.

In Figure 4.5, we give a specification which uses the flyweight pattern to create and access glyphs (i.e., refs of pairs of characters and fonts) of a particular font f. We have a function newglyph to create new glyphs, which does the caching described above, using a predicate variable I to refer to the table invariant; and a function getdata to get the character and font information from a glyph.

Furthermore, these functions will be created by a call to another function, make_flyweight, which receives a font as an argument and will return appropriate newglyph and getdata functions.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{Flyweight}(I : (\mathsf{char} \rightharpoonup \mathsf{glyph}) \to \mathsf{prop},} \\
& \quad \mathsf{newglyph} : \mathsf{char} \to \bigcirc\mathsf{glyph}, \\
& \quad \mathsf{getdata} : \mathsf{glyph} \to \bigcirc(\mathsf{char} \times \mathsf{font}), \\
& \quad f : \mathsf{font}) \equiv \\
1 & \quad \forall c, h.\ \langle I(h) \rangle \\
& \qquad \mathsf{newglyph}(c) \\
& \qquad \langle a : \mathsf{glyph}.\ I([h|c : a]) \wedge (c \in \mathrm{dom}(h) \supset a = h(c)) \rangle \\
& \quad \& \\
2 & \quad \forall l, h, c.\ \langle I(h) \wedge l = h(c) \rangle \\
& \qquad \mathsf{getdata}(l) \\
& \qquad \langle a : \mathsf{char} \times \mathsf{font}.\ I(h) \wedge a = (c, f) \rangle
\end{array}
$$

where $\mathsf{glyph} \triangleq \mathsf{ref}\ (\mathsf{char} \times \mathsf{font})$

Figure 4.5: Flyweight Specification

$$
\begin{array}{ll}
1 & \exists \mathsf{make\_flyweight} : \mathsf{font} \to \bigcirc((\mathsf{char} \to \bigcirc\mathsf{glyph}) \times \\
& \qquad\qquad\qquad\qquad (\mathsf{glyph} \to \bigcirc(\mathsf{char} \times \mathsf{font}))). \\
2 & \quad \forall f.\ \{\mathsf{emp}\} \\
& \qquad \mathsf{run\ make\_flyweight}(f) \\
& \qquad \{a.\ \exists I.\ I([]) \wedge \mathsf{Flyweight}(I, \mathsf{fst}\ a, \mathsf{snd}\ a, f)\ \mathsf{spec}\}
\end{array}
$$

Figure 4.6: Flyweight Factory Specification

In the opening, we informally parametrize our specification over the predicate variable I, the function variable newglyph, the function variable getdata, and the variable f of font type. The reason we do this instead of existentially quantifying over them will become clear shortly, once we see the factory function that creates flyweight constructors.

On line 1, we specify the effect of a call $\mathsf{newglyph}(c)$ procedure. Its precondition is the predicate $I(h)$, which asserts that the abstract state contains glyphs for all of the characters in the domain of the partial function h. Its postcondition changes to a state $I([h|c : a])$, which is a partial function extended to include the character c, with the additional condition that if c was already in h's domain, the same glyph value is returned.

The intuition for this specification is that $I(h)$ abstractly represents a memo table (i.e., the function h), which characterizes all the glyphs allocated so far.

On line 2, we specify the getdata function. This just says that if we call $\mathsf{getdata}(l)$ on a glyph l, then we are returned the data (the alphabetic character and font) for this glyph.

The specification of the flyweight factory is given in Figure 4.6. Here, we assert the existence of a function make_flyweight, which takes a font f as an input argument, and returns two functions to serve as the getchar and getdata functions of the flyweight. In the postcondition, we assert the existence of some private state I, which contains the table used to cache glyph creations.

This pattern commonly arises when encoding aggressively object-oriented designs in a

```
1   make_flyweight ≡
2     λf :font.
3     [letv t = newtable() in
4      letv newglyph =
5          [λc.[letv x = lookup(t, c) in
6               run case(x,None → [letv r = [new_{char×font}(c, f)] in
7                                  letv_ = update(t, c, r) in r],
8                        Some(r) → [r])]] in
9      letv getdata = [λr. [!r]] in
10     (newglyph, getdata)]]

11  J : font × table × (char ⇀ glyph) → prop
12  J(f, t, h) ≡ table(t, h) ∗ refs(f, h)

13  refs(f, h) = ∀*c ∈ dom(h). h(c) ↦ (c, f)

14  ∀*x ∈ ∅. P(x) ≜ emp
15  ∀*x ∈ {y} ⊎ Y. P(x) ≜ P(y) ∗ ∀*x ∈ Y. P(x)
```

Figure 4.7: Flyweight Implementation

higher-order style — we call a function, which creates a hidden state, and returns other procedures which are the only way to access that state. This style of specification resembles the existential encodings of objects into type theory. The difference is that instead of making the fields of an object an existentially quantified *value* [38], we make use of existentially-quantified *state*.

In Figure 4.7, we define make_flyweight and its predicates. In this implementation we have assumed the existence of a hash table implementation with operations newtable, lookup, and update, whose specifications we give in Figure 4.8. The make_flyweight function definition takes a font argument f, and then in its body it creates a new table t. It then constructs two functions as closures which capture this state (and the argument f) and operate on it. In lines 4-8, we define $newglyph$, which takes a character and checks to see (line 6) if it is already in the table. If it is not (lines 6-7), it allocates a new glyph reference, stores it in the table, and returns the reference. Otherwise (line 8), it returns the existing reference from the table. On line 9, we define $getdata$, which dereferences its pointer argument and returns the result. This implementation does no writes, fulfilling the promise made in the specification. The definition of the invariant state I describes the state of the table t (as the partial function h), which are hidden from clients.

Observe how the post-condition to make_flyweight nests the existential state I within the validity assertion to specialize the flyweight spec to the *dynamically* created table. Each created flyweight factory receives its own private state, and we can reuse specifications and proofs with no possibility that the wrong getdata will be called on the wrong reference, even though they have compatible types.

1 $\langle \mathsf{emp} \rangle \mathsf{newtable} \langle a : \mathsf{table}.\ table(a, []) \rangle$

2 $\{table(t, h)\}$
3 $\mathsf{lookup}(t, c)$
4 $\{a : \mathsf{option}(\mathsf{glyph}).\ table(t, h)$
5 $\qquad \wedge((c \notin \mathrm{dom}(h) \wedge a = \mathsf{None}) \vee (c \in \mathrm{dom}(h) \wedge a = \mathsf{Some}(h(c))))\}$

6 $\{table(t, h)\}$
7 $\mathsf{update}(t, c, g)$
8 $\{a : 1.\ table(t, [h|c : g])\}$

Figure 4.8: Hash Table Specification

4.3.1 Verification of the Implementation

To prove the correctness of the implementation, we will work inside-out, first giving specifications and correctness proofs for the "methods" of the factory, and then using these to prove the correctness of the make_flyweight function.

Lemma 60. *(Correctness of newglyph) Define the function* $newglyph$ *as follows:*

$$
\begin{array}{l}
newglyph \equiv \lambda c.\ [\mathsf{letv}\ x = \mathsf{lookup}(t, c)\ \mathsf{in} \\
\qquad run\ \mathsf{case}(x, \mathsf{None} \rightarrow [\mathsf{letv}\ r = [\mathsf{new}_{\mathsf{glyph}}(c, f)]\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\ \mathsf{letv}\ \_ = \mathsf{update}(t, c, r)\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\ r] \\
\qquad\qquad\qquad \mathsf{Some}(r) \rightarrow [r])]
\end{array}
$$

This function satisfies the following specification:

$$\{J(f, t, h)\} newglyph(c) \{a : \mathsf{glyph}.\ \exists h' \supseteq h. J(f, t, h') \wedge h'(c) = a\}$$

Proof. Note that in this definition, the t and f variables occur free. We are going to prove our specification valid with respect to these variables, so that we can substitute them with whatever actual terms in context that we need to use.

1 Assume we are in the precondition state $J(f, t, h)$
2 Hence we are in the precondition state $table(t, h) * \forall^* c \in \mathrm{dom}(h).\ h(c) \mapsto (c, f)$
3 $\mathsf{letv}\ x = \mathsf{lookup}(t, c)\ \mathsf{in}$
4 We now have $table(t, h) * \forall^* c' \in \mathrm{dom}(h).\ h(c) \mapsto (c', f)$
$\wedge\ ((c \notin \mathrm{dom}(h) \wedge x = \mathsf{None}) \vee (c \in \mathrm{dom}(h) \wedge x = \mathsf{Some}(h(c))))$
5 $\mathsf{case}(x,$
6 $\quad \mathsf{None} \rightarrow$
7 $\quad\quad$ Now it follows that $x = \mathsf{None}$,
8 $\quad\quad$ Hence $table(t, h) * \forall^* c' \in \mathrm{dom}(h).\ h(c') \mapsto (c', f) \wedge c \notin \mathrm{dom}(h) \wedge x = \mathsf{None}$
9 $\quad\quad [\mathsf{letv}\ r = [\mathsf{new}_{\mathsf{glyph}}((c, f))]\ \mathsf{in}$
10 $\quad\quad$ So $r \mapsto (c, f) * \forall^* c' \in \mathrm{dom}(h).\ h(c') \mapsto (c', f) * table(t, h) \wedge c \notin \mathrm{dom}(h)$
11 $\quad\quad \mathsf{letv}\ \_ = \mathsf{update}(t, c, r)\ \mathsf{in}$

So $r \mapsto (c, f) * \forall^* c' \in \text{dom}(h).\ h(c') \mapsto (c', f) * table(t, [h|c : r]) \wedge c \notin \text{dom}(h)$
So $\exists h'.\ \forall^* c' \in \text{dom}(h').\ h(c') \mapsto (c', f) * table(t, h') \wedge h' = [h|c : r]$
$r]$
We can choose the witness to the existential to be h', yielding
$\exists h' \supseteq h.\ table(t, h') \wedge h'(c) = a * \forall^* c' \in \text{dom}(h).\ h(c') \mapsto (c', f)$
Hence $\exists h' \supseteq h.\ J(f, t, h') \wedge h'(c) = a$
$\mathsf{Some}(r) \rightarrow$
We know $x = \mathsf{Some}(r)$,
Hence $table(t, h) * \forall^* c' \in \text{dom}(h).\ h(c') \mapsto (c', f) \wedge x = \mathsf{Some}(h(c))$
So $J(f, t, h) \wedge h(c) = r$
$[r])$
Hence $a.\ J(f, t, h) \wedge h(c) = a$
Choose the witness to the existential to be h, yielding
$\exists h' \supseteq h. J(f, t, h') \wedge h'(c) = a$
Hence $\exists h' \supseteq h. J(f, t, h') \wedge h'(c) = a$

□

Lemma 61. *(Correctness of $getdata$) Define the $getdata$ function as follows:*

$$getdata \equiv \lambda r.\ [!r]$$

This function satisfies the following specification:

$$\langle J(f, t, h) \wedge r = h(c) \rangle getdata(r) \langle a : \mathsf{char} \times \mathsf{font}.\ J(f, t, h) \wedge a = (c, f) \rangle$$

Proof. The correctness proof for this function is very simple, amounting to a single application of the frame rule, together with the dereference rule.

