Logical Methods in Computer Science
Vol. 2 (5:1) 2006, pp. 1–33
www.lmcs-online.org

Submitted Dec. 21, 2005
Published Nov. 3, 2006

SEMANTICS OF SEPARATION-LOGIC TYPING AND HIGHER-ORDER FRAME RULES FOR ALGOL-LIKE LANGUAGES *

LARS BIRKEDAL [a], NOAH TORP-SMITH [b], AND HONGSEOK YANG [c]

[a,b] IT University of Copenhagen, Denmark
e-mail address: {birkedal,noah}@itu.dk

[c] Seoul National University, Korea
e-mail address: hyang@ropas.snu.ac.kr

Abstract. We show how to give a coherent semantics to programs that are well-specified in a version of separation logic for a language with higher types: idealized algol extended with heaps (but with immutable stack variables). In particular, we provide simple sound rules for deriving higher-order frame rules, allowing for local reasoning.

1. Introduction

Separation logic [17, 16, 5, 14, 8, 4] is a Hoare-style program logic, and variants of it have been applied to prove correct interesting pointer algorithms such as copying a dag, disposing a graph, the Schorr-Waite graph algorithm, and Cheney's copying garbage collector. The main advantage of separation logic compared to ordinary Hoare logic is that it facilitates *local reasoning*, formalized via the so-called *frame rule* using a connective called *separating conjunction*. The development of separation logic has mostly focused on *low-level* languages with heaps and pointers, although in recent work [9] it was shown how to extend separation logic to a language with a simple kind of procedures, and a second-order frame rule was proved sound.

Our aim here is to extend the study of separation logic to *high-level* languages, in particular to *higher-order* languages, in such a way that a wide collection of frame rules are sound, thus allowing for local reasoning in the presence of higher-order procedures. For concreteness, we choose to focus on the language of idealized algol extended with heaps and pointers and we develop a semantics for this language in which all commands and procedures are appropriately local. Our approach is to refine the type system of idealized algol extended with heaps, essentially by making specifications be types, and give semantics to well-specified programs. Thus we develop a *separation-logic type system* for idealized algol extended with heaps. It is a dependent type theory and the types include Hoare triples, rules corresponding to the rules of separation logic, and subtyping rules formalizing higher-order versions of the frame rule of separation logic.

2000 ACM Subject Classification: F.3, D.3.
Key words and phrases: Program Verification, Separation Logic.
* An extended abstract of this paper appeared at LICS'05.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-2 (5:1) 2006

Our type system is related to modern proposals for type systems for low-level imperative languages, such as TAL [7], in that types may express state changes (since they include forms of Hoare triples as types). The type system for TAL was proved sound using an operational semantics. We provide a soundness proof of our type system using a denotational semantics which we, moreover, formally relate to the standard semantics for idealized algol [11, 18]. The denotational semantics of a well-typed program is given by induction on its typing derivation and the relation to the standard semantics for idealized algol is then used to prove that the semantics is *coherent* (i.e., is independent of the chosen typing derivation). We should perhaps stress that soundness is not a trivial issue: Reynolds has shown [9] that already the soundness of the second-order frame rule is tricky, by proving that if a proof system contains the second-order frame rule and the conjunction rule, together with the ordinary frame rule and the rule of Consequence, then the system becomes inconsistent. The semantics of our system proves that if we drop the conjunction rule, then we get soundness of all higher-order frame rules, including the second-order one. We also show how to get soundness of all higher-order frame rules *without* dropping the conjunction rule, by instead restricting attention to so-called *precise* predicates (see Section 5).

In idealized algol, variables are allocated on a stack and they are mutable (i.e., one can assign to variables). We only consider *immutable* variables (as in the ML programming language) for simplicity. The reason for this choice is that all mutation then takes place in the heap and thus we need not bother with so-called *modifies clauses* on frame rules, which become complicated to state already for the second-order frame rule [9].

We now give an intuitive overview of the technical development. Recall that the standard semantics of idealized algol is given using the category CPO of pointed complete partial orders and continuous functions. Thus types are interpreted as pointed complete partial orders and terms (programs) are interpreted as continuous functions. The semantics of our refined type system is given by refining the standard semantics. A type θ in our refined type system specifies which elements of the "underlying" type in the standard semantics satisfy the specification corresponding to θ and are appropriately local (to ensure soundness of the frame rules), that is, it "extracts" those elements. Moreover, the semantics also equates elements, which cannot be distinguished by clients, that is, it quotients some of the extracted elements. Corresponding to these two aspects of the semantics we introduce two categories, $\mathcal{C}$ and $\mathcal{D}$, where $\mathcal{C}$ just contains the extracted elements and $\mathcal{D}$ is a quotient of $\mathcal{C}$. Thus there is a faithful functor from $\mathcal{C}$ to CPO and a full functor from $\mathcal{C}$ to $\mathcal{D}$. We show that the categories $\mathcal{C}$ and $\mathcal{D}$ are cartesian closed and have additional structure to interpret the higher-order frame rules, and that the mentioned functors preserve all this structure. The semantics of our type system is then given in the category $\mathcal{D}$ and the functors relating $\mathcal{C}$, $\mathcal{D}$, and CPO are then used to prove coherence of the semantics. In fact, as mentioned above, our type system is a *dependent* type theory, with dependent product type $\Pi_i\theta$ intuitively corresponding to the specification given by universally quantifying i in the specification corresponding to θ (the usual Curry-Howard correspondence). For this reason the semantics is really not given in $\mathcal{D}$ but rather in the family fibration $\mathit{Fam}(\mathcal{D}) \to \mathsf{Set}$ over $\mathcal{D}$.

The remainder of this paper is organized as follows. In Section 2, we define the storage model and assertion language used in this paper, thus setting the stage for our model. In Section 3, we provide the syntax of the version of idealized algol we use in this paper. In particular, we introduce our separation-logic type system, which includes an extended subtype relation. We also include two extended examples of typings in our typing system, one of which exemplifies the use of a third-order frame rule. In Section 4, we present the main contribution of the paper, a model which allows a sound interpretation, which we also show to be coherent and in harmony with the standard semantics. For simplicity, we omit treatment of the conjunction rule in Sections 3 and 4 — in Section 5 we show how to treat the conjunction rule. In the last sections we give pointers to related and future work, and conclude.

An extended abstract of this paper was presented at the LICS 2005 conference. Compared to the conference paper, the present paper includes proofs, more detailed examples of the use of the typing system, and a treatment of the conjunction rule.

2. Storage Model and Assertion Language

We use the usual storage model of separation logic with one minor modification: we make explicit the shape of stack storage. Let $\mathsf{Ids} = \{i, j, \ldots\}$ be a countably infinite set of variables, and let Δ range over finite subsets of Ids. We use the following semantic domains:

$$\begin{array}{rcrcl} & & \mathit{Loc} & \stackrel{\text{def}}{=} & \mathit{PositiveInt}, \\ & & \mathit{Val} & \stackrel{\text{def}}{=} & \mathit{Int}, \\ \eta & \in & [\![\Delta]\!] & \stackrel{\text{def}}{=} & \Delta \to \mathit{Val}, \\ h & \in & \mathit{Heap} & \stackrel{\text{def}}{=} & \mathit{Loc} \rightharpoonup_{\mathsf{fin}} \mathit{Val}, \\ (\eta, h) & \in & \mathit{State}(\Delta) & \stackrel{\text{def}}{=} & [\![\Delta]\!] \times \mathit{Heap}. \end{array}$$

In this storage model, locations are positive integers, so that they can be manipulated by arithmetic operations. The set Δ models the set of variables in scope, and an element η in $[\![\Delta]\!]$ specifies the values of those stack variables. We sometimes call η an *environment* instead of a *stack*, in order to emphasize that all variables are immutable. An element h in *Heap* denotes a heap; the domain of h specifies the set of allocated cells, and the actual action of h determines the contents of those allocated cells. We recall the disjointness predicate $h\#h'$ and the (partial) heap combination operator $h \cdot h'$ from separation logic. The predicate $h\#h'$ means that $\mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset$; and, $h \cdot h'$ is defined only for such disjoint heaps h and h', and in that case, it denotes the combined heap $h \cup h'$.

Properties of states are expressed using the assertion language of classical separation logic [17]: [1]

$$\begin{array}{lcl} E & ::= & i \mid 0 \mid 1 \mid E + E \mid E - E, \\ P & ::= & E = E \mid E \mapsto E \mid \mathsf{emp} \mid P * P \mid \mathsf{true} \mid P \wedge P \mid P \vee P \mid \neg P \mid \forall i.\, P. \mid \exists i.\, P. \end{array}$$

The assertion $E \mapsto E'$ means that the current heap has only one cell E and, moreover, that the content of the cell is E'. When we do not care about the contents, we write $E \mapsto -$; formally, this is an abbreviation of $\exists i.\, E \mapsto i$ for some i not occurring in E. The next two assertions, emp and $P * Q$, are the most interesting features of this assertion language. The empty predicate emp means that the current heap is empty, and the separating conjunction $P * Q$ means that the current heap can be partitioned into two parts, one satisfying P and another satisfying Q.

As in the storage model, we make explicit which set of free variables we are considering an expression or an assertion under. Thus, letting fv be a function that takes an expression or an assertion and returns the set of free variables, we often write assertions as $\Delta \vdash P$ to

[1] The assertion language of separation logic also contains the separating implication $-\!\!*$. Since that connective does not raise any new issues in connection with the present work, we omit it here.

indicate that $\mathsf{fv}(P) \subseteq \Delta$, and that P is currently being considered for environments of the shape Δ. Likewise, we often write $\Delta \vdash E$ for expressions.

The interpretations of an expression $\Delta \vdash E$ and an assertion $\Delta \vdash P$ are of the forms

$$[\![\Delta \vdash E]\!] : [\![\Delta]\!] \to \mathit{Val}, \qquad [\![\Delta \vdash P]\!] : [\![\Delta]\!] \to \mathcal{P}(\mathit{Heap}).$$

The interpretation of expressions is standard, just like that of assertions. We include part of the definition of the interpretation of assertions here.

$$\begin{array}{rcl} [\![\Delta \vdash E \mapsto E']\!]_\eta & = & \mathsf{if}\ ([\![\Delta \vdash E]\!]_\eta \leq 0)\ \mathsf{then}\ \emptyset\ \mathsf{else}\ \big\{[[\![\Delta \vdash E]\!]_\eta {\to} [\![\Delta \vdash E']\!]_\eta]\big\}, \\ [\![\Delta \vdash \mathsf{emp}]\!]_\eta & = & \{[]\}, \\ [\![\Delta \vdash P * P']\!]_\eta & = & \{h \cdot h' \mid h \# h' \wedge h \in [\![\Delta \vdash P]\!]_\eta \wedge h' \in [\![\Delta \vdash P']\!]_\eta\}, \\ [\![\Delta \vdash \forall i.\ P]\!]_\eta & = & \{h \mid \forall n \in \mathit{Val}.\ h \in [\![\Delta \cup \{i\} \vdash P]\!]_{\eta[i \to n]}\}. \end{array}$$

3. Programming Language

The programming language is Reynolds's idealized algol [18] adapted for "separation-logic typing." It is a call-by-name typed lambda calculus, extended with heap operations, dependent functions, and Hoare-triple types. As explained in the introduction, we only consider immutable variables.

The types of the language are defined as follows. We write $\Delta \vdash \theta : \mathsf{Type}$ for a type θ in context Δ. The set of types is defined by the following inference rules (in which P and Q range over assertions):

$$\frac{\Delta \vdash P \quad \Delta \vdash Q}{\Delta \vdash \{P\}\text{-}\{Q\} : \mathsf{Type}} \qquad \frac{\Delta \vdash \theta : \mathsf{Type} \quad \Delta \vdash P}{\Delta \vdash \theta \otimes P : \mathsf{Type}}$$

$$\frac{\Delta \cup \{i\} \vdash \theta : \mathsf{Type}}{\Delta \vdash \Pi_i \theta : \mathsf{Type}}\ (i \notin \Delta) \qquad \frac{\Delta \vdash \theta : \mathsf{Type} \quad \Delta \vdash \theta' : \mathsf{Type}}{\Delta \vdash \theta \to \theta' : \mathsf{Type}}$$

Note that the types are *dependent types*, in that they may depend on variables i (see the first rule above). One way to understand a type is to read it as a specification for terms, i.e., through the Curry-Howard correspondence. A Hoare-triple type $\{P\}\text{-}\{Q\}$ is a direct import from separation logic; it denotes a set of commands c that satisfy the Hoare triple $\{P\}c\{Q\}$. An invariant extension $\theta \otimes P$ is satisfied by a term M if and only if for one part of the heap, the behavior of M satisfies θ and for the other part of the heap, M maintains the invariant P. We remark that $\theta \otimes P$ allows M to transfer cells between the θ-part of the heap and the P-part of the heap. For instance, $\{P\}\text{-}\{Q\} \otimes P_0$ intuitively consists of the following commands c: given an input state satisfying $P * P_0$, so that the input state may be split into a P-part and a P_0-part, command c changes these two parts, sometimes transferring cells between the two, such that in the end, the P-part satisfies Q and the P_0-part satisfies P_0.

The type $\Pi_i \theta$ is a dependent product type, as in standard dependent type theory (under Curry-Howard it corresponds to the specification given by universally quantifying i in the specification corresponding to θ). Intuitively, $\Pi_i \theta$ denotes functions from integers such that given an integer n, they return a value satisfying $\theta[n/i]$. For example, the type $\Pi_i \{j \mapsto -\}\text{-}\{j \mapsto i!\}$ specifies a factorial function that computes the factorial of i and stores the result in the heap cell j.

The pre-terms of the language are given by the following grammar:

$$\begin{aligned} M ::= {} & x \mid \lambda x\colon \theta.M \mid MM \mid \lambda i.M \mid ME \\ \mid {} & \mathsf{fix}\, M \mid \mathsf{ifz}\, E\, M\, M \mid \mathsf{skip} \mid M; M \\ \mid {} & \mathsf{let}\, i = \mathsf{new}\, \mathsf{in}\, M \mid \mathsf{free}(E) \mid [E] := E \mid \mathsf{let}\, i = [E]\, \mathsf{in}\, M, \end{aligned}$$

where E is an integer expression defined in Section 2. The language has the usual constructs for a higher-order imperative language with heap operations, but it has two distinct features. First, it treats the integer expressions as "second class": the terms M never have the integer type, and all integer expressions inside a term are from the separate grammar for E defined in Section 2. Second, no "integer variables" i can be modified in this language; only heap cells can be modified. Note that the language has two forms of abstraction and application, one for general terms and the other for integer expressions. A consequence of this stratification is that all integer expressions terminate, because the grammar for E does not contain the recursion operator.

The language has four heap operations. Command $\mathsf{let}\, i = \mathsf{new}\, \mathsf{in}\, M$ allocates a heap cell, binds i to the address of the allocated cell, and executes the command M.[2] An allocated cell i can be disposed by $\mathsf{free}(i)$. The remaining two commands access the content of a cell. The command $[i] := E'$ changes the content of cell i by E'; and $\mathsf{let}\, j = [i]\, \mathsf{in}\, M$ reads the content of cell i, binds j to the read value, and executes M. Note that the allocation and lookup commands involve the "continuation", and make the bound variable available in the continuation; such indirect-style commands are needed because all variables are immutable.

In this paper, we assume a hygiene condition on integer variables i, in order to avoid the (well-known) issue of variable capturing. That is, we assume that no terms or types in the paper use a single symbol i for more than one bound variables, or for a bound variable and a free variable at the same time.

The typing rules of the language decide a judgment of the form $\Gamma \vdash_\Delta M : \theta$, where Γ is a list of type assignments to identifiers $\Gamma = x_1\colon \theta_1, \ldots, x_n\colon \theta_n$, and where the set Δ contains all the free variables appearing in Γ, M, θ.

The type system is shown in Figures 1 and 2. For notational simplicity we have omitted some obvious side-conditions of the form $\Delta \vdash \theta : \mathsf{Type}$ which ensure that, for a judgment $\Gamma \vdash_\Delta M : \theta$, the set Δ always contains all the free variables appearing in Γ, M, θ, and that the type assignment Γ is always well-formed. There are three classes of rules. The first class consists of the rules from the simply typed lambda calculus extended with dependent product types and recursion. The second class consists of the rules for the imperative constructs, all of which come from separation logic. The last class consists of the subsumption rule based on the subtype relation $\preceq_\Delta$, which is the most interesting part of our type system. The proof rules for $\preceq_\Delta$ are shown in Figure 2. They define a preorder between types with free variables in Δ, and include all the usual structural subtyping rules in the chapter 15 of [13]. The rules specific to our system are: the covariant structural rule for $\theta \otimes P$; the encoding of Consequence in Hoare logic; the generalized frame rule that adds an invariant to all types; and the distribution rules for an added invariant assertion.

