

BI-Hyperdoctrines, Higher-order Separation Logic, and Abstraction*

BODIL BIERING, LARS BIRKEDAL, and NOAH TORP-SMITH

Department of Theoretical Computer Science

IT University of Copenhagen

We present a precise correspondence between separation logic and a simple notion of *predicate* BI, extending the earlier correspondence given between part of separation logic and *propositional* BI. Moreover, we introduce the notion of a BI hyperdoctrine and show that it soundly models classical and intuitionistic first- and higher-order predicate BI, and use it to show that we may easily extend separation logic to *higher-order*. We also demonstrate that this extension is important for program proving, since it provides sound reasoning principles for data abstraction in the presence of aliasing.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution and Maintenance—*documentation*; D.2.8 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*Assertions, Logics of programs, Specification techniques*

General Terms: Reliability, Theory, Verification

Additional Key Words and Phrases: Separation Logic, Hyperdoctrines, Abstraction

Contents

1	Introduction	3
2	BI Hyperdoctrines	5
2.1	Hyperdoctrines	5
2.2	BI Hyperdoctrines	7
3	Separation Logic modeled by BI-hyperdoctrines	10
3.1	The pointer model.	11
3.2	The pointer model as a BI hyperdoctrine.	12
3.3	An intuitionistic model.	13
3.4	The permissions model.	13
4	Some Consequences for Separation Logic	13
4.1	Formalizing Separation Logic	13

* An extended abstract of the present paper appeared in the proceedings of ESOP'05.

Lars Birkedal's and Noah Torp-Smith's research was partially supported by Danish Natural Science Research Council Grant 51-00-0315 and Danish Technical Research Council Grant 56-00-0309.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 1999 ACM 0164-0925/99/0100-0111 \$00.75

2	• Biering, Birkedal, Torp-Smith	
4.2	Logical Characterizations of Classes of Assertions	15
4.3	Predicates via Fixed Points	16
5	Higher-Order Separation Logic	17
5.1	Program Logic Judgments	19
5.2	Inference Rules	21
5.3	Informal Explanation of Rules	23
5.4	Soundness	24
5.5	A Derived Rule	26
6	Data Abstraction via Existential Quantification	27
6.1	Reasoning about Abstract Priority Queues	27
6.2	Sample Programs	28
6.3	Implementations of Priority Queues	30
6.3.1	Sorted, Singly-linked Lists	30
6.3.2	Doubly-linked Lists	31
6.4	Representation Independence	32
7	Some Applications of Universal Quantification	35
7.1	Polymorphic Types via Universal Quantification	35
7.2	Invariance	36
8	Related Work	36
9	Conclusion	37
A	Proof of Theorem 2.1	38
B	Implementation Using slists	39
C	Implementation Using dlists	41
D	Hints for Proof of slistEnque	43
E	Hints for Proof of dlistDeque	43

1. INTRODUCTION

Variants of the recent formalism of *separation logic* [Reynolds 2002; Ishtiaq and O’Hearn 2001] have been used to prove correct many interesting algorithms involving pointers, both in sequential and concurrent settings [O’Hearn 2004; Yang 2001; Birkedal et al. 2004]. Separation logic is a Hoare-style program logic, and its main advantage over traditional program logics is that it facilitates *local reasoning*. Different extensions of core separation logic [Reynolds 2002] have been used to prove correct various algorithms. For example, Yang [Yang 2001] extended the core logic with lists and trees and in [Birkedal et al. 2004] the logic was extended with finite sets and relations. Thus it is natural to ask whether one has to make a new extension of separation logic for every proof one wants to make. This would be unfortunate for formal verification of proofs in separation logic since it would make the enterprise of formal verification burdensome and dubious. We argue in this paper that there is a natural single underlying logic in which it is possible to *define* the various extensions and prove the expected properties thereof; this is then the single logic that should be employed for formal verification.

Part of the pointer model of separation logic, namely that given by heaps (but not stacks), has been related to *propositional BI*, the logic of bunched implications introduced by O’Hearn and Pym [O’Hearn and Pym 1999]. In this paper we show how the correspondence may be extended to a precise correspondence between all of the pointer model (including stacks) and a simple notion of *predicate BI*. We introduce the notion of a *BI hyperdoctrine*, a simple extension of Lawvere’s notion of hyperdoctrine [Lawvere 1969], and show that it soundly models predicate BI. The notion of predicate BI we consider is different from the one studied in [Pym 2002; 2004], which has a BI structure on contexts. However, we believe that our notion of predicate BI with its class of BI hyperdoctrine models is the right one for separation logic (Pym aimed to model multiplicative quantifiers; separation logic only uses additive quantifiers). To make this point, we show that the pointer model of separation logic exactly corresponds to the interpretation of predicate BI in a simple BI hyperdoctrine. This correspondence also allows us to see that it is simple to extend separation logic to *higher-order* separation logic. Now we briefly explain this extension and outline why it is important for program proving.

The force of separation logic comes from both its language of assertions – which is a variant of propositional BI [Pym 2002] – and its language of specifications, or Hoare triples. In the present paper, we extend both of these. First, we introduce an assertion language which is a variant of *higher-order predicate BI*. The extension from the traditional assertion language of separation logic simply allows function types, has a type **Prop** of proposition, and allows quantification over variables of all types. Thus the assertion language is higher-order, in the usual sense that it allows quantification over predicates. Next, we present a specification logic for a simple second-order programming language. We provide models for both the new assertion language and the specification logic, and provide inference rules for deriving valid specifications. As it turns out, it is technically straightforward to do so; this emphasizes that our notion of higher-order predicate BI is the correct one for separation logic.

Next we consider the effectiveness of higher-order separation logic and argue,

with the use of several examples, that it is quite effective. In particular, we show that higher-order separation logic can be used in a natural way to model data abstraction, via existential quantification over predicates corresponding to abstract resource invariants; we do so by means of a detailed example, which involves two implementations of priority queues. We believe this way of reasoning about data abstraction is more natural than the recently suggested abstract predicates of Parkinson and Bierman [Parkinson and Bierman 2005]. Moreover, we show that, using universal quantification over predicates, we can prove correct polymorphic operations on polymorphic data types, e.g., reversing a list of elements described by an arbitrary predicate. For this to be useful, however, it is clear that a higher-order programming language would be preferable (such that one could program many more useful polymorphic operations, e.g., the **map** function for lists) — we have chosen to stick with the simpler second-order language here to communicate more easily the ideas of higher-order separation logic.

Before proceeding with the technical development we give an intuitive justification of the use of BI hyperdoctrines to model higher-order predicate BI. A powerful way of obtaining models of BI is by means of functor categories (presheaves), using Day’s construction to obtain a doubly-closed structure on the functor category [Pym et al. 2004]. Such functor categories can be used to model *propositional* BI in two different senses: In the first sense, one models *provability*, entailment between propositions, and it works because the lattice of subobjects of the terminal object in such functor categories form a BI algebra (a doubly cartesian closed preorder). In the second sense, one models *proofs*, and it works because the whole functor category is doubly cartesian closed. Here we seek models of provability of *predicate* BI. Since the considered functor categories are toposes and hence model higher-order predicate logic, one might think that a straightforward extension is possible. But, alas, it is not the case. In general, for this to work, *every* lattice of subobjects (for any object, not only for the terminal object) should be a BI algebra and, moreover, to model substitution correctly, the BI algebra structure should be preserved by pulling back along any morphism. We show this can only be the case if the BI algebra structure is trivial, that is, coincides with the cartesian structure (see Theorem 2.2). Our theorem holds for any topos, not just for the functor categories just mentioned. Hence we need to consider a wider class of models for predicate BI than just toposes and this justifies the notion of a BI hyperdoctrine. The intuitive reason that BI hyperdoctrines work, is that predicates are not required to be modeled by subobjects, they can be something more general. Another important point of BI hyperdoctrines is that they are easy to come by: given any complete BI algebra B , there is a canonical BI hyperdoctrine in which predicates are modeled as B -valued functions; this is explained in detail in Example 2.2.

The rest of the paper is organized as follows. In Section 2, we first recall Lawvere’s notion of a *hyperdoctrine* [Lawvere 1969] and briefly recall how it can be used to model intuitionistic and classical first- and higher-order predicate logic. More details about this can be found in the handbook chapter [Pitts 2001] and the book [Jacobs 1999]. We then introduce the concept of a BI-hyperdoctrine and show that it models BI. In Section 3, we show that the standard pointer model of BI is an instance of our class of models. The new class of models provides a straightforward way to give semantics to a *higher-order* extension of BI, and we

discuss ramifications of this extension for separation logic in Section 4. In Section 5, we introduce the programming language considered in this paper. It is a simple extension of the standard programming language of separation logic with simple procedures and calls to these. We use the higher-order logic just introduced to give a specification logic for the programming language. In Section 6, we present examples which illustrate how this specification logic can be used to reason about data abstraction and representation independence, using existential quantification over predicates. In Section 7 we present some simple applications of universal quantification over predicates in program proving. In the last sections we give pointers to related work, and conclude.

This paper is a full version of an extended abstract presented at the ESOP 2005 conference. Compared to the conference version, this paper includes more detailed proofs, and a much more extensive discussion of applications of higher-order separation logic in program proving, in particular for data abstraction.

2. BI HYPERDOCTRINES

We first introduce Lawvere’s notion of a hyperdoctrine [Lawvere 1969] and briefly recall how it can be used to model intuitionistic and classical first- and higher-order predicate logic (see, for example, the handbook chapter [Pitts 2001] and Jacobs’ book [Jacobs 1999] for more explanations). We then define the notion of a BI hyperdoctrine, which is a straightforward extension of the standard notion of hyperdoctrine, and explain how it can be used to model predicate BI logic.

2.1 Hyperdoctrines

A first-order hyperdoctrine is a categorical structure tailored to model first-order predicate logic with equality. The structure has a base category \mathcal{C} for modeling the types and terms, and a \mathcal{C} -indexed category \mathcal{P} for modeling formulas.

Definition 2.1. Let \mathcal{C} be a category with finite products. A *first-order hyperdoctrine* \mathcal{P} over \mathcal{C} is a contravariant functor $\mathcal{P} : \mathcal{C}^{op} \rightarrow \mathbf{Poset}$ from \mathcal{C} into the category of partially ordered sets and monotone functions, with the following properties.