Assume our state is $table(t, h) * refs(f, h) \wedge r = h(c)$
Hence $c \in \text{dom}(h)$
This is $table(t, h) * refs(f, h - [c : r]) * r \mapsto (c, f)$
$[!r]$
So $a.\ table(t, h) * (refs(f, h - [c : r]) * r \mapsto (c, f) \wedge a = (c, f)$
So $a.\ J(f, t, h) \wedge a = (c, f)$
□

Lemma 62. *(Correctness of* make_flyweight*) The* make_flyweight *function meets the specification in Figure 4.6.*

Now, we can prove the correctness of the make_flyweight function. To do this, we will assume a font argument f, and then prove the correctness of the body of the function.

Proof.

We begin with an assumed emp precondition.
[letv $t =$ newtable() in
Now our state is $table(t, [])$
letv $newglyph = [N]$ in (where N is the definition of $newglyph$ in Lemma 60)

5 letv $getdata = [G]$ in (where G is the definition of $getdata$ in Lemma 61)
6 Now our state is $table(t, []) \wedge newglyph = N \wedge getdata = G$
7 Since the two equalities are pure, we may assume them ambiently.
8 Then we can instantiate the specs of $newglyph$ and $getdata$ to know that
9 $\{J(f,t,\sigma)\}\mathsf{run}\ N(c)\{a : \mathsf{glyph}.\ \exists\sigma' \supseteq \sigma. J(f,t,\sigma') \wedge \sigma'(c) = a\}$
10 and
11 $\{J(f,t,\sigma) \wedge r = \sigma(c)\}\mathsf{run}\ G(r)\{a : \mathsf{char} \times \mathsf{font}.\ J(f,t,\sigma) \wedge a = (c,f)\}$
12 Then since valid specs allow us to introduce validity assertions, we know
$J(f,t,[]) \wedge \text{Flyweight}(\lambda\sigma.\ J(f,t,\sigma), newglyph, getdata, f)\ \mathsf{spec}$
13 We can existentially quantify, choosing a witness $I' = \lambda\sigma.\ J(f,t,\sigma)$ to get
$\exists I'.\ I'([]) \wedge \text{Flyweight}(I', newglyph, getdata, f)\ \mathsf{spec}$
14 $(newglyph, getdata)]$
15 Now we know $\exists I.\ I([]) \wedge \text{Flyweight}(I, \mathsf{fst}\ a, \mathsf{snd}\ a, f)\ \mathsf{spec}$

□

4.4 Subject-Observer

The subject-observer pattern is one of the most characteristic patterns of object-oriented programming, and is extensively used in GUI toolkits. This pattern features a mutable data structure called the *subject*, and a collection of data structures called *observers* whose invariants depend on the state of the subject. Each observer registers a callback function with the subject to ensure it remains in sync with the subject. Then, whenever the subject changes state, it iterates over its list of callback functions, notifying each observer of its changed state. While conceptually simple, this is a lovely problem for verification, since every observer can have a different invariant from all of the others, and the implementation relies on maintaining lists of callback functions in the heap.

In our example, we will model this pattern with one type of subjects, and three functions. A subject is simply a pair, consisting of a pointer to a number, the subject state; and a list of observer actions, which are imperative procedures to be called with the new value of the subject whenever it changes. There is a function newsub to create new subjects; a function register, which attaches observer actions to the subject; and finally a function broadcast, which updates a subject and notifies all of its observers of the change.

We give a specification for the subject-observer pattern below:

1 $\exists sub : A_s \times \mathbb{N} \times \mathsf{seq}\ ((\mathbb{N} \Rightarrow \mathsf{prop}) \times (\mathbb{N} \to \bigcirc 1)).$
2 $\exists \mathsf{newsub} : \mathbb{N} \to \bigcirc A_s,$
3 $\exists \mathsf{register} : A_s \times (\mathbb{N} \to \bigcirc 1) \to \bigcirc 1,$
4 $\exists \mathsf{broadcast} : A_s \times \mathbb{N} \to \bigcirc 1.$

5 $\forall n.\ \langle \mathsf{emp} \rangle \mathsf{newsub}(n) \langle a : A_s.\ sub(a, n, \epsilon) \rangle$
$\&$
6 $\forall f, O, s, n, os. (\forall i, k. \langle O(i) \rangle f(k) \langle a : 1.\ O(k) \rangle)$

$$
\begin{array}{ll}
7 & \quad \Rrightarrow \langle sub(s, n, os) \rangle \\
8 & \quad\quad \mathsf{register}(s, f) \\
9 & \quad\quad \langle a : 1.\ sub(s, n, (O, f) \cdot os) \rangle \\
 & \ \& \\
10 & \forall s, i, os, k.\ \langle sub(s, i, os) * obs(os) \rangle \\
 & \quad\quad \mathsf{broadcast}(s, k) \\
 & \quad\quad \langle a : 1.\ sub(s, k, os) * obs\_at(os, k) \rangle
\end{array}
$$

$$
\begin{array}{ll}
obs(\epsilon) & \equiv \mathsf{emp} \\
obs((O, f) \cdot os) & \equiv (\exists i.\ O(i)) * obs(os)
\end{array}
$$

$$
\begin{array}{ll}
obs\_at(\epsilon, k) & \equiv \mathsf{emp} \\
obs\_at((O, f) \cdot os, k) & \equiv O(k) * obs\_at(os, k)
\end{array}
$$

On line 1 we assert the existence of a three-place predicate $sub(s, n, os)$. The first argument is the subject s, whose state this predicate represents. The second argument n is the data the observers depend on, and the field os is a sequence of callbacks paired with their invariants. That is, os is a sequence of pairs, each consisting of the observer functions which act on a state, preceeded by the predicate describing what that state should be.

On lines 2-4, we assert the existence of newsub, register and broadcast, which create a new subject, register a callback, and broadcast a change, respectively.

register is a higher order function, which takes a subject and an observer action as its two arguments. The observer action is a function of type $\mathbb{N} \to \bigcirc 1$, which can be read as saying it takes the new value of the subject and performs a side-effect. Because register take a procedure as an argument, its specification must say how this observer action should behave. register's specification on lines 6-9 accomplishes this via an implication over Hoare triples. It says that *if* the function f is a good observer callback, *then* it can be safely registered with the subject. Here, a "good callback" f is one that takes an argument k and sends an observer state to $O(k)$. If this condition is satisfied, then $\mathsf{register}(s, f)$ will add the pair (O, f) to the sequence of observers in the sub predicate.

One point worth focusing on is that a given subject can have many different observers o_i, each with a *different* invariant O_i and callback f_i. This means that the $sub(s, n, os)$ predicate is a genuinely higher-order one, since it contains a list of arbitrary predicate values. This is natural, given that the core of the subject-observer pattern is to call a list of unknown higher-order imperative procedures, but it is still worth pointing out explicitly.

broadcast updates a subject and all its interested observers. The precondition state of $\mathsf{broadcast}(s, k)$ requires the subject state $sub(s, n, os)$, and all of the observer states $obs(os)$. The definition $obs(os)$ takes the list of observers and yields the separated conjunction of the observer states. So when broadcast is invoked, it can modify the subject and any of its observers. Then, after the call, the postcondition puts the sub predicate and all of the observers in the same state k. The $obs\_at(os, k)$ function generates the separated conjunction of all the O predicates, all in the same state k.

The implementation follows:

1 $\quad A_s \equiv \mathsf{ref}\ \mathbb{N} \times \mathsf{list}\ (\mathbb{N} \to \bigcirc 1)$

```
2   sub(s, n, os) ≡fst s ↦ n * list(snd s, map snd os) ∧ Good(os)

3   Good(ϵ)              ≡ ⊤
4   Good((O, f) · os) ≡ (∀i, k. {O(i)}run f(k){a : 1. O(k)}) spec ∧ Good(os)

5   register(s, f) ≡ [letv cell = [!(snd s)] in
6                     letv r = [new_listcell(ℕ→◯1)(cell)] in
7                     snd s := Cons(f, r)]

8   broadcast(s, k) ≡
9         [letv _ = [fst s := k] in loop(k, snd s)]

10  loop(k, list) ≡
        [letv cell = [!list] in
11       run case(cell,Nil → [()],
12                     Cons(f, tl) → [letv _ = f(k) in
13                                    run loop(k, tl)])

14  newsub(n) ≡ [letv data = new_ℕ(n) in
15                letv callbacks = new_listcell(ℕ→◯1)(Nil) in
16                (data, callbacks)]
```

In line 1, we define the concrete type of the subject A_s to be a pair of a pointer to a reference, and a mutable list of callback functions. On line 2, we define the three-place subject predicate, $sub(s, n, os)$. The first two subclauses of the predicate's body describe the physical layout of the subject, and assert that the first component of s should point to n, and that the second component of s should be a linked list containing the function pointers in os. (The $list$ predicate is described in Figure 4.3, when we give the definition of the iterator predicates.)

Then we require that os be "Good". $Good$-ness is defined on lines 3 and 4, and says a sequence of predicates and functions is good when every (O, f) pair in the sequence satisfies the same validity requirement the specification of register demanded – that is, that each observer function f update O properly. (Note that we make use of our ability to nest specifications within assertions, in order to constrain the behavior of code stored in the heap.)

Next, we give the implementations of register and broadcast. register, on lines 5-7, adds its argument to the list of callbacks. Though the code is trivial, its correctness depends on the fact the $Good$ predicate holds for the extended sequence — we use the fact that the argument f updates O properly to establish that the extended list remains $Good$.

broadcast, on lines 8-9, updates the subject's data field (the first component), and then calls loop (on lines 10-13) to invoke all the callbacks. loop$(k,$ snd $s)$ just recurs over the list and calls each callback with argument k. The correctness of this function also relies on the $Good$ predicate – each time we call one of the functions in the observer list, we use the hypothesis of its behavior given in $Good(os)$ to be able to make progress in the proof.