The generalized frame rule, $\theta \preceq_\Delta \theta \otimes P_0$, means that if a program satisfies θ and an assertion P_0 does not "mention" any cells described by θ, then the program preserves P_0. Note that this rule indicates that the types in our system are *tight* [5, 17]: if a program satisfies θ, it can only access heap cells "mentioned" in θ. This is why an assertion P_0 for

[2] We consider single-cell allocation only in order to simplify the presentation; it is straightforward to adapt our results to a language with allocation of n consecutive cells.

$$\frac{}{\Gamma, x\colon \theta \vdash_\Delta x : \theta} \qquad \frac{\Gamma, x\colon \theta \vdash_\Delta M : \theta'}{\Gamma \vdash_\Delta \lambda x\colon \theta.M : \theta \to \theta'} \qquad \frac{\Gamma \vdash_\Delta M : \theta' \to \theta \quad \Gamma \vdash_\Delta M' : \theta'}{\Gamma \vdash_\Delta M M' : \theta}$$

$$\frac{\Gamma \vdash_{\Delta \cup \{i\}} M : \theta'}{\Gamma \vdash_\Delta \lambda i.M : \Pi_i \theta'} \, (i \notin \mathsf{fv}(\Gamma, \Delta)) \qquad \frac{\Gamma \vdash_\Delta M : \Pi_i \theta \quad \Delta \vdash E}{\Gamma \vdash_\Delta M E : \theta[E/i]} \qquad \frac{\Gamma \vdash_\Delta M : \theta \to \theta}{\Gamma \vdash_\Delta \mathsf{fix}\, M : \theta}$$

$$\frac{\Gamma \vdash_\Delta M : \{P \wedge E{=}0\}\text{-}\{Q\} \quad \Gamma \vdash_\Delta M' : \{P \wedge E{\neq}0\}\text{-}\{Q\}}{\Gamma \vdash_\Delta \mathsf{ifz}\, E\, M\, M' : \{P\}\text{-}\{Q\}} \qquad \frac{\Delta \vdash P}{\Gamma \vdash_\Delta \mathsf{skip} : \{P\}\text{-}\{P\}}$$

$$\frac{\Gamma \vdash_\Delta M : \{P\}\text{-}\{P'\} \quad \Gamma \vdash_\Delta M' : \{P'\}\text{-}\{Q\}}{\Gamma \vdash_\Delta (M; M') : \{P\}\text{-}\{Q\}} \qquad \frac{\Delta \vdash E}{\Gamma \vdash_\Delta \mathsf{free}(E) : \{E {\mapsto} -\}\text{-}\{\mathsf{emp}\}}$$

$$\frac{\Gamma \vdash_{\Delta \cup \{i\}} M : \{i {\mapsto} - * P\}\text{-}\{Q\}}{\Gamma \vdash_\Delta \mathsf{let}\, i = \mathsf{new}\,\mathsf{in}\, M : \{P\}\text{-}\{Q\}} \, (i \notin \mathsf{fv}(\Gamma, \Delta, P, Q)) \qquad \frac{\Delta \vdash E \quad \Delta \vdash E'}{\Gamma \vdash_\Delta [E]{:=}E' : \{E {\mapsto} -\}\text{-}\{E {\mapsto} E'\}}$$

$$\frac{\Gamma \vdash_{\Delta \cup \{i\}} M : \{E {\mapsto} i * P\}\text{-}\{Q\}}{\Gamma \vdash_\Delta \mathsf{let}\, i{=}[E]\,\mathsf{in}\, M : \{\exists i.\, E {\mapsto} i * P\}\text{-}\{Q\}} \, (i \notin \mathsf{fv}(\Gamma, \Delta, E, Q)) \qquad \frac{\Gamma \vdash_\Delta M : \theta \quad \theta \preceq_\Delta \theta'}{\Gamma \vdash_\Delta M : \theta'}$$

Figure 1: Typing Rules

Inference Rules

$$\frac{\theta \preceq_\Delta \theta \quad \theta' \preceq_\Delta \theta''}{\theta \preceq_\Delta \theta''} \qquad \frac{\theta_0' \preceq_\Delta \theta_0 \quad \theta_1 \preceq_\Delta \theta_1'}{(\theta_0 \to \theta_1) \preceq_\Delta (\theta_0' \to \theta_1')} \qquad \frac{\theta \preceq_{\Delta \cup \{i\}} \theta'}{(\Pi_i \theta) \preceq_\Delta (\Pi_i \theta')} \, (i \notin \Delta)$$

$$\frac{\theta \preceq_\Delta \theta'}{(\theta \otimes P) \preceq_\Delta (\theta' \otimes P)} \qquad \frac{\forall \eta \in [\![\Delta]\!].\, [\![P']\!]_\eta \subseteq [\![P]\!]_\eta \wedge [\![Q]\!]_\eta \subseteq [\![Q']\!]_\eta}{\{P\}\text{-}\{Q\} \preceq_\Delta \{P'\}\text{-}\{Q'\}}$$

Axioms

$$\theta \preceq_\Delta \theta \qquad \theta \preceq_\Delta \theta \otimes P$$
$$(\{P\}\text{-}\{Q\}) \otimes P_0 \simeq_\Delta \{P * P_0\}\text{-}\{Q * P_0\} \qquad (\Pi_i \theta) \otimes P \simeq_\Delta \Pi_i(\theta \otimes P) \;\; (\text{when } i {\notin} \Delta)$$
$$(\theta \otimes Q) \otimes P \simeq_\Delta \theta \otimes (Q * P) \qquad (\theta \to \theta') \otimes P \simeq_\Delta (\theta \otimes P \to \theta' \otimes P)$$

Figure 2: Rules for the Subtyping Relation $\preceq_\Delta$

"unmentioned" cells is preserved by the program. For instance, if a program M has a type of the form

$$\theta_1 \to \ldots \to \theta_n \to \{P\}\text{-}\{Q\},$$

the tightness of the type says that all the cells that M can directly access must appear in the pre-condition P. Thus, if no cells in an assertion P_0 appear in P, program M maintains P_0, as long as argument procedures maintain it. Such a fact can, indeed, be inferred by the generalized frame rule together with the distribution rules:

$$\begin{array}{ll}
 & \theta_1 \to \ldots \to \theta_n \to \{P\}\text{-}\{Q\} \\
\preceq_\Delta & \quad (\because\ \theta \preceq_\Delta \theta \otimes P_0) \\
 & (\theta_1 \to \ldots \to \theta_n \to \{P\}\text{-}\{Q\}) \otimes P_0 \\
\preceq_\Delta & \quad (\because\ (\theta \to \theta') \otimes P_0 \preceq_\Delta (\theta \otimes P_0 \to \theta' \otimes P_0)) \\
 & (\theta_1 \otimes P_0 \to \ldots \to \theta_n \otimes P_0 \to \{P\}\text{-}\{Q\} \otimes P_0) \\
\preceq_\Delta & \quad (\because\ \{P\}\text{-}\{Q\} \otimes P_0 \preceq_\Delta \{P * P_0\}\text{-}\{Q * P_0\}) \\
 & (\theta_1 \otimes P_0 \to \ldots \to \theta_n \otimes P_0 \to \{P * P_0\}\text{-}\{Q * P_0\}).
\end{array}$$

The generalized frame rule, the distribution rules, and the structural subtyping rule for function types all together give many interesting higher-order frame rules, including the second-order frame rule. The common mechanism for obtaining such a rule is: first, add an invariant assertion by the generalized frame rule, and then, propagate the added assertion all the way down to a base triple type by the distribution rules. The structural subtyping rule for the function type allows us to apply this construction for a sub type-expression in an appropriate covariant or contravariant way. For instance, we can derive a third-order frame rule as follows:

$$\begin{array}{ll}
 & (\{P_1\}\text{-}\{Q_1\} \to \{P_2\}\text{-}\{Q_2\}) \to \{P_3\}\text{-}\{Q_3\} \\
\preceq_\Delta & \quad (\because\ \theta \preceq_\Delta \theta \otimes P) \\
 & \Big((\{P_1\}\text{-}\{Q_1\} \to \{P_2\}\text{-}\{Q_2\}) \to \{P_3\}\text{-}\{Q_3\}\Big) \otimes P \\
\preceq_\Delta & \quad (\because\ (\theta \to \theta') \otimes P \simeq_\Delta (\theta \otimes P \to \theta' \otimes P)) \\
 & (\{P_1\}\text{-}\{Q_1\}{\otimes}P \to \{P_2\}\text{-}\{Q_2\}{\otimes}P) \to \{P_3\}\text{-}\{Q_3\}{\otimes}P \\
\preceq_\Delta & \quad (\because\ \text{structural subtyping}) \\
 & (\{P_1\}\text{-}\{Q_1\}{\otimes}P \to \{P_2\}\text{-}\{Q_2\}) \to \{P_3\}\text{-}\{Q_3\}{\otimes}P \\
\preceq_\Delta & \quad (\because\ \{P_0\}\text{-}\{Q_0\} \otimes P \simeq_\Delta \{P_0 * P\}\text{-}\{Q_0 * P\}) \\
 & (\{P_1{*}P\}\text{-}\{Q_1{*}P\} \to \{P_2\}\text{-}\{Q_2\}) \to \{P_3{*}P\}\text{-}\{Q_3{*}P\}.
\end{array}$$

3.1. **Example Proofs in the Type System.** We illustrate how the type system works, with the verification of two example programs.

The first example is a procedure that disposes a linked list. With this example we demonstrate how a standard proof in separation logic yields a typing in our type system. Let $\mathsf{lst}(i)$ be an assertion which expresses that the heap contains a linked list i terminating with 0, and all the cells in the heap are in the list.[3] We define a procedure *Dlist* for list disposal as follows:

$$\mathit{Dlist} \stackrel{\text{def}}{=} \mathsf{fix}\ \lambda f\colon (\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}).\, \Big(\lambda i.\ \mathsf{ifz}\ i\ \big(\mathsf{skip}\big)\ \big(\mathsf{let}\ j = [i]\ \mathsf{in}\ f(j); \mathsf{free}(i)\big)\Big).$$

The program *Dlist* takes a linked list i, and disposes the list, first the tail and then the head of the list.

[3]Formally, $\mathsf{lst}(i)$ is the (parameterized) assertion that satisfies the equivalence:

$$\mathsf{lst}(i) \iff (i = 0 \land \mathsf{emp}) \lor (\exists j.\, (i \mapsto j) * \mathsf{lst}(j))$$

— it can be defined as the minimal fixed point, expressible in higher-order separation logic [3].

We derive the typing judgment $\vdash_{\{\}} \mathit{Dlist} : (\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\})$. Note that this derivation captures the correctness of *Dlist*, because the judgment means that when *Dlist* is given a linked list i as argument, then it disposes all the cells in the list.

The main part of the derivation is a proof tree for the false branch of the conditional statement. Let Γ be $f\colon (\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\})$. The proof tree for the false branch is given below:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\{i,j\}} f : \Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}}}{\Gamma \vdash_{\{i,j\}} f(j) : \{\mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}}}{\Gamma \vdash_{\{i,j\}} f(j) : \{i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{i{\mapsto}j\}}\,1 \qquad \dfrac{\dfrac{}{\Gamma \vdash_{\{i,j\}} \mathsf{free}(i) : \{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}}}{\Gamma \vdash_{\{i,j\}} \mathsf{free}(i) : \{i{\mapsto}j\}\text{-}\{\mathsf{emp}\}}\,2}{\dfrac{\Gamma \vdash_{\{i,j\}} (f(j); \mathsf{free}(i)) : \{i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}}{\dfrac{\Gamma \vdash_{\{i\}} (\mathsf{let}\, j{=}[i]\, \mathsf{in}\, f(j); \mathsf{free}(i)) : \{\exists j.\, i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}}{\Gamma \vdash_{\{i\}} (\mathsf{let}\, j{=}[i]\, \mathsf{in}\, f(j); \mathsf{free}(i)) : \{\mathsf{lst}(i) \wedge i{\neq}0\}\text{-}\{\mathsf{emp}\}}\,3}}$$

Most of the steps in this tree use syntax-directed rules, such as those for the sequential composition and procedure application. The only exceptions are the steps marked by 1, 2 and 3, where we apply the subsumption rule. These steps express structural rules in separation logic. Step 1 is an instance of the ordinary frame rule, and attaches the invariant $(i{\mapsto}j)$ to the pre- and post-conditions of the triple type $\{\mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}$. The other steps are an instance of Consequence. Step 2 strengthens the pre-condition of $\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}$, and step 3 replaces the pre-condition of $\{\exists j. i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}$ by the equivalent assertion $\mathsf{lst}(j) \wedge i{\neq}0$. In the tree above, we have not shown how to derive the necessary subtype relations in 1, 2 and 3. They are straightforward to derive:

$$\begin{aligned} \{\mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\} &\preceq_{\{i,j\}} (\{\mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}) \otimes i{\mapsto}j && (\because\ \theta \ \preceq_\Delta\ \theta \otimes R)\\ &\preceq_{\{i,j\}} \{\mathsf{lst}(j) * i{\mapsto}j\}\text{-}\{\mathsf{emp} * i{\mapsto}j\} && (\because\ \{P\}\text{-}\{Q\} \otimes R \ \simeq_\Delta\ \{P * R\}\text{-}\{Q * R\})\\ &\preceq_{\{i,j\}} \{i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{i{\mapsto}j\} && (\because\ \forall\eta.\ [\![P{*}Q]\!]_\eta = [\![Q{*}P]\!]_\eta \wedge [\![\mathsf{emp}{*}P]\!]_\eta = [\![P]\!]_\eta) \end{aligned}$$

$$\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\} \preceq_{\{i,j\}} \{i{\mapsto}j\}\text{-}\{\mathsf{emp}\}\ (\because\ \forall\eta.\ [\![i{\mapsto}j]\!]_\eta \subseteq [\![i{\mapsto}-]\!]_\eta)$$

$$\{\exists j.\, i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\} \preceq_{\{i\}} \{\mathsf{lst}(i) \wedge i{\neq}0\}\text{-}\{\mathsf{emp}\}\ (\because \forall\eta.\ [\![\exists j. i{\mapsto}j{*}\mathsf{lst}(j)]\!]_\eta = [\![\mathsf{lst}(i) \wedge i{\neq}0]\!]_\eta).$$

The complete derivation of $\vdash_{\{\}} \mathit{Dlist} : \Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}$ is shown in Figure 3.

The second example is a client program that uses a randomized memory manager. The verification of this program demonstrates the use of a third-order frame rule.

The randomized memory manager is a module with two methods, *Malloc* for allocating a cell and *Mfree* for deallocating a cell. The memory manager maintains a free list whose starting address is stored in the cell l. When *Malloc* is called, the module first checks this free list $[l]$. If the free list is not empty, *Malloc* takes one cell from the list and returns it to the client. Otherwise, *Malloc* makes a system call, obtains a new cell from the operating system, and returns this cell to the client. When *Mfree* is called to deallocate cell i, the randomized memory manager first flips a coin. Then, depending on the result of the coin, it either adds the cell i to the free list or returns the cell to the operating system. Note that randomization is used only in *Mfree*. We will focus on the method *Mfree*.

Let $\mathsf{inv}(l)$ be the assertion $\exists l'. (l{\mapsto}l') * \mathsf{lst}(l')$, which expresses that cell l stores the starting address of a linked list. The following program implements the *Mfree* method of

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\Gamma \vdash_{\{i,j\}} f{:}\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}}}{\Gamma \vdash_{\{i,j\}} f(j){:}\{\mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}}
}{\Gamma \vdash_{\{i,j\}} f(j){:}\{i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{i{\mapsto}j\}}\,1
\quad
\dfrac{\dfrac{}{\Gamma \vdash_{\{i,j\}} \mathsf{free}(i){:}\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}}}{\Gamma \vdash_{\{i,j\}} \mathsf{free}(i){:}\{i{\mapsto}j\}\text{-}\{\mathsf{emp}\}}\,2
}{
\dfrac{\dfrac{\Gamma \vdash_{\{i,j\}} (f(j);\mathsf{free}(i)){:}\{i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}}{\Gamma \vdash_{\{i\}} \mathsf{let}\,j{=}[i]\,\mathsf{in}\,(f(j);\mathsf{free}(i)){:}\{\exists j.\, i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\}}}{\Gamma \vdash_{\{i\}} \mathsf{let}\,j{=}[i]\,\mathsf{in}\,(f(j);\mathsf{free}(i)){:}\{\mathsf{lst}(i) \wedge i{\neq}0\}\text{-}\{\mathsf{emp}\}}\,3
}
\quad
\dfrac{\dfrac{}{\Gamma \vdash_{\{i\}} \mathsf{skip}{:}\{\mathsf{emp}\}\text{-}\{\mathsf{emp}\}}}{\Gamma \vdash_{\{i\}} \mathsf{skip}{:}\{\mathsf{lst}(i) \wedge i{=}0\}\text{-}\{\mathsf{emp}\}}\,4
}{
\dfrac{\dfrac{\Gamma \vdash_{\{i\}} \big(\mathsf{ifz}\ i\ (\mathsf{skip})\ (\mathsf{let}\,j{=}[i]\,\mathsf{in}\,(f(j);\mathsf{free}(i)))\big){:}\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}}{\Gamma \vdash_{\{\}} \big(\lambda i.\,\mathsf{ifz}\ i\ (\mathsf{skip})\ (\mathsf{let}\,j{=}[i]\,\mathsf{in}\,(f(j);\mathsf{free}(i)))\big){:}\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}}}{\vdash_{\{\}} \big(\lambda f.\lambda i.\,\mathsf{ifz}\ i\ (\mathsf{skip})\ (\mathsf{let}\,j{=}[i]\,\mathsf{in}\,(f(j);\mathsf{free}(i)))\big){:}(\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}) \to (\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\})}
}
$$

$$\vdash_{\{\}} \mathit{Dlist}{:}\Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}$$

In the tree, Γ is $f : \Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}$; and at 1 - 4 of the tree, the subsumption rule is used with the following subtype relations:

$$
\begin{array}{rcl}
\{\mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\} & \preceq_{\{i,j\}} & \{i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{i{\mapsto}j\} \\
\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\} & \preceq_{\{i,j\}} & \{i{\mapsto}j\}\text{-}\{\mathsf{emp}\} \\
\{\exists j.\, i{\mapsto}j * \mathsf{lst}(j)\}\text{-}\{\mathsf{emp}\} & \preceq_{\{i\}} & \{\mathsf{lst}(i) \wedge i{\neq}0\}\text{-}\{\mathsf{emp}\} \\
\{\mathsf{emp}\}\text{-}\{\mathsf{emp}\} & \preceq_{\{i\}} & \{\mathsf{lst}(i) \wedge i{=}0\}\text{-}\{\mathsf{emp}\}
\end{array}
$$

Figure 3: Derivation of the Typing Judgment $\vdash_{\{\}} \mathit{Dlist} : \Pi_i\{\mathsf{lst}(i)\}\text{-}\{\mathsf{emp}\}$

the randomized memory manager:

$$
\begin{array}{rcl}
\mathit{Mfree} & : & \Big(\big(\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\}\big) \to \big(\Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}\big)\Big) \otimes \mathsf{inv}(l) \\
\mathit{Mfree} & \stackrel{\text{def}}{=} & \lambda \mathit{cflip}.\ \lambda i.\ \mathit{cflip}(i); \mathsf{let}\, i'{=}[i]\, \mathsf{in}\, \Big(\mathsf{ifz}\ i'\ (\mathsf{free}(i))\ \big(\mathsf{let}\, l'{=}[l]\, \mathsf{in}\, ([i]{:=}l'; [l]{:=}i)\big)\Big)
\end{array}
$$

Note that before disposing cell i, method *Mfree* uses the cell to store the result of flipping a coin by calling *cflip* with i. The declared type of the method *Mfree* has the form $\theta \otimes \mathsf{inv}(l)$. The θ part expresses that the method has the expected behavior externally, and the $\mathsf{inv}(l)$ part indicates that it maintains the module invariant internally. The derivation of the declared type is shown in Figure 4.