- (1) For each object X , the partially ordered set $\mathcal{P}(X)$ is a Heyting algebra.
- (2) For each morphism $f : X \rightarrow Y$ in \mathcal{C} , the monotone function $\mathcal{P}(f) : \mathcal{P}(Y) \rightarrow \mathcal{P}(X)$ is a Heyting algebra homomorphism.
- (3) For each diagonal morphism $\Delta_X : X \rightarrow X \times X$ in \mathcal{C} , the left adjoint to $\mathcal{P}(\Delta_X)$ at the top element $\top \in \mathcal{P}(X)$ exists. In other words, there is an element $=_X$ of $\mathcal{P}(X \times X)$ satisfying that for all $A \in \mathcal{P}(X \times X)$,

$$\top \leq \mathcal{P}(\Delta_X)(A) \quad \text{iff} \quad =_X \leq A.$$

- (4) For each product projection $\pi : \Gamma \times X \rightarrow \Gamma$ in \mathcal{C} , the monotone function $\mathcal{P}(\pi) : \mathcal{P}(\Gamma) \rightarrow \mathcal{P}(\Gamma \times X)$ has both a left adjoint $(\exists X)_\Gamma$ and a right adjoint $(\forall X)_\Gamma$:

$$\begin{aligned} A \leq \mathcal{P}(\pi)(A') & \quad \text{if and only if} \quad (\exists X)_\Gamma(A) \leq A' \\ \mathcal{P}(\pi)(A') \leq A & \quad \text{if and only if} \quad A' \leq (\forall X)_\Gamma(A). \end{aligned}$$

Moreover, these adjoints are natural in Γ , i.e., given $s : \Gamma \rightarrow \Gamma'$ in \mathcal{C} ,

$$\begin{array}{ccc} \mathcal{P}(\Gamma' \times X) & \xrightarrow{\mathcal{P}(s \times \text{id}_X)} & \mathcal{P}(\Gamma \times X) & & \mathcal{P}(\Gamma' \times X) & \xrightarrow{\mathcal{P}(s \times \text{id}_X)} & \mathcal{P}(\Gamma \times X) \\ (\exists X)_{\Gamma'} \downarrow & & \downarrow (\exists X)_{\Gamma} & & (\forall X)_{\Gamma'} \downarrow & & \downarrow (\forall X)_{\Gamma} \\ \mathcal{P}(\Gamma') & \xrightarrow{\mathcal{P}(s)} & \mathcal{P}(\Gamma) & & \mathcal{P}(\Gamma') & \xrightarrow{\mathcal{P}(s)} & \mathcal{P}(\Gamma). \end{array}$$

The elements of $\mathcal{P}(X)$, where X ranges over objects of \mathcal{C} , are referred to as \mathcal{P} -predicates.

Interpretation of first-order logic in a first-order hyperdoctrine.. Given a (first-order) signature with types X , function symbols $f : X_1, \dots, X_n \rightarrow X$, and relation symbols $R \subset X_1, \dots, X_n$, a *structure* for the signature in a first-order hyperdoctrine \mathcal{P} over \mathcal{C} assigns an object $\llbracket X \rrbracket$ in \mathcal{C} to each type, a morphism $\llbracket f \rrbracket : \llbracket X_1 \rrbracket \times \dots \times \llbracket X_n \rrbracket \rightarrow \llbracket X \rrbracket$ to each function symbol, and a \mathcal{P} -predicate $\llbracket R \rrbracket \in \mathcal{P}(\llbracket X_1 \rrbracket \times \dots \times \llbracket X_n \rrbracket)$ to each relation symbol. Any term t over the signature, with free variables in $\Gamma = \{x_1 : X_1, \dots, x_n : X_n\}$ and of type X say, is interpreted as a morphism $\llbracket t \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket X \rrbracket$, where $\llbracket \Gamma \rrbracket = \llbracket X_1 \rrbracket \times \dots \times \llbracket X_n \rrbracket$, by induction on the structure of t (in the standard manner in which terms are interpreted in categories).

Each formula φ with free variables in Γ is interpreted as a \mathcal{P} -predicate $\llbracket \varphi \rrbracket \in \mathcal{P}(\llbracket \Gamma \rrbracket)$ by induction on the structure of φ using the properties given in Definition 2.1. For atomic formulas $R(t_1, \dots, t_n)$, the interpretation is given by

$$\mathcal{P}(\langle \llbracket t_1 \rrbracket, \dots, \llbracket t_n \rrbracket \rangle)(\llbracket R \rrbracket).$$

In particular, the atomic formula $t =_X t'$ is interpreted by the \mathcal{P} -predicate

$$\mathcal{P}(\langle \llbracket t \rrbracket, \llbracket t' \rrbracket \rangle)(=_{\llbracket X \rrbracket}).$$

The interpretation of other formulas is defined by structural induction. Assume φ, φ' are formulas with free variables in Γ and that ψ is a formula with free variables in $\Gamma \cup \{x : X\}$. Then,

$$\begin{aligned} \llbracket \top \rrbracket &= \top_H & \llbracket \varphi \wedge \varphi' \rrbracket &= \llbracket \varphi \rrbracket \wedge_H \llbracket \varphi' \rrbracket \\ \llbracket \perp \rrbracket &= \perp_H & \llbracket \varphi \vee \varphi' \rrbracket &= \llbracket \varphi \rrbracket \vee_H \llbracket \varphi' \rrbracket \\ & & \llbracket \varphi \rightarrow \varphi' \rrbracket &= \llbracket \varphi \rrbracket \rightarrow_H \llbracket \varphi' \rrbracket \\ \llbracket \forall x : X. \psi \rrbracket &= (\forall \llbracket X \rrbracket)_{\llbracket \Gamma \rrbracket}(\llbracket \psi \rrbracket) \in \mathcal{P}(\llbracket \Gamma \rrbracket) \\ \llbracket \exists x : X. \psi \rrbracket &= (\exists \llbracket X \rrbracket)_{\llbracket \Gamma \rrbracket}(\llbracket \psi \rrbracket) \in \mathcal{P}(\llbracket \Gamma \rrbracket), \end{aligned}$$

where \wedge_H, \vee_H , etc., is the Heyting algebra structure on $\mathcal{P}(\llbracket \Gamma \rrbracket)$.

A formula φ with free variables in Γ is said to be *satisfied* if $\llbracket \varphi \rrbracket$ is the top element of $\mathcal{P}(\llbracket \Gamma \rrbracket)$. This notion of satisfaction is *sound* for intuitionistic predicate logic, in the sense that all provable formulas are satisfied. Moreover, it is *complete* in the sense that a formula is provable if it is satisfied in all structures in first-order hyperdoctrines. A first-order hyperdoctrine \mathcal{P} is sound for *classical* predicate logic in case all the fibres $\mathcal{P}(X)$ are Boolean algebras.

Definition HYPERDOCTRINES. A (general) *hyperdoctrine* is a first-order hyperdoctrine with the following additional properties: \mathcal{C} is cartesian closed, and there is a Heyting algebra H and a natural bijection $\Theta_X : \text{Obj}(\mathcal{P}(X)) \simeq \mathcal{C}(X, H)$.

A hyperdoctrine is sound for higher-order intuitionistic predicate logic: the Heyting algebra H is used to interpret the type **Prop** of propositions and higher types (e.g., \mathbf{Prop}^X , the type for predicates over X), are interpreted by exponentials in \mathcal{C} . The natural bijection Θ_X is used to interpret substitution of formulas in formulas: Suppose φ is a formula with a free variable q of type **Prop** and with remaining free variables in Γ , and that ψ is a formula with free variables in Γ . Then $\llbracket \psi \rrbracket \in \mathcal{P}(\llbracket \Gamma \rrbracket)$, $\llbracket \varphi \rrbracket \in \mathcal{P}(\llbracket \Gamma \rrbracket \times H)$, and $\varphi[\psi/q]$ (φ with ψ substituted in for q) is interpreted by $\mathcal{P}(\langle \text{id}, \Theta(\llbracket \psi \rrbracket) \rangle)(\llbracket \varphi \rrbracket)$. For more details see, e.g., the handbook chapter [Pitts 2001].

Again it is the case that a hyperdoctrine \mathcal{P} is sound for *classical* higher-order predicate logic in case all the fibres $\mathcal{P}(X)$ are Boolean algebras.

Example CANONICAL HYPERDOCTRINE OVER A TOPOS. Let \mathcal{E} be a topos. It is well-known that \mathcal{E} models higher-order predicate logic, by interpreting types as objects in \mathcal{E} , terms as morphisms in \mathcal{E} and predicates as subobjects in \mathcal{E} . The topos \mathcal{E} induces a canonical \mathcal{E} -indexed hyperdoctrine $\text{Sub}_{\mathcal{E}} : \mathcal{E}^{op} \rightarrow \mathbf{Poset}$, which maps an object X in \mathcal{E} to the poset of subobjects of X in \mathcal{E} and a morphism $f : X \rightarrow Y$ to the pullback functor $f^* : \text{Sub}(Y) \rightarrow \text{Sub}(X)$. Then the standard interpretation of predicate logic in \mathcal{E} coincides with the interpretation of predicate logic in the hyperdoctrine $\text{Sub}_{\mathcal{E}}$. Compared to the standard interpretation in toposes, however, hyperdoctrines do not require that predicates are always modeled by subobjects but can come from some other universe. This means that hyperdoctrines describe a wider class of models than toposes do.

2.2 BI Hyperdoctrines

Recall that a Heyting algebra is a bi-cartesian closed partial order, i.e., a partial order, which, when considered as a category, is cartesian closed (\top , \wedge , \rightarrow) and has finite coproducts (\perp , \vee). Further recall that a *BI algebra* is a Heyting algebra, which has an additional symmetric monoidal closed structure (\mathbf{I} , $*$, \multimap) [Pym 2002].

We now present a straightforward extension of (first-order) hyperdoctrines, which models first and higher-order predicate BI.

Definition BI HYPERDOCTRINES.

- A first-order hyperdoctrine \mathcal{P} over \mathcal{C} is a *first-order BI hyperdoctrine* in case all the fibres $\mathcal{P}(X)$ are BI algebras and all the reindexing functions $\mathcal{P}(f)$ are BI algebra homomorphisms.
- A *BI hyperdoctrine* is a first-order BI hyperdoctrine with the additional properties that \mathcal{C} is cartesian closed, and there is a BI algebra B and a natural bijection $\Theta_X : \text{Obj}(\mathcal{P}(X)) \simeq \mathcal{C}(X, B)$.

First-order predicate BI is first-order predicate logic with equality, extended with formulas \mathbf{I} , $\varphi * \psi$, $\varphi \multimap \psi$ satisfying the following rules (in any context Γ including the free variables of the formulas):

$$\begin{array}{ccc}
 (\varphi * \psi) * \theta \vdash_{\Gamma} \varphi * (\psi * \theta) & \varphi * (\psi * \theta) \vdash_{\Gamma} (\varphi * \psi) * \theta & \vdash_{\Gamma} \varphi \leftrightarrow \varphi * \mathbf{I} \\
 \varphi * \psi \vdash_{\Gamma} \psi * \varphi & \frac{\varphi \vdash_{\Gamma} \psi \quad \theta \vdash_{\Gamma} \omega}{\varphi * \theta \vdash_{\Gamma} \psi * \omega} & \frac{\varphi * \psi \vdash_{\Gamma} \theta}{\varphi \vdash_{\Gamma} \psi \multimap \theta}
 \end{array}$$

Our notion of predicate BI should not be confused with the one presented in Pym's book [Pym 2002]; the latter seeks to include a BI structure on contexts but

we do not attempt to do that here, since this is not what is used in separation logic. In particular, weakening at the level of variables is always allowed:

$$\frac{\varphi \vdash_{\Gamma} \psi}{\varphi \vdash_{\Gamma \cup \{x:X\}} \psi}.$$

We interpret first-order predicate BI in a first-order BI hyperdoctrine simply by extending the interpretation of first-order logic in first-order hyperdoctrine defined above by:

$$\begin{aligned} \llbracket \mathbf{1} \rrbracket &= I_B \\ \llbracket \varphi * \psi \rrbracket &= \llbracket \varphi \rrbracket *_B \llbracket \psi \rrbracket \\ \llbracket \varphi \multimap \psi \rrbracket &= \llbracket \varphi \rrbracket \multimap_B \llbracket \psi \rrbracket, \end{aligned}$$

where I_B , $*_B$ and \multimap_B is the monoidal closed structure in the BI algebra $\mathcal{P}(\llbracket \Gamma \rrbracket)$. We then have:

THEOREM 2.1. *The interpretation of first-order predicate BI given above is sound and complete.*

Likewise, BI hyperdoctrines form sound and complete models for higher-order predicate BI. Of course, a (first-order) BI hyperdoctrine is sound for classical BI in case all the fibres $\mathcal{P}(X)$ are Boolean BI algebras and all the reindexing functions $\mathcal{P}(f)$ are Boolean BI algebra homomorphisms. Here is a canonical example of a BI hyperdoctrine.