Below, we give a simple piece of client code using this interface.

```
1   {emp}
2   letv s = newsub(0) in
3   {sub(s, 0, ϵ)}
4   letv d = new_ℕ((0)) in
```

5 letv $b = \mathsf{new}_{\mathsf{bool}}((\mathsf{true}))$ in
6 $\{sub(s, 0, \epsilon) * d \mapsto 0 * b \mapsto \mathsf{true}\}$
7 letv $() = \mathsf{register}(s, f)$ in
8 $\{sub(s, 0, (double, f) \cdot \epsilon) * double(0) * b \mapsto \mathsf{true}\}$
9 letv $() = \mathsf{register}(s, g)$ in
10 $\{sub(s, 0, (even, g) \cdot (double, f) \cdot \epsilon) * double(0) * even(0)\}$
11 $\mathsf{broadcast}(s, 5)$
12 $\{sub(s, 5, (even, g) \cdot (double, f) \cdot \epsilon) * double(5) * even(5)\}$
13 $\{sub(s, 5, (even, g) \cdot (double, f) \cdot \epsilon) * d \mapsto 10 * b \mapsto \mathsf{false}\}$

14 $f \equiv \lambda n : \mathbb{N}.\ [d := 2 \times n]$
15 $double(n) \equiv d \mapsto (2 \times n)$
16 $g \equiv \lambda x : \mathbb{N}.\ [b := even?(x)]$
17 $even(n) \equiv b \mapsto even?(n)$

We start in the empty heap, and create a new subject s on line 2. On line 4, we create a new reference to 0, and on line 5, we create a reference to true. So on line 6, the state consists of a subject state, and two references. On line 7, we call register on the function f (defined on line 14), which sets d to twice its argument. To the observer list in sub, we add f and the predicate $double$ (defined on line 15), which asserts that indeed, d points to two times the predicate argument. On line 8, we call register once more, this time with the function g (defined on line 16) as its argument, which stores a boolean indicating whether its argument was even into the pointer b. Again, the state of sub changes, and we equip g with the $even$ predicate (defined on line 17) indicating that b points to a boolean indicating whether the predicate argument was even or not. Since $d \mapsto 0$ and $b \mapsto \mathsf{true}$ are the same as $double(0)$ and $even(0)$, so we can write them in this form on line 10. We can now invoke $\mathsf{broadcast}(s, 5)$ on line 11, and correspondingly the states of all three components of the state shift in line 12. In line 13, we expand $double$ and $even$ to see d points to 10 (twice 5), and b points to false (since 5 is odd).

Discussion. One nice feature of the proof of the subject-observer implementation is that the proofs are totally oblivious to the concrete implementations of the notification callbacks, or to any details of the observer invariants. Just as existential quantification hides the details of a module implementation from the clients, the universal quantification in the specification of register and broadcast hides all details of the client callbacks from the proof of the implementation – since they are free variables, we are unable to make any assumptions about the code or predicates beyond the ones explicitly laid out in the spec. Another benefit of the passage to higher-order logic is the smooth treatment of observers with differing invariants; higher-order quantification lets us store and pass formulas around, making it easy to allow each callback to have a totally different invariant.

4.4.1 Correctness Proofs for Subject-Observer

Proof of the register **Function**

Proof.

1 Assume we have f, O, s, cs, n, os and the specification $\forall i, k.\ \langle O(i)\rangle f(k)\langle a : 1.\ O(k)\rangle$
2 Assume we are in the prestate $\{sub((s, cs), n, os)\}$
3 This state is equivalent to $s \mapsto n * list(cs, map\ \mathsf{snd}\ \ os) \wedge Good(os)$
4 By the definition of $list$, we know
$list(cs, map\ \mathsf{snd}\ \ os) = \exists cell.\ cs \mapsto cell * listcell(cell, map\ \mathsf{snd}\ \ os)$
5 $[\mathsf{letv}\ cell = [!cs]\ \mathsf{in}$
6 We can drop the existential now, giving a state of
$s \mapsto n * cs \mapsto cell * listcell(cell, map\ \mathsf{snd}\ \ os) \wedge Good(os)$
7 $\mathsf{letv}\ r = [\mathsf{new}_{\mathsf{ref\ list}\ (\mathbb{N} \to \bigcirc \mathbf{1})}(cell)]\ \mathsf{in}$
8 We add $r \mapsto cell$ to the state, and fold the definition of list, to get
$s \mapsto n * cs \mapsto - * list(r, map\ \mathsf{snd}\ \ os) \wedge Good(os)$
9 $cs := \mathsf{Cons}(f, r)]$
10 Therefore $s \mapsto n * cs \mapsto \mathsf{Cons}(f, r) * list(r, map\ \mathsf{snd}\ \ os) \wedge Good(os)$
11 Therefore $s \mapsto n * list(cs, map\ \mathsf{snd}\ \ ((O, f) \cdot os)) \wedge Good(os)$
12 Since we assume $\forall i, k.\ \{O(i)\}\mathsf{run}\ f(k)\{a : 1.\ O(k)\}$, we can conjoin it to the state to get $s \mapsto n * list(cs, map\ \mathsf{snd}\ \ ((O, f) \cdot os)) \wedge Good((O, f) \cdot os)$
13 This is $sub((s, cs), n, (O, f) \cdot os)$
□

Proof of the newsub **Function**

Proof.

1 Assume we have a variable n and the initial state emp
2 Now consider the body of the newsub function:
3 $[\mathsf{letv}\ data = \mathsf{new}_{\mathbb{N}}(n)\ \mathsf{in}$
4 The state is $data \mapsto n$
5 $\mathsf{letv}\ callbacks = \mathsf{new}_{\mathsf{listcell}\mathbb{N} \to \bigcirc 1}(\mathsf{Nil})\ \mathsf{in}$
6 The state is $data \mapsto n * callbacks \mapsto \mathsf{Nil}$
7 This is equivalent to $data \mapsto n * list(callbacks, \epsilon)$
8 This is equivalent to $data \mapsto n * list(callbacks, \epsilon) \wedge Good(\epsilon)$
9 This is equivalent to $sub((data, callbacks), n, \epsilon)$
10 $(data, callbacks)]$
11 Therefore $sub(a, n, \epsilon)$ with return value a
□

Proof of the broadcast **Function**

To prove this function, we first need to prove an auxilliary lemma about the loop function:
Lemma 63. *(*loop *Invariant) For all k, fs and os we have*

$\{Good(os) \wedge list(fs, map\ \mathsf{snd}\ os) * obs(os)\}\mathsf{loop}(k, fs)\{a : 1.\ list(fs, map\ \mathsf{snd}\ \ os) * obs\_at(os, k)\}$

Proof.

Assume we have a suitable k and fs, and then proceed by induction on os.
Pull $Good(os)$ into the context as a pure assertion, and then frame it away.
Suppose $os = \epsilon$:
  Then our precondition is $list(fs, \epsilon) * obs(\epsilon)$
  This is equivalent to $fs \mapsto \mathsf{Nil}$
  [letv $cell = [!fs]$ in
  So we know $fs \mapsto \mathsf{Nil} \wedge cell = \mathsf{Nil}$
  Since $cell = \mathsf{Nil}$ and equational reasoning, we know the remainder of the program is $\langle\rangle$
  $\langle\rangle$]
  So the state is still $list(fs, \epsilon)$
  This is equivalent to $list(fs, \epsilon) * \mathsf{emp}$
  This is equivalent to $list(fs, \epsilon) * obs\_at(\epsilon, k)$
  This is equivalent to $Good(\epsilon) \wedge list(fs, \epsilon) * obs\_at(\epsilon, k)$
  This is equivalent to $Good(os) \wedge list(fs, \epsilon) * obs\_at(\epsilon, k)$
Suppose $os = (O, f) \cdot os'$:
  Then our precondition is $list(fs, f \cdot map\ \mathsf{snd}\ os') * obs((O, f) \cdot os')$
  This is equivalent to $\exists fs'.\ fs \mapsto \mathsf{Cons}(f, fs') * list(fs', map\ \mathsf{snd}\ os') * \exists j.\ O(j) * obs(os')$
  By dropping existentials, $fs \mapsto \mathsf{Cons}(f, fs') * list(fs', map\ \mathsf{snd}\ os') * O(j) * obs(os')$
  [letv $cell = [!fs]$ in
  We know $fs \mapsto cell * list(fs', map\ \mathsf{snd}\ os') * O(j) * obs(os') \wedge cell = \mathsf{Cons}(f, fs')$
  Since $cell = \mathsf{Cons}(f, fs')$, we can use equational reasoning to eliminate the case
  Since we assumed $Good(os)$, we know that $Good(O, f) \cdot os'$
  Hence we know that $(\forall i, k.\ \{O(i)\}\mathsf{run}\ f(k)\{a : 1.\ O(k)\})\ \mathsf{spec} \wedge Good(os')$
  Therefore we know that $(\forall i, k.\ \{O(i)\}\mathsf{run}\ f(k)\{a : 1.\ O(k)\})$
  letv $\_ = f(k)$ in
  We know $fs \mapsto cell * list(fs', map\ \mathsf{snd}\ os') * O(k) * obs(os') \wedge cell = \mathsf{Cons}(f, fs')$
  Add $Good(os')$ to the precondition, since it is a consequence of $Good(os)$, before
  $\mathsf{loop}(k, fs')$]
  Then by induction, we can conclude
  $Good(os') \wedge fs \mapsto cell * list(fs', map\ \mathsf{snd}\ os') * O(k) * obs\_at(os', k) \wedge cell = \mathsf{Cons}(f, fs')$
  This is equivalent to $Good(os') \wedge list(fs, map\ \mathsf{snd}\ os) * obs\_at(os, k)$
  Since we know $Good(os)$, we can strengthen this to
  $Good(os) \wedge list(fs, map\ \mathsf{snd}\ os) * obs\_at(os, k)$
Hence $list(fs, map\ \mathsf{snd}\ os) * obs\_at(os, k)$

□

There is an interesting feature of this proof which distinguishes it from the proofs we gave for the iterator implementation. In this proof, we reason about branches by doing a case analysis in the program logic, and then using the specification-level equality to justify *simplifying* the program we are proving. That is, when we consider an empty input list, we can use the equational theory of the lambda calculus to simplify the program we are proving to eliminate that case altogether.