We now consider the following client of the randomized memory manager.

$$
\begin{array}{rcl}
\mathit{Rd} & : & \Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \\
\mathit{Rd} & \stackrel{\text{def}}{=} & \lambda i.\ \mathsf{let}\, i' = [i]\, \mathsf{in}\, [i]{:=}i'{+}1 \\
\\
\mathit{Client} & : & (\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}) \to \{j{\mapsto}-\}\text{-}\{\mathsf{emp}\} \\
\mathit{Client} & \stackrel{\text{def}}{=} & \lambda \mathit{mfree}.\ (\mathit{mfree}\ \mathit{Rd}\ j)
\end{array}
$$

The client *Client* takes a “randomized” method *mfree* for deallocating a cell. Then, it instantiates the method with the (degenerate) “random function” *Rd*, and calls the instantiated method to dispose cell j. Suppose that *Client* is “linked” with the *Mfree* of the randomized

Consider Γ, Δ such that $\mathit{cflip} \notin \mathsf{dom}(\Gamma)$ and $l', i', i \notin \Delta$ but $l \in \Delta$. Define Γ', *FBranch*, and *Body* as follows:

$$\begin{array}{rcl}
\Gamma' & \stackrel{\text{def}}{=} & \Gamma,\ \mathit{cflip} : \Pi_i\{i\mapsto -\}\text{-}\{i\mapsto -\} \\
\mathit{FBranch} & \stackrel{\text{def}}{=} & \mathsf{let}\, l'{=}[l]\ \mathsf{in}\ ([i]{:=}l';\, [l]{:=}i) \\
\mathit{Body} & \stackrel{\text{def}}{=} & \mathsf{let}\, i'{=}[i]\ \mathsf{in}\ (\mathsf{ifz}\ i'\ (\mathsf{free}(i))\ \mathit{FBranch})
\end{array}$$

The term *Mfree* and its subterms *FBranch* and *Body* are typed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash_{\Delta\cup\{i,i',l'\}} [i]{:=}l'{:}\{i\mapsto -\}\text{-}\{i\mapsto l'\}}}{\Gamma' \vdash_{\Delta\cup\{i,i',l'\}} [i]{:=}l'{:}\{l\mapsto - * i\mapsto -\}\text{-}\{l\mapsto - * i\mapsto l'\}}\,1 \qquad \dfrac{\dfrac{}{\Gamma' \vdash_{\Delta\cup\{i,i',l'\}} [l]{:=}i{:}\{l\mapsto -\}\text{-}\{l\mapsto i\}}}{\Gamma' \vdash_{\Delta\cup\{i,i',l'\}} [l]{:=}i{:}\{l\mapsto - * i\mapsto l'\}\text{-}\{l\mapsto i * i\mapsto l'\}}\,2}{\dfrac{\Gamma' \vdash_{\Delta\cup\{i,i',l'\}} ([i]{:=}l';\, [l]{:=}i){:}\{l\mapsto - * i\mapsto -\}\text{-}\{l\mapsto i * i\mapsto l'\}}{\Gamma' \vdash_{\Delta\cup\{i,i',l'\}} ([i]{:=}l';\, [l]{:=}i){:}\{l\mapsto l' * i\mapsto i' * \mathsf{lst}(l')\}\text{-}\{l\mapsto i * i\mapsto l' * \mathsf{lst}(l')\}}\,3}}{\Gamma' \vdash_{\Delta\cup\{i,i'\}} \mathsf{let}\, l'{=}[l]\ \mathsf{in}\ ([i]{:=}l';\, [l]{:=}i){:}\{\exists l'.\, l\mapsto l' * i\mapsto i' * \mathsf{lst}(l')\}\text{-}\{l\mapsto i * i\mapsto l' * \mathsf{lst}(l')\}}}{\Gamma' \vdash_{\Delta\cup\{i,i'\}} \mathit{FBranch}{:}\{i\mapsto i' * \mathsf{inv}(l) \wedge i'{\neq}0\}\text{-}\{\mathsf{inv}(l)\}}\,4$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash_{\Delta\cup\{i,i'\}} \mathsf{free}(i){:}\{i\mapsto -\}\text{-}\{\mathsf{emp}\}}}{\Gamma' \vdash_{\Delta\cup\{i,i'\}} \mathsf{free}(i){:}\{i\mapsto i' * \mathsf{inv}(l) \wedge i'{=}0\}\text{-}\{\mathsf{inv}(l)\}}\,5 \quad \Gamma' \vdash_{\Delta\cup\{i,i'\}} \mathit{FBranch}{:}\{i\mapsto i' * \mathsf{inv}(l) \wedge i'{\neq}0\}\text{-}\{\mathsf{inv}(l)\}}{\Gamma' \vdash_{\Delta\cup\{i,i'\}} (\mathsf{ifz}\ i'\ (\mathsf{free}(i))\ \mathit{FBranch}){:}\{i\mapsto i' * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}}{\Gamma' \vdash_{\Delta\cup\{i\}} \mathsf{let}\, i'{=}[i]\ \mathsf{in}\ (\mathsf{ifz}\ i'\ (\mathsf{free}(i))\ \mathit{FBranch}){:}\{\exists i'.i\mapsto i' * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}}{\Gamma' \vdash_{\Delta\cup\{i\}} \mathit{Body}{:}\{i\mapsto - * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}\,6$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash_{\Delta\cup\{i\}} \mathit{cflip}(i){:}\{i\mapsto -\}\text{-}\{i\mapsto -\}}}{\Gamma' \vdash_{\Delta\cup\{i\}} \mathit{cflip}(i){:}\{i\mapsto - * \mathsf{inv}(l)\}\text{-}\{i\mapsto - * \mathsf{inv}(l)\}}\,7 \qquad \Gamma' \vdash_{\Delta\cup\{i\}} \mathit{Body}{:}\{i\mapsto - * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}{\Gamma' \vdash_{\Delta\cup\{i\}} \mathit{cflip}(i);\, \mathit{Body}{:}\{i\mapsto - * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}}{\Gamma' \vdash_{\Delta} \lambda i.\ \mathit{cflip}(i);\, \mathit{Body}{:}\Pi_i\{i\mapsto - * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}}{\dfrac{\Gamma \vdash_{\Delta} \mathit{Mfree}{:}\big(\Pi_i\{i\mapsto - * \mathsf{inv}(l)\}\text{-}\{i\mapsto - * \mathsf{inv}(l)\}\big) \to \big(\Pi_i\{i\mapsto - * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}\big)}{\Gamma \vdash_{\Delta} \mathit{Mfree}{:}\Big(\big(\Pi_i\{i\mapsto -\}\text{-}\{i\mapsto -\}\big) \to \big(\Pi_i\{i\mapsto -\}\text{-}\{\mathsf{emp}\}\big)\Big) \otimes \mathsf{inv}(l)}\,8}$$

where the steps marked by 1-8 use the subsumption rule.

Figure 4: Derivation of the Typing Judgment for *Mfree*

memory manager, that is, that it is applied to *Mfree*. We prove the correctness of this application by deriving the typing judgment $\vdash_{\{j,l\}} (\mathit{Client}\ \mathit{Mfree}) : \{j\mapsto - * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}$.

The derivation of the mentioned typing judgment consists of three parts: the sub proof-trees for *Mfree* and *Client*, and the part that links these two proof trees. The sub proof-trees for *Mfree* and *Client* are shown in Figures 4 and 5. Note that the internal free list $[l]$ of the memory manager does not appear in the proof tree for *Client* in Figure 5; all the rules in the tree concern just cell j, the only cell that *Client* directly manipulates.

$$
\dfrac{
\dfrac{}{\Gamma \vdash_{\{j,l\}} \mathit{mfree}{:}\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\}{\to}\Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\Gamma \vdash_{\{j,l,i,i'\}} [i]{:=}i'{+}1{:}\{i{\mapsto}-\}\text{-}\{i{\mapsto}i'{+}1\}}
}{\Gamma \vdash_{\{j,l,i,i'\}} [i]{:=}i'{+}1{:}\{i{\mapsto}i'\}\text{-}\{i{\mapsto}-\}}\;1
}{\Gamma \vdash_{\{j,l,i\}} \mathsf{let}\; i'{=}[i] \;\mathsf{in}\; [i]{:=}i'{+}1{:}\{\exists i'.\, i{\mapsto}i'\}\text{-}\{i{\mapsto}-\}}
}{\Gamma \vdash_{\{j,l,i\}} \mathsf{let}\; i'{=}[i] \;\mathsf{in}\; [i]{:=}i'{+}1{:}\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\}}\;2
}{\Gamma \vdash_{\{j,l\}} \mathit{Rd}{:}\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\}}
}{
\dfrac{
\dfrac{\Gamma \vdash_{\{j,l\}} \mathit{mfree}\ \mathit{Rd}{:}\Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}}{\Gamma \vdash_{\{j,l\}} \mathit{mfree}\ \mathit{Rd}\ j{:}\{j{\mapsto}-\}\text{-}\{\mathsf{emp}\}}
}{\vdash_{\{j,l\}} \mathit{Client}{:}(\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}) \to \{j{\mapsto}-\}\text{-}\{\mathsf{emp}\}}
}
$$

where Γ is $\mathit{mfree} : \Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}$. In the tree, the subsumption rule is applied at 1 and 2, and in both cases, it uses subtype relations that express Consequence.

Figure 5: Typing Derivation for *Client*

It is the third-order frame rule that lets us ignore the internal free list $[l]$ of the memory manager when constructing the proof tree for *Client*. The third-order frame rule adds the missing free list $[l]$ to the derived type for *Client*, so that we can link *Client* with *Mfree*, without producing a type error. More precisely, the rule allows the following derivation:

$$
\dfrac{
\dfrac{\vdash_{\{j,l\}} \mathit{Client} : (\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}) \to \{j{\mapsto}-\}\text{-}\{\mathsf{emp}\}}
{\vdash_{\{j,l\}} \mathit{Client} : \big((\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}) \otimes \mathsf{inv}(l)\big) \to \{j{\mapsto}-{*}\mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}\;1
}{\vdash_{\{j,l\}} \mathit{Client}\ \mathit{Mfree} : \{j{\mapsto}- * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}}
$$

Here the step marked by 1 is an instance of the third-order frame rule, and it applies the subsumption rule with the subtype relation proved below:

$$
\begin{aligned}
& \big(\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}\big) \to \{j{\mapsto}-\}\text{-}\{\mathsf{emp}\} \\
\preceq_{\{j,l\}} \quad & (\because\ \theta \preceq_\Delta \theta \otimes P) \\
& \Big(\big(\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}\big) \to \{j{\mapsto}-\}\text{-}\{\mathsf{emp}\}\Big) \otimes \mathsf{inv}(l) \\
\preceq_{\{j,l\}} \quad & (\because\ (\theta \to \theta') \otimes P \simeq_\Delta (\theta \otimes P \to \theta' \otimes P)) \\
& \Big(\big(\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}\big) \otimes \mathsf{inv}(l)\Big) \to \Big(\{j{\mapsto}-\}\text{-}\{\mathsf{emp}\} \otimes \mathsf{inv}(l)\Big) \\
\preceq_{\{j,l\}} \quad & (\because\ \{P\}\text{-}\{Q\} \otimes R \simeq_\Delta \{P * R\}\text{-}\{Q * R\}) \\
& \Big(\big(\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}\big) \otimes \mathsf{inv}(l)\Big) \to \{j{\mapsto}- * \mathsf{inv}(l)\}\text{-}\{\mathsf{emp} * \mathsf{inv}(l)\} \\
\preceq_{\{j,l\}} \quad & (\because\ \forall\eta.\ [\![P]\!]_\eta = [\![P * \mathsf{emp}]\!]_\eta) \\
& \Big(\big(\Pi_i\{i{\mapsto}-\}\text{-}\{i{\mapsto}-\} \to \Pi_i\{i{\mapsto}-\}\text{-}\{\mathsf{emp}\}\big) \otimes \mathsf{inv}(l)\Big) \to \{j{\mapsto}- * \mathsf{inv}(l)\}\text{-}\{\mathsf{inv}(l)\}.
\end{aligned}
$$

4. Semantics

In this section we present our main contribution, the semantics that formalizes the underlying intuitions of the separation-logic type system. In particular, we formalize the following three intuitive properties of the type system:

(1) The types in the separation-logic type system refine the conventional types. A separation-logic type specifies a stronger property of a term, and restricts clients of such terms by asking them to only depend upon what can be known from the type. For instance, the type $\{1 \mapsto 3\}$-$\{1 \mapsto 0\}$ of a term M indicates not just that M is a command, but also that M stores 0 to cell 1 if cell 1 contains 3 initially. Moreover, this type forces clients to run M only when cell 1 contains 3.
(2) The higher-order frame rules in the type system imply that all programs behave locally.
(3) The type system, however, does not change the computational behavior of each program.

We formalize the first intuitive property by means of partial equivalence relations. Roughly, each type θ in our semantics determines a partial equivalence relation (in short, per) over the meaning of the "underlying type" $\overline{\theta}$. The domain of a per over a set A is a subset of A; this indicates that θ indeed specifies a stronger property than $\overline{\theta}$. The other part of a per, namely the equivalence relation part, explains that the type system restricts the clients, so that no type-checked clients can tell apart two equivalent programs. For instance, $\{1 \mapsto 3\}$-$\{1 \mapsto 0\}$ determines a per over the set of all commands. The domain of this per consists of commands satisfying $\{1 \mapsto 3\}$-$\{1 \mapsto 0\}$, and the per equates two such commands if they behave identically when cell 1 contains 3 initially. The equivalence relation implies that type-checked clients run a command of $\{1 \mapsto 3\}$-$\{1 \mapsto 0\}$ only when cell 1 contains 3.

We justify the other two intuitive properties by proving technical lemmas about our semantics. For number 2, we prove the soundness of all the subtyping rules, including the generalized frame rule and the distribution rules. For number 3, we prove that our semantics has been obtained by extracting and then quotienting semantic elements in the conventional semantics; yet, this extraction and quotienting does not reduce the computational information of semantic elements.

In this section, we first define categories $\mathcal{C}$ and $\mathcal{D}$, corresponding to the extraction and quotienting, respectively. Next we give the interpretation of types and terms. Finally, we connect our semantics with the conventional semantics, and prove that our semantics is indeed obtained by extracting and quotienting from the conventional semantics.

To make the paper accessible for a wider audience, we have decided to present the categories $\mathcal{C}$ and $\mathcal{D}$ and the proofs of their properties in a very concrete way — it is possible to give equivalent, but more abstract, descriptions of $\mathcal{C}$ and $\mathcal{D}$ and use known abstract results ¿From category theory to prove some of their properties (e.g., cartesian closure). For simplicity, we use the Hoare powerdomain to model the nondeterminism of commands in the semantics. Our results can be adapted to other alternatives, such as the Plotkin powerdomain for countable nondeterminism, using the idea from the chapter 9.3.2 of [23].

4.1. Categories $\mathcal{C}$ and $\mathcal{D}$.

We construct $\mathcal{C}$ and $\mathcal{D}$ by modifying the category CPO of pointed cpos and continuous functions. For $\mathcal{C}$, we impose a parameterized per on each cpo, and extract only those morphisms in CPO that preserve such pers (at all instantiations). The pers formalize that each type θ corresponds to a specification over the underlying type $\overline{\theta}$, and the preservation of the pers guarantees that all the morphisms in $\mathcal{C}$ satisfy the corresponding specifications. The parameterization of each per gives an additional guarantee that all morphisms in $\mathcal{C}$ behave locally (in the sense of higher-order frame rules). The other category $\mathcal{D}$ is a quotient of $\mathcal{C}$. Intuitively, the quotienting of $\mathcal{C}$ reflects that our type system also restricts the clients of a term; thus, more terms become equivalent observationally.

We define the "extracting" category $\mathcal{C}$ first. Let *Pred* be the set of *predicates*, i.e., subsets of *Heap*. We recall the semantic version of separating connectives, *emp* and $*$, on *Pred*. For $p, q \in \mathit{Pred}$,

$$\begin{aligned} h \in \mathit{emp} &\iff h = \lambda n.\mathit{undef}, \\ h \in p * q &\iff \exists h_1 h_2.\, h_1 \cdot h_2 = h \wedge h_1 \in p \wedge h_2 \in q. \end{aligned}$$

The category $\mathcal{C}$ is defined as follows:

- objects: (A, R) where A is a pointed cpo, and R is a family of admissible pers[4] indexed by predicates such that
$$\forall p, q \in \mathit{Pred}.\, R(p) \subseteq R(p * q);$$
- morphisms: $f\colon (A, R) \to (B, S)$ is a continuous function from A to B such that
$$\forall p \in \mathit{Pred}.\, f[R(p) \to S(p)]f,$$
i.e., f maps $R(p)$ related elements to $S(p)$ related elements.

Intuitively, an object (A, R) denotes a specification parameterized by invariant extension. The first component A denotes the underlying set from which we select "correct" elements. $R(\mathit{emp})$ denotes the initial specification of this object where no invariant is added by the frame rule. The domain $|R(\mathit{emp})|$ of per $R(\mathit{emp})$ indicates which elements satisfy the specification, and the equivalence relation on $|R(\mathit{emp})|$ expresses how the specification is also used to limit the interaction of a client: the client can only do what the specification guarantees, so more elements become equivalent observationally. The per $R(p)$ at another predicate p denotes an extended specification by the invariant p.

We illustrate the intuition of $\mathcal{C}$ with a "Hoare-triple" object $[p, q]$ for $p, q \in \mathit{Pred}$. Let comm be the set of all functions c from *Heap* to $\mathcal{P}(\mathit{Heap} \cup \{\mathit{wrong}\})$ that satisfy safety monotonicity and the frame property:

- *Safety Monotonicity*: for all $h, h_0 \in \mathit{State}$, if $h \# h_0$ and $\mathit{wrong} \notin c(h)$, then $\mathit{wrong} \notin c(h \cdot h_0)$;
- *Frame Property*: for all $h, h_0, h_1' \in \mathit{State}$, if $h \# h_0$, $\mathit{wrong} \notin c(h)$, and $h_1' \in c(h \cdot h_0)$, then there exists h' such that $h_1' = h' \cdot h_0$ and $h' \in c(h)$.

The above two properties are from the work on separation logic, and they form a sufficient and necessary condition that commands satisfy the (first-order) frame rule [24]. Note that the safety monotonicity and frame property are equivalent to the following condition:[5]

$$\text{if } h \# h_0 \text{ and } \mathit{wrong} \text{ isn't in } c(h), \text{ then } c(h \cdot h_0) \subseteq \{h' \cdot h_0 \mid h' \in c(h) \text{ and } h' \# h_0\}.$$

[4] A per R_0 on A is admissible iff $(\bot, \bot) \in R_0$ and R_0 is a sub-cpo of $A \times A$.