Example BI HYPERDOCTRINE OVER A COMPLETE BI ALGEBRA. Let B be a complete BI algebra, i.e., it has all joins and meets. It determines a BI hyperdoctrine over the category **Set** as follows. For each set X , let $\mathcal{P}(X) = B^X$, the set of functions from X to B , ordered pointwise. Given $f : X \rightarrow Y$, $\mathcal{P}(f) : B^Y \rightarrow B^X$ is the BI algebra homomorphism given by composition with f . For example if $s, t \in \mathcal{P}(Y)$, i.e., $s, t : Y \rightarrow B$, then $\mathcal{P}(f)(s) = s \circ f : X \rightarrow B$ and $s * t$ is defined pointwise as $(s * t)(y) = s(y) * t(y)$. Equality predicates $=_X$ in $B^{X \times X}$ are defined by

$$=_X(x, x') \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } x = x' \\ \perp & \text{if } x \neq x' \end{cases},$$

where \top and \perp are the greatest and least elements of B , respectively. The quantifiers use set-indexed joins (\bigvee) and meets (\bigwedge). Specifically, given $A \in B^{\Gamma \times X}$ one has

$$(\exists X)_{\Gamma}(A) \stackrel{\text{def}}{=} \lambda i \in \Gamma. \bigvee_{x \in X} A(i, x) \qquad (\forall X)_{\Gamma}(A) \stackrel{\text{def}}{=} \lambda i \in \Gamma. \bigwedge_{x \in X} A(i, x)$$

in B^{Γ} . The conditions in Definition 2.2 are trivially satisfied (Θ is the identity).

There are plenty of examples of complete BI algebras: for any Grothendieck topos \mathcal{E} with an additional symmetric monoidal closed structure, $\text{Sub}_{\mathcal{E}}(1)$ is a complete BI algebra, and for any monoidal category \mathcal{C} such that the monoid is cover preserving with respect to the Grothendieck topology J , $\text{Sub}_{\text{Sh}(\mathcal{C}, J)}(1)$ is a complete BI algebra [Biering 2004; Pym et al. 2004].

The following theorem shows that to get interesting models of higher-order predicate BI, it does not suffice to consider BI hyperdoctrines arising as the canonical

hyperdoctrine over a topos (as in Example 2.1). Indeed this is the reason for introducing the more general BI hyperdoctrines.

THEOREM 2.2. *Let \mathcal{E} be a topos and suppose $\text{Sub}_{\mathcal{E}} : \mathcal{E}^{op} \rightarrow \mathbf{Poset}$ is a BI hyperdoctrine. Then the BI structure on each lattice $\text{Sub}_{\mathcal{E}}(X)$ is trivial, i.e., for all $\varphi, \psi \in \text{Sub}_{\mathcal{E}}(X)$, $\varphi * \psi \leftrightarrow \varphi \wedge \psi$.*

PROOF. Let \mathcal{E} be a topos and suppose $\text{Sub}_{\mathcal{E}} : \mathcal{E}^{op} \rightarrow \mathbf{Poset}$ is a BI hyperdoctrine. Let X be an object of \mathcal{E} and let $\varphi, \psi, \psi' \in \text{Sub}_{\mathcal{E}}(X)$. Furthermore let Y be the domain of the mono φ , and notice that the lattice $\text{Sub}_{\mathcal{E}}(Y)$ can be characterized by

$$\text{Sub}_{\mathcal{E}}(Y) = \{\psi \wedge \varphi \mid \psi \in \text{Sub}_{\mathcal{E}}(X)\}. \quad (1)$$

Furthermore, notice that the order on $\text{Sub}_{\mathcal{E}}(Y)$ is inherited from $\text{Sub}_{\mathcal{E}}(X)$, i.e.,

$$\text{For all } \chi, \chi' \in \text{Sub}_{\mathcal{E}}(Y), \chi \vdash_Y \chi' \text{ iff } \chi \vdash_X \chi'. \quad (2)$$

Since \wedge is modeled by pullback which by assumption preserves $*$, the following equations hold in $\text{Sub}_{\mathcal{E}}(Y)$ (and therefore also in $\text{Sub}_{\mathcal{E}}(X)$):

$$(\varphi \wedge \psi) *_Y (\varphi \wedge \psi') \leftrightarrow \varphi \wedge (\psi *_X \psi') \quad (3)$$

and

$$(\varphi \wedge \psi) \text{--} *_Y (\varphi \wedge \psi') \leftrightarrow \varphi \wedge (\psi \text{--} *_X \psi'). \quad (4)$$

By assumption, $\text{Sub}_{\mathcal{E}}(Y)$ forms a BI algebra with connectives $*_Y, \text{--} *_Y$ and I_Y , so using the characterization of subobjects of Y given in (1), yields the following rule for each $\chi \in \text{Sub}_{\mathcal{E}}(X)$:

$$\frac{(\varphi \wedge \psi) *_Y (\varphi \wedge \psi') \vdash_Y \chi \wedge \varphi}{\varphi \wedge \psi \vdash_Y (\varphi \wedge \psi') \text{--} *_Y (\chi \wedge \varphi)}$$

Using (2), (3), and (4) we deduce

$$\frac{\varphi \wedge (\psi *_X \psi') \vdash_X \chi \wedge \varphi}{\varphi \wedge \psi \vdash_X \varphi \wedge (\psi' \text{--} *_X \chi)}$$

for all $\varphi, \psi, \psi', \chi \in \text{Sub}_{\mathcal{E}}(X)$, which implies

$$\frac{\frac{\varphi \wedge (\psi *_X \psi') \vdash_X \chi \wedge \varphi}{\varphi \wedge \psi \vdash_X \psi' \text{--} *_X \chi}}{(\varphi \wedge \psi) *_X \psi' \vdash_X \chi} \quad (5)$$

Inserting $\varphi \wedge (\psi *_X \psi')$ for χ into (5) yields

$$\frac{\varphi \wedge (\psi *_X \psi') \vdash_X \varphi \wedge (\psi *_X \psi')}{(\varphi \wedge \psi) *_X \psi' \vdash_X \varphi \wedge (\psi *_X \psi')}. \quad (6)$$

Since the entailment above the line in (6) always holds,

$$(\varphi \wedge \psi) *_X \psi' \vdash_X \varphi \wedge (\psi *_X \psi').$$

This gives us projections for $*_X$ by letting ψ be \top :

$$(\varphi *_X \psi') \dashv\vdash_X (\varphi \wedge \top) *_X \psi' \vdash_X \varphi \wedge (\top *_X \psi') \vdash_X \varphi.$$

Now, let χ be the subobject $(\varphi \wedge \psi) *_X \psi'$, then $\chi \leftrightarrow \chi \wedge \varphi$ due to the projections for $*_X$. Using (5) downwards-up, gives

$$\frac{(\varphi \wedge \psi) *_X \psi' \vdash_X (\varphi \wedge \psi) *_X \psi'}{\varphi \wedge (\psi *_X \psi') \vdash_X (\varphi \wedge \psi) *_X \psi'} \quad (7)$$

By (6) and (7) we conclude that for all $\varphi, \psi, \psi' \in \text{Sub}_{\mathcal{E}}(X)$,

$$\varphi \wedge (\psi *_X \psi') \leftrightarrow (\varphi \wedge \psi) *_X \psi'. \quad (8)$$

We already noted the projections for $*_X$, so $\top *_X I_X \vdash_X I_X$, which entails $\top \leftrightarrow I_X$. Let ψ be \top in (8), then $\varphi \wedge (\top *_X \psi') \leftrightarrow (\varphi \wedge \top) *_X \psi'$ and so $\varphi \wedge \psi' \leftrightarrow \varphi *_X \psi'$, as claimed. \square

In fact, it is possible to make a slight strengthening of Theorem 2.2. We say that a logic has *full subset types* [Jacobs 1999] if the following conditions are satisfied.

- For each formula $\varphi(x_1, \dots, x_n)$, there is a type $\{x_1:\tau_1, \dots, x_n:\tau_n \mid \varphi(x_1, \dots, x_n)\}$.
- For a term N of type $\{x_1:\tau_1, \dots, x_n:\tau_n \mid \varphi(x_1, \dots, x_n)\}$, in a context Γ , there is a term $\mathfrak{o}(N)$ of type $\tau_1 \times \dots \times \tau_n$ in Γ .
- The rule

$$\frac{\Gamma, y:\{x:X \mid \varphi\} \mid \theta[\mathfrak{o}(y)/x] \vdash \psi[\mathfrak{o}(y)/x]}{\Gamma, x:X \mid \theta, \varphi \vdash \psi} \quad (9)$$

is valid.

One can then show

PROPOSITION 2.1. *If our notion of predicate BI has full subset types, then for all formulas φ, ψ in a context Γ ,*

$$\varphi \wedge \psi \dashv\vdash_{\Gamma} \varphi *_X \psi.$$

The proof may be found in Appendix A. The following is an easy consequence.

COROLLARY 2.1. *Any BI hyperdoctrine which satisfies the rules for full subset types is trivial.*

The BI hyperdoctrine S , which we define below and which corresponds to the standard pointer model of separation logic, satisfies all of the above except the downward direction of (9). When this is the case, we say that the logic has subset types, but not *full* subset types [Jacobs 1999].

3. SEPARATION LOGIC MODELED BY BI-HYPERDOCTRINES

We briefly recall the standard pointer model of separation logic (for a more thorough presentation see, for instance, [Reynolds 2002]) and then show how it can be construed as a BI hyperdoctrine over Set .

The core assertion language of separation logic (which we will henceforth also call separation logic) is often defined as follows. There is a single type Val of values. Terms t are defined by a grammar

$$t ::= x \mid n \mid t + t \mid t - t \mid \dots,$$

where $n : \text{Val}$ are constants for all integers n . Formulas, also called assertions, are defined by

$$\varphi ::= \top \mid \perp \mid t = t \mid t \mapsto t \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi * \varphi \mid \varphi \multimap \varphi \mid \emptyset \mid \forall x. \varphi \mid \exists x. \varphi$$

The symbol \emptyset is used in separation logic for the unit of BI.