Now, the proof of the broadcast function is quite easy:

1 Assume we have s, i, os, k with a precondition of $sub(s, i, os) * obs(os)$
2 This is equivalent to $Good(os) \wedge \mathsf{fst}\ s \mapsto i * list(\mathsf{snd}\ s, map\ \mathsf{snd}\ \ os) * obs(os)$
3 $[\mathsf{letv}\ \_ = [\mathsf{fst}\ s := k]\ \mathsf{in}$
4 This yields $Good(os) \wedge \mathsf{fst}\ s \mapsto k * list(\mathsf{snd}\ s, map\ \mathsf{snd}\ \ os) * obs(os)$
5 $\mathsf{loop}(k, \mathsf{snd}\ s)]$
6 This yields $Good(os) \wedge \mathsf{fst}\ s \mapsto k * list(\mathsf{snd}\ s, map\ \mathsf{snd}\ \ os) * obs\_at(os, k)$
7 This is equivalent to $sub(s, k, os) * obs\_at(os, k)$

Chapter 5

Proving the Union-Find Disjoint Set Algorithm

5.1 Introduction

In this chapter, we introduce the technique of "ramification", as a way of recovering local reasoning in the face of imperative programs with global invariants.

The union-find disjoint set data structure [10] is a technique for efficiently computing canonical representives for equivalence classes of values. The basic technique for doing so is to represent each value in the equivalence class as a node in a tree — but unlike the usual implementation of trees, each node does not contain a pointer to its children, but rather the children each maintain a pointer to the parent. The root of the tree has no parent pointer, and represents the canonical representative for an equivalence class.

The canonical representative can be found (the find operation) by following the parent pointers to the root of the tree. Similarly, two disjoint sets can be merged (the union operation), by finding their canonical representatives and setting one to point to the other.

As described, this data structure is no better than using a linked list. However, two optimizations give rise to an extremely efficient implementation [11]. First, the root node can be modified to keep track of a bound on the maximum height, so that whenever two sets are merged, the shorter tree can be made a subtree of the deeper one. Second, the algorithm can make use of *path compression* – whenever the find operation is called, it can set all of the nodes on the path to the root to point directly at the root. Together, these optimizations permit performing a sequence of n union and find operations in $O(n \cdot \alpha(n))$ time, where α is the inverse Ackermann function [46].

This permitted Huet [15] to give a simple implementation of near-linear-time unification algorithms, and variants of this idea are used in the proofs of congruence closure algorithms [32].

However, path compression is an idiom difficult to accomodate within the framework of separation logic. In informal reasoning about the union-find data structure, we do not explicitly track all the elements of a union-find data structure in our reasoning — instead, we rely on the fact that path compression only makes changes to the heap which our global program invariant should be insensitive to. However, separation logic is a resource-aware logic, which demands to know the footprint of any command. So we cannot simply leave the other elements of the

equivalence class out of the invariant, since union may read and modify them.

The solution I propose in this chapter is to use a global invariant structured in a way which preserves modular reasoning, both in the style of separation logic, and in the interference-insensitive style of the usual informal proofs.

However, we do not only want to hide interference! One of the features which makes union-find so elegant is that the union operation features a well-structured use of aliased mutable state. When merging two equivalence classes, a single update allows efficiently *broadcasting* the change to every element of both classes in almost constant time. So we need a specification technique that should also let us specify global interference in a clean, well-structured way. We achieve this by introducing a *ramification operator* in our specification, which gives us an abstract way of characterizing the information propagated globally.

To understand the idea of ramifications, we look back to McCarthy's original paper introducing the frame problem [24]. There, he described the frame problem as the problem of how to specify what parts of a state were *unaffected* by the action of a command, which inspired the name of the frame rule in separation logic. In that paper, he also described the *qualification problem*. He observed that many commands (such as a flipping a light switch turning on a light bulb) have numerous implicit preconditions (such as there being a bulb in the light socket), and dubbed the problem of identifying these implicit preconditions the qualification problem.

Some years later, Finger [9] observed that the qualification problem has a dual: actions can have indirect effects that are not explicitly stated in their specification (e.g., turning on the light can startle the cat). He called the problem of deducing these implicit consequences the "ramification problem" — is there a simple way to represent all of the indirect consequences of an action? If so, then we have a way of modularly specifying *global* effects. This is the idea we will adopt to deal with the broadcast nature of union in the union-find algorithm.

5.2 The Specification

In Figure 5.1, we give our specification of the union-find algorithm.

On lines 1-5 of the specification, we specify that there is an abstract type τ of nodes of the disjoint set forest, and three operations newset, find, and union, which create new nodes, find canonical representatives, and merge equivalence classes, respectively. Furthermore, there is a monolithic abstract predicate $H(\phi)$, which describes the entire disjoint-set forest all at once.

This monolithic predicate represents of one of the two tricks of this specification. Our first trick is to replay the key idea of separation logic, only "one level up". Even though we have a single abstract predicate describing the whole forest, we can recover separation-style modularity by indexing the abstract predicate with a formula ϕ, which gives a small (in fact, nearly degenerate) "separation logic" for describing elements of these equivalence classes.

The datatype of formulas is given in the display above the specification. We give it both as an inductive type, to illustrate why it is definable in higher-order, and as a grammar (which is more readable, and what we use in the specifications). Formulas have three grammatical productions. First, we have the forms I and $\phi \otimes \psi$, which are an (intuitionistic) unit and separating conjunction for the elements of this little logic. We also have an atomic formula $\mathsf{elt}(x, y)$, which says that x is a term whose canonical representative is y.

$$\begin{array}{lcl} \mathsf{formula} & = & I \mid \mathsf{Elt\ of}\ \tau \times \tau \mid \mathsf{Tensor\ of\ formula} \times \mathsf{formula} \\ \phi, \psi & ::= & I \mid \mathsf{elt}(x,y) \mid \phi \otimes \psi \end{array}$$

$$\begin{array}{lcl} R & : & (\tau \to \tau) \times \mathsf{formula} \to \mathsf{formula} \\ R(\rho, I) & = & I \\ R(\rho, \psi \otimes \phi) & = & R(\rho,\psi) \otimes R(\rho,\phi) \\ R(\rho, \mathsf{elt}(x,y)) & = & \mathsf{elt}(x, \rho(y)) \end{array}$$

$$\boxed{\phi \vdash \mathsf{elt}(x,y)}$$

$$\frac{}{\mathsf{elt}(x,y) \vdash \mathsf{elt}(x,y)} \qquad \frac{\phi \vdash \mathsf{elt}(x,y)}{\phi \otimes \psi \vdash \mathsf{elt}(x,y)} \qquad \frac{\psi \vdash \mathsf{elt}(x,y)}{\phi \otimes \psi \vdash \mathsf{elt}(x,y)}$$

1 $\exists \tau : \star$
2 $\exists H : \mathsf{formula} \to \mathsf{prop}$
3 $\exists \mathsf{newset} : \bigcirc \tau$
4 $\exists \mathsf{find} : \tau \to \bigcirc \tau$
5 $\exists \mathsf{union} : \tau \times \tau \to \bigcirc \mathbf{1}$
6 $\{H(\phi)\}\mathsf{newset}\{a : \tau.\ H(\phi \otimes \mathsf{elt}(a,a))\}$
$\&$
7 $\{H(\phi) \wedge (\phi \vdash \mathsf{elt}(x,y))\}\mathsf{find}(x)\{a : \tau.\ H(\phi) \wedge a = y\}$
$\&$
8 $\{H(\phi) \wedge (\phi \vdash \mathsf{elt}(x,y)) \wedge (\phi \vdash \mathsf{elt}(u,v))\}\mathsf{union}(x,u)\{a : 1.\ H(R([y/v],\phi))\}$
$\&$
9 $\{\forall \phi, a, b, x, y.\ H(\phi \otimes \mathsf{elt}(a,b) \otimes \mathsf{elt}(x,y)) \supset a \neq x\}$
$\&$
10 $\{\forall \phi.\ H(\phi) \supset H(I)\}$
$\&$
11 $\{\forall \phi, \psi.\ H(\phi \otimes \psi) \iff H(\psi \otimes \phi)\}$
$\&$
12 $\{\forall \phi.\ H(I \otimes \phi) \iff H(\phi)\}$
$\&$
13 $\{\forall \phi, \psi, \theta.\ H(\phi \otimes (\psi \otimes \theta)) \iff H((\phi \otimes \psi) \otimes \theta)\}$

Figure 5.1: Specification of Union Find Algorithm

These formulas have the usual resource-aware interpretation of separation logic, so that $\mathsf{elt}(x, y) \otimes \mathsf{elt}(a, b)$ implies that $x \neq a$.

We also give a simple judgement on formulas $\phi \vdash \mathsf{elt}(x, y)$, which lets us say that ϕ entails drawing the conclusion that x's representative is y. This is of course a trivial judgement, since the language of formulas is so simple.

In addition to this separation logic, our second trick is embodied in the modal operator $R(\rho, \phi)$, which we call a "ramification operator". Here, ρ is a substitution, and ϕ is a formula. R is defined to operate homomorphically on the structure of formulas, with its action on atomic formulas being $R(\rho, \mathsf{elt}(x, y)) = \mathsf{elt}(x, \rho(y))$. Intutively, a ramification $R(\rho, \phi)$ says to replace the canonical witnesses in the domain of the substitution with the result of the substitution. This lets us specify the aliasing effects of global updates in a modular fashion.

So we have the tools to reason both locally and globally in our specifications. An example of local reasoning can be seen on line 6 of Figure 5.1. Here, we say that given a state $H(\phi)$, calling newset will result in a new state $H(\phi \otimes \mathsf{elt}(a, a))$. This functions a bit like a global axiom for creating new equivalence classes, since we explicitly quantify over the frame.

By quantifying over ϕ, we can *implement* the frame rule for our library. This is similar to the interpretation of the frame rule in our underlying semantics – there, we interpret Hoare triples to mean all the assertions that can be safely framed on, and here, we quantify over all possible frames.

On line 7, we see the necessity of this kind of interpretation. Our specification for find is very simple – it says that in any state which entails $\mathsf{elt}(x, y)$, calling $\mathsf{find}(x)$ will return y. The ϕ is unchanged from precondition to postcondition, and so the user of this library does not need to know anything about any elements other than x. However, due to path compression, we can modify many other nodes in the forest, a fact which our domain-specific logic conceals.

On line 8, we give the specification of union. Here, we say that if x's witness is y, and u's witness is v, then calling $\mathsf{union}(x, u)$ will equate the two equivalence classes, setting u's witness to y. Furthermore, since this is globally visible, we need to push this ramification over the entire set of known nodes ϕ. Observe that unlike in the previous function we do not want local reasoning, since the purpose the union operation is to globally broadcast the update. But the use of a ramification operator does *structure* this update.