[5] The inclusion is one way only. For a counterexample, consider two disjoint heaps $h{=}[1{\to}0]$ and $h_0{=}[2{\to}0]$ and the command

$$\vdash\ \mathsf{let}\, j{=}\mathsf{new}\,\mathsf{in}\, \big(\mathsf{free}(j); (\mathsf{ifz}\ (j{-}2)\ ([1] := 5)\ ([1] := 6))\big) : \{1 \mapsto -\}\text{-}\{1 \mapsto -\}.$$

When this command is run in h, it nondeterministically assigns 5 or 6 to location 1, but when it is run in a bigger heap $h \cdot h_0$, the command always assigns 6 to the same location.

The set comm is the first component of the Hoare-triple object $[p, q]$, where the order on comm is given by:

$$c \sqsubseteq c' \iff \forall h.\, c(h) \subseteq c'(h).$$

The real meaning of $[p, q]$ is given by the second component R. For each predicate p_0, the domain of $R(p_0)$ consists of all "commands" in comm that satisfy $\{p * p_0\}$-$\{q * p_0\}$:

$$c \in |R(p_0)| \iff \forall h \in p * p_0.\, c(h) \subseteq q * p_0.$$

The equivalence relation $R(p_0)$ relates c and c' in $|R(p_0)|$ iff c and c' behave the same for the inputs in $p * p_0 * \mathit{true}$:

$$\begin{array}{rcl} \mathit{true} & = & \{h \mid h \in \mathit{Heap}\} \\ c[R(p_0)]c' & \iff & \forall h \in p * p_0 * \mathit{true}.\, c(h) = c'(h). \end{array}$$

This equivalence relation means that the type system allows a client to execute c or c' in h only when h satisfies $p * p_0 * p'$ for some p', which is added by the frame rule. We remark that the $*$ operator in the definition of $|R(p_0)|$ is allowed to partition the heap differently before and after the execution of c. For instance, when

$$p = \{[1{\to}1]\},\ q = \{[2{\to}0]\},\ \text{and}\ p_0 = \{[2{\to}0, 3{\to}0],\ [1{\to}0, 3{\to}1]\},$$

the initial heap h in the definition is split into cell 1 for p and cells $2, 3$ for p_0, but the final heap is split into cell 2 for q and cells $1, 3$ for p_0.

The category $\mathcal{C}$ is cartesian closed, has all small products, and contains the least fixpoint operator. The terminal object is $(\{\perp\}, CR)$ where $CR(p)$ is $\{(\perp, \perp)\}$ for all p, and the small products are given pointwise; for instance, $(A, R) \times (B, S)$ is $(A \times B, \{R(p) \times S(p)\}_p)$. The exponential of (A, R) and (B, S) is subtle, and its per component involves the quantification over all predicates. The cpo component of the exponential $(A, R) \Rightarrow (B, S)$ is the continuous function space $A \Rightarrow B$, and the per component of $(A, R) \Rightarrow (B, S)$, denoted $R \Rightarrow S$, is defined as follows:

$$\begin{array}{rcl} f \in |(R \Rightarrow S)(p)| & \iff & \forall q \in \mathit{Pred}.\, f[R(p * q) \to S(p * q)]f, \\ f[(R \Rightarrow S)(p)]g & \iff & f, g \in |(R \Rightarrow S)(p)| \ \wedge\ \forall q \in \mathit{Pred}.\, f[R(p * q) \to S(p * q)]g. \end{array}$$

Note that the right hand sides of the above equivalences quantify over all $*$-extension $p * q$ of p. This quantification ensures that $R \Rightarrow S$ satisfies the requirement

$$\forall p, p' \in \mathit{Pred}.\ (R \Rightarrow S)(p) \subseteq (R \Rightarrow S)(p * p')$$

in the category $\mathcal{C}$.

Lemma 4.1. *$\mathcal{C}$ is cartesian closed, and has all small products.*

Proof. First, we prove that for every (small) family $\{(A_i, R_i)\}_{i \in I}$ of objects in $\mathcal{C}$, its product is $(\Pi_{i \in I} A_i, \Pi_{i \in I} R_i)$ and the i-th projection π_i is $\lambda x.x(i)$. Here we write $(\Pi_{i \in \emptyset} A_i, \Pi_{i \in \emptyset} R_i)$ for $(\{\perp\}, CR)$. It is straightforward to show that $(\Pi_{i \in I} A_i, \Pi_{i \in I} R_i)$ is an object in $\mathcal{C}$ and π_i is a morphism in $\mathcal{C}$. So, we focus on proving the usual universality requirement for the product. Consider an object (B, S) and a family $f_i : (B, S) \to (A_i, R_i)_{i \in I}$ of morphisms in $\mathcal{C}$. We need to prove that there exists a unique morphism k from (B, S) to $(\Pi_{i \in I} A_i, \Pi_{i \in I} R_i)$, such that

$$\forall i \in I.\ f_i = \pi_i \circ k.$$

The above formula is equivalent to saying that k is $g = \lambda b.\lambda i.f_i(b)$. In particular, when $I = \emptyset$, k has to be the unique function $g' = \lambda b.\perp$. Note that these characterizations give the uniqueness of k. We prove the existence of k, by showing that g and g' are morphisms in $\mathcal{C}$. The continuity of g and g' is well-known. The relation preservation of g' also easily follows, since CR is a family of complete relations. For the relation preservation of g, we use the fact that f_i's are the morphisms in $\mathcal{C}$. Pick an arbitrary predicate p, and choose b, b' from B such that $b[S(p)]b'$. Then,

$$\begin{aligned}\forall i \in I.\ f_i(b)[R_i(p)]f_i(b') &\iff \forall i \in I.\ g(b)(i)\big[R_i(p)\big]g(b')(i) && (\because \text{ the definition of } g)\\ &\iff g(b)\big[(\Pi_{i\in I}R_i)(p)\big]g(b') && (\because \text{ the definition of } \Pi_{i\in I}R_i).\end{aligned}$$

Next, we prove that $(A \Rightarrow B, R \Rightarrow S)$ is an exponential of (A, R) and (B, S), with the evaluation morphism $\mathsf{ev} = \lambda(f, x).f(x)$. It is straightforward to prove that ev is a morphism in $\mathcal{C}$ and $(A \Rightarrow B, R \Rightarrow S)$ is an object in $\mathcal{C}$. So, we focus on the universality requirement for the exponentials. Consider a morphism $f\colon (C, T) \times (A, R) \to (B, S)$ in $\mathcal{C}$. We need to show that there exists a unique morphism $\mathsf{curry}(f) : (C, T) \to (A \Rightarrow B, R \Rightarrow S)$ such that

$$\forall (c, a) \in C \times A.\ f(c, a) = \mathsf{ev}(\mathsf{curry}(f)(c), a).$$

Since $\mathsf{ev}(\mathsf{curry}(f)(c), a) = \mathsf{curry}(f)(c)(a)$, the above is equivalent to $\mathsf{curry}(f) = \lambda c.\lambda a.f(c, a)$. Note that this characterizes $\mathsf{curry}(f)$ completely, so it gives the uniqueness of $\mathsf{curry}(f)$. It remains to prove that $\mathsf{curry}(f)$ is a morphism in $\mathcal{C}$. It is well-known that $\mathsf{curry}(f)$ is a continuous function from C to $A \Rightarrow B$. Thus, we only prove the relation preservation of $\mathsf{curry}(f)$, using the fact that $f[T(p) \times R(p) \to S(p)]f$ for all p. Pick arbitrary predicate p and c, c' in C such that $c[T(p)]c'$. Then, for all predicates q, we have that $c[T(p * q)]c'$, because $T(p) \subseteq T(p * q)$. Thus,

$$\begin{aligned}&\forall q.\forall a, a' \in A.\ a[R(p * q)]a' \implies f(c, a)[S(p * q)]f(c', a')\\ \iff\ & (\because \text{ the definition of } R \Rightarrow S)\\ &(\lambda a.f(c, a))\big[(R \Rightarrow S)(p)\big](\lambda a'.f(c', a'))\\ \iff\ & (\because \text{ the definition of } \mathsf{curry}(f))\\ &\mathsf{curry}(f)(c)\big[(R \Rightarrow S)(p)\big]\mathsf{curry}(f)(c').\end{aligned}$$

□

Lemma 4.2. *For every object (A, R) in $\mathcal{C}$, the least fixpoint operator $\mathsf{lfix}_A\colon [A \Rightarrow A] \to A$ on A is a morphism in $\mathcal{C}$.*

Proof. Pick arbitrary predicate p, and continuous functions f, g of type $A \to A$, such that $f[(R \Rightarrow R)(p)]g$; equivalently, $f[R(p * q) \to R(p * q)]g$ for all q. We need to show that $\mathsf{lfix}(f)[R(p)]\mathsf{lfix}(g)$. Note that since R is admissible, it is sufficient to prove that $f^k(\perp)[R(p)]g^k(\perp)$ for all $k \geq 0$. This sufficient condition holds because $f[R(p) \Rightarrow R(p)]g$ and $\perp[R(p)]\perp$. □

Another important feature of $\mathcal{C}$ is that it validates higher-order frame rules. Let $\mathcal{P}_r$ be the preorder $(Pred, \sqsubseteq)$ with $\sqsubseteq$ defined by predicate extension:

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

Category $\mathcal{C}$ has an "invariant-extension" functor inv from $\mathcal{C} \times \mathcal{P}_r$ to $\mathcal{C}$ defined by:

$$\mathsf{inv}((A, R), p) = (A, R(p * -)) \ \text{ and } \ \mathsf{inv}(f, p \sqsubseteq q) = f.$$

Functor inv corresponds to the type constructor $\otimes$ in our language; given a "type" (A, R) and a predicate p, inv extends (A, R) by adding the invariant p. For instance, when a triple object $[p', q']$ is extended with p, it becomes $[p' * p, q' * p]$.

Functor inv validates the subtyping rules that express higher-order frame rules: the generalized frame rule $\theta \preceq_\Delta \theta{\otimes}P$ and the rules for distributing $\otimes$ over each type constructor.

We first show that the functoriality of inv gives the soundness of the generalized frame rule. Note that $emp \sqsubseteq p$ for all predicates p, and that $\mathsf{inv}(-, emp)$ is the identity functor on $\mathcal{C}$. Thus, for each (A, R), the functoriality of inv gives a morphism from (A, R) to $\mathsf{inv}((A, R), p)$. This morphism gives the soundness of the subtyping rule $\theta \preceq_\Delta \theta \otimes P$.

The soundness of the other distribution rules follows from the fact that for all p, $\mathsf{inv}(-, p)$ preserves most of the structure of $\mathcal{C}$. For instance, $\mathsf{inv}(-, p)$ preserves the exponential of $\mathcal{C}$, because for all objects (A, R) and (B, S) and all predicates q, we have that

$$\begin{array}{rcl} f[(R(p*-) \Rightarrow S(p*-))(q)]g & \iff & \forall q'.f[R(p*(q*q')) \to S(p*(q*q'))]g \\ & \iff & \forall q'.f[R((p*q)*q') \to S((p*q)*q')]g \\ & \iff & f[(R \Rightarrow S)(p*q)]g. \end{array}$$

Lemma 4.3. *For each predicate p, $\mathsf{inv}(-, p)$ preserves the cartesian closed structure and all the small products of $\mathcal{C}$ on the nose.*

Proof. It is sufficient to prove that $\mathsf{inv}(-, p)$ preserves exponential objects, small product objects, evaluation morphisms, and projection morphisms. First, we prove the preservation of the small product objects and projection morphisms. Consider a family $\{(A_i, R_i)\}_{i \in I}$ of objects in $\mathcal{C}$. The following shows that the product $\Pi_{i \in I}(A_i, R_i)$ of this family is preserved by $\mathsf{inv}(-, p)$:

$$\begin{array}{rll} \mathsf{inv}\big(\Pi_{i\in I}(A_i,\ R_i),\ p\big) & = \mathsf{inv}\big(\big(\Pi_{i\in I}A_i,\ \Pi_{i\in I}R_i\big),\ p\big) & (\because\ \text{the definition of products in } \mathcal{C}) \\ & = \big(\Pi_{i\in I}A_i,\ (\Pi_{i\in I}R_i)(p*-)\big) & (\because\ \text{the definition of } \mathsf{inv}) \\ & = \big(\Pi_{i\in I}A_i,\ \Pi_{i\in I}(R_i(p*-))\big) & \\ & = \Pi_{i\in I}\big(A_i,\ R_i(p*-)\big) & \\ & = \Pi_{i\in I}\big(\mathsf{inv}((A_i,\ R_i),\ p)\big) & (\because\ \text{the definition of } \mathsf{inv}). \end{array}$$

Since $\mathsf{inv}(f, p) = f$, functor inv preserves the i-th projection from $\Pi_{i \in I}(A_i, R_i)$.

Next, we show that $\mathsf{inv}(-, p)$ preserves the exponential objects and evaluation morphisms in $\mathcal{C}$. Let (A, R) and (B, S) be objects in $\mathcal{C}$. By what we have shown before this lemma, we have that

$$(R \Rightarrow S)(p*-) \;=\; R(p*-) \Rightarrow S(p*-).$$

¿From this follows the preservation of exponential objects:

$$\begin{array}{rll} \mathsf{inv}\big((A,R) \Rightarrow (B,S),\ p\big) & = \mathsf{inv}\big((A \Rightarrow B,\ R \Rightarrow S),\ p\big) & (\because \text{Def. of exponentials in } \mathcal{C}) \\ & = \big(A \Rightarrow B,\ (R \Rightarrow S)(p*-)\big) & (\because \text{Def. of } \mathsf{inv}) \\ & = \big(A \Rightarrow B,\ (R(p*-) \Rightarrow S(p*-))\big) & \\ & = \big(A, R(p*-)\big) \Rightarrow \big(B, S(p*-)\big) & (\because \text{Def. of exponentials in } \mathcal{C}) \\ & = \mathsf{inv}\big((A,R),p\big) \Rightarrow \mathsf{inv}\big((B,S),p\big) & (\because \text{Def. of } \mathsf{inv}). \end{array}$$

Functor $\mathsf{inv}(-, p)$ preserves the the evaluation morphism ev for $(A, R) \Rightarrow (B, S)$, because $\mathsf{inv}(-, p)$ preserves the products and exponentials and $\mathsf{inv}(f, p)$ only changes the type of f, not modifying its "meaning" (i.e., $\mathsf{inv}(f, p) = f$). □

Lemma 4.4. *For all predicates p and q, $\mathsf{inv}(-, p) \circ \mathsf{inv}(-, q) = \mathsf{inv}(-, p * q)$.*

Proof. Both $\mathsf{inv}(-, p) \circ \mathsf{inv}(-, q)$ and $\mathsf{inv}(-, p * q)$ map a morphism f to the same f with perhaps different domain and codomain. Thus, if they act the same on the objects in $\mathcal{C}$, they must act the same on the morphisms. In fact, they do act the same on the objects;

for each (A, R) in $\mathcal{C}$,

$$\begin{array}{rcll}(\mathsf{inv}(-,p) \circ \mathsf{inv}(-,q))(A,R) & = & (A, R(p * (q * -))) & (\because \text{ the definition of } \mathsf{inv}) \\ & = & (A, R((p*q) * -)) & (\because \ * \text{ is associative}) \\ & = & \mathsf{inv}(-, p*q)(A,R) & (\because \text{ the definition of } \mathsf{inv}).\end{array}$$

□

For now, the final remark on $\mathcal{C}$ is that the triple-object generator $[-,-]$ can be made into a functor, whose morphism action validates the subtyping rule for Consequence. Let $\mathcal{P}$ be the set of predicates ordered by the subset inclusion $\subseteq$. Generator $[-,-]$ can be extended to a functor tri from $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$ to $\mathcal{C}$:

$$\mathsf{tri}(p,q) = [p,q] \ \text{ and } \ \mathsf{tri}(p' \subseteq p, q \subseteq q')(c) = c.$$

Note that tri is contravariant in the first argument and covariant on the second argument. This mixed variance reflects that the pre-condition of a triple can be strengthened, and the post-condition can be weakened; thus, it validates the subtyping rule for Consequence. We also note that the subtyping rule that moves an invariant assertion into the pre- and post-conditions is sound.

Lemma 4.5. *For each predicate p, let $- * p \colon \mathcal{P} \to \mathcal{P}$ be a functor that maps a predicate q to $q * p$. Then,*

$$\mathsf{inv}(-,p) \circ \mathsf{tri} = \mathsf{tri}(- * p, - * p).$$

Proof. Both $\mathsf{inv}(-,p) \circ \mathsf{tri}$ and $\mathsf{tri}(- * p, - * p)$ map the morphisms in $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$ to inclusions between pointed cpos. Thus, it is sufficient to prove that $\mathsf{inv}(-,p) \circ \mathsf{tri}$ and $\mathsf{tri}(- * p, - * p)$ act the same on objects. Pick an arbitrary object (p', q') in $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$. Then, by the definition of inv and tri, there exist families R, S of pers such that

$$(\mathsf{comm}, R) = (\mathsf{inv}(-,p) \circ \mathsf{tri})(p', q') \quad \text{and} \quad (\mathsf{comm}, S) = (\mathsf{tri}(- * p, - * p))(p', q').$$

Thus, to prove $(\mathsf{inv}(-,p) \circ \mathsf{tri})(p', q') = (\mathsf{tri}(- * p, - * p))(p', q')$, we only need to show $R = S$. For each predicate p_0, the domains of $R(p_0)$ and $S(p_0)$ are the same, because

$$\begin{array}{rcll} c \in |R(p_0)| & \iff & \forall h \in p' * (p * p_0).\ c(h) \subseteq q' * (p * p_0) & (\because \text{ Def. of } \mathsf{inv}(\mathsf{tri}(p', q'), p)) \\ & \iff & \forall h \in (p' * p) * p_0.\ c(h) \subseteq (q' * p) * p_0 & (\because \ * \text{ is associative}) \\ & \iff & c \in |S(p_0)| & (\because \text{ Def. of } \mathsf{tri}(p' * p, q' * p)).\end{array}$$

And, $R(p_0)$ and $S(p_0)$ specify the same relation on their domains, because

$$\begin{array}{rcll} c[R(p_0)]c' & \iff & \forall h \in p' * (p * p_0) * \mathit{true}.\ c(h) = c'(h) & (\because \text{ Def. of } \mathsf{inv}(\mathsf{tri}(p', q'), p)) \\ & \iff & \forall h \in (p' * p) * p_0 * \mathit{true}.\ c(h) = c'(h) & (\because \ * \text{ is associative}) \\ & \iff & c[S(p_0)]c' & (\because \text{ Def. of } \mathsf{tri}(p' * p, q' * p)).\end{array}$$

□

The category $\mathcal{D}$ is obtained from $\mathcal{C}$ by equating morphisms according to an equivalence relation $\sim$. Morphisms f and g in $\mathcal{C}[(A,R),(B,S)]$ are related by $\sim$ iff

$$\forall p \in \mathit{Pred}.\ f[R(p) \to S(p)]g.$$

$\sim$ is an equivalence relation; it is reflexive, because every morphism in $\mathcal{C}[(A,R),(B,S)]$ should map $R(p)$-related elements to $S(p)$-related elements, for all p; it is symmetric and transitive because, for all p, $R(p)$ and $S(p)$ are symmetric and transitive. The interesting property of $\sim$ is that it is preserved by all the structure of $\mathcal{C}$:

Lemma 4.6 (Preservation)**.** *The relation $\sim$ is preserved by the following operators in $\mathcal{C}$:*

- *the functor* $\mathsf{inv}(-, p \sqsubseteq q)$ *on* $\mathcal{C}$*, for all predicates* p, q *such that* $p \sqsubseteq q$*;*
- *the composition of morphisms;*
- *the currying of morphisms; and*
- *the pairing into all the small products.*

Proof. First, we prove the preservation by inv. Let p and q be predicates such that $p \sqsubseteq q$. Pick arbitrary two morphisms $f, g\colon (A, R) \to (B, S)$ in $\mathcal{C}$ such that $f \sim g$. We will show that $\mathsf{inv}(f, p \sqsubseteq q) \sim \mathsf{inv}(g, p \sqsubseteq q)$. Morphism $\mathsf{inv}(f, p \sqsubseteq q)$ and $\mathsf{inv}(g, p \sqsubseteq q)$ both have the type $(A, R(p * -)) \to (B, S(q * -))$. Thus, proving $\mathsf{inv}(f, p \sqsubseteq q) \sim \mathsf{inv}(g, p \sqsubseteq q)$ amounts to showing the formula:

$$\forall r \in \mathit{Pred}.\ f[R(p * r) \to S(q * r)]g.$$

The formula holds, because $f[R(p * r) \to S(p * r)]g$ and $S(p * r) \subseteq S(q * r)$ for all r.