Note that the above is just another way of defining a signature (specification of types, function symbols and predicate symbols) for first-order predicate BI with a single type Val , function symbols $+, -, \dots : \text{Val}, \text{Val} \rightarrow \text{Val}$, constants $n : \text{Val}$, and relation symbol $\mapsto \subseteq \text{Val}, \text{Val}$.

3.1 The pointer model.

The standard pointer model of separation logic is usually presented as follows. It consists of a set $\llbracket \text{Val} \rrbracket$ interpreting the type Val and a set $\llbracket \text{Loc} \rrbracket$ of locations such that $\llbracket \text{Loc} \rrbracket \subseteq \llbracket \text{Val} \rrbracket$ and binary functions on $\llbracket \text{Val} \rrbracket$ interpreting the function symbols $+, -$. The set $H = \llbracket \text{Loc} \rrbracket \multimap_{fin} \llbracket \text{Val} \rrbracket$ of finite partial functions from $\llbracket \text{Loc} \rrbracket$ to $\llbracket \text{Val} \rrbracket$, ordered discretely, is referred to as the set of *heaps*. The set of heaps has a partial binary operation $*$ defined by

$$h_1 * h_2 = \begin{cases} h_1 \cup h_2 & \text{if } h_1 \# h_2 \\ \text{undefined} & \text{otherwise,} \end{cases}$$

where $\#$ is the binary relation on heaps defined by $h_1 \# h_2$ iff $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$. The interpretation of the relation $\mapsto \subseteq \llbracket \text{Val} \rrbracket \times \llbracket \text{Val} \rrbracket$ is the subset of singleton heaps, that is, for $h \in H$, $h \in \mapsto$ iff $h = \{(v_1, v_2)\}$ for some values v_1, v_2 . To define the standard interpretation of terms and formulas, one assumes a partial function $s : \text{Var} \multimap_{fin} \llbracket \text{Val} \rrbracket$, called a stack (also called a store in the literature). The interpretation of terms depends on the stack and is defined by

$$\begin{aligned} \llbracket x \rrbracket s &= s(x) \\ \llbracket n \rrbracket s &= \llbracket n \rrbracket \\ \llbracket t_1 \pm t_2 \rrbracket s &= \llbracket t_1 \rrbracket s \pm \llbracket t_2 \rrbracket s \end{aligned}$$

The interpretation of formulas is standardly given by a forcing relation $s, h \vdash \varphi$, where $\text{FV}(\varphi) \subseteq \text{dom}(s)$, as follows

$$\begin{aligned} s, h \vdash t_1 = t_2 &\text{ iff } \llbracket t_1 \rrbracket s = \llbracket t_2 \rrbracket s \\ s, h \vdash t_1 \mapsto t_2 &\text{ iff } \text{dom}(h) = \{\llbracket t_1 \rrbracket s\} \text{ and } h(\llbracket t_1 \rrbracket s) = \llbracket t_2 \rrbracket s \\ s, h \vdash \emptyset &\text{ iff } h = \emptyset \\ s, h \vdash \top &\text{ always} \\ s, h \vdash \perp &\text{ never} \\ s, h \vdash \varphi * \psi &\text{ iff there exists } h_1, h_2 \in H. h_1 * h_2 = h \text{ and} \\ &\quad s, h_1 \vdash \varphi \text{ and } s, h_2 \vdash \psi \\ s, h \vdash \varphi \multimap \psi &\text{ iff for all } h', h' \# h \text{ and } s, h' \vdash \varphi \text{ implies } s, h * h' \vdash \psi \\ s, h \vdash \varphi \vee \psi &\text{ iff } s, h \vdash \varphi \text{ or } s, h \vdash \psi \\ s, h \vdash \varphi \wedge \psi &\text{ iff } s, h \vdash \varphi \text{ and } s, h \vdash \psi \\ s, h \vdash \varphi \rightarrow \psi &\text{ iff } s, h \vdash \varphi \text{ implies } s, h \vdash \psi \\ s, h \vdash \forall x. \varphi &\text{ iff for all } v \in \llbracket \text{Val} \rrbracket. s[x \mapsto v], h \vdash \varphi \\ s, h \vdash \exists x. \varphi &\text{ iff there exists } v \in \llbracket \text{Val} \rrbracket. s[x \mapsto v], h \vdash \varphi \end{aligned}$$

We now show how this pointer model is an instance of a BI-hyperdoctrine of a complete Boolean BI algebra (cf. Example 2.2).

3.2 The pointer model as a BI hyperdoctrine.

Let $(H_\perp, *)$ be the discretely ordered set of heaps with a bottom element added to represent undefined, and let $* : H_\perp \times H_\perp \rightarrow H_\perp$ be the total extension of $* : H \times H \rightarrow H$ satisfying $\perp * h = h * \perp = \perp$, for all $h \in H_\perp$. This defines a partially ordered commutative monoid with the empty heap $\{\}$ as the unit for $*$. The powerset of H , $\mathcal{P}(H)$ (without \perp) is a complete Boolean BI algebra, ordered by inclusion and with monoidal closed structure given by (for $U, V \in \mathcal{P}(H)$):

$$\begin{aligned} \text{---}I & \text{ is } \{\emptyset\} \\ \text{---}U * V & := \{h * h' \mid h \in U \wedge h' \in V\} \setminus \{\perp\} \\ \text{---}U \multimap V & := \bigcup \{W \subseteq H \mid (W * U) \subseteq V\}. \end{aligned}$$

It can easily be verified directly that this defines a complete Boolean BI algebra; it also follows from more abstract arguments in [Pym et al. 2004; Biering 2004].

Let S be the BI hyperdoctrine induced by the complete Boolean BI algebra $\mathcal{P}(H)$ as in Example 2.2. To show that the interpretation of separation logic in this BI hyperdoctrine exactly corresponds to the standard pointer model presented above we spell out the interpretation of separation logic in S .

A term t in a context $\Gamma = \{x_1 : \mathbf{Val}, \dots, x_n : \mathbf{Val}\}$ is interpreted as a morphism between sets:

$$\begin{aligned} \text{---}[[x_i : \mathbf{Val}]] & = \pi_i, \text{ where } \pi_i : \mathbf{Val}^n \rightarrow \mathbf{Val} \text{ is the } i\text{'th projection,} \\ \text{---}[[n]] & \text{ is the map } [[n]] : [\Gamma] \rightarrow 1 \rightarrow [\mathbf{Val}] \text{ which sends the unique element of the} \\ & \text{one-point set } 1 \text{ to } [[n]], \\ \text{---}[[t_1 \pm t_2]] & = [[t_1] \pm [t_2]] : [\Gamma] \rightarrow [\mathbf{Val}] \times [\mathbf{Val}] \rightarrow [\mathbf{Val}], \text{ where } [[t_i]] : [\Gamma] \rightarrow [\mathbf{Val}], \text{ for} \\ & i = 1, 2. \end{aligned}$$

The interpretation of a formula φ in a context $\Gamma = \{x_1 : \mathbf{Val}, \dots, x_n : \mathbf{Val}\}$ is given inductively as follows. Let $I = [[\mathbf{Val}]] \times \dots \times [[\mathbf{Val}]] = [[\mathbf{Val}]]^n$ and write \bar{v} for elements of I . Then φ is interpreted as an element of $\mathcal{P}(I)$ as follows:

$$\begin{aligned} [[t_1 \mapsto t_2]](\bar{v}) & = \{h \mid \text{dom}(h) = \{[[t_1]](\bar{v})\} \text{ and } h([[t_1]](\bar{v})) = [[t_2]](\bar{v})\} \\ [[t_1 = t_2]](\bar{v}) & = H \text{ if } [[t_1]](\bar{v}) = [[t_2]](\bar{v}), \emptyset \text{ otherwise} \\ [[\Gamma]](*) & = H \\ [[\perp]](*) & = \emptyset \\ [[\emptyset]](*) & = \{h \mid \text{dom}(h) = \emptyset\} \\ [[\varphi \wedge \psi]](\bar{v}) & = [[\varphi]](\bar{v}) \cap [[\psi]](\bar{v}) \\ [[\varphi \vee \psi]](\bar{v}) & = [[\varphi]](\bar{v}) \cup [[\psi]](\bar{v}) \\ [[\varphi \rightarrow \psi]](\bar{v}) & = \{h \mid h \in [[\varphi]](\bar{v}) \text{ implies } h \in [[\psi]](\bar{v})\} \\ [[\varphi * \psi]](\bar{v}) & = [[\varphi]](\bar{v}) * [[\psi]](\bar{v}) \\ & = \{h_1 * h_2 \mid h_1 \in [[\varphi]](\bar{v}) \text{ and } h_2 \in [[\psi]](\bar{v})\} \setminus \{\perp\} \\ [[\varphi \multimap \psi]](\bar{v}) & = [[\varphi]](\bar{v}) \multimap [[\psi]](\bar{v}) \\ & = \{h \mid [[\varphi]](\bar{v}) * \{h\} \subseteq [[\psi]](\bar{v})\} \\ [[\forall x : \mathbf{Val}. \varphi]](\bar{v}) & = \bigcap_{v_x \in [[\mathbf{Val}]]} ([[\varphi]](v_x, \bar{v})) \\ [[\exists x : \mathbf{Val}. \varphi]](\bar{v}) & = \bigcup_{v_x \in [[\mathbf{Val}]]} ([[\varphi]](v_x, \bar{v})) \end{aligned}$$

Now it is easy to verify by structural induction on formulas φ that the interpretation given in the BI hyperdoctrine S corresponds exactly to the forcing semantics given earlier:

THEOREM 3.1. $h \in \llbracket \varphi \rrbracket(v_1, \dots, v_n)$ iff $[x_1 \mapsto v_1, \dots, x_n \mapsto v_n], h \vdash \varphi$.

As a consequence, we of course obtain the well-known result that separation logic is sound for classical first-order BI. But, more interestingly, the correspondence also shows that we may easily extend separation logic to higher-order since the BI hyperdoctrine S soundly models higher-order BI. We expand on this in the next section, which also discusses other consequences of the above correspondence. First, however, we explain that one can also obtain such a correspondence for other versions of separation logic.

3.3 An intuitionistic model.

Consider again the set of heaps $(H_\perp, *)$ with an added bottom \perp , as above. We now define the order by

$$h_1 \sqsupseteq h_2 \quad \text{iff} \quad \text{dom}(h_1) \subseteq \text{dom}(h_2) \text{ and for all } x \in \text{dom}(h_1). h_1(x) = h_2(x).$$

Let I be the set of sieves on H , i.e., downwards closed subsets of H , ordered by inclusion. This is a complete BI algebra, as can be verified directly or by an abstract argument [Biering 2004; Pym et al. 2004].