On lines 9-13, we add axioms corresponding to our domain-specific logic. On line 9, we say that disjoint elements are disjoint, and on line 10 we say that we can forget the existence of elements (i.e., that this logic is like intuitionistic separation logic). On lines 11-13, we simply say that formulas are commutative, unital and associative.

In Figure 5.2 we give the invariant for the union-find data structure. The node type τ is a pointer to a term of type σ, which is either a Child value containing the parent of the current node, or a $\mathsf{Root}(w, n)$ value containing the witness plus a number to maintain balance.[1]

Then on line 3, we say that $H(\phi)$ holds when there are D, p, and w such that $G(D, p, w, \phi)$ holds. We require D to be a finite set of nodes D, $p : D \rightharpoonup D$ to be a partial map of nodes to parents, and a map $w : (D - \mathrm{dom}(p)) \to D$. The set of nodes D represent all the elements that have been allocated, and the parent map p maps each node to its parent. The function p is partial

[1] We do not track the ranks of subtrees in our invariant to avoid obscuring the essential simplicity of the techniques underpinning ramification, though it is straightforward to add.

1 $\sigma = \mathsf{Child\ of\ ref}\ \sigma \mid \mathsf{Root\ of\ option}\ (\mathsf{ref}\ \sigma) \times \mathbb{N}$
2 $\tau = \mathsf{ref}\ \sigma$
3 $H(\phi) \triangleq \exists D \subseteq \tau, p \in D \rightharpoonup D, w \in (D - \mathrm{dom}(p)) \to D.\ G(D, p, w, \phi)$

4 $G(D, p, w, \phi) = p^+$ strict partial order $\wedge$
5 w injective $\wedge$
6 $(D, p, w) \models \phi \wedge$
7 $heap(D, p, w)$

8 $D, p, w \models I$ iff always
9 $D, p, w \models \phi \otimes \psi$ iff $\exists D_1, D_2.\ D = D_1 \uplus D_2$ and $D_1, p, w \models \phi$ and $D_2, p, w \models \psi$
10 $D, p, w \models \mathsf{elt}(x, y)$ iff $x \in D \wedge \exists z.\ (x, z) \in p^* \wedge z$ maximal $\wedge\ w(z) = y$

11 $heap(D, p, w) = \forall^* l \in \mathrm{dom}(p).\ l \mapsto \mathsf{Child}(p(l))$
12 $* \forall^* l \in (D - \mathrm{dom}(p)).\ \exists n.\ l \mapsto \mathsf{Root}(\mathsf{Some}(w(l)), n)$

Figure 5.2: Concrete Definition of Union Find Invariants

since some nodes are root nodes of the disjoint-set forest. The map $w : (D - \mathrm{dom}(p)) \to D$ sends those roots to the appropriate canonical witness.

Then, on lines 4-7, we define exactly how G works. In order to ensure that the graph structure actually has no cycles, on line 4 we impose the condition that the transitive closure of p is a strict partial order. That is, we require that p^+ is a transitive, irreflexive relation. Since the domain D is finite, this also insures that p^+ has no infinite ascending chains.

In terms of graph structure, this requirement means that the nodes form a directed acyclic graph, and the fact that it arises as the closure of a function means that no node has more than one parent. Together, these conditions ensure that the nodes form a forest. On line 5, we assert that w is an injective function, which ensures that the canonical representatives for different equivalence classes are all distinct from one another. On line 6, we assert that the triple (D, p, w) models our formulas.

These formulas are a small subset of separation logic. We have the formula I, which is always satisfied. (So this logic is like an intuitionistic rather than classical separation logic.) Then we have the formula $\phi \otimes \psi$, which corresponds to the usual separating conjunction, in that the resource n (the collection of disjoint-set nodes) is split into two parts, one of which must support ϕ and the other of which must support ψ. Note that the whole of the parent function p and the canonical witness function w functions are passed to both branches. This is the information that will let us ensure that global constraints are maintained in local invariants. Finally, the atomic proposition $\mathsf{elt}(x, y)$ asserts that x is a node whose canonical witness is y, by saying that x is in D, and that there is some maximal z above x in the reflexive transitive closure of p (viewed as a partial order), such that $w(z) = y$. (Note that z is maximal precisely when it is not in the domain of p.)

Then, the predicate $heap(D, p, w)$ asserts that every node in D is correctly realized by some physical node in the heap. Every child node must point to its parent, and every root node must

```
1   newset = [letv r = [new_σ(Root(None, 0))] in
2             letv ⟨⟩ = [r := Root(Some(r), 0)] in
3             r]

4   findroot(x) = [letv i = [!x] in
5                  run case(i,
6                           Root(w, n) → [(x, w, n)],
7                           Child(p) → [letv (r, w, n) = findroot(p) in
8                                       letv _ = [x := Child(r)] in
9                                       (r, w, n)]

10  find(x) = [letv (_, w, _) = findroot(x) in w]

11  union(x, y) = [letv (r, u, m) = findroot x in
12                 letv (s, v, n) = findroot y in
13                 run if r ≠ s then
14                       if m < n then
15                         [letv ⟨⟩ = [s := Root(u, n)] in
16                          r := Child(s)]
17                       else if n < m then
18                         [s := Child(r)]
19                       else
20                         [letv ⟨⟩ = [r := Root(u, n + 1)] in
21                          s := Child(r)]
22                     else
23                       [⟨⟩]]
```

Figure 5.3: Implementation of Union-Find Algorithm

point to its witness and some count of its tree rank.

In Figure 5.3, we give the implementation of these functions. On lines 1-3, we give the code for creating a new element. This works by allocating a pointer of type τ, and then updating it so that it points to itself as its canonical witness.

The find function is defined on line 10, but it works by deferring almost all of its work to an auxiliary function findroot. This function is defined on lines 4-9, and it works by recursively following the parent pointers of each node. When it reaches a root, it returns a triple (r, w, n), containing all three of the physical root r, the witness value w, and the tree rank n. On a recursive call (i.e., when the argument is a child node), we take the return triple, and before returning, we implement path compression, by updating the child's parent to be the root node r. The find function simply calls findroot, and ignores its return values except for the witness.

The reason we have this auxilliary function becomes evident in the union function, on lines 11-23. Given arguments (x, y), what union does is to first call findroot on both x and y (on lines 11 and 12). Then, on line 13, we compare the two physical roots for inequality. If they are equal,

then there is no work to be done (lines 22-23). Otherwise, we compare the two returned tree ranks, and add the smaller root as a child of the larger one (lines 14-19), and increment the size counter if they are the same rank (lines 20-21).

5.3 Correctness Proofs

All of these proofs have a similar structure. First, we prove a lemma about how the properties of the parent order p can change. Second, we prove how this changes (or doesn't change) the satisfaction of a formula ϕ. Third, we use these two properties to verify the program itself, in an annotated specification style.

Lemma 64. *(Soundness of Entailment) If* $D, p, w \models \phi$ *and* $\phi \vdash \mathsf{elt}(u, v)$*, then* $D, p, w \models \mathsf{elt}(u, v)$.

Proof. This proof follows from an easy induction on the derivation of $\phi \vdash \mathsf{elt}(u, v)$. □

Lemma 65. *(Disjointness of Elements) The formula* $\forall \phi, a, b, x, y.\ H(\phi \otimes \mathsf{elt}(a, b) \otimes \mathsf{elt}(x, y)) \supset a \neq x$ *is valid.*

Proof.

1 Assume ϕ, a, b, x, y and $H(\phi \otimes \mathsf{elt}(a, b) \otimes \mathsf{elt}(x, y))$
2 So we know that there are D, p, and w such that $D, p, w \models \phi \otimes \mathsf{elt}(a, b) \otimes \mathsf{elt}(x, y)$
3 By the definition of $\otimes$, we know that there are disjoint D_1, D_2 and D_3, such that
$D_1, p, w \models \phi$
$D_2, p, w \models \mathsf{elt}(a, b)$
$D_3, p, w \models \mathsf{elt}(x, y)$
4 So we know $a \in D_2$ and $x \in D_3$
5 Since D_2 and D_3 are disjoint, $a \neq x$

□

Lemma 66. *(Structural Properties) The following properties are valid:*

- $\forall \phi.\ H(\phi) \supset H(I)$
- $\forall \phi, \psi.\ H(\phi \otimes \psi) \iff H(\psi \otimes \phi)$
- $\forall \phi.\ H(I \otimes \phi) \iff H(\phi)$
- $\forall \phi, \psi, \theta.\ H(\phi \otimes (\psi \otimes \theta)) \iff H((\phi \otimes \psi) \otimes \theta)$

Proof. These properties follow immediately from the semantics of assertions. □

5.3.1 Proof of find

Suppose $R \subseteq D \times D$ is a strict partial order. We say $x \in D$ is *maximal* when there is no y such that $(x, y) \in R$. Note that when we have a partial function $f : D \rightharpoonup D$ such that f^+ is a strict partial order, $x \in D$ is maximal precisely when $f(x)$ is not defined. (If $f(x)$ were defined, then $(x, f(x)) \in f^+$. Hence it cannot be defined for any maximal element.)

Lemma 67. *(Path Compression Lemma) Suppose* D *is a finite set and* $f : D \rightharpoonup D$ *is a partial function on* D*, such that* f^+ *is a strict partial order. Now suppose* $(x, y) \in f^+$ *with* y *maximal.*

Then, $[f|x : y]$ has the same domain as f, and $[f|x : y]^+$ is a strict partial order such that for all $u, v \in D$, $(u, v) \in [f|x : y]^+$ with v maximal if and only if $(u, v) \in f^+$ with v maximal.

Proof.

1. Since $(x, y) \in f^+$, we know that $x \in \text{dom}(f)$. Hence $[f|x : y]$ has the same domain as f.
2. For $[f|x : y]^+$ to be a strict partial order, it must be an irreflexive transitive relation. Since it is the transitive closure of a function, it is immediately transitive.

   To show that it is irreflexive, assume that $(u, u) \in [f|x : y]^+$. Therefore there is some sequence $(u, [f|x : y](u), \ldots, [f|x : y]^{n+1}(u) = u)$.