Second, we prove the preservation by the composition of morphisms. Consider morphisms $f, f'\colon (A, R) \to (B, S)$ and $g, g'\colon (B, S) \to (C, T)$ such that $f \sim f'$ and $g \sim g'$. Then, for all predicates p and all $a, a' \in A$ such that $a[R(p)]a'$, we have that $f(a)[S(p)]f'(a')$, so $g(f(a))[T(p)]g'(f'(a'))$. This proves that $(g \circ f) \sim (g' \circ f')$.

Third, we show the preservation by the currying operator. Consider morphisms f, f' from $(C, T) \times (A, R)$ to (B, S), such that $f \sim f'$. Pick an arbitrary predicate p, and choose $T(p)$-related c, c' from C. Then, for all predicates q, we have that $c[T(p * q)]c'$, because $T(p) \subseteq T(p * q)$. Thus,

$$\begin{array}{ll} & \forall q. \forall a, a' \in A.\ a[R(p * q)]a' \Rightarrow f(c, a)[S(p * q)]f'(c', a') \\ \iff & \quad (\because \text{ the definition of } R \Rightarrow S) \\ & (\lambda a. f(c, a))\big[(R \Rightarrow S)(p)\big](\lambda a'. f'(c', a')) \\ \iff & \quad (\because \text{ the definition of } \mathsf{curry}(f)) \\ & \mathsf{curry}(f)(c)\big[(R \Rightarrow S)(p)\big]\mathsf{curry}(f')(c') \end{array}$$

What we have just proved shows that $\mathsf{curry}(f) \sim \mathsf{curry}(f')$.

Finally, we prove the preservation by the pairing into the small products. Consider a family $\{(A_i, R_i)\}_{i \in I}$ of objects in $\mathcal{C}$. Pick two families of morphisms in $\mathcal{C}$, $\{f_i\}_{i \in I}$ and $\{f'_i\}_{i \in I}$, such that

$$\forall i \in I. \quad f_i\colon (B, S) \to (A_i, R_i), \quad f'_i\colon (B, S) \to (A_i, R_i), \quad \text{and} \quad f_i \sim f'_i.$$

We need to show the following equivalence:

$$\Big(\lambda b\colon B. \lambda i\colon I. f_i(b)\Big) \sim \Big(\lambda b'\colon B. \lambda i\colon I. f'_i(b')\Big).$$

For all predicates p and all b, b' in B such that $b[S(p)]b'$, we have that

$$\forall i \in I.\ f_i(b)[R_i(p)]f'_i(b').$$

Thus,

$$(\lambda i\colon I.\ f_i(b))[(\Pi_{i \in I} R_i)(p)](\lambda i\colon I.\ f'_i(b')).$$

This relationship gives the required equivalence. □

Lemma 4.6 ensures that taking a quotient of morphisms in $\mathcal{C}$ gives a well-defined category, which we call $\mathcal{D}$. Category $\mathcal{D}$ inherits all the interesting structure of $\mathcal{C}$ by Lemma 4.6; it is cartesian closed, has all small products, and has a functor $\mathsf{inv}'\colon \mathcal{D} \times \mathcal{P}_r \to \mathcal{D}$ that preserves the CCC structure and the small products of $\mathcal{D}$. Let E be the "quotienting" functor from $\mathcal{C}$ to $\mathcal{D}$, and $\mathsf{tri}'\colon \mathcal{P}^{\mathsf{op}} \times \mathcal{P} \to \mathcal{D}$ the composition of E with tri. We summarize the main property of $\mathcal{D}$ in the following two lemmas:

Lemma 4.7. *The category $\mathcal{D}$ is a CCC with all small products, and has two functors $\mathsf{inv}' \colon \mathcal{D} \times \mathcal{P}_r \to \mathcal{D}$ and $\mathsf{tri}' \colon \mathcal{P}^{\mathsf{op}} \times \mathcal{P} \to \mathcal{D}$ such that*

(1) $\mathsf{inv}'(-, p)$ *preserves all the CCC structure and the small products of $\mathcal{D}$;*
(2) $\mathsf{inv}'(-, p) \circ \mathsf{inv}'(-, q) = \mathsf{inv}'(-, p * q)$*; and*
(3) $\mathsf{inv}'(-, p) \circ \mathsf{tri}' = \mathsf{tri}'(- * p, - * p)$.

Proof. First, we prove that $\mathcal{D}$ has all the small products. Let $\{(A_i, R_i)\}_{i \in I}$ be a small family of objects in $\mathcal{D}$. We show that the product of this family is $(\Pi_{i \in I} A_i, \Pi_{i \in I} R_i)$ and the i-th projection is $[\pi_i]$, where $[f]$ means the equivalence class of the morphism f. Consider an arbitrary family $\{[f_i] \colon (B, S) \to (A_i, R_i)\}_{i \in I}$ of morphisms in $\mathcal{D}$. This family induces some family $\{f_i\}_{i \in I}$ in $\mathcal{C}$. Since $(\Pi_{i \in I} A_i, \Pi_{i \in I} R_i)$ is the product in $\mathcal{C}$, there exists a morphism $\langle f_i \rangle_{i \in I} \colon (B, S) \to (\Pi_{i \in I} A_i, \Pi_{i \in I} R_i)$ such that $\pi_i \circ \langle f_i \rangle_{i \in I} = f_i$ for all $i \in I$. The equivalence class $[\langle f_i \rangle_{i \in I}]$ of this morphism is the required unique morphism in $\mathcal{D}$. It makes the required diagrams for the products commute, because

$$\forall i \in I.\ [\pi_i] \circ [\langle f_i \rangle_{i \in I}] = [\pi_i \circ \langle f_i \rangle_{i \in I}] = [f_i].$$

For the uniqueness, suppose that $[k]$ is another morphism in $\mathcal{D}$ that makes the diagram commutes. Then, $[k]$ must be equal to $[k] = [\langle f_i \rangle_{i \in I}]$, as shown below:

$$\begin{aligned}
(\forall i \in I.\ [\pi_i] \circ [k] = [f_i]) &\iff (\forall i \in I.\ [\pi_i \circ k] = [f_i]) && (\because \sim \text{ is preserved by } \circ) \\
&\implies [\langle \pi_i \circ k \rangle_{i \in I}] = [\langle f_i \rangle_{i \in I}] && (\because \sim \text{ is preserved by the pairing}) \\
&\iff [k] = [\langle f_i \rangle_{i \in I}].
\end{aligned}$$

Second, we show that $\mathcal{D}$ has the exponentials. Let $((A, R), (B, S))$ be a pair of objects in $\mathcal{D}$. We prove that $(A \Rightarrow B, R \Rightarrow S)$ is an exponential of this pair, and the evaluation morphism is the equivalence class $[\mathsf{ev}]$. Consider a morphism $[f] \colon (C, T) \times (A, R) \to (B, S)$ in $\mathcal{D}$. We need to prove that the universality requirement holds for $[f]$: there exists a unique morphism $[g] \colon (C, T) \to (A \Rightarrow B, R \Rightarrow S)$ in $\mathcal{D}$ such that

$$[f] = [\mathsf{ev}] \circ \langle [g] \circ [\pi_0], [\pi_1] \rangle.$$

The equation in the requirement implies that $[g]$ should be equal to $[\mathsf{curry}(f)]$:

$$\begin{aligned}
[f] = [\mathsf{ev}] \circ \langle [g] \circ [\pi_0], [\pi_1] \rangle &\implies [f] = [\mathsf{ev}] \circ \langle [g \circ \pi_0], [\pi_1] \rangle && (\because \text{the composition preserves } \sim) \\
&\implies [f] = [\mathsf{ev}] \circ [\langle g \circ \pi_0, \pi_1 \rangle] && (\because \text{the pairing preserves } \sim) \\
&\implies [f] = [\mathsf{ev} \circ \langle g \circ \pi_0, \pi_1 \rangle] && (\because \text{the composition preserves } \sim) \\
&\implies [\mathsf{curry}(f)] = [\mathsf{curry}(\mathsf{ev} \circ \langle g \circ \pi_0, \pi_1 \rangle)] && (\because \mathsf{curry} \text{ preserves } \sim) \\
&\implies [\mathsf{curry}(f)] = [g].
\end{aligned}$$

Thus, $\mathcal{D}$ has at most one morphism $[g]$ that satisfies the universality requirement. We now show that $[\mathsf{curry}(f)]$ satisfies the requirement. By the definition of curry, we have that

$$f = \mathsf{ev} \circ \langle \mathsf{curry}(f) \circ \pi_0, \pi_1 \rangle.$$

This equation implies that $[\mathsf{curry}(f)]$ makes the required diagram commute:

$$\begin{aligned}
f = \mathsf{ev} \circ \langle \mathsf{curry}(f) \circ \pi_0, \pi_1 \rangle &\implies [f] = [\mathsf{ev} \circ \langle \mathsf{curry}(f) \circ \pi_0, \pi_1 \rangle] \\
&\implies [f] = [\mathsf{ev}] \circ [\langle \mathsf{curry}(f) \circ \pi_0, \pi_1 \rangle] && (\because \circ \text{ preserves } \sim) \\
&\implies [f] = [\mathsf{ev}] \circ \langle [\mathsf{curry}(f) \circ \pi_0], [\pi_1] \rangle && (\because \text{pairing preserves } \sim) \\
&\implies [f] = [\mathsf{ev}] \circ \langle [\mathsf{curry}(f)] \circ [\pi_0], [\pi_1] \rangle && (\because \circ \text{ preserves } \sim).
\end{aligned}$$

Finally, we prove the three properties of inv'. Note that the categories $\mathcal{C}$ and $\mathcal{D}$ have the same collection of objects, and they have the same exponentials and same small products, as far as the objects are concerned. Moreover, for objects, the functors $\mathsf{inv}'(-, p)$ and

$\mathsf{inv}(-,p)$ are identical. Thus, $\mathsf{inv}'(-,p)\colon \mathcal{D} \to \mathcal{D}$ preserves the exponential objects and small product objects in $\mathcal{D}$ if and only if $\mathsf{inv}(-,p)$ preserves those in $\mathcal{C}$; the right hand side of this equivalence holds by Lemma 4.3. The functor $\mathsf{inv}'(-,p)$ also preserves $[\mathsf{ev}]$ and $[\pi_i]$, because $\mathsf{inv}'([f],p) = [\mathsf{inv}(f,p)] = [f]$. So, $\mathsf{inv}'(-,p)$ preserves the CCC structure and the small products.

For the second property of inv', we note that the equation in the property holds for the objects, because for all predicates r, functors $\mathsf{inv}'(-,r)$ and $\mathsf{inv}(-,r)$ behave the same on the objects, and $\mathsf{inv}(-,p) \circ \mathsf{inv}(-,q) = \mathsf{inv}(-,p*q)$. The equation also holds for the morphisms, because $\mathsf{inv}'([f],r) = [f]$ for all f, r.

For the third property of inv', we recall that $\mathsf{tri}' = E \circ \mathsf{tri}$. Thus, it is sufficient to show that

$$\mathsf{inv}'(-,p) \circ E \circ \mathsf{tri} = E \circ \mathsf{tri}(- * p, - * p).$$

The equation holds for the objects; E is the identity on the objects, inv and inv' are the same for objects, and $\mathsf{inv}(-,p) \circ \mathsf{tri} = \mathsf{tri}(- * p, - * p)$. For the morphisms, the equation also holds, because both sides of the equation map each morphism in $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$ to the equivalence class of an inclusion. $\square$

Lemma 4.8. *The functor E from $\mathcal{C}$ to $\mathcal{D}$ is full, preserves the CCC structure as well as small products, and makes the following diagrams commute:*

$$\begin{tikzcd} \mathcal{C} \times \mathcal{P}_r \arrow[r, "\mathsf{inv}"] \arrow[d, "E \times \mathsf{Id}"'] & \mathcal{C} \arrow[d, "E"] \\ \mathcal{D} \times \mathcal{P}_r \arrow[r, "\mathsf{inv}'"] & \mathcal{D} \end{tikzcd} \qquad \begin{tikzcd} \mathcal{P}^{\mathsf{op}} \times \mathcal{P} \arrow[r, "\mathsf{tri}"] \arrow[d, "\mathsf{Id}"'] & \mathcal{C} \arrow[d, "E"] \\ \mathcal{P}^{\mathsf{op}} \times \mathcal{P} \arrow[r, "\mathsf{tri}'"] & \mathcal{D} \end{tikzcd}$$

Proof. The categories $\mathcal{C}$ and $\mathcal{D}$ have the same collection of objects, and their CCC structure and small products are identical, as far as the objects are concerned. Since E is the identity on objects, it preserves the exponential objects and small product objects. Moreover, E preserves the evaluation and projection morphisms, because the evaluation and projection morphisms in $\mathcal{D}$ are just the equivalence classes of the corresponding morphisms in $\mathcal{C}$, and E maps f to its equivalence class $[f]$. Thus, functor E preserves the CCC structure and the small products of $\mathcal{C}$.

The commutative diagram for inv holds for the objects, because inv and inv' behave the same for the objects and E is the identity on the objects. To show that the diagram also holds for the morphisms, we pick an arbitrary morphism $(f, p \sqsubseteq q)$ in $\mathcal{C} \times \mathcal{P}_r$. Then,

$$(\mathsf{inv}' \circ (E \times \mathsf{Id}))(f, p \sqsubseteq q) \;=\; \mathsf{inv}'([f], p \sqsubseteq q) \;=\; [f] \;=\; (E \circ \mathsf{inv})(f, p \sqsubseteq q).$$

Finally, the commutative diagram for tri' is the definition of tri', so it must hold. $\square$

4.2. **Interpretation of the Language.** We interpret the language in two steps. First, we define the semantics $[\![-]\!]^{\mathcal{C}}$ in the family fibration $\mathit{Fam}(\mathcal{C}) \to \mathsf{Set}$. Each base set in the fibration models all the possible environments for a fixed shape of the stack (i.e., a fixed set of integer variables Δ). For instance, the object $\{(A,R)_\eta\}_{\eta \in [\![\Delta]\!]}$ assumes that all the available integer variables are in Δ, and it specifies a type dependent on the values of such variables, given by η. The types and terms of our language are interpreted using the categorical structure of the fibration. Next, we quotient the semantics $[\![-]\!]^{\mathcal{C}}$ to get more abstract, official interpretation $[\![-]\!]$, which uses category $\mathcal{D}$ instead of $\mathcal{C}$.

4.2.1. *Semantics* $[\![-]\!]^{\mathcal{C}}$ *in* $Fam(\mathcal{C}) \to \mathsf{Set}$. The interpretation is explicit about the set of variables under which we consider types, type assignments, and terms. Write $\Delta \vdash \Gamma$ to mean that $\Delta \vdash \Gamma(x) : \mathsf{Type}$, for all x in the domain of Γ.

The semantics of $\Delta \vdash \theta(: \mathsf{Type})$ and $\Delta \vdash \Gamma$ is given by a family of objects in $\mathcal{C}$ indexed by the environments in $[\![\Delta]\!]$. The precise definition of $[\![\theta]\!]^{\mathcal{C}}$ and $[\![\Gamma]\!]^{\mathcal{C}}$ is given as follows: for η in $[\![\Delta]\!]$,

$$\begin{aligned}
[\![\Delta \vdash \{P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_{\eta} &= \mathsf{tri}([\![\Delta \vdash P]\!]_{\eta}, [\![\Delta \vdash Q]\!]_{\eta}),\\
[\![\Delta \vdash \theta \otimes P]\!]^{\mathcal{C}}_{\eta} &= \mathsf{inv}([\![\Delta \vdash \theta]\!]^{\mathcal{C}}_{\eta}, [\![\Delta \vdash P]\!]_{\eta}),\\
[\![\Delta \vdash \theta \to \theta']\!]^{\mathcal{C}}_{\eta} &= [\![\Delta \vdash \theta]\!]^{\mathcal{C}}_{\eta} \Rightarrow [\![\Delta \vdash \theta']\!]^{\mathcal{C}}_{\eta},\\
[\![\Delta \vdash \Pi_i \theta]\!]^{\mathcal{C}}_{\eta} &= \Pi_{n \in Val} [\![\Delta \cup \{i\} \vdash \theta]\!]^{\mathcal{C}}_{\eta[i \to n]},\\
[\![\Delta \vdash \Gamma]\!]^{\mathcal{C}}_{\eta} &= \Pi_{x \in \mathsf{dom}(\Gamma)} [\![\Delta \vdash \Gamma(x)]\!]^{\mathcal{C}}_{\eta}.
\end{aligned}$$

Note that tri is used to interpret the triple type $\{P\}\text{-}\{Q\}$, and inv to interpret the invariant extension $\theta \otimes P$.