Now let T be the BI hyperdoctrine induced by the complete BI algebra I as in Example 2.2. The interpretation of predicate BI in this BI hyperdoctrine corresponds exactly to the intuitionistic pointer model of separation logic, which is presented using a forcing style semantics in [Ishtiaq and O’Hearn 2001].

3.4 The permissions model.

It is also possible to fit the permissions model of separation logic from [Bornat et al. 2005] into the framework presented here. The main point is that the set of heaps, which in that model map locations to values and permissions, has a binary operation $*$, which makes $(H_\perp, *)$ a partially ordered commutative monoid.

Remark 3.1. The correspondences between separation logic and BI hyperdoctrines given above illustrate that what matters for the interpretation of separation logic is the choice of BI algebra. Indeed, the main relevance of the topos-theoretic constructions in [Pym et al. 2004] for models of separation logic is that they can be used to construct suitable BI-algebras (as subobject lattices in categories of sheaves).

4. SOME CONSEQUENCES FOR SEPARATION LOGIC

We have shown above that it is completely natural and straightforward to interpret first-order predicate BI in first-order BI-hyperdoctrines and that the standard pointer model of separation logic corresponds to a particular case of BI-hyperdoctrine. Based on this correspondence, in this section we draw some further consequences for separation logic.

4.1 Formalizing Separation Logic

The strength of separation logic has been demonstrated in numerous papers before. In the early days of separation logic, it was shown that it could handle simple programs for copying trees, deleting lists, etc. The first proof of a more realistic program appeared in Yang’s thesis [Yang 2001], in which he showed correctness

of the Schorr-Waite graph marking algorithm. Later, a proof of correctness of Cheney’s garbage collection algorithm was published in [Birkedal et al. 2004], and other examples of correctness proofs of non-trivial algorithms may be found in [Bornat et al. 2004]. In all of these papers, different simple extensions of core separation logic were used. For example, Yang used lists and binary trees as parts of his term language, and Birkedal et. al. introduced expression forms for finite sets and relations. It would seem that it is a weakness of separation logic that one has to come up with suitable extensions of it every time one has to prove a new program correct. In particular, it would make machine-verifiable formalizations of such proofs more burdensome and dubious if one would have to alter the underlying logic for every new proof.

The right way to look at these “extensions” is that they are really trivial definitional extensions of one and the same logic, namely the internal logic of the classical BI hyperdoctrine S presented in Section 3. The internal language of a BI hyperdoctrine \mathcal{P} over \mathcal{C} is formed as follows: to each object of \mathcal{C} one associates a type, to each morphism of \mathcal{C} one associates a function symbol, and to each predicate in $\mathcal{P}(X)$ one associates a relation symbol. The terms and formulas over this signature (considered as a higher-order signature [Jacobs 1999]) form the internal language of the BI hyperdoctrine. There is an obvious structure for this language in \mathcal{P} .

Let $2 = \{\perp, \top\}$ be a two-element set (the subobject classifier of \mathbf{Set}). There is a canonical map $\iota : 2 \rightarrow \mathcal{P}(H)$ which maps \perp to $\{\}$ (the bottom element of the BI algebra $\mathcal{P}(H)$) and \top to H (the top element of $\mathcal{P}(H)$).

Definition 4.1. Let φ be an S -predicate over a set X , i.e., a function $\varphi : X \rightarrow \mathcal{P}(H)$. Call φ *pure* if φ factors through ι .

Thus $\varphi : X \rightarrow \mathcal{P}(H)$ is pure if there exists a map $\chi_\varphi : X \rightarrow 2$ such that

$$\begin{array}{ccc} X & \xrightarrow{\varphi} & \mathcal{P}(H) \\ & \searrow \chi_\varphi & \nearrow \iota \\ & 2 & \end{array}$$

commutes. This corresponds to the notion of pure predicate traditionally used in separation logic [Reynolds 2002].

The sub-logic of pure predicates is simply the standard classical higher-order logic of \mathbf{Set} , and thus it is sound for classical higher-order logic. Hence one can use classical higher-order logic for defining lists, trees, finite sets and relations in the standard manner using pure predicates and prove the standard properties of these structures, as needed for the proofs presented in the papers referred to above. In particular, notice that recursive definitions of predicates, which in the papers [Yang 2001; Birkedal et al. 2004; Bornat et al. 2004] are defined at the meta level, can be defined inside the higher-order logic itself. For machine verification one would thus only need to formalize one and the same logic, namely a sufficient fragment of the internal logic of the BI hyperdoctrine (with obvious syntactic rules for when a formula is pure). The internal logic itself is “too big” (it can have class-many types and function symbols, e.g.); hence the need for a fragment thereof, say classical higher-order logic with natural numbers.

4.2 Logical Characterizations of Classes of Assertions

Different classes of assertions, precise, monotone, and pure, are introduced by Reynolds [Reynolds 2002], who notices that special axioms for these classes of assertions are valid. Such special axioms are exploited in the proof of Cheney’s garbage collector [Birkedal et al. 2004], where pure assertions are moved in and out of the scope of iterated separating conjunctions, and in the paper [O’Hearn et al. 2004], where properties of precise assertions are crucially applied to verify soundness of the hypothetical frame rule. The different classes of assertions are defined semantically and the special axioms are validated using the semantics. We show how the higher-order features of higher-order separation logic allows a logical characterization of the classes of assertions, and logical proofs of the properties earlier taken as axioms. This is, of course, important for machine verification, since it means that the special classes of assertions and their properties can be expressed *in the logic*.

To simplify notation we just present the characterizations for *closed* assertions, the extension to open assertions is straightforward. Recall that closed assertions are interpreted in S as functions from 1 to $\mathcal{P}(H)$, i.e., as subsets of H .

In the proofs below, we use assertions which describe heaps in a canonical way. Since a heap h has finite domain, there is a unique (up to permutation) way to write an assertion $p_h \equiv l_1 \mapsto n_1 * \dots * l_k \mapsto n_k$ such that $\llbracket p_h \rrbracket = \{h\}$.

Precise assertions. The traditional definition of a precise assertion is semantic, inasmuch as an assertion q is precise if, and only if, for all states (s, h) , there is at most one subheap h_0 of h such that $(s, h_0) \vdash q$. The following proposition logically characterizes closed precise assertions (at the semantic level, this characterization of precise predicates has been mentioned before [O’Hearn et al. 2003]).

PROPOSITION 4.1. *The closed assertion q is precise if, and only if, the assertion*

$$\forall p_1, p_2 : \text{Prop. } (p_1 * q) \wedge (p_2 * q) \leftrightarrow (p_1 \wedge p_2) * q \quad (10)$$

is valid in the BI hyperdoctrine S .

PROOF. The “only-if” direction is trivial, so we focus on the other implication. Thus suppose (10) holds for q , and let h be a heap with two different subheaps h_1, h_2 for which $h_i \in \llbracket q \rrbracket$. Let p_1, p_2 be canonical assertions describing the heaps $h \setminus h_1$ and $h \setminus h_2$, respectively. Then $h \in \llbracket (p_1 * q) \wedge (p_2 * q) \rrbracket$, so $h \in \llbracket (p_1 \wedge p_2) * q \rrbracket$, whence there is a subheap $h' \subseteq h$ with $h' \in \llbracket p_1 \wedge p_2 \rrbracket$. This is a contradiction. \square

One can verify properties for precise assertions *in the logic* without using semantic arguments. For example, one can show that $q_1 * q_2$ is precise if q_1 and q_2 are by the following logical argument: Suppose (10) holds for q_1, q_2 . Then,

$$\begin{aligned} (p_1 * (q_1 * q_2)) \wedge (p_2 * (q_1 * q_2)) &\Rightarrow ((p_1 * q_1) * q_2) \wedge ((p_2 * q_1) * q_2) \\ \Rightarrow ((p_1 * q_1) \wedge (p_2 * q_1)) * q_2 &\Rightarrow ((p_1 \wedge p_2) * q_1) * q_2 \\ \Rightarrow (p_1 \wedge p_2) * (q_1 * q_2), \end{aligned}$$

as desired.

Monotone assertions. A closed assertion q is defined to be *monotone* if, and only if, whenever $h \in \llbracket q \rrbracket$ then also $h' \in \llbracket q \rrbracket$, for all extensions $h' \supseteq h$.

PROPOSITION 4.2. *The closed assertion q is monotone if, and only if, the assertion $\forall p:\text{Prop}. p * q \rightarrow q$ is valid in the BI hyperdoctrine S .*

This is easily verified, and again, one can show the usual rules for monotone assertions in the logic (without semantical arguments) using this characterization.

Pure assertions. Recall from above that an assertion q is pure iff its interpretation factors through 2. Thus a closed assertion is pure iff its interpretation is either \emptyset or H .

PROPOSITION 4.3. *The closed assertion q is pure if, and only if, the assertion*

$$\forall p_1, p_2:\text{Prop}. (q \wedge p_1) * p_2 \leftrightarrow q \wedge (p_1 * p_2) \quad (11)$$

is valid in the BI hyperdoctrine S .

PROOF. Again, the interesting direction here is the “if” implication. Hence, suppose (11) holds for the assertion q , and that $h \in \llbracket q \rrbracket$. For any heap h_0 , we must then show that $h_0 \in \llbracket q \rrbracket$. This is done via the verification of two claims.

Fact 1: For all $h' \subseteq h$, $h' \in \llbracket q \rrbracket$. Proof: Let p_1 be a canonical description of h' , and p_2 a canonical description of $h \setminus h'$. Then $h \in \llbracket q \wedge (p_1 * p_2) \rrbracket$, so $h \in \llbracket (q \wedge p_1) * p_2 \rrbracket$. This means there is a split $h_1 * h_2 = h$ with $h_1 \in \llbracket q \wedge p_1 \rrbracket$ and $h_2 \in \llbracket p_2 \rrbracket$. But then, $h_2 = h \setminus h'$, so $h_1 = h'$, and thus, $h' \in \llbracket q \rrbracket$.

Fact 2: For all $h' \supseteq h$, $h' \in \llbracket q \rrbracket$. Proof: Let p_1 and p_2 be canonical descriptions of h and $h' \setminus h$, respectively. Then, $h' \in \llbracket (q \wedge p_1) * p_2 \rrbracket$, so $h' \in \llbracket q \wedge (p_1 * p_2) \rrbracket$, and in particular, $h' \in \llbracket q \rrbracket$, as desired.

Using Facts 1 and 2, we deduce $h \in \llbracket q \rrbracket \Rightarrow \emptyset \in \llbracket q \rrbracket \Rightarrow h_0 \in \llbracket q \rrbracket$. \square

4.3 Predicates via Fixed Points

Consider the following predicate clist, taken from [Parkinson and Bierman 2005]. It is required to satisfy the following recursive equation:

$$\text{clist} = \lambda(x, s).x = \text{null} \vee (\exists j, k. x \mapsto j, k * P(j, s) * \text{clist}(k, s)),$$

for some specific P . Solutions to such equations are definable in higher-order separation logic. Indeed, we may define both minimal and maximal fixed points for any monotone operator on predicates, using standard encodings of fixed points. To wit, consider for notational simplicity an arbitrary predicate

$$q : \text{Prop} \vdash \varphi(q) : \text{Prop}$$

satisfying that q only occurs positively in φ . Then

$$\mu q. \varphi(q) = \forall q. (\varphi(q) \rightarrow q) \rightarrow q$$

is the least fixed point for φ in the obvious sense that $\varphi(\mu q. \varphi(q)) \rightarrow \mu q. \varphi(q)$ and $\forall p. (\varphi(p) \rightarrow p) \rightarrow (\mu q. \varphi(q) \rightarrow p)$ holds in the logic. Likewise,

$$\nu q. \varphi(q) = \exists q. (q \rightarrow \varphi(q)) \wedge q$$

is the maximal fixed point for φ .