   Now consider whether x is in the sequence. If it is not, then this sequence is equal to $(u, f(u), \ldots, f^{n+1}(u) = u)$, which is a contradiction, since f^+ is irreflexive. If x is in the sequence, then we know that either x is the last element, or the second-to-last element (since $f(x) = y$, and y is maximal). Suppose x is the last element. Then we know that $u = x$, and hence $[f|x : y](u) = y$, and so we have a contradiction. Suppose y is the last element. Then the first element is y, and there is no $f(y)$, since y is a maximal element, and this is also a contradiction.

   Therefore there is no $(u, u) \in [f|x : y]^+$, and so $[f|x : y]^+$ is irreflexive.

3. $\Leftarrow$ Suppose that (u, v) in f^+ with v maximal. Then there is some k such that $f^k(u) = v$. So we have a sequence $u, f(u), \ldots, f^k(u)$.

   If any of the $f^i(u) = x$ for $i < k$, then we know that $v = y$, since both v and y are maximal. Therefore, it follows that the sequence $u, [f|x : y](u), \ldots, [f|x : y]^i(u), y$ shows that $(u, v) \in [f|x : y]^+$ with v maximal.

   If none of the $f^i(u) = x$, then it follows that the sequence $u, f(u), \ldots, f^k(u)$ is exactly the same as $u, [f|x : y](u), \ldots, [f|x : y]^k(u)$, and so $(u, v) \in [f|x : y]^+$ with v maximal.

   $\Rightarrow$ Suppose $(u, v) \in [f|x : y]^+$ with v maximal. Then we know that there is some sequence $u, [f|x : y](u), \ldots, [f|x : y]^{k+1}(u)$ with $[f|x : y]^{k+1}(u) = v$.

   Now, either x occurs in this sequence, or not. Suppose that x does not occur in the sequence – that is, for every $i \leq k + 1$, $[f|x : y]^i(u) \neq x$. Then it follows that this sequence is the same as $(u, \ldots, f^{k+1}(u) = v)$. Since $[f|x : y]$ and f have the same domain, it follows that v is maximal in f, as well. Hence $(u, v) \in f^+$ with v maximal.

   Now, suppose that x occurs in the sequence at position i. It cannot occur at $i = k+1$, since $f^{k+1}(u) = v$ is the last element, and we know v is maximal. But since $[f|x : y](x) = y$, we know that x is not maximal in $[f|x : y]$. Therefore it occurs at $i \leq k$.

   However, it also cannot occur at any $i < k$, since y is maximal in $[f|x : y]$, and if x occured at $i < k$, then the sequence would end at $i + 1$ with y, and we know the sequence is $k + 1$ elements long. Therefore, x occurs at position $i = k$, and $(u, \ldots, [f|x : y]^k(u), [f|x : y]^{k+1}(u) = v) = (u, \ldots, f^k(u) = x, [f|x : y]^{k+1}(u) = y = v)$.

   Now, we know that $(x, y) \in f^+$ with y maximal in f. Therefore, there is a sequence $(x, \ldots, f^{n+1}(x) = y)$. Therefore, the sequence $(u, \ldots, f^k(u) = x, \ldots, f^{n+k+1}(u) =$

$y = v$) is in f^+, with v maximal in f.

□

Lemma 68. *(Satisfaction Depends on Maximality) Suppose D is a finite set and $f, g : D \rightharpoonup D$ are partial functions on D, such that f^+ and g^+ are strict partial orders such that for all $x, y \in D$, we have that $(x, y) \in f^+$ with y maximal in f if and only if $(x, y) \in g^+$ with y maximal in g.*

Then for all ϕ, w and $D' \subseteq D$, if $D', f, w \models \phi$, then $D', g, w \models \phi$.

Proof. This lemma follows by induction on ϕ.

- Case $\phi = I$. This follows immediately from the definition of satisfaction.
- Case $\phi = \psi \otimes \theta$.

1 By assumption, we have $D', f, w \models \psi \otimes \theta$.
2 From the definition of satisfaction, we get D_1, D_2 such that $D' = D_1 \uplus D_2$ and $D_1, f, w \models \psi$ and $D_2, f, w \models \theta$.
3 By induction, we get $D_1, g, w \models \psi$
4 By induction, we get $D_2, g, w \models \theta$
5 By definition, we get $D_1 \cup D_2, g, w \models \psi \otimes \theta$

- Case $\phi = \mathsf{elt}(x, y)$.

1 By assumption, $D', f, w \models \mathsf{elt}(x, y)$
2 So we know that $x \in D'$ and there exists an z such that $(x, z) \in f^*$ with z maximal and $y = w(z)$
3 By definition of reflexive transitive closure, either $(x, z) \in f^+$ or $x = z$
4 Suppose $(x, z) \in f^+$:
5 Then by hypothesis, $(x, z) \in g^+$ with m maximal.
6 Hence $D', g, w \models \mathsf{elt}(x, z)$
7 Suppose $x = z$:
8 Since x is maximal, we know $x \notin \mathrm{dom}(f)$.
9 Suppose $x \in \mathrm{dom}(g)$.
10 Then there is a sequence $(x, g(x), \ldots, u) \in g^+$ with u maximal
11 Then $(x, f(x), \ldots, u) \in f^+$ with u maximal
12 This is a contradiction, since x is maximal in f
13 So $x \notin \mathrm{dom}(g)$
14 Hence $(x, x) \in g^*$ and x maximal in g
15 Hence $(x, z) \in g^*$ and z maximal in g
16 Hence $D', g, w \models \mathsf{elt}(x, y)$

□

Now, we can specify and prove the correctness of findroot.

Lemma 69. *(Correctness of* findroot*) The* findroot *function satisfies the following specification:*

1 $\{G(D, f, w, \phi) \wedge (D, f, w) \models \mathsf{elt}(u, v)\}$
2 $\mathsf{findroot}(u)$

3 $\{(r, c, n) : \tau \times \tau \times \mathbb{N}.$
4 $\quad \exists f'.\ G(D, f', w, \phi) \wedge c = v \wedge (u, r) \in f'^{*} \wedge r$ *maximal* $\wedge\ w(r) = c \wedge$ *dom*$(f) =$ *dom*(f')
5 $\quad \wedge\ (f, f')$ *have same maximal relationships*$\}$

Here, f and f' "having the same maximal relationships" means that $(u, v) \in f^{+}$ with v maximal if and only if $(u, v) \in (f')^{+}$ with v maximal.

Proof.

1 Assume our precondition state is $G(D, f, w, \phi) \wedge (D, f, w) \models \mathsf{elt}(u, v)$
2 Then we have a sequence $(u, \ldots, f^k(u) = x)$ with x maximal and $w(x) = v$.
3 Then we know that f^{+} is a strict partial order, and $D, f, w \models \phi$ and $heap(D, f, w)$ and $D, f, w \models \mathsf{elt}(u, v)$
4 Now, do a case split on whether or not $u \in \mathrm{dom}(f)$ (i.e., is maximal).
5 If $u \in \mathrm{dom}(f)$:
6 Then we have a sequence $(u, f(u), \ldots, f^k(u) = x)$ with x maximal and $w(x) = v$
7 Hence $(D, f, w) \models \mathsf{elt}(f(u), v)$
8 Next, $heap(D, f, w) \supset (u \hookrightarrow \mathsf{Child}(f(u))$
9 $[\mathsf{letv}\ i = [!u]\ \mathsf{in}$
10 So we add $i = \mathsf{Child}(f(u))$ to the state, and simplify the case statement
11 $\mathsf{letv}\ (r, c, n) = \mathsf{findroot}(f(u))\ \mathsf{in}$
12 Then $G(D, f', w, \phi) \wedge (f(u), r) \in f'^{*} \wedge r$ maximal $\wedge\ c = w(r) \wedge c = v$ for some f'
13 with the same domain and maximal relationships as f
14 We know that $(f(u), r) \in f'^{*}$ and r is maximal
15 If $f(u) = r$, then $(f(u), r) \in f^{*}$ and r is maximal
16 If $f(u) \neq r$, then:
17 $(f(u), r) \in f'^{+}$ and r is maximal
18 Therefore $(f(u), r) \in f^{+}$ and r is maximal, since f and f' have the same domain and maximal relationships
19 Therefore $(f(u), r) \in f^{*}$ and r is maximal
20 From line 6, $(f(u), x) \in f^{*}$ and x maximal
21 Therefore $r = x$
22 From line 19, $(u, r) \in f^{+}$ and r is maximal
23 Hence $(u, r) \in f'^{+}$ and r is maximal, since f and f' have the same domain and maximal relationships
24 So we know that $r = x$, and that $(u, r) \in f'^{*}$
25 So $D, f', w \models \phi$ and $(f')^{+}$ is a strict partial order and w injective and $heap(D, f', w)$,
26 Since we know r remains maximal in f', we know that $(D, f', w) \models \mathsf{elt}(u, v)$, since satisfaction depends only on maximality
27 Therefore, due to path compression, $[f'|u : r]^{+}$ is a strict partial order with the same maximal relationships and domain as f'
28 Hence $D, [f'|u : r], w \models \phi$ and $\mathrm{dom}([f'|u : r]) = \mathrm{dom}(f)$ and $(u, r) \in [f'|u : r]^{*}$, since satisfaction depends only on maximality
29 Now we will perform updates to make the physical heap match this logical heap
30 $\mathsf{letv}\ \_ = [u := \mathsf{Child}(r)]\ \mathsf{in}$

31 Hence $heap(D, [f'|u : r], w)$, together with the pure formulas above
32 $(r, c, n)]$
33 Choosing as the existential witness $[f'|u : r]$, we get
34 Hence $(r, c, n).\ \exists f'.\ G(D, f', w, \phi) \wedge c = v \wedge w(r) = c \wedge$
$\mathrm{dom}(f) = \mathrm{dom}(f') \wedge (u, r) \in f'^* \wedge r$ maximal $\wedge$
(f, f') have the same maximal relationships.
35 If $u \notin \mathrm{dom}(f)$:
36 Then $heap(D, f, w) \supset (u \hookrightarrow \mathsf{Root}(\mathsf{Some}(p), n))$ for some n and $p = w(u)$
37 [letv $i = [!u]$ in
38 So we add $i = \mathsf{Root}(\mathsf{Some}(w(u)), n)$ to the state, and simplify the case statement
39 $(u, p, n)]$
40 Note $(u, u) \in f^*$ and u maximal, and that $p = w(u)$, and that $\mathrm{dom}(f) = \mathrm{dom}(f)$
41 The fact that $D, f, w \models \mathsf{elt}(u, v)$ implies $v = w(u)$
42 And obviously f has the same maximal relationships as f
43 Hence $(r, c, n).\ \exists f''.\ G(D, f'', w, \phi) \wedge c = v \wedge w(r) = c \wedge$
$\mathrm{dom}(f) = \mathrm{dom}(f'') \wedge (u, r) \in f''^* \wedge r$ maximal $\wedge$
(f, f'') have the same maximal relationships.
44 (with the choice of f as witness for f'')

□

Lemma 70. *(The* find *Function is Correct) The* find *function meets the specification in Figure 5.1.*
Proof. This proof is easy, since findroot does almost all the work.