Each subtype relation $\theta \preceq_{\Delta} \theta'$ is interpreted as a family of morphisms in $\mathcal{C}$ of the shape

$$\{\lambda x.\, x : [\![\Delta \vdash \theta]\!]^{\mathcal{C}}_{\eta} \to [\![\Delta \vdash \theta']\!]^{\mathcal{C}}_{\eta}\}_{\eta \in [\![\Delta]\!]}.$$

Note that every morphism in the family is implemented (or realized) by the identity function. In order for this definition to typecheck, the underlying cpo of the source object $[\![\theta]\!]^{\mathcal{C}}_{\eta}$ should be included in that of the target $[\![\theta']\!]^{\mathcal{C}}_{\eta}$, and the parameterized per of the source should imply that of the target for all instantiations. In the lemma below, we prove that both of these requirements hold.

Lemma 4.9. *If a subtype relation $\theta \preceq_{\Delta} \theta'$ is derivable, then for all η in $[\![\Delta]\!]$,*

(1) *objects $[\![\Delta \vdash \theta]\!]^{\mathcal{C}}_{\eta}$ and $[\![\Delta \vdash \theta']\!]^{\mathcal{C}}_{\eta}$ have the same underlying cpo, and*
(2) *their per parts R and R' satisfy that $\forall p.\, R(p) \subseteq R'(p)$.*

Proof. The proof proceeds by the induction on the derivation of $\theta \preceq_{\Delta} \theta'$. First, we consider the base cases where $\theta \preceq_{\Delta} \theta'$ is proved by an axiom. In all the base cases except the generalized frame rule, objects $[\![\theta]\!]^{\mathcal{C}}_{\eta}$ and $[\![\theta']\!]^{\mathcal{C}}_{\eta}$ are identical, because inv preserve all categorical structure used to interpret types (Lemmas 4.3, 4.4 and 4.5). When $\theta \preceq_{\Delta} \theta'$ is derived by the generalized frame rule, so that $\theta' = \theta \otimes P$ for some P, object $[\![\theta']\!]^{\mathcal{C}}_{\eta}$ is $\mathsf{inv}([\![\theta]\!]^{\mathcal{C}}_{\eta}, [\![P]\!]_{\eta})$. Thus, by the definition of inv, there exist A and R such that

$$[\![\theta']\!]^{\mathcal{C}}_{\eta} = (A, R([\![P]\!]_{\eta} * -)) \;\text{ and }\; [\![\theta]\!]^{\mathcal{C}}_{\eta} = (A, R).$$

The above two equations show that $[\![\theta']\!]^{\mathcal{C}}_{\eta}$ and $[\![\theta]\!]^{\mathcal{C}}_{\eta}$ have the same underlying cpo. They also imply the requirement for the parameterized pers, because $R(p) \subseteq R(p * [\![P]\!]_{\eta}) = R([\![P]\!]_{\eta} * p)$ for all p.

Second, we consider the case that Consequence is applied in the last step of the derivation. In this case, the derivation of $\theta \preceq_{\Delta} \theta'$ has the following shape:

$$\frac{\forall \eta' \in [\![\Delta]\!].\;\; [\![P']\!]_{\eta'} \subseteq [\![P]\!]_{\eta'} \wedge [\![Q]\!]_{\eta'} \subseteq [\![Q']\!]_{\eta'}}{\{P\}\text{-}\{Q\} \preceq_{\Delta} \{P'\}\text{-}\{Q'\}}$$

By the definition of the semantics of types, both $[\![\{P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_{\eta}$ and $[\![\{P'\}\text{-}\{Q'\}]\!]^{\mathcal{C}}_{\eta}$ have comm as their underlying cpo. We will now show that their parameterised pers also satisfy the requirement in the lemma. Let R, R' be parameterized pers of $[\![\{P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_{\eta}$ and $[\![\{P'\}\text{-}\{Q'\}]\!]^{\mathcal{C}}_{\eta}$, respectively. Then, for all p and $c_0, c_1 \in \mathsf{comm}$,

$$\begin{aligned}
c_0[R(p)]c_1 &\iff (\forall h \in [\![P]\!]_\eta{*}p{*}\mathit{true}.\ c_0(h){=}c_1(h)) \wedge (c_0, c_1 \in |R(p)|) \\
&\iff (\forall h \in [\![P]\!]_\eta{*}p{*}\mathit{true}.\ c_0(h){=}c_1(h)) \wedge (\forall h \in [\![P]\!]_\eta{*}p.\, (c_0(h), c_1(h) \subseteq [\![Q]\!]{*}p)) \\
&\implies (\forall h \in [\![P']\!]_\eta{*}p{*}\mathit{true}.\ c_0(h){=}c_1(h)) \wedge (\forall h \in [\![P']\!]_\eta{*}p.\, (c_0(h), c_1(h) \subseteq [\![Q']\!]{*}p)) \\
&\iff (\forall h \in [\![P']\!]_\eta{*}p{*}\mathit{true}.\ c_0(h){=}c_1(h)) \wedge (c_0, c_1 \in |R'(p)|) \\
&\iff c_0[R'(p)]c_1.
\end{aligned}$$

The implication above uses the assumption that P' is the strengthening of P and Q' is the weakening of Q, and all the equivalences are simply the rolling or unrolling of some definition. We have just shown that $R(p) \subseteq R'(p)$ for all p, as required.

Third, we consider the cases of inference rules for the type constructors, $\rightarrow$, Π and $\otimes$. All these cases follow from the induction hypothesis and the definition of appropriate functors, which are used to interpret $\rightarrow$, Π and $\otimes$. We illustrate this general pattern by proving the case of $\rightarrow$. Suppose that the last step of the derivation of $\theta \preceq_\Delta \theta'$ has the form:

$$\frac{\theta'_0 \preceq_\Delta \theta_0 \quad \theta_1 \preceq_\Delta \theta'_1}{\theta_0 \rightarrow \theta_1 \preceq_\Delta \theta'_0 \rightarrow \theta'_1}$$

For $i = 0, 1$, let $(A_i, R_i) = [\![\theta_i]\!]^{\mathcal{C}}_\eta$ and $(A'_i, R'_i) = [\![\theta'_i]\!]^{\mathcal{C}}_\eta$. Then, by the induction hypothesis, we have that

$$A'_0 = A_0,\ \ A'_1 = A_1,\ \ (\forall p.\, R'_0(p) \subseteq R_0(p)),\ \text{ and }\ (\forall p.\, R_1(p) \subseteq R'_1(p)).$$

So, the underlying cpos of $[\![\theta_0 \rightarrow \theta_1]\!]^{\mathcal{C}}_\eta$ and $[\![\theta'_0 \rightarrow \theta'_1]\!]^{\mathcal{C}}_\eta$ are the same cpo of continuous functions from A_0 to A_1. The remaining requirement is to show that $(R_0 \Rightarrow R_1)(p) \subseteq (R'_0 \Rightarrow R'_1)(p)$ for all p, and it is proved below:

$$\begin{aligned}
f[(R_0 \Rightarrow R_1)(p)]g &\iff \forall p_0.\, f[R_0(p{*}p_0) \rightarrow R_1(p{*}p_0)]g && (\because \text{Def. of } R_0 \Rightarrow R_1) \\
&\implies \forall p_0.\, f[R'_0(p{*}p_0) \rightarrow R'_1(p{*}p_0)]g && (\because \forall q. R'_0(q){\subseteq}R_0(q) \wedge R_1(q){\subseteq}R'_1(q)) \\
&\iff f[(R'_0 \Rightarrow R'_1)(p)]g && (\because \text{Def. of } R'_0 \Rightarrow R'_1).
\end{aligned}$$

Finally, we consider the inference rule for transitivity. Suppose that the last step of the derivation of $\theta \preceq_\Delta \theta'$ has the form:

$$\frac{\theta \preceq_\Delta \theta_0 \quad \theta_0 \preceq_\Delta \theta'}{\theta \preceq_\Delta \theta'}$$

By the induction hypothesis, all of $[\![\theta]\!]^{\mathcal{C}}_\eta$, $[\![\theta_0]\!]^{\mathcal{C}}_\eta$ and $[\![\theta']\!]^{\mathcal{C}}_\eta$ have the same underlying cpos. Let R, R_0, R' be parameterized pers of $[\![\theta]\!]^{\mathcal{C}}_\eta$, $[\![\theta_0]\!]^{\mathcal{C}}_e ta$ and $[\![\theta']\!]^{\mathcal{C}}_\eta$, respectively. By the induction hypothesis again, we have that

$$\forall p.\ R(p)\ \subseteq\ R_0(p)\ \subseteq\ R'(p).$$

We have just shown that the lemma holds in this case. $\square$

Finally, we define the semantics of each typing judgment $\Gamma \vdash_\Delta M : \theta$ by an indexed family of morphisms in $\mathcal{C}$ of the form:

$$\{f_\eta : [\![\Delta \vdash \Gamma]\!]^{\mathcal{C}}_\eta \rightarrow [\![\Delta \vdash \theta]\!]^{\mathcal{C}}_\eta\}_{\eta \in [\![\Delta]\!]}.$$

The semantics is given by induction on the derivation of the judgment, and it is shown in Figure 6. The interpretation uses the categorical structure of $\mathcal{C}$ in a standard way. The only specific parts are the interpretation of basic imperative operations, where we use six basic semantic constants

$$\mathit{skip},\ \mathit{seq},\ \mathit{new},\ \mathit{read},\ \mathit{free},\ \text{and}\ \mathit{write},$$

which are also defined in the figure.

For this interpretation of terms, the question of well-definedness arises, because of the introduction and elimination of dependent function type $\Pi_i\theta$. The semantic definition of $\lambda i.M$ assumes that if Γ does not contain the variable i, it is interpreted as the same object in $\mathcal{C}$ no matter how we change or even drop the value of i in the index. The definition of $[\![ME]\!]^{\mathcal{C}}$ assumes that the reindexing precisely models the substitution. The following lemmas show that these two assumptions indeed hold.

Lemma 4.10. *If* $i \notin \Delta$ *and* $\Delta \vdash \theta$, *then*

$$\forall \eta \in [\![\Delta]\!].\, \forall n \in \mathit{Val}.\ [\![\Delta \vdash \theta]\!]^{\mathcal{C}}_{\eta} = [\![\Delta \cup \{i\} \vdash \theta]\!]^{\mathcal{C}}_{\eta[i \to n]}.$$

Proof. The lemma can be proved by straightforward induction on the structure of θ. We omit the details. □

Lemma 4.11. *If* $i \notin \Delta$ *and* $\Delta \vdash P$, *then*

$$\forall \eta \in [\![\Delta]\!].\, \forall n \in \mathit{Val}.\ [\![\Delta \vdash \Gamma]\!]^{\mathcal{C}}_{\eta} = [\![\Delta \cup \{i\} \vdash \Gamma]\!]^{\mathcal{C}}_{\eta[i \to n]}.$$

Proof. The lemma follows from Lemma 4.10, as shown below:

$$\begin{array}{rcll} [\![\Delta \cup \{i\} \vdash \Gamma]\!]^{\mathcal{C}}_{\eta[i \mapsto n]} & = & \Pi_{x \in \mathsf{dom}(\Gamma)} [\![\Delta \cup \{i\} \vdash \Gamma(x)]\!]^{\mathcal{C}}_{\eta[i \mapsto n]} & \\ & = & \Pi_{x \in \mathsf{dom}(\Gamma)} [\![\Delta \vdash \Gamma(x)]\!]^{\mathcal{C}}_{\eta} & (\because\ \text{Lemma 4.10}) \\ & = & [\![\Delta \vdash \Gamma]\!]^{\mathcal{C}}_{\eta}. & \end{array}$$

□

Lemma 4.12. *If* $i \notin \Delta$, $\Delta \cup \{i\} \vdash \theta$, *and* $\Delta \vdash E$, *then*

$$\forall \eta \in [\![\Delta]\!].\ [\![\Delta \vdash \theta[E/i]]\!]^{\mathcal{C}}_{\eta} = [\![\Delta \cup \{i\} \vdash \theta]\!]^{\mathcal{C}}_{\eta[i \to [\![E]\!]_{\eta}]}.$$

Proof. This lemma holds because the reindexing of the family fibration $\mathit{Fam}(\mathcal{C}) \to \mathsf{Set}$ preserves on the nose all the categorical structure that is used to interpret types. A more concrete, direct proof can be obtained by induction on the structure of θ. We omit the details. □

4.2.2. *Semantics* $[\![-]\!]$ *in* $\mathit{Fam}(\mathcal{D}) \to \mathsf{Set}$. The official semantics $[\![-]\!]$ of the language uses the fibration $\mathit{Fam}(\mathcal{D}) \to \mathsf{Set}$, rather than $\mathit{Fam}(\mathcal{C}) \to \mathsf{Set}$. It is obtained by applying the embedding functor $E\colon \mathcal{C} \to \mathcal{D}$ to the semantics $[\![-]\!]^{\mathcal{C}}$ of the previous section. Concretely, the semantics $[\![-]\!]$ is defined as follows: for all $\eta \in [\![\Delta]\!]$,

$$\begin{array}{rcccl} [\![\Delta \vdash \theta]\!]_{\eta} & = & E([\![\Delta \vdash \theta]\!]^{\mathcal{C}}_{\eta}) & = & [\![\Delta \vdash \theta]\!]^{\mathcal{C}}_{\eta}, \\ [\![\Delta \vdash \Gamma]\!]_{\eta} & = & E([\![\Delta \vdash \Gamma]\!]^{\mathcal{C}}_{\eta}) & = & [\![\Delta \vdash \Gamma]\!]^{\mathcal{C}}_{\eta}, \\ [\![\theta \preceq_{\Delta} \theta']\!]_{\eta} & = & E([\![\theta \preceq_{\Delta} \theta']\!]^{\mathcal{C}}_{\eta}), & & \\ [\![\Gamma \vdash_{\Delta} M : \theta]\!]_{\eta} & = & E([\![\Gamma \vdash_{\Delta} M : \theta]\!]_{\eta}). & & \end{array}$$

Note that in the first two equations, we use the fact that E is the identity on objects.

$$
\begin{aligned}
[\![\Gamma, x\colon \theta \vdash_\Delta x : \theta]\!]^{\mathcal{C}}_\eta \rho &= \rho(x) \\
[\![\Gamma \vdash_\Delta \lambda x\colon \theta. M : \theta \to \theta']\!]^{\mathcal{C}}_\eta \rho &= \lambda c.\ [\![\Gamma, x\colon \theta \vdash_\Delta M : \theta']\!]^{\mathcal{C}}_\eta (\rho[x \to c]) \\
[\![\Gamma \vdash_\Delta M M' : \theta]\!]^{\mathcal{C}}_\eta \rho &= ([\![\Gamma \vdash_\Delta M : \theta' \to \theta]\!]^{\mathcal{C}}_\eta \rho)\ ([\![\Gamma \vdash_\Delta M' : \theta']\!]^{\mathcal{C}}_\eta \rho) \\
[\![\Gamma \vdash_\Delta \lambda i. M : \Pi_i \theta]\!]^{\mathcal{C}}_\eta \rho &= \lambda n.\ [\![\Gamma \vdash_{\Delta \cup \{i\}} M : \theta]\!]^{\mathcal{C}}_{\eta[i \to n]} \rho \\
[\![\Gamma \vdash_\Delta M E : \theta[E/i]]\!]^{\mathcal{C}}_\eta \rho &= ([\![\Gamma \vdash_\Delta M : \Pi_i \theta]\!]^{\mathcal{C}}_\eta \rho)\ ([\![E]\!]_\eta) \\
[\![\Gamma \vdash_\Delta M : \theta']\!]^{\mathcal{C}}_\eta \rho &= ([\![\theta \preceq_\Delta \theta']\!]^{\mathcal{C}}_\eta)\ ([\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathcal{C}}_\eta \rho) \\
[\![\Gamma \vdash_\Delta \mathsf{fix}\, M : \theta]\!]^{\mathcal{C}}_\eta \rho &= \mathsf{lfix}\ ([\![\Gamma \vdash_\Delta M : \theta \to \theta]\!]^{\mathcal{C}}_\eta \rho) \\
[\![\Gamma \vdash_\Delta \mathsf{ifz}\, E\, M\, M' : \{P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_\eta \rho &= \mathsf{if}\ ([\![\Delta \vdash E]\!]_\eta = 0) \\
&\ \mathsf{then}\ [\![\Gamma \vdash_\Delta M : \{P \wedge E{=}0\}\text{-}\{Q\}]\!]^{\mathcal{C}}_\eta \rho \\
&\ \mathsf{else}\ [\![\Gamma \vdash_\Delta M' : \{P \wedge E{\neq}0\}\text{-}\{Q\}]\!]^{\mathcal{C}}_\eta \rho \\
[\![\Gamma \vdash_\Delta M; M' : \{P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_\eta \rho &= \mathit{seq}\ ([\![\Gamma \vdash_\Delta M{:}\{P\}\text{-}\{P'\}]\!]^{\mathcal{C}}_\eta \rho, \\
&\qquad\quad [\![\Gamma \vdash_\Delta M'{:}\{P'\}\text{-}\{Q\}]\!]^{\mathcal{C}}_\eta \rho) \\
[\![\Gamma \vdash_\Delta \mathsf{skip} : \{P\}\text{-}\{P\}]\!]^{\mathcal{C}}_\eta \rho &= \mathit{skip}\ (\bot) \\
[\![\Gamma \vdash_\Delta \mathsf{let}\, i{=}\mathsf{new}\, \mathsf{in}\, M : \{P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_\eta \rho &= \mathit{new}\ (\lambda n.\ [\![\Gamma \vdash_{\Delta \cup \{i\}} M{:}\{i{\mapsto}-{*}P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_{\eta[i \to n]} \rho) \\
[\![\Gamma \vdash_\Delta \mathsf{let}\, i{=}[E]\, \mathsf{in}\, M{:}\{\exists i. E{\mapsto}i{*}P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_\eta \rho &= \mathit{read}\ ([\![\Delta \vdash E]\!]_\eta) \\
&\qquad\quad (\lambda n.\ [\![\Gamma \vdash_{\Delta \cup \{i\}} M{:}\{E{\mapsto}i{*}P\}\text{-}\{Q\}]\!]^{\mathcal{C}}_{\eta[i \to n]} \rho) \\
[\![\Gamma \vdash_\Delta \mathsf{free}(E) : \{E{\mapsto}-\}\text{-}\{\mathsf{emp}\}]\!]^{\mathcal{C}}_\eta \rho &= \mathit{free}\ ([\![\Delta \vdash E]\!]_\eta)\ (\bot) \\
[\![\Gamma \vdash_\Delta [E]{:=}E' : \{E{\mapsto}-\}\text{-}\{E{\mapsto}E'\}]\!]^{\mathcal{C}}_\eta \rho &= \mathit{write}\ ([\![\Delta \vdash E]\!]_\eta, [\![\Delta \vdash E']\!]_\eta)\ (\bot)
\end{aligned}
$$