5. HIGHER-ORDER SEPARATION LOGIC

We present a programming language and use the higher-order assertion language of the pointer-model BI-hyperdoctrine S to give a specification logic for the programming language. The programming language is a simple extension of that of standard separation logic with calls to simple procedures, and the program logic includes standard rules for these.

Programming Language. The programming language uses a restricted set of terms of type Int , referred to as *expressions*, and uses *booleans*, which consists of a restricted (heap-independent) set of terms of type Prop . E and B range over these, and they are generated by the grammars:

$$\begin{aligned} E &::= n \mid x \mid E + E \mid E - E \mid E \times E \mid \text{null} \\ B &::= E = E \mid E \leq E \mid B \wedge B \mid \dots \end{aligned}$$

Formally, booleans have type Prop in our system, but we sometimes write $B : \text{Bool}$ if they can be generated from this grammar. Moreover, officially we always consider expressions and formulas in context and thus write $\Delta \vdash E : \text{Int}$, $\Delta \vdash B : \text{Bool}$, and $\Delta \vdash P : \text{Prop}$ for expressions, booleans, and general assertions.

The syntax of the programming language is given by the following grammar. Here, k ranges over a set of function names, and x ranges over a set of program variables.

$$\begin{aligned} c &::= \mathbf{skip} \\ & \mid x := k_i(E_1, \dots, E_{m_i}) \\ & \mid \mathbf{newvar} \ x; c \\ & \mid x := E \\ & \mid x := [E] \\ & \mid [E] := E' \\ & \mid x := \mathbf{cons}(E_1, \dots, E_m) \\ & \mid \mathbf{dispose}(E) \\ & \mid \mathbf{if} \ B \ \mathbf{then} \ c \ \mathbf{else} \ c \ \mathbf{fi} \\ & \mid \mathbf{while} \ B \ \mathbf{do} \ c \ \mathbf{od} \\ & \mid c; c \\ & \mid \mathbf{let} \ k_1(x_1, \dots, x_{m_1}) = c_1 \\ & \quad \vdots \\ & \quad k_n(x_1, \dots, x_{m_n}) = c_n \\ & \mid \mathbf{in} \ c \ \mathbf{end} \\ & \mid \mathbf{return} \ e \end{aligned}$$

There are some restrictions on the programs, and a program is called *well-formed* if it meets these restrictions. This could be expressed formally with auxiliary grammars, but we refrain from that here. The restrictions include:

- There is always a **return** at the end of a function body.
- A function name is declared at most once in a **let**.
- There are the right number of parameters in function calls.
- Function bodies do not modify non-local variables (other than *ret*).

Function Specifications. There is a judgment

$$\Delta \vdash \gamma : \text{FSpec}$$

stating that γ is a well-formed *function specification* in the context Δ . Function specifications are used to record assumptions about functions used in programs. The judgment is given by

$$\frac{\frac{\Delta \vdash P : \text{Prop} \quad \Delta \vdash Q : \text{Prop}}{\Delta \vdash \{P\} k \{Q\} : \text{FSpec}} \quad \Delta \vdash \gamma : \text{FSpec} \quad \Delta \vdash \gamma' : \text{FSpec}}{\Delta \vdash \gamma \wedge \gamma' : \text{FSpec}} \\ \frac{\Delta, x : \tau \vdash \gamma : \text{FSpec}}{\Delta \vdash \lambda x : \tau. \gamma : \text{FSpec}} \text{ where } \lambda \in \{\exists, \forall\}$$

The set of free variables for a function specification is defined as the free variables in the assertions occurring in it.

Specifications. We introduce syntax for commands and specifications. There is a judgment

$$\Delta; \Pi \vdash c : \text{comm}, \tag{12}$$

which asserts that the program c is well-formed in the context Δ and *semantic function environment* Π . A semantic function environment maps function names k to pairs (\vec{x}, c) , where \vec{x} is a vector of integer variables and c is a command from the programming language. Such an environment is well-formed if the function bodies only modify local variables (and *ret*, by the **return** command):

$$\Pi \text{ ok iff } \forall (x, c) \in \text{cod}(\Pi). \text{Mod}(c) = \emptyset.$$

The definition of the judgment (12) is omitted here.

The *specifications* of higher-order separation logic is given by a judgment

$$\Delta; \Pi \vdash \delta : \text{Spec},$$

which asserts that δ is a well-formed specification in the context Δ and semantic function environment Π . This judgment is given by

$$\frac{\frac{\Delta; \Pi \vdash c : \text{comm} \quad \Delta \vdash P : \text{Prop} \quad \Delta \vdash Q : \text{Prop}}{\Delta; \Pi \vdash \{P\} c \{Q\} : \text{Spec}} \quad \Delta; \Pi \vdash \delta : \text{Spec} \quad \Delta; \Pi \vdash \delta' : \text{Spec}}{\Delta; \Pi \vdash \delta \wedge \delta' : \text{Spec}} \\ \frac{\Delta, x : \tau; \Pi \vdash \delta : \text{Spec}}{\Delta; \Pi \vdash \lambda x : \tau. \delta : \text{Spec}} \lambda \in \{\exists, \forall\}$$

The set $\text{FV}(\delta)$ of free variables of a specification δ is the set of free variables in the assertions and the *modified* variables in the commands occurring in δ . The set $\text{Mod}(\delta)$ of modified variables of δ is the set of modified variables in the commands occurring in δ .

Operational Semantics. The operational semantics of the programming language is given by a judgment

$$(\Pi, c, s, h) \Downarrow (s', h').$$

The proviso here is that $s \in \Delta$ for some Δ in which $\Delta; \Pi \vdash c:\text{comm}$ holds, and intuitively the judgment says that the state (s, h) is transformed to the state (s', h') by the program c . The judgment is the same as in the paper [Parkinson and Bierman 2005] and it is given by the clauses in Fig. 1. We occasionally use Δ for the domain of s in the definition of the judgment, for example, in the second rule (for assignment). Furthermore, the notation $h - \{n\}$ is used to denote the heap which is like h , but with n taken out of its domain.

The configuration (Π, c, s, h) is called *safe* if $(\Pi, c, s, h) \not\Downarrow \text{wrong}$. A configuration may either terminate in a state (s', h') , diverge, or go wrong.

Note that, since this semantics is the same as the operational semantics of the language of Parkinson and Bierman [Parkinson and Bierman 2005], the properties needed to prove the frame rule, namely safety monotonicity and the frame property, are valid for all programs of the language. These properties are:

Safety Monotonicity. For all well-formed semantic function environments Π , programs c , stacks s , and heaps h , if (Π, c, s, h) is safe, then for all heaps h' disjoint from h , $(\Pi, c, s, h \cup h')$ is also safe.

The Frame Property. For all well-formed semantic function environments Π , programs c , stacks s , and heaps h , if (Π, c, s, h) is safe and h' is disjoint from h , then $(\Pi, c, s, h \cup h') \Downarrow (s', h'')$, implies that there is h_0 such that $h'' = h_0 \cup h'$ and $(\Pi, c, s, h) \Downarrow (s', h_0)$.

5.1 Program Logic Judgments

A list Γ of function specifications with the function names all distinct, is called an *environment*. We shall define the judgment

$$\Delta; \Gamma \models \delta:\text{Spec},$$

which states that in the context Δ , given the assumptions about functions recorded in Γ , the specification δ holds. This judgment is defined in several straightforward steps, and it is basically the same as the corresponding judgment in [Parkinson and Bierman 2005].

First, we give the semantics of a triple, relative to a context and a semantic function environment. The semantics of $\llbracket \Delta, \Pi \vdash \delta:\text{Spec} \rrbracket$ is a map from $\llbracket \Delta \rrbracket$ to the domain $\{\mathbf{true}, \mathbf{false}\}$, and it is given by (some obvious type annotations are omitted):

$$\begin{aligned} \llbracket \Delta, \Pi \vdash \{P\} c \{Q\} \rrbracket &\text{ iff } \forall h \in \llbracket \Delta \vdash P \rrbracket s. \\ &\quad - (\Pi, c, s, h) \text{ is safe, and} \\ &\quad - (\Pi, c, s, h) \Downarrow (s', h') \text{ implies } h' \in \llbracket \Delta \vdash Q \rrbracket s' \\ \llbracket \Delta, \Pi \vdash \delta \wedge \delta' \rrbracket s &\text{ iff } \llbracket \Delta, \Pi \vdash \delta \rrbracket s \text{ and } \llbracket \Delta, \Pi \vdash \delta' \rrbracket s \\ \llbracket \Delta, \Pi \vdash \exists x:\tau. \delta \rrbracket s &\text{ iff } \llbracket \Delta, \Pi \vdash \delta \rrbracket s_{[x \rightarrow v]} \text{ for some } v \in \llbracket \tau \rrbracket \\ \llbracket \Delta, \Pi \vdash \forall x:\tau. \delta \rrbracket s &\text{ iff } \llbracket \Delta, \Pi \vdash \delta \rrbracket s_{[x \rightarrow v]} \text{ for all } v \in \llbracket \tau \rrbracket. \end{aligned}$$

We call $\Delta, \Pi \vdash \delta$ *valid* and write $\Delta, \Pi \models \delta$ iff $\llbracket \Delta, \Pi \vdash \delta \rrbracket s = \mathbf{true}$ for all $s \in \llbracket \Delta \rrbracket$.