1 Assume a precondition of $H(\phi)$ and $\phi \vdash \mathsf{elt}(u, v)$
2 This means we have $G(D, f, w, \phi)$ for some D, f, and w.
3 Furthermore, we also know that $D, f, w \models \mathsf{elt}(u, v)$.
4 Now, expand the call $\mathsf{find}(u)$:
5 [letv $(\_, v', \_) = \mathsf{findroot}(u)$ in
6 Now we know $G(D, f', w, \phi) \wedge \mathrm{dom}(f) = \mathrm{dom}(f') \wedge v' = v$
7 $v']$
8 Now we know $G(D, f', w, \phi) \wedge \mathrm{dom}(f) = \mathrm{dom}(f') \wedge a = v$
□

5.3.2 Proof of newset

Lemma 71. *(Satisfaction and Allocation) Suppose $D, f, w \models \phi$ and $x \notin D$.*
Then $D \cup \{x\}, f, [w|x : x] \models \phi \otimes \mathsf{elt}(x, x)$
Proof.

1 To prove this, we want to exhibit D_1, D_2 such that $D \cup \{x\} = D_1 \uplus D_2$,
and $D_1, f, [w|x : x] \models \phi$ and $D_2, f, [w|x : x] \models \mathsf{elt}(x, x)$.
2 Take $D_1 = D$ and $D_2 = \{x\}$, which are disjoint since $x \notin D$.
3 Note that $\{x\}, f, [w|x : x] \models \mathsf{elt}(x, x)$, since:

$x \in \{x\}$
$(x, x) \in f^*$ (since this is a reflexive relation)
x is maximal (since it is not in the domain of f)
$[w|x : x](x) = x$.
Now it remains to be shown that $D, f, [w|x : x] \models \phi$
By induction on ϕ, we will show
$\forall D, \phi$, if $x \notin D$ and $D, f, w \models \phi$ then $D, f, [w|x : x] \models \phi$
Case $\phi = I$:
This case is immediate, since $D, f, [w|x : x] \models I$ by definition
Case $\phi = \psi \otimes \theta$:
Since $D, f, w, \models \psi \otimes \theta$, there are D_1, D_2 so $D_1, f, w \models \psi$ and $D_2, f, w \models \theta$ and $D = D_1 \uplus D_2$
Since $x \notin D$, we know $x \notin D_1$ and $x \notin D_2$
By induction, we know $D_1, f, [w|x : x] \models \psi$
By induction, we know $D_2, f, [w|x : x] \models \theta$
Hence $D, f, [w|x : x] \models \phi$
Case $\phi = \mathsf{elt}(u, v)$
So we know $u \in D$ and that $(u, z) \in f^*$ and z maximal and $w(z) = v$ for some z
Since $x \notin D$, $x \notin D - \mathrm{dom}(f)$, and so it follows that $[w|x : x](z) = v$
Hence $D, f, [w|x : x] \models \mathsf{elt}(u, v)$
Hence $D, f, [w|x : x] \models \mathsf{elt}(u, v)$

□

Lemma 72. *The* newset *procedure meets the specification in Figure 5.1.*

Proof.

Assume we have a precondition state $H(\phi)$, and consider the body of newset.
Then we know that f^+ is a strict partial order, and $D, f, w \models \phi$ and $heap(D, f, w)$ and $D, f, w \models \mathsf{elt}(u, v)$
$[\mathsf{letv}\ r = [\mathsf{new}_\sigma(\mathsf{Root}(\mathsf{None}, 0))]\ \mathsf{in}$
Now the state is $heap(D, f, w) * r \mapsto \mathsf{Root}(\mathsf{None}, 0)$, plus the pure predicates.
$\mathsf{letv}\ \_ = [r := \mathsf{Root}(\mathsf{Some}(r), 0)]\ \mathsf{in}$
Now the state is $heap(D, f, w) * r \mapsto \mathsf{Root}(\mathsf{Some}(r), 0)$, plus the pure predicates.
Since $heap(D, f, w)$ has a pointer for each $l \in D$, it follows that $r \notin D$.
Thus, we know that $D', f, [w|r : r] \models \phi \otimes \mathsf{elt}(r, r)$ where $D' = D \uplus \{r\}$
f^+ is still a strict partial order which is a subset of $D' \times D'$
So $f \in D' \rightharpoonup D'$
$[w|r : r] \in (D' - \mathrm{dom}(f)) \to D'$
It is clear that $heap(D, f, w) * r \mapsto \mathsf{Root}(\mathsf{Some}(r), 0)$ is equivalent to $heap(D', f, [w|r : r])$
Hence $G(D', f, [w|r : r], \phi \otimes \mathsf{elt}(r, r))$
$r]$
Hence $a.\ G(D', f, [w|a : a], \phi \otimes \mathsf{elt}(a, a))$
Hence $a.\ H(\phi \otimes \mathsf{elt}(a, a))$

□

5.3.3 Proof of union

Lemma 73. *If f^+ is a strict partial order, $(u, v) \in f^+$ and v maximal, and $(x, y) \in f^+$ and y maximal, and $v \neq y$, then it follows that for $g = [f|v : y]$,*

1. *dom(g) = dom$(f) \uplus \{v\}$*
2. *g^+ is a strict partial order*
3. *If $(a, b) \in f^*$ with b maximal, either $b \neq v$ and $(a, b) \in g^*$ with b maximal, or $b = v$ and $(a, y) \in g^*$ with y maximal.*

Proof.

1. Since v is maximal in f, it follows that $v \notin \text{dom}(f)$, and hence $\text{dom}([f|v : y]) = \text{dom}(f) \uplus \{v\}$
2. To be a strict partial order, it suffices that there is no $(z, z) \in g^+$.

    1 Assume $(z, z) \in g^+$
    2 Then there is a $k \geq 1$ such that $g^k(z) = z$
    3 Now, we'll show that v does not occur in the sequence $z, g(z), \ldots, g^k(z)$
    4 To do this, we'll first show that y does not occur in the sequence $z, g(z), \ldots, g^k(z)$
    5 Since y is maximal for f and $v \neq y$, we know y is maximal for $g = [f|v : y]$
    6 As a result, y can only occur as the last element $g^k(z)$
    7 But since $g^k(z) = z$ and $z \in \text{dom}(g)$, we know that $y \neq g^k(z)$,
    and so it cannot occur as the last element, either.
    8 As a result, v cannot occur at any $i < k$, since then $g^{i+1}(z) = y$,
    9 and we know this cannot happen
    10 We also know v cannot occur at $i = k$, since $g^k(z) = z = g^0(z)$, and
    11 we know v cannot occur at $i = 0$
    12 Therefore for $i \leq k$, we have $g^i(z) \neq v$.
    13 Hence we have $g^i(z) = f^i(z)$
    14 Therefore $z, f(z), \ldots, f^k(z) = z$ shows that $(z, z) \in f^+$
    15 But f^+ is a strict partial order, which is a contradiction.

3. If $(a, b) \in f^*$ with b maximal, either $b \neq v$ and $(a, b) \in g^*$ with b maximal, or $b = v$ and $(a, y) \in g^*$ with y maximal.

    1 Assume $(a, b) \in f^*$ with b maximal.
    2 Then either $a = b$, or $(a, b) \in f^+$
    3 Suppose $a = b$:
    4 Then either $b = v$ or not
    5 Suppose $b = v$:
    6 Then $(a, y) = (v, y) \in g \subseteq g^*$, and
    7 y is maximal since $y \notin \text{dom}(g) = \text{dom}(f) \cup \{v\}$

Suppose $b \neq v$:
Then $(a, y) = (b, b) \in g^*$, since b is maximal since $b \notin \text{dom}(g) = \text{dom}(f) \cup \{v\}$
Suppose $(a, b) \in f^+$:
Then we have a $k > 0$ such that $f^k(a) = b$
Now, either there is an $i \leq k$ such that $f^i(a) = v$, or not.
Suppose $f^i(a) = v$:
Since v is maximal, it is the last element, so $b = v$
Then we know that for all $j < i$, $f^j(a) \neq v$, since v is maximal in f
Therefore for all $j \leq i$, $f^j(a) = g^j(a)$
Therefore $g^{i+1}(a) = y$
Hence $(a, y) \in g^*$ and y is maximal, since $y \notin \text{dom}(g)$
Suppose there is no i such that $f^i(a) = v$:
Then we know that for all $j \leq k$, $f^j(a) = g^j(a)$, and $b \neq v$
Hence $(a, b) \in g^*$ and b is maximal since $b \notin \text{dom}(f) \cup \{v\} = \text{dom}(g)$

□

Lemma 74. *(Ramification) Suppose $D, f, w \models \phi$ and $(x, y) \in f^*$ with y maximal and $(u, v) \in f^*$ with v maximal, and $y \neq v$, and w injective. Let w' be the restriction of $[w|v : z]$ to exclude y, and let $g = [f|y : v]$.*

Then, $D, g, w' \models R([z/w(v), z/w(y)], \phi)$.

Proof. This proof follows by induction on the structure of ϕ.