where *skip*, *seq*, *new*, *read*(m), *free*(m), and *write*(m, m') are the following morphisms in $\mathcal{C}$:

$$
m{\mapsto}- \in \mathit{Pred} \ \stackrel{\text{def}}{=}\ \{[m \to n] \mid n \in \mathit{Val}\} \qquad\qquad m{\mapsto}n \in \mathit{Pred} \ \stackrel{\text{def}}{=}\ \{[m \to n]\}
$$

$$
\begin{aligned}
\mathit{skip}_p &: 1 \to \mathsf{tri}(p, p) \\
\mathit{skip}_p &\stackrel{\text{def}}{=} \lambda x.\, \lambda h. \{h\} \\[4pt]
\mathit{seq}_{p,p',q} &: \mathsf{tri}(p, p') \times \mathsf{tri}(p', q) \to \mathsf{tri}(p, q) \\
\mathit{seq} &\stackrel{\text{def}}{=} \lambda(c, c').\, \lambda h.\, \{\mathit{wrong} \mid \mathit{wrong} \in c(h)\} \cup \bigcup\{c'(h') \mid h' \in c(h)\} \\[4pt]
\mathit{new}_{p,q} &: (\Pi_{n \in \mathit{Val}} \mathsf{tri}(n{\mapsto}- * p, q)) \to \mathsf{tri}(p, q) \\
\mathit{new} &\stackrel{\text{def}}{=} \lambda c.\, \lambda h.\ \bigcup\{c(n)([n \to n'] \cdot h) \mid n, n' \in \mathit{Val}\ \wedge\ n \notin \mathsf{dom}(h)\} \\[4pt]
\mathit{read}(m)_{(\{p_n\}_n, q)} &: (\Pi_{n \in \mathit{Val}} \mathsf{tri}(m{\mapsto}n * p_n, q)) \to \mathsf{tri}(\bigcup\{m{\mapsto}n * p_n \mid n \in \mathit{Val}\},\ q) \\
\mathit{read}(m) &\stackrel{\text{def}}{=} \lambda c.\, \lambda h.\, \mathsf{if}\ m \in \mathsf{dom}(h)\ \mathsf{then}\ c(h(m))(h)\ \mathsf{else}\ \{\mathit{wrong}\} \\[4pt]
\mathit{free}(m) &: 1 \to \mathsf{tri}(m{\mapsto}-, \mathit{emp}) \\
\mathit{free}(m) &\stackrel{\text{def}}{=} \lambda x.\, \lambda h.\, \mathsf{if}\ m \in \mathsf{dom}(h)\ \mathsf{then}\ \{h[m \to \mathit{undef}]\}\ \mathsf{else}\ \{\mathit{wrong}\} \\[4pt]
\mathit{write}(m, m') &: 1 \to \mathsf{tri}(m{\mapsto}-, m{\mapsto}m') \\
\mathit{write}(m, m') &\stackrel{\text{def}}{=} \lambda x.\, \lambda h.\, \mathsf{if}\ m \in \mathsf{dom}(h)\ \mathsf{then}\ \{h[m \to m']\}\ \mathsf{else}\ \{\mathit{wrong}\}
\end{aligned}
$$

Figure 6: Interpretation of Terms

We point out that $[\![-]\!]$ can be presented in a compositional style, using the categorical structure of the fibration $\mathit{Fam}(\mathcal{D}) \to \mathsf{Set}$.[6] In that presentation, the types are interpreted using exponentials, small products, inv' and tri' for $\mathcal{D}$; and the terms are interpreted by appropriate categorical combinators and the embedding of the six constants in Figure 6. This direct definition of $[\![-]\!]$ is identical to the semantics in this section, because the embedding functor E preserves all the relevant categorical structure (Lemma 4.8).

4.3. **Adequacy.** Our semantics of terms needs further justification in two ways. First, the interpretation of a typing judgment needs to be shown coherent. The interpretation is defined over a proof derivation of the judgment, so two different derivations of the same judgment might have different denotations. This is troublesome for us especially, because our goal is to give a semantics of a programming language with a separation-logic type system, instead of a semantics of a proof in separation logic. Second, the connection with the standard semantics needs to be provided. Our semantics uses subsumption which never arises in the standard interpretation. Thus, it could be substantially different from the standard interpretation. In this section, we provide justification for both of these two issues.

We consider another interpretation $[\![-]\!]^{\mathsf{CPO}}$ of our language, called standard interpretation, which ignores all assertions in the types. In the standard interpretation, $\{P\}$-$\{Q\}$ means the same thing no matter what P and Q are, and for all P, $\theta \otimes P$ and θ have identical interpretations. Let tri'' be the constant functor from $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$ to CPO such that $\mathsf{tri}''(p, q) = \mathsf{comm}$, and let inv'' be a functor given by the first projection from $\mathsf{CPO} \times \mathcal{P}_r$ to CPO. The standard interpretation is the interpretation in Section 4.2.1, where we use CPO, tri'' and inv'' instead of $\mathcal{C}$, tri and inv. It interprets types and type assignments just like the interpretation in Section 4.2.1, but it uses functors on CPO, instead of those on $\mathcal{C}$.

Lemma 4.13. *If a subtype relation $\theta \preceq_\Delta \theta'$ is derivable, then θ and θ' have the identical denotation in the standard interpretation.*

Proof. We prove the lemma by induction on the derivation of the subtype relation $\theta \preceq_\Delta \theta'$. First, we consider the case that the subtype relation is derived by an axiom. In all six axioms, θ and θ' are both Hoare-triple types, or they are different only for the invariant added by $\otimes$. Note that in the standard interpretation, all triple types mean the same cpo comm and the added invariants by $\otimes$ are ignored. Thus, we have that $[\![\theta]\!]^{\mathsf{CPO}}_\eta = [\![\theta']\!]^{\mathsf{CPO}}_\eta$ for all environments $\eta \in [\![\Delta]\!]$. Next, we consider the cases where some inference rule is applied at the last step of the derivation. Pick an environment η in $[\![\Delta]\!]$. If the last rule in the derivation is Consequence, both θ and θ' are Hoare-triple objects, so $[\![\theta]\!]^{\mathsf{CPO}}_\eta$ and $[\![\theta']\!]^{\mathsf{CPO}}_\eta$ are the same cpo comm. If the last applied rule is an inference rule other than Consequence, $[\![\theta]\!]^{\mathsf{CPO}}_\eta$ and $[\![\theta']\!]^{\mathsf{CPO}}_\eta$ are obtained by applying the same functor on the denotations of their subparts. By applying the induction hypothesis to these subparts, we can prove the lemma. For instance, if the last applied rule is the structural rule for $\to$, there are $\theta_0, \theta_0', \theta_1, \theta_1'$ such that

$$\theta = \theta_0 \to \theta_1,\ \ \theta = \theta_0' \to \theta_1',\ \ \theta_0' \preceq_\Delta \theta_0,\ \text{ and }\ \theta_1 \preceq_\Delta \theta_1'.$$

By the induction hypothesis, $[\![\theta_i]\!]^{\mathsf{CPO}}_\eta = [\![\theta_i']\!]^{\mathsf{CPO}}_\eta$ for $i = 0, 1$. This implies that $[\![\theta]\!]^{\mathsf{CPO}}_\eta$ and $[\![\theta]\!]^{\mathsf{CPO}}_\eta$ are identical. □

[6]The conference version of this paper defined $[\![-]\!]$ in such a style.

The standard interpretation defines the meaning of typing judgments $\Gamma \vdash_\Delta M : \theta$, by repeating the clauses in Figure 6. Although the interpretation is given inductively on the typing derivation, Lemma 4.13 ensures that $[\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathsf{CPO}}$ does not depend on derivations, because it guarantees that $[\![\theta \preceq_\Delta \theta']\!]^{\mathsf{CPO}}$ is the identity morphism. As usual, we can give the operational semantics, and prove the computational adequacy of the standard interpretation. Since this is completely standard, we omit it.

The standard interpretation is closely related to the semantics $[\![-]\!]^{\mathcal{C}}$ in Section 4.2.1. Note that from the category $\mathcal{C}$ to CPO, there is a forgetful functor F that maps an object (A, R) to A, and a morphism f to f. This forgetful functor preserves all the categorical structure of $\mathcal{C}$ that we use to interpret the types of our language:

Lemma 4.14. *F is a faithful functor that preserves the CCC structure and the small products of $\mathcal{C}$, and makes the following diagrams commute.*

$$\begin{array}{ccc} \mathcal{C} \times \mathcal{P}_r & \xrightarrow{\mathsf{inv}} & \mathcal{C} \\ {\scriptstyle F\times\mathsf{Id}}\downarrow & & \downarrow{\scriptstyle F} \\ \mathsf{CPO} \times \mathcal{P}_r & \xrightarrow{\mathsf{inv}''} & \mathsf{CPO} \end{array} \qquad \begin{array}{ccc} \mathcal{P}^{\mathsf{op}} \times \mathcal{P} & \xrightarrow{\mathsf{tri}} & \mathcal{C} \\ {\scriptstyle \mathsf{Id}}\downarrow & & \downarrow{\scriptstyle F} \\ \mathcal{P}^{\mathsf{op}} \times \mathcal{P} & \xrightarrow{\mathsf{tri}''} & \mathsf{CPO} \end{array}$$

Proof. First, we prove that the forgetful functor F preserves the exponentials and small products of $\mathcal{C}$. For this, it is sufficient to prove the preservation of four elements: exponential objects, small product objects, evaluation morphisms, and projection morphisms. Note that both the CCC structure and small products of $\mathcal{C}$ are defined using the corresponding structure of CPO; the first components of exponential objects and small product objects of $\mathcal{C}$ are defined by exponential objects and small product objects of CPO, and evaluation morphisms and projection morphisms in $\mathcal{C}$ are precisely evaluation morphisms and projection morphisms in CPO. Since F projects the first component of each object in $\mathcal{C}$ and maps each morphism in $\mathcal{C}$ to itself, it preserves the required four elements. For instance, for all objects $(A, R), (B, S)$ in $\mathcal{C}$, the first component of their exponential $(A, R) \Rightarrow (B, S)$ is the cpo $A \Rightarrow B$ of continuous functions from A to B, which is precisely the exponential of A and B in CPO. Thus, $F((A, R) \Rightarrow (B, S))$ is $F(A) \Rightarrow F(B)$.

Next, we prove that the diagram for inv and inv'' commutes. Since inv'' is the projection of the first component, $\mathsf{inv}'' \circ (F \times \mathsf{Id})$ is $F \circ \mathsf{fst}$. So, it suffices to show that $F \circ \mathsf{fst} = F \circ \mathsf{inv}$. Consider objects $((A, R), p)$, $((B, S), q)$ and a morphism $(f, p \sqsubseteq q)\colon ((A, R), p) \to ((B, S), q)$ in $\mathcal{C} \times \mathcal{P}_r$. Then,

$$\begin{array}{l} (F \circ \mathsf{inv})((A, R), p) \;=\; F(A, R(p * -)) \;=\; A \;=\; (F \circ \mathsf{fst})((A, R), p), \quad \text{and} \\ (F \circ \mathsf{inv})(f, p \sqsubseteq q) \;=\; F(f) \;=\; f \;=\; (F \circ \mathsf{fst})(f, p \sqsubseteq q). \end{array}$$

Thus, $F \circ \mathsf{fst} = F \circ \mathsf{inv}$, as required.

Finally, we prove the commutative diagram for tri and tri''. Consider objects (or predicate pairs) $(p, q), (p', q')$ and a morphism $(p' \subseteq p, q \subseteq q')\colon (p, q) \to (p', q')$ in $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$. Then,

$$(F \circ \mathsf{tri})(p, q) \;=\; F([p, q]) \;=\; \mathsf{comm} \quad \text{and} \quad (F \circ \mathsf{tri})(p' \subseteq p, q \subseteq q') \;=\; \mathsf{id}.$$

Thus, $F \circ \mathsf{tri}$ is the constant functor to comm, so it is identical to tri''. □

Lemma 4.14 implies that the interpretation of types in CPO factors through the interpretation in $\mathcal{C}$. The following lemma show that the interpretation of terms has a similar property.

Proposition 4.15. *The functor $F\colon \mathcal{C} \to \mathsf{CPO}$ preserves the interpretation of terms: for all typing judgments $\Gamma \vdash_\Delta M : \theta$ and all $\eta \in [\![\Delta]\!]$,*

$$F([\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathcal{C}}_\eta) = [\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathsf{CPO}}_\eta.$$

Proof. Pick an arbitrary $\eta \in [\![\Delta]\!]$ and choose any $\rho' \in [\![\Gamma]\!]^{\mathsf{CPO}}_\eta$. Then,

$$F([\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathcal{C}}_\eta)(\rho') = [\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathcal{C}}_\eta\ \rho',$$

because $F(f)$ only changes the "type" of f, not the implementation of f. Thus, it is sufficient to show that

$$\forall \eta, \rho'.\ [\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathcal{C}}_\eta\ \rho' = [\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathsf{CPO}}_\eta\ \rho'.$$

We prove this equality by induction on the derivation of $\Gamma \vdash_\Delta M : \theta$. Since $[\![-]\!]^{\mathsf{CPO}}$ and $[\![-]\!]$ use the same clauses to define the meaning of $\Gamma \vdash_\Delta M\colon \theta$, the induction easily goes through in all cases. For instance, consider the case where the subsumption rule is applied at the last step of the derivation. For all environments $\eta \in [\![\Delta]\!]$ and all $\rho' \in [\![\Gamma]\!]^{\mathsf{CPO}}_\eta$,

$$\begin{aligned}
[\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathcal{C}}_\eta\ \rho' &= [\![\theta_0 \preceq_\Delta \theta]\!]^{\mathcal{C}}_\eta\ ([\![\Gamma \vdash_\Delta M : \theta_0]\!]^{\mathcal{C}}_\eta\ \rho') \\
&= [\![\Gamma \vdash_\Delta M : \theta_0]\!]^{\mathcal{C}}_\eta\ \rho' && (\because\ [\![\theta_0 \preceq_\Delta \theta]\!]^{\mathcal{C}}_\eta\, x = x) \\
&= [\![\Gamma \vdash_\Delta M : \theta_0]\!]^{\mathsf{CPO}}_\eta\ \rho' && (\because\ \text{Induction Hypothesis}) \\
&= ([\![\theta_0 \preceq_\Delta \theta]\!]^{\mathsf{CPO}}_\eta) \circ ([\![\Gamma \vdash_\Delta M : \theta_0]\!]^{\mathsf{CPO}}_\eta\ \rho') && (\because\ [\![\theta_0 \preceq_\Delta \theta]\!]^{\mathsf{CPO}}_\eta\ x = x) \\
&= [\![\Gamma \vdash_\Delta M : \theta]\!]^{\mathsf{CPO}}_\eta.
\end{aligned}$$

□

Recall that the official semantics $[\![-]\!]$ of our language is obtained by applying the full functor E to the semantics $[\![-]\!]^{\mathcal{C}}$, and that the functor F is faithful. Together with these facts, Lemma 4.14 and Proposition 4.15 show that the official semantics $[\![-]\!]$ is obtained from the standard interpretation $[\![-]\!]^{\mathsf{CPO}}$ by first selecting some elements, and then quotienting those selected elements.

Corollary 4.16. *The semantics $[\![-]\!]$ is coherent: the semantics of a typing judgment does not depend on derivations.*

Proof. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two derivations of a judgment $\Gamma \vdash_\Delta M : \theta$. We note that the standard semantics is coherent; only the subsumption rule is not syntax-directed, but in the standard semantics, this rule does not contribute to the interpretation, because all the subtype relations $\theta \preceq_\Delta \theta'$ denote the family of identity morphisms. Thus, for all environments $\eta \in [\![\Delta]\!]$, we have

$$[\![\mathcal{P}_1]\!]^{\mathsf{CPO}}_\eta = [\![\mathcal{P}_2]\!]^{\mathsf{CPO}}_\eta.$$

Then, by Proposition 4.15 and the faithfulness of F,

$$\begin{aligned}
[\![\mathcal{P}_1]\!]^{\mathsf{CPO}}_\eta = [\![\mathcal{P}_2]\!]^{\mathsf{CPO}}_\eta \quad &\Longrightarrow \quad F([\![\mathcal{P}_1]\!]^{\mathcal{C}}_\eta) = F([\![\mathcal{P}_2]\!]^{\mathcal{C}}_\eta) \\
&\Longrightarrow \quad [\![\mathcal{P}_1]\!]^{\mathcal{C}}_\eta = [\![\mathcal{P}_2]\!]^{\mathcal{C}}_\eta \qquad (\because\ F \text{ is faithful}) \\
&\Longrightarrow \quad E([\![\mathcal{P}_1]\!]^{\mathcal{C}}_\eta) = E([\![\mathcal{P}_2]\!]^{\mathcal{C}}_\eta) \\
&\Longrightarrow \quad [\![\mathcal{P}_1]\!]_\eta = [\![\mathcal{P}_2]\!]_\eta \qquad (\because\ \text{Definition of } [\![-]\!]).
\end{aligned}$$

□

5. Conjunction Rule

The conjunction rule is often omitted from Hoare logic, but it is a useful proof rule that lets one combine two Hoare triples about a single command. In our type system, it can be expressed as follows:

$$\frac{\Gamma \vdash_\Delta M : \{P\}\text{-}\{Q\} \quad \Gamma \vdash_\Delta M : \{P'\}\text{-}\{Q'\}}{\Gamma \vdash_\Delta M : \{P \wedge P'\}\text{-}\{Q \wedge Q'\}}$$

Unfortunately, we cannot immediately include the conjunction rule in our type system. In [9], Reynolds has proved that if a proof system contains the conjunction rule and the second-order frame rule, together with Consequence and the ordinary (first-order) frame rule, then the system becomes inconsistent. More specifically, Reynolds's result implies that once the conjunction rule is added to our type system, we can derive $\vdash_{\{\}} \mathsf{skip} : \{(\exists x, y.x{\mapsto}y) * \mathsf{true}\}\text{-}\{\mathsf{false}\}$, which incorrectly expresses that skip diverges when the input heap is not empty.