$$\begin{array}{c}
(\Pi, \mathbf{skip}, s, h) \Downarrow (s, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \\
\hline
(\Pi, x := E, s, h) \Downarrow (s_{[x \rightarrow n]}, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \\
\hline
(\Pi, \mathbf{return} E, s, h) \Downarrow (s_{[\text{ret} \rightarrow n]}, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \quad n \in \text{dom}(h) \quad h(n) = n' \\
\hline
(\Pi, x := [E], s, h) \Downarrow (s_{[x \rightarrow n']}, h) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \quad \llbracket \Delta \vdash E':\text{Int} \rrbracket s = n' \quad n \in \text{dom}(h) \\
\hline
(\Pi, [E] := E', s, h) \Downarrow (s, h_{[n \rightarrow n']}) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \quad n \in \text{dom}(h) \\
\hline
(\Pi, \mathbf{dispose}(E), s, h) \Downarrow (s, h - \{n\}) \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \quad n \notin \text{dom}(h) \\
\hline
(\Pi, x := [E], s, h) \Downarrow \text{wrong} \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \quad n \notin \text{dom}(h) \\
\hline
(\Pi, [E] := E', s, h) \Downarrow \text{wrong} \\
\hline
\llbracket \Delta \vdash E:\text{Int} \rrbracket s = n \quad n \notin \text{dom}(h) \\
\hline
(\Pi, \mathbf{dispose}(E), s, h) \Downarrow \text{wrong} \\
\hline
\{n, n+1, \dots, n+m\} \perp \text{dom}(h) \quad (\llbracket \Delta \vdash E_i:\text{Int} \rrbracket s = n_i)_{i=0, \dots, m} \\
\hline
(\Pi, x := \mathbf{cons}(E_0, \dots, E_m), s, h) \Downarrow (s_{[x \rightarrow n]}, h_{[n+i \rightarrow n_i]_{i=0, \dots, m}}) \\
\hline
(\Pi, c_1, s, h) \Downarrow (s', h') \quad (\Pi, c_2, s', h') \Downarrow (s'', h'') \\
\hline
(\Pi, c_1; c_2, s, h) \Downarrow (s'', h'') \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket s = \mathbf{false} \quad (\Pi, c_1, s, h) \Downarrow (s', h') \\
\hline
(\Pi, \mathbf{if} B \mathbf{then} c_0 \mathbf{else} c_1 \mathbf{fi}) \Downarrow (s', h') \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket s = \mathbf{true} \quad (\Pi, c_0, s, h) \Downarrow (s', h') \\
\hline
(\Pi, \mathbf{if} B \mathbf{then} c_0 \mathbf{else} c_1 \mathbf{fi}) \Downarrow (s', h') \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket s = \mathbf{false} \\
\hline
(\Pi, \mathbf{while} B \mathbf{do} c \mathbf{od}, s, h) \Downarrow (s, h) \\
\hline
\llbracket \Delta \vdash B:\text{Bool} \rrbracket s = \mathbf{true} \quad (\Pi, c; \mathbf{while} B \mathbf{do} c \mathbf{od}, s, h) \Downarrow (s', h') \\
\hline
(\Pi, \mathbf{while} B \mathbf{do} c \mathbf{od}, s, h) \Downarrow (s', h') \\
\hline
\Pi(k) = ((x_1, \dots, x_m), c_k) \quad (\Pi, c_k, [x_i \rightarrow n_i], h) \Downarrow (s', h') \\
\hline
(\llbracket \Delta \vdash E_i:\text{Int} \rrbracket s = n_i)_{i=1, \dots, m} \\
\hline
(\Pi, x = k(E_1, \dots, E_m), s, h) \Downarrow (s_{[x \rightarrow s'(\text{ret})]}, h') \\
\hline
(\Pi, c, s_{[x \rightarrow \text{null}]}, h) \Downarrow (s', h') \quad s(x) = v \\
\hline
(\Pi, \mathbf{newvar} x; c, s, h) \Downarrow (s'_{[x \rightarrow v]}, h') \\
\hline
(\Pi \cup (k_1 \rightarrow ((x_1, \dots, x_{n_1}), c_1), \dots, k_n \rightarrow ((x_1, \dots, x_{n_k}), c_n)), c, s, h) \Downarrow (s', h') \\
\hline
(\Pi, \mathbf{let} k_1(x_1, \dots, x_{n_1}) = c_1, \dots, k_n(x_1, \dots, x_{n_n}) = c_n \mathbf{in} c, s, h) \Downarrow (s', h')
\end{array}$$

Fig. 1. Operational Semantics of the Programming Language

LEMMA 5.1. *Let δ be a specification, $x:\tau$ a variable, and $\Delta \vdash t:\tau$ a term. Further, let $s \in \llbracket \Delta \rrbracket$, and Π be well-formed. Then,*

$$\llbracket \Delta; \Pi \vdash \delta[t/x] \rrbracket s \text{ iff } \llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket s_{[x \rightarrow v]},$$

where $v = \llbracket \Delta \vdash t:\tau \rrbracket s$.

There is a similar semantics for function specifications. This semantics is a map

$$\llbracket \Delta, \Pi \vdash \gamma:\text{FSpec} \rrbracket : \llbracket \Delta \rrbracket \rightarrow \{\mathbf{true}, \mathbf{false}\},$$

and it is given in much the same way as the corresponding map for specifications. The only difference is the base case, which is given by

$$\begin{aligned} \llbracket \Delta; \Pi \vdash \{P\} k \{Q\} \rrbracket s \text{ iff } \llbracket \Delta'; \Pi \vdash \{P\} c_m \{Q\} \rrbracket s \\ \text{where } \Pi(k) = ((x_1, \dots, x_{n_m}), c_m), \end{aligned}$$

where Δ' is Δ with those x_i that do not appear in Δ added (with type Int).

LEMMA 5.2. *Let γ be a specification, $x:\tau$ a variable, and $\Delta \vdash t:\tau$ a term. Further, let $s \in \llbracket \Delta \rrbracket$, and Π be well-formed. Then,*

$$\llbracket \Delta; \Pi \vdash \gamma[t/x] \rrbracket s \text{ iff } \llbracket \Delta, x:\tau; \Pi \vdash \gamma \rrbracket s_{[x \rightarrow v]},$$

where $v = \llbracket \Delta \vdash t:\tau \rrbracket s$.

As mentioned, an environment is a list of function specifications. The semantics of an environment is given componentwise:

$$\llbracket \Delta, \Pi \vdash \Gamma \rrbracket s \text{ iff } \llbracket \Delta, \Pi \vdash \gamma \rrbracket s \text{ for all } \gamma \in \Gamma.$$

LEMMA 5.3. *Let $\Delta \vdash t:\tau$ be a term, $s \in \llbracket \Delta \rrbracket$, and Γ an environment. Then,*

$$\llbracket \Delta; \Pi \vdash \Gamma[t/x] \rrbracket s \text{ iff } \llbracket \Delta, x:\tau; \Pi \vdash \Gamma \rrbracket s_{[x \rightarrow v]},$$

where $v = \llbracket \Delta \vdash t:\tau \rrbracket s$.

Finally, the semantics of specifications, relative to a context and an environment, is defined by

$$\begin{aligned} \Delta; \Gamma \models \delta \text{ iff for all well-formed } \Pi \text{ and all } s \in \llbracket \Delta \rrbracket, \\ \llbracket \Delta; \Pi \vdash \Gamma \rrbracket s \text{ implies } \llbracket \Delta; \Pi \vdash \delta \rrbracket s. \end{aligned}$$

The relevant substitution lemma for this semantics is:

LEMMA 5.4. *Let $\Delta \vdash t:\tau$ be a term. Then*

$$\Delta, x:\tau; \Gamma \models \delta \text{ implies } \Delta; \Gamma[t/x] \models \delta[t/x].$$

5.2 Inference Rules

We define a judgment

$$\Delta; \Gamma \vdash \delta,$$

for deriving valid specifications. The inference rules are given in Fig. 2. We first explain some of the rules at an intuitive level, and then show soundness.

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash \{P\} \mathbf{skip} \{P\}} \quad \frac{}{\Delta; \Gamma \vdash \{P[E/x]\} x := E \{P\}} \quad x \notin \text{FV}(E) \\
\frac{}{\Delta; \Gamma \vdash \{P[E/ret]\} \mathbf{return} E \{P\}} \quad \frac{\{P\} k(\vec{x}) \{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P[\vec{E}/\vec{x}]\} y = k(\vec{E}) \{Q[\vec{E}, y/\vec{x}, ret]\}} \\
\frac{}{\Delta; \Gamma \vdash \{\mathbf{emp} \wedge x = m\} x := \mathbf{cons}(E_1, \dots, E_n) \{x \mapsto \mathcal{E}_1[m/x], \dots, \mathcal{E}_n[m/x]\}} \\
\frac{}{\Delta; \Gamma \vdash \{E \mapsto -\} \mathbf{dispose}(E) \{\mathbf{emp}\}} \\
\frac{}{\Delta; \Gamma \vdash \{E \mapsto n \wedge x = m\} x := [E] \{E[m/x] \mapsto n \wedge x = n\}} \\
\frac{}{\Delta; \Gamma \vdash \{E \mapsto -\} [E] := E' \{E \mapsto E'\}} \\
\Delta; \Gamma \vdash \{P_1\} c_1 \{Q_1\} \\
\vdots \\
\Delta; \Gamma \vdash \{P_n\} c_n \{Q_n\} \\
\frac{\Delta; \Gamma, \{P_1\} k_1 \{Q_1\}, \dots, \{P_n\} k_n \{Q_n\} \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P\} \mathbf{let} k_1(\vec{x}_1) = c_1, \dots, k_n(\vec{x}_n) = c_n \mathbf{in} c \{Q\}} \\
\frac{\Delta; \Gamma \vdash \{P\} c_1 \{P'\} \quad \Delta; \Gamma \vdash \{P'\} c_2 \{Q\}}{\Delta; \Gamma \vdash \{P\} c_1; c_2 \{Q\}} \\
\frac{\Delta, x:\text{int}; \Gamma \vdash \{P \wedge x = \text{null}\} c \{Q\}}{\Delta; \Gamma \vdash \{P\} \mathbf{newvar} x \mathbf{in} c \mathbf{end} \{Q\}} \quad x \notin \text{FV}(P) \\
\frac{\Delta; \Gamma \vdash \{P \wedge B\} c_1 \{Q\} \quad \Delta; \Gamma \vdash \{P \wedge \neg B\} c_2 \{Q\}}{\Delta; \Gamma \vdash \{P\} \mathbf{if} B \mathbf{then} c_1 \mathbf{else} c_2 \mathbf{fi} \{Q\}} \\
\frac{}{\Delta; \Gamma \vdash \{P \wedge B\} c \{P\}} \\
\frac{}{\Delta; \Gamma \vdash \{P\} \mathbf{while} B \mathbf{do} c \mathbf{od} \{P \wedge \neg B\}} \\
\frac{\Delta; \Gamma \vdash \{P\} c \{Q\}}{\Delta, \Delta'; \Gamma \vdash \{P\} c \{Q\}} \quad \Delta \cap \Delta' = \emptyset \\
\frac{\Delta, \Delta'; \Gamma \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P\} c \{Q\}} \quad \Delta' \text{ disjoint from } \Delta, P, c, Q, \Gamma \\
\frac{\Delta \vdash P \Rightarrow P' \quad \Delta; \Gamma \vdash \{P'\} c \{Q'\} \quad \Delta \vdash Q' \Rightarrow Q}{\Delta; \Gamma \vdash \{P\} c \{Q\}} \\
\frac{\Delta, x:\tau; \Gamma, \gamma \vdash \delta}{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta} \quad x \notin \text{FV}(\Gamma) \cup \text{Mod}(\delta) \\
\frac{\Delta, x:\tau; \Gamma \vdash \delta}{\Delta; \Gamma \vdash \forall x:\tau. \delta} \quad x \notin \text{FV}(\Gamma) \cup \text{Mod}(\delta) \\
\frac{\Delta; \Gamma \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P * P'\} c \{Q * P'\}} \quad \text{Mod}(c) \cap \text{FV}(P') = \emptyset
\end{array}$$

Fig. 2. Program Logic

5.3 Informal Explanation of Rules

The first two rules are the usual rules for **skip** and assignment from Hoare logic. The rule for **return** is similar to the rule for assignment, since **return** simply amounts to an assignment to the special variable *ret*.