Case $\phi = I$:
This case is immediate since $D, g, w' \models I$ and $R([z/w(v), z/w(y)], I) = I$
Case $\phi = \psi \otimes \theta$:
By satisfaction of ϕ, we have D_1, D_2 such that $D = D_1 \uplus D_2$ and
$D_1, f, w \models \psi$ and
$D_2, f, w \models \theta$
By induction, we have $D_1, g, w' \models R([z/w(v), z/w(y)], \psi)$
By induction, we have $D_2, g, w' \models R([z/w(v), z/w(y)], \theta)$
By the definition of satisfaction, $D, g, w' \models R([z/w(v), z/w(y)], \psi) \otimes R([w(v)/w(y)], \theta)$
By the definition of R, we have $D, g, w' \models R([z/w(v), z/w(y)], \phi)$
Case $\phi = \mathsf{elt}(a, b)$:
We know that $a \in D$, $(a, c) \in f^*$ with c maximal, and $w(c) = b$
Therefore either $c \neq y$ and $(a, c) \in g^*$ with c maximal, or
$c = y$ and $(a, v) \in g^*$ with v maximal
Suppose $c \neq y$:
Since w is injective, we know that $w(c) \neq w(y)$. Hence $[z/w(y)]w(c) = w(c)$
Consider whether c is v.
If $c \neq v$:
Since w is injective, we know that $w(c) \neq w(v)$. Hence $[z/w(v)]w(c) = w(c)$
Hence $[z/w(v), z/w(y)]w(c) = w(c)$
So $\mathsf{elt}(a, b) = R([z/w(v), z/w(y)], \mathsf{elt}(a, b))$

Furthermore $w'(c) = w(c)$
Hence $D, g, w' \models \mathsf{elt}(a, b)$
Hence $D, g, w' \models R([z/w(v), z/w(y)], \mathsf{elt}(a, b))$
If $c = v$:
Since $w(c) = w(v)$, we have $[z/w(v)]w(c) = z$
Hence $[z/w(v), z/w(y)]w(c) = z$
Note $w'(v) = z$
Hence $D, g, w' \models \mathsf{elt}(a, z)$
Hence $D, g, w' \models R([z/w(v), z/w(y)], \mathsf{elt}(a, b))$
Suppose $c = y$:
Then, $b = w(c) = w(y)$, so that $[z/w(v), z/w(y)]b = z$
So $R([z/w(v), z/w(y)], \mathsf{elt}(a, b)) = \mathsf{elt}(a, z)$
Furthermore, we know that $(a, v) \in g^*$ and v maximal and $w'(v) = z$
So $D, g, w' \models \mathsf{elt}(a, z)$
So $D, g, w' \models R([z/w(v), z/w(y)], \mathsf{elt}(a, b))$

□

Lemma 75. *(Correctness of* union*) The* union *function meets the specification in Figure 5.1.*

Proof.

Assume we have a precondition $H(\phi)$ and $\phi \vdash \mathsf{elt}(x, y)$ and $\phi \vdash \mathsf{elt}(u, v)$
Then we have D, f, w, such that $G(D, f, w, \phi)$
Furthermore $D, f, w \models \mathsf{elt}(x, y)$
Now consider the body of $\mathsf{union}(x, u)$
[$\mathsf{letv}\ (r, y', m) = \mathsf{findroot}(x)\ \mathsf{in}$
So we have f' such that $G(D, f', w, \phi)$ and
$(x, r) \in f'^*$ and r maximal in f'
f and f' have the same maximal relationships and domain
$y = w(r)$
$y = y'$
Furthermore $D, f', w \models \mathsf{elt}(u, v)$, so
$\mathsf{letv}\ (s, v', n) = \mathsf{findroot}(u)\ \mathsf{in}$
So we have f'' such that $G(D, f'', w, \phi)$ and
$(u, s) \in f''^*$ and s maximal in f''
f' and f'' have the same maximal relationships and domain
$v = w(s)$
$v = v'$
So we can substitute y for y' and v for v'
Since f and f' have the same maximal relationships, and
since f' and f'' have the same maximal relationships,
we know that f and f'' have the same maximal relationships
Since $(x, r) \in f'^*$ and r maximal in f', we know $(x, r) \in f''^*$ and r maximal in f''
Now case analyze on whether $r = s$:
If $r = s$:

Then we can simplify the remaining program to $\langle\rangle$
Now, note that since $r = s$, $w(r) = w(s)$, and so $y = v$ and $R([y/v], \phi) = \phi$
Hence we can hide D, f'', w to get $H(\phi)$
If $r \neq s$:
Now, case analyze on the ranks m and n:
If $m < n$:
We can simplify the if-then-else, and continue
letv $\langle\rangle = [s := \mathsf{Root}(y, n)]$ in
$r := \mathsf{Child}(s)$
Now take w' to be the restriction of $[w|s : y]$ to exclude r
Now take $g = [f''|r : s]$
Since $w(r) = y$, and w is injective, w' is still injective
From the ramification lemma, we know $D, g, w' \models R([y/y, y/v], \phi)$
This is the same as $R([y/v], \phi)$
The two updates ensure that $heap(D, g, w')$ hold
Hiding D, g, w', we get $H(R([y/v], \phi)$
If $n < m$:
We can simplify the if-then-else, and continue
$s := \mathsf{Child}(r)$
Now take w' to be the restriction of $[w|r : y]$ to exclude s
Now take $g = [f''|s : r]$
By the ramification lemma, $D, g, w' \models R([y/y, y/v], \phi)$
This is the same as $R([y/v], \phi)$
The update ensures that $heap(D, g, w')$ holds
Hiding D, g, w', we get $H(R([y/v], \phi)$
If $m = n$:
letv $\langle\rangle = [r := \mathsf{Root}(y, m + 1)]$ in
$s := \mathsf{Child}(r)$
Now take w' to be the restriction of $[w|r : y]$ to exclude s
Now take $g = [f''|s : r]$
By the ramification lemma, $D, g, w' \models R([y/y, y/v], \phi)$
This is the same as $R([y/v], \phi)$
The update ensures that $heap(D, g, w')$ holds
Hiding D, g, w', we get $H(R([y/v], \phi)$

□

Acknowledgements I would like to thank Peter O'Hearn for pointing out the connection of our work with the ramification problem of AI.

Bibliography

[1] A. Banerjee and D.A. Naumann. Representation independence, confinement and access control [extended abstract]. *ACM SIGPLAN Notices*, 37(1):166–177, 2002. ISSN 0362-1340.

[2] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, pages 387–411, 2008.

[3] M. Barnett and D.A. Naumann. Friends Need a Bit More: Maintaining Invariants Over Shared State. *Mathematics of Program Construction (MPC)*, 2004.

[4] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. *Proceedings CASSIS 2004*, 2005.

[5] N. Benton. Abstracting allocation. In *Computer Science Logic*, pages 182–196. Springer, 2006.

[6] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 29(5):24, 2007. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1275497.1275499.

[7] Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. In Helmut Seidl, editor, *FoSSaCS*, volume 4423 of *LNCS*, pages 93–107. Springer, 2007. ISBN 978-3-540-71388-3.

[8] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: http://doi.acm.org/10.1145/964001.964020.

[9] J. J. Finger. *Exploiting constraints in design synthesis*. PhD thesis, Stanford University, Stanford, CA, USA, 1987.

[10] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/116873.116878.

[11] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[13] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de larithmtique dordre supérieur*. PhD thesis, Université Paris VII, 1972.

[14] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. ISSN 0001-0782.

[15] Gérard Huet. *Resolution d'equations dans les langages d'ordre 1, 2, ..., ω*. PhD thesis, Univ. de Paris VII, Paris, France, 1976.

[16] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: Proof rules and implementation. *Proceedings FTfJP*, 2005.

[17] S.P. Jones and S.L.P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[18] N.J. Kokholm. An extended library of collection classes for .NET. Master's thesis, IT University of Copenhagen, Copenhagen, Denmark, 2004.

[19] Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying event-driven programs using ramified frame properties. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '10, pages 63–76, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-891-9. doi: 10.1145/1708016.1708025. URL `http://doi.acm.org/10.1145/1708016.1708025`.

[20] Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06*, pages 83–86, New York, NY, USA, 2006. ACM. ISBN 1-59593-586-X. doi: http://doi.acm.org/10.1145/1181195.1181213.

[21] G.T. Leavens, A.L. Baker, and C. Ruby. JML: a Java modeling language. *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.

[22] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *European Symposium on Programming (ESOP)*, pages 80–94. Springer, 2007. ISBN 978-3-540-71314-2. doi: http://dx.doi.org/10.1007/978-3-540-71316-6_7.

[23] Barbara H. Liskov and Jeannette M. Wing. Behavioural subtyping using invariants and constraints. In *Formal Methods for Distributed Processing: a Survey of Object-Oriented Approaches*, pages 254–280. Cambridge University Press, New York, NY, USA, 2001. ISBN 0-521-77184-6.

[24] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[25] R. Milner. Implementation and applications of Scott's logic for computable functions. *ACM sigplan notices*, 7(1):1–6, 1972. ISSN 0362-1340.

[26] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):470–502, 1988.

[27] Eugenio Moggi. Notions of computation and monads. *Information and Compututation*, 93 (1):55–92, 1991. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/0890-5401(91)90052-4.

[28] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable

ADTs in Hoare type theory. *European Symposium on Programming (ESOP)*, 2007.

[29] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. *ACM SIGPLAN Notices*, 45(1):261–274, 2010. ISSN 0362-1340.

[30] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings ICFP*, pages 62–73, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: http://doi.acm.org/10.1145/1159803.1159812.

[31] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *In Proceedings of International Conference on Functional Programming'08*, 2008.

[32] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557 – 580, 2007. ISSN 0890-5401. doi: DOI: 10.1016/j.ic.2006.08.009. Special Issue: 16th International Conference on Rewriting Techniques and Applications.

[33] M. Parkinson. Class invariants: The end of the road. *Proceedings IWACO*, 2007.

[34] Matthew Parkinson. *Local Reasoning for Java*. PhD in Computer Science, University of Cambridge, August 2005.

[35] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *SIGPLAN Not.*, 40(1):247–258, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1047659.1040326.

[36] Matthew Parkinson and Dino Distefano. jStar: Towards practical verification for Java. In *OOPSLA*, 2008, to appear.

[37] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. ISSN 0960-1295. doi: http://dx.doi.org/10.1017/S0960129501003322.

[38] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Get by anonymous ftp from `ftp.dcs.ed.ac.uk` in `pub/bcp/friendly.ps.Z`. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[39] C. Pierik, D. Clarke, and F.S. de Boer. Creational Invariants. *Proceedings FTfJP*, 2004.

[40] J.C. Reynolds. The essence of Algol. In *ALGOL-like Languages, Volume 1*, page 88. Birkhauser Boston Inc., 1997. ISBN 0817638806.

[41] John C. Reynolds. An introduction to specification logic. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, page 442, London, UK, 1984. Springer-Verlag. ISBN 3-540-12896-4.

[42] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9.

[43] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame

rules for higher-order store. In *Computer Science Logic*, pages 440–454. Springer, 2009.

[44] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 351–368. ACM, 2007. ISBN 978-1-59593-786-5.

[45] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

[46] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[47] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. ISBN 978-3-540-74406-1.

[48] J. Vouillon and P.A. Melliès. Semantic types: A fresh look at the ideal model for types. *ACM SIGPLAN Notices*, 39(1):52–63, 2004. ISSN 0362-1340.