In the case of the second-order frame rule, several solutions have been proposed to overcome this problem. In this section we adopt one of the proposals, modify the separation-logic type system accordingly, and extend the modified system with the conjunction rule. Then, we define an adequate semantics of the new type system, thereby showing that all the higher-order frame rules can be used with the conjunction rule, as long as the frame rules add only precise invariants.

We recall the definition of precise predicates in separation logic [9]. A predicate p is precise if and only if for every heap h, there is at most one subheap h_0 of h (i.e., $h_0 \cdot h_1 = h$ for some h_1) such that $h_0 \in p$. We also call an assertion $\Delta \vdash P$ *precise* when $[\![P]\!]_\eta$ is a precise predicate for all $\eta \in [\![\Delta]\!]$.

The proposal that we use is to restrict the second-order frame rule such that it is used with only precise assertions. We adopt the proposal in our separation-logic type system by limiting the second parameter of the type constructor $\otimes$ to precise assertions. Note that in the resulting restricted type system, only precise assertions can be added as invariants, because the generalized frame rule $\theta \preceq_\Delta \theta \otimes P$ is applicable only with a precise assertion P. Thus, the second or third order frame rule can add only precise assertions as invariants. We may then extend the restricted type system with the conjunction rule. Note that the result of this extension, denoted $\mathbf{T}$, includes the conjunction rule and all (restricted) higher-order frame rules. In the remainder of this section, we focus on giving an adequate semantics of $\mathbf{T}$.

Before giving the semantics of $\mathbf{T}$, we point out that requiring invariants to be precise is not as restrictive as it seems; all the examples in Section 3.1 use precise invariants only, so they typecheck in $\mathbf{T}$.

The semantics of the type system $\mathbf{T}$ is given by categories $\mathcal{C}_0$ and $\mathcal{D}_0$. The category $\mathcal{C}_0$ is identical to $\mathcal{C}$, except that the per component of each object is parameterized by *precise* predicates, instead of all predicates. An object in $\mathcal{C}_0$ is a pair of cpo A and parameterized per R, such that (1) the parameterization of R is over *precise* predicates, and (2) for all *precise* predicates p, q, the per $R(p)$ implies $R(p * q)$, i.e., $R(p) \subseteq R(p * q)$. A morphism $f\colon (A, R) \to (B, S)$ in $\mathcal{C}_0$ is a continuous function ¿From A to B that maps $R(p)$-related elements to $S(p)$-related elements for all precise p. The other category $\mathcal{D}_0$ is constructed by quotienting morphisms in $\mathcal{C}_0$, in the same way as $\mathcal{D}$ is constructed from $\mathcal{C}$.

The categories $\mathcal{C}_0$ and $\mathcal{D}_0$ have all the categorical structure that we have used in the semantics in Section 4. They are cartesian closed categories with all the small products, and they have functors for invariant extension and Hoare triples. The only subtlety is that the preorder $\mathcal{P}_r$, which is used for functors for invariant extension in Section 4, is now replaced by the preorder of precise predicates with the following order $\sqsubseteq_p$: for all precise predicates p, q,

$$p \sqsubseteq_p q \iff \text{there exists a precise } r \text{ such that } p * r = q.$$

This categorical structure is preserved by the functors for invariant extension, the forgetful functor $F_0 \colon \mathcal{C}_0 \to \mathsf{CPO}$, and the quotienting functor $E_0 \colon \mathcal{C}_0 \to \mathcal{D}_0$, in the way expressed by Lemmas 4.3, 4.14, and 4.8. All the definitions and results in Section 4.2 and 4.3 can easily be transferred to $\mathcal{C}_0$ and $\mathcal{D}_0$, as long as they are concerned with $\mathbf{T}$ without the conjunction rule. We now explain how to soundly interpret the conjunction rule.

Define a continuous function *con* from $\mathsf{comm} \times \mathsf{comm}$ to comm as follows:

$$\begin{array}{rcl} \mathit{wrong} \in \mathit{con}(c, c')(h) & \iff & \mathit{wrong} \in c(h) \cup c'(h) \\ h' \in \mathit{con}(c, c')(h) & \iff & h' \in c(h) \cap c'(h) \end{array}$$

Function *con* is the key element in our interpretation of the conjunction rule. Intuitively, $\mathit{con}(c, c')$ is a command that is better than c and c': it satisfies more Hoare triples than c and c', as long as we consider triples with sufficiently strong preconditions, those which ensure that both c and c' run without generating *wrong*.

Lemma 5.1. *Function con is well-defined. In particular, for all* $(c, c') \in \mathsf{comm} \times \mathsf{comm}$*,* $\mathit{con}(c, c')$ *satisfies the safety monotonicity and frame property.*

Proof. The continuity follows from the fact that $\mathit{con}(c, -)$ and $\mathit{con}(-, c)$ preserve arbitrary nonempty unions. Here we focus on proving that *con* is a well-defined function. Pick $(c, c') \in \mathsf{comm} \times \mathsf{comm}$. To prove that $\mathit{con}(c, c') \in \mathsf{comm}$, we should show that $\mathit{con}(c, c')$ satisfies the safety monotonicity and the frame property.

- *Safety Monotonicity:* Consider heaps h_0, h_1 such that $\mathit{wrong} \notin \mathit{con}(c, c')(h_0)$ and $h_0 \# h_1$. Then, *wrong* is neither in $c(h_0)$ nor in $c'(h_0)$. Thus, by the safety monotonicity of c and c', we have that $\mathit{wrong} \notin c(h_0 \cdot h_1)$ and $\mathit{wrong} \notin c'(h_0 \cdot h_1)$. This implies that $\mathit{wrong} \notin \mathit{con}(c, c')(h_0 \cdot h_1)$, as required.
- *Frame Property:* Suppose that $h_0 \# h_1$, $\mathit{wrong} \notin \mathit{con}(c, c')(h_0)$, and $h' \in \mathit{con}(c, c')(h_0 \cdot h_1)$. Note that while proving the previous item, we have shown two facts: (1) $\mathit{con}(c, c')(h_0 \cdot h_1)$ does not contain *wrong*, and (2) neither $c(h_0)$ nor $c'(h_0)$ contains *wrong*. The first fact implies that $\mathit{con}(c, c')(h_0 \cdot h_1) = c(h_0 \cdot h_1) \cap c'(h_0 \cdot h_1)$, because by the definition of *con*,

  $$\begin{array}{rcl} c(h_0 \cdot h_1) \cap c'(h_0 \cdot h_1) & \subseteq & \mathit{con}(c, c')(h_0 \cdot h_1) \\ & \subseteq & \big(c(h_0 \cdot h_1) \cap c'(h_0 \cdot h_1)\big) \cup \{\mathit{wrong}\}. \end{array}$$

  Since h' is in $\mathit{con}(c, c')(h_0 \cdot h_1)$ and $\mathit{con}(c, c')(h_0 \cdot h_1) = c(h_0 \cdot h_1) \cap c'(h_0 \cdot h_1)$, heap h' is in $c(h_0 \cdot h_1)$ as well as in $c'(h_0 \cdot h_1)$. Moreover, by the second fact proved in the previous item, $\mathit{wrong} \notin c(h_0)$ and $\mathit{wrong} \notin c'(h_0)$. Thus, we can apply the frame property of c and c' here. Once the property is applied, we obtain subheaps h_0', h_0'' of h' such that

  $$h_0' \cdot h_1 = h_0'' \cdot h_1 = h' \;\;\wedge\;\; h_0' \in c(h_0) \;\;\wedge\;\; h_0'' \in c'(h_0).$$

Note that the equalities force h'_0 and h''_0 to be the same. So, h'_0 should be in $c(h_0) \cap c'(h_0) = con(c, c')(h_0)$. We have just proved that h'_0 is the heap required by the frame property of $con(c, c')$.

□

For all predicates p, q, define an object $[p, q]$ in $\mathcal{C}_0$ just like the corresponding triple object in $\mathcal{C}$, except that the second component of $[p, q]$ is a family of pers indexed by precise predicates. The following lemma expresses that *con* properly models a semantic version of the conjunction rule in $\mathcal{C}_0$.

Lemma 5.2. *For all predicates p, q, p', q', function con is a morphism in $\mathcal{C}_0$ that has type $[p, q] \times [p', q'] \to [p \cap p', q \cap q']$.*

Proof. Let R, S, T be pers parameterized by precise predicates, such that

$$(\mathsf{comm}, R) = [p, q], \quad (\mathsf{comm}, S) = [p', q'], \quad \text{and} \quad (\mathsf{comm}, T) = [p \cap p', q \cap q'].$$

Because of Lemma 5.1, *con* is a well-defined continuous function from $\mathsf{comm} \times \mathsf{comm}$ to comm. Thus, it suffices to show that for all precise predicates r,

$$con[R(r) \times S(r) \to T(r)]con.$$

Consider precise predicate r, and command pairs $(c_0, c'_0), (c_1, c'_1)$, such that

$$(c_0, c'_0)[R(r) \times S(r)](c_1, c'_1).$$

First, we show that $con(c_0, c'_0)$ and $con(c_1, c'_1)$ are in the domain of per $T(r)$. We focus on $con(c_0, c'_0)$, because $con(c_1, c'_1) \in |T(r)|$ can be proved similarly. Pick a heap h in $(p \cap p') * r$. Then, h is in $p * r$ and $p' * r$. Note that c_0 and c'_0 are in $|R(r)|$ and $|S(r)|$, and R and S are the per components of $[p, q]$ and $[p', q']$. Thus, neither $c_0(h)$ nor $c'_0(h)$ contains *wrong*, $c_0(h) \subseteq q * r$, and $c'_0 \subseteq q' * r$. Thus,

$$con(c_0, c'_0)(h) \;\; = \;\; c_0(h) \cap c'_0(h) \;\; \subseteq \;\; p * r \cap q' * r \;\; = \;\; (p \cap q') * r.$$

The first equality follows from the definition of *con*, because $wrong \notin c_0(h)$ and $wrong \notin c'_0(h)$. And the last equality holds, because for all precise predicates r_0, $- * r_0$ distributes over $\cap$.

Next, we show that $con(c_0, c'_0)$ and $con(c_1, c'_1)$ are $T(r)$-related. Since both $con(c_0, c'_0)$ and $con(c_1, c'_1)$ are in $|T(r)|$, it is enough to prove that

$$\forall h \in (p \cap p') * r * true.\;\; con(c_0, c'_0)(h) = con(c_1, c'_1)(h).$$

Pick h from $(p \cap p') * r * true$. Then, $h \in p * t * true$ and $h \in p' * t * true$. Since $c_0[R(p)]c_1$ and $c'_0[S(p)]c'_1$, these two membership relations of h imply that none of $c_0(h), c_1(h), c'_0(h), c'_1(h)$ contains *wrong*, $c_0(h) = c_1(h)$, and $c'_0(h) = c'_1(h)$. Thus,

$$con(c_0, c'_0)(h) \;\; = \;\; c_0(h) \cap c'_0(h) \;\; = \;\; c_1(h) \cap c'_1(h) \;\; = \;\; con(c_1, c'_1)(h).$$

Since none of $c_0(h), c_1(h), c'_0(h), c'_1(h)$ contains *wrong*, the first and last equalities follow from the definition of *con*. □

The conjunction rule

$$\frac{\Gamma \vdash_\Delta M : \{P\}\text{-}\{Q\} \quad \Gamma \vdash_\Delta M : \{P'\}\text{-}\{Q'\}}{\Gamma \vdash_\Delta M : \{P \wedge P'\}\text{-}\{Q \wedge Q'\}}$$

is now interpreted as follows:

$$[\![\Gamma \vdash_\Delta M : \{P \wedge P'\}\text{-}\{Q \wedge Q'\}]\!]^X_\eta = con' \circ \langle [\![\Gamma \vdash_\Delta M : \{P\}\text{-}\{Q\}]\!]^X_\eta,\ [\![\Gamma \vdash_\Delta M : \{P'\}\text{-}\{Q'\}]\!]^X_\eta \rangle$$

where X is $\mathcal{C}_0$, $\mathcal{D}_0$ or CPO. The standard semantics in CPO and the filtering semantics in $\mathcal{C}_0$ uses *con* for con', and in a direct-style presentation, the quotienting semantics in $\mathcal{D}_0$ uses the equivalence class $[con]$ for con'. Note that in the standard semantics, the conjunction rule is interpreted as the identity, because $con \circ \langle f, f \rangle = f$, for all morphisms f in CPO.

Since E' and F' preserve the semantic entities for con', they preserve the interpretation of terms in the three semantics. ¿From this preservation, the coherence of the quotienting semantics follows. Moreover, since the conjunction rule means the identity in the standard semantics, the preservation of interpretations also implies that the conjunction rule is always implemented by the identity function in all three semantics, thereby reflecting the fact that the rule does not have any computational meaning.

6. Related Work

The (first order) frame rule was discovered in the early days of separation logic [5], and it was a main reason for the success of that logic. For example, it was vital in the proofs of garbage collection algorithms in [21] and [4]. Recently, the second-order frame rule, which allows reasoning about simple first-order modules, was discovered [9]. This naturally encouraged the question of whether there are more general frame rules that apply to higher types.

Other type systems which track state changes have been proposed in the work on typed assembly languages [7, 2, 20]. Their main focus is to obtain sound rules for proving the safety of programs. Thus, they mostly use easy-to-define conventional operational semantics, and prove the soundness of the proof system syntactically (i.e., by subject reduction and progress lemmas), or "logically" [20]: each type is interpreted as a subset of a single universe of "meanings," and a typing judgment is interpreted as a specification for the behavior of programs, like a Hoare triple in separation logic. Our separation-logic type system is more refined in that it allows the full power of separation logic in the types and, moreover, we also treat higher-order procedures.

The semantics of idealized algol has been studied intensively [11, 18, 10, 14]. Normally, the semantics is parameterized by the shape of the memory. The indexing in the fibration in our semantics follows this tradition, and it models the shape of the stack. However, the other indexing of our semantics, the indexing by invariant predicates over heaps, has not been used in the literature before.

The construction of the category $\mathcal{D}$ is an instance of the Kripke quotient by Mitchell and Moggi [6]. The families of pers in $\mathcal{D}$ form a Kripke logical relation on CPO indexed by the preorder category $\mathcal{P}_r$; our condition on each family ensures that the requirement of Kripke monotonicity holds. This Kripke logical relation produces $\mathcal{D}$ by Mitchell and Moggi's construction.

The idea of proving coherence by relating two languages comes from Reynolds [19]. Reynolds proved the coherence of the semantics of typed lambda calculus with subtyping, by connecting it with the semantics of untyped lambda calculus. We use the general direction of Reynolds's proof, but the details of our proof are quite different from Reynolds's, because we consider very different languages.

7. Conclusion and Future Directions

We have presented a type system for idealized algol extended with heaps that includes separation-logic specifications as types and, moreover, defined the coherent semantics of idealized algol typed with this system.

One shortcoming of our type system is that the higher-order frame rules in the system allow only static modularity [12]. With the higher-order frame rules alone, we cannot capture all the the information hiding aspect of dynamically allocated data structures as needed for modeling abstract data types. However, it is well-known that abstract data types can be modeled using existential types and we are currently considering to enrich the assertion language with predicate variables, as in the recently introduced higher-order version of separation logic [3], and to extend the types with dependent product and sums over *predicates*.

Yet another future direction is to define a parametric model. Uday Reddy pointed out that separation-logic types should validate stronger reasoning principles for data abstraction than ordinary types, because they let us control what clients can access more precisely. Formalizing his intuition is the goal of the parametricity semantics. We currently plan to use category $\mathcal{C}'$ which replaces each *predicate-indexed* family of *pers* in $\mathcal{C}$ by a *relation-indexed* family of *saturated relations*: an object in $\mathcal{C}'$ is a cpo paired with a family T of *binary* relations such that (1) T is indexed by a "typed" relation $r\colon p \leftrightarrow q$ on heaps (i.e., $r \subseteq p \times q$); (2) for each predicate p, T at the diagonal relation Δ_p is a per; (3) for all $r\colon p \leftrightarrow q$, $T(r)$ is a saturated relation between pers $T(\Delta_p)$ and $T(\Delta_q)$; (4) $T(r) \subseteq T(r * r')$. The morphisms in $\mathcal{C}'$ are continuous functions that preserve the families of relations. This category has all the categorical structure of $\mathcal{C}$ that we used in the semantics of this paper. However, it is difficult to interpret the triple types such that the memory allocator new lives in the category. Overcoming this problem will be the focus of our research in this direction.

Finally, we would like to extend the relational separation logic [22] to higher-order, following the style of system $\mathcal{R}$ [1], and we want to explore the Curry-Howard correspondence of our type system with specification logic [15].

Acknowledgements

We have benefitted greatly from discussions with Uday Reddy, Peter O'Hearn, and David Naumann. We would like to thank anonymous referees and Rasmus Lerchedahl Petersen for providing useful suggestions, which in particular helped us to improve the presentation of the paper. Yang was supported by grant No. R08-2003-000-10370-0 ¿From the Basic Research Program of the Korea Science & Engineering Foundation. Yang, Birkedal and Torp-Smith were supported by Danish Technical Research Council Grant 56-00-0309.

References

[1] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Comput. Sci.*, 121(1-2):9–58, December 1993.

[2] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *Proc. of LICS'03*, 2003.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP'05*, Edinburgh, Scotland, April 2005.

[4] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proc. of POPL'04*, pages 220 – 231, Venice, Italy, 2004.

[5] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of POPL'01*, 2001.

[6] J. C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Appl. Logic*, 51:99–124, 1991.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. and Syst.*, 21(3):527 – 568, 1999.

[8] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Local reasoning about programs that alter data structures. In *Proc. of CSL'01*, pages 1 – 19, Paris, France, September 2001.

[9] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of POPL'04*, pages 268 – 280, Venice, Italy, 2004.

[10] Peter W. O'Hearn and Robert D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995.

[11] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.

[12] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proc. of POPL'05*, Long Beach, CA, USA, January 2005.

[13] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] U. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1):129 – 160, March 2004.

[15] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

[16] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, Houndsmill, Hampshire, 2000.

[17] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55 – 74, Copenhagen, Denmark, 2002.

[18] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[19] John C. Reynolds. The meaning of types — from intrinsic to extrinsic semantics. Research Series RS–00–32, BRICS, DAIMI, Department of Computer Science, University of Aarhus, December 2000. http://www.brics.dk/RS/00/32/.

[20] G. Tan, A. W. Appel, K. N. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In *Proc. of VMCAI '04*, January 2004.

[21] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.

[22] H Yang. Relational separation logic. Submitted to Theoretical Comput. Sci., October 2004.

[23] Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001. (Technical Report UIUCDCS-R-2001-2227).

[24] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of the 5th Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer-Verlag, April 2002.