The rule

$$\frac{\{P\} k(\vec{x}) \{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P[\vec{E}/\vec{x}]\} y = k(\vec{E}) \{Q[\vec{E}, y/\vec{x}, \text{ret}]\}}$$

for a function call says that in order to call a function, the precondition for the function must be satisfied. This precondition is recorded in the environment, along with the corresponding postcondition.

The next four rules which involve the heap-manipulating constructs of the programming language, are the standard rules of separation logic, adapted to our setting. Note that the specifications are “tight” in the sense that they only mention the heap cells that are actually manipulated by the commands. For example, the rule

$$\frac{}{\Delta; \Gamma \vdash \{\text{emp} \wedge x = m\} x := \mathbf{cons}(\vec{E})\{x \mapsto \vec{E}[m/x]\}}$$

for **cons** produces a new cell when run in an empty heap. Note that this does *not* mean that **cons** can only be executed in an empty heap. The last rule of the system,

$$\frac{\Delta; \Gamma \vdash \{P\} c \{Q\}}{\Delta; \Gamma \vdash \{P * P'\} c \{Q * P'\}} \text{Mod}(c) \cap \text{FV}(P') = \emptyset,$$

called *the frame rule*, implies that one can infer a *global* specification from a *local* specification like the one for **cons**. Hence, **cons** can be executed in *any* heap, described by the predicate *P* (in which *x* does not occur freely), by the following instance of the frame rule:

$$\frac{\Delta; \Gamma \vdash \{\text{emp} \wedge x = m\} x := \mathbf{cons}(\vec{E})\{x \mapsto \vec{E}[m/x]\}}{\Delta; \Gamma \vdash \{P \wedge x = m\} x := \mathbf{cons}(\vec{E})\{P * (x \mapsto \vec{E}[m/x])\}}.$$

The rule

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \{P_1\} c_1 \{Q_1\} \\ \vdots \\ \Delta; \Gamma \vdash \{P_n\} c_n \{Q_n\} \\ \Delta; \Gamma, \{P_1\} k_1 \{Q_1\}, \dots, \{P_n\} k_n \{Q_n\} \vdash \{P\} c \{Q\} \end{array}}{\Delta; \Gamma \vdash \{P\} \mathbf{let} k_1(\vec{x}_1) = c_1, \dots, k_n(\vec{x}_n) = c_n \mathbf{in} c \{Q\}}$$

for function definitions is the usual one from Hoare logic with procedures [Hoare 1971]. The rules for **while** and **if-then-else** are also standard. The next two rules are structural and allow certain straightforward manipulations of contexts. The rule of consequence is standard, and the rules

$$\frac{\Delta, x:\tau; \Gamma, \gamma \vdash \delta}{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta} x \notin \text{FV}(\Gamma)$$

$$\frac{\Delta, x:\tau; \Gamma \vdash \delta}{\Delta; \Gamma \vdash \forall x:\tau. \delta} x \notin \text{Mod}(\delta)$$

are straightforward adaptations of standard rules of predicate logic. (Note that by the convention that variables in contexts Δ are all distinct, $x \notin \text{FV}(\delta)$ in the first rule and $x \notin \text{FV}(\Gamma)$ in the second rule.) They are used later for reasoning about data abstraction and parametricity. Note here that x may be of any type τ , including higher types for predicates, see the examples in Sections 6 and 7.

5.4 Soundness

THEOREM 5.1. *If a specification*

$$\Delta; \Gamma \vdash \delta$$

can be derived from the rules in Fig. 2, then it is valid.

PROOF. By induction. For each rule of form

$$\frac{\Delta; \Gamma \vdash \delta}{\Delta'; \Gamma' \vdash \delta'}, \quad (13)$$

we check $\Delta'; \Gamma' \models \delta'$, under the assumption $\Delta; \Gamma \models \delta$. For axioms of the form

$$\overline{\Delta; \Gamma \vdash \delta},$$

the proof obligation is to show $\Delta; \Gamma \models \delta$.

Consider the rule for **skip**:

$$\overline{\Delta; \Gamma \vdash \{P\} \mathbf{skip} \{P\}}$$

Although trivial, we show soundness of this rule here, to exercise the definitions. Let Π be a well-formed semantic function environment. It suffices to show

$$\llbracket \Delta; \Pi \vdash \{P\} \mathbf{skip} \{P\} \rrbracket s$$

for all $s \in \llbracket \Delta \rrbracket$. Let $h \in \llbracket P \rrbracket s$. Then,

$$(\Pi, \mathbf{skip}, s, h) \Downarrow (s, h),$$

and clearly, $h \in \llbracket P \rrbracket s$, so this rule is sound.

Soundness of the rule for assignment

$$\overline{\Delta; \Gamma \vdash \{P[E/x]\} x := E \{P\}}$$

depends, as usual, on the standard substitution lemma for assertions (not included in the review in Section 3).

The rule for **return**

$$\overline{\Delta; \Gamma \vdash \{P[E/ret]\} \mathbf{return} E \{P\}}$$

is essentially just an instance of the assignment rule.

Now consider the rule for function call:

$$\frac{\{P\} k_i(x_1, \dots, x_{n_i}) \{Q\} \in \Gamma}{\Delta, \Gamma \vdash \{P[E_1/x_1 \cdots E_{n_i}/x_{n_i}]\} y = k_i(E_1, \dots, E_{n_i}) \{Q[E_1/x_1 \cdots E_{n_i}/x_{n_i}, y/ret]\}}$$

To show soundness, suppose $\{P\} k_i(x_1, \dots, x_{n_i}) \{Q\} \in \Gamma$. Let $s \in \llbracket \Delta \rrbracket$, and let Π be a well-formed semantic function environment with $\llbracket \Delta; \Pi \models \Gamma \rrbracket s$. In particular,

$$\llbracket \Delta; \Pi \vdash \{P\} k_i(x_1, \dots, x_{n_i}) \{Q\} \rrbracket s,$$

so if $\Pi(k_i) = ((x_1, \dots, x_{n_i}), c_i)$, then $\llbracket \Delta; \Pi \vdash \{P\} c_i \{Q\} \rrbracket s$. Now, suppose

$$h \in \llbracket P[E_1/x_1 \cdots E_{n_i}/x_{n_i}] \rrbracket s = \llbracket P \rrbracket s_{[x_1 \rightarrow v_1, \dots, x_{n_i} \rightarrow v_{n_i}]},$$

where $v_j = \llbracket \Delta \vdash E_j : \text{Int} \rrbracket s$ for $j = 1, \dots, n_i$, by the substitution lemma. This means that if

$$(\Pi, c_i, s_{[x_1 \rightarrow v_1, \dots, x_{n_i} \rightarrow v_{n_i}]}, h) \Downarrow (s', h'),$$

then $h' \in \llbracket Q \rrbracket s'$. Since Π is well-formed, c_i does not modify any variables, so s' is of the form

$$s' = s_{[x_1 \rightarrow v_1, \dots, x_{n_i} \rightarrow v_{n_i}, \text{ret} \rightarrow s'(\text{ret})]},$$

and by the substitution lemma, $h' \in \llbracket Q[E_1/x_1 \cdots E_{n_i}/x_{n_i}, s'(\text{ret})/\text{ret}] \rrbracket s$. By the operational semantics for function calls,

$$(\Pi, y = k_i(E_1, \dots, E_{n_i}), s, h) \Downarrow (s_{[y \rightarrow s'(\text{ret})]}, h'),$$

and thus, the rule holds.

The first rule for existentials is

$$\frac{\Delta, x:\tau; \Gamma, \gamma \vdash \delta}{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta} \quad x \notin \text{Mod}(\delta) \cup \text{FV}(\Gamma)$$

Suppose that for all well-formed Π and $s \in \llbracket \Delta, x:\tau \rrbracket$,

$$\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \gamma \rrbracket s \text{ implies } \llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket s,$$

and let $\llbracket \Delta; \Pi \vdash \Gamma \rrbracket s$ and $\llbracket \Delta; \Pi \vdash \exists x:\tau. \gamma \rrbracket s$. This means $\llbracket \Delta; \Pi \vdash \gamma \rrbracket s_{[x \rightarrow v]}$ for some $v \in \llbracket \tau \rrbracket$. Since $x \notin \text{FV}(\Gamma)$, $\llbracket \Delta; \Pi \vdash \Gamma \rrbracket s_{[x \rightarrow v]}$. This implies $\llbracket \Delta; \Pi \vdash \delta \rrbracket s_{[x \rightarrow v]}$, and since $x \notin \text{Mod}(\delta)$, we have $\llbracket \Delta; \Pi \vdash \delta \rrbracket s$.

The other rule for existentials is

$$\frac{\Delta; \Gamma, \exists x:\tau. \gamma \vdash \delta}{\Delta, x:\tau; \Gamma, \gamma \vdash \delta} \quad x \notin \text{Mod}(\delta).$$

For soundness, first suppose τ is inhabited and that for all well-formed Π and $s \in \llbracket \Delta \rrbracket$,

$$\llbracket \Delta; \Pi \vdash \Gamma, \exists x:\tau. \gamma \rrbracket s \text{ implies } \llbracket \Delta; \Pi \vdash \delta \rrbracket s,$$

and suppose $\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \gamma \rrbracket s$. Since τ is inhabited, this means

$$\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \gamma \rrbracket s_{[x \rightarrow s(x)]},$$

and since $x \notin \text{FV}(\Gamma)$, this implies

$$\llbracket \Delta, x:\tau; \Pi \vdash \Gamma, \exists x:\tau. \gamma \rrbracket s,$$

and thus, $\llbracket \Delta; \Pi \vdash \delta \rrbracket s$, as desired. If τ is an empty type, one can make an easy case analysis on whether x occurs in γ or not.

Soundness of the downwards rule for universals is easy. For soundness of the upwards rule:

$$\frac{\Delta; \Gamma \vdash \forall x:\tau. \delta}{\Delta, x:\tau; \Gamma \vdash \delta} \quad x \notin \text{Mod}(\delta),$$

suppose for all well-formed Π and $s \in \llbracket \Delta \rrbracket$,

$$\llbracket \Delta; \Pi \vdash \Gamma \rrbracket s \text{ implies } \llbracket \Delta; \pi \vdash \forall x:\tau. \delta \rrbracket s,$$

and let $s' \in \llbracket \Delta, x:\tau \rrbracket$. Suppose $\llbracket \Delta, x:\tau; \Pi \vdash \Gamma \rrbracket s'$. Since $x \notin \text{FV}(\Gamma)$,

$$\llbracket \Delta; \Pi \vdash \Gamma \rrbracket (s' - x),$$

and this implies

$$\llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket (s' - x)_{[x \rightarrow v]}, \text{ for all } v \in \llbracket \tau \rrbracket.$$

This means in particular,

$$\llbracket \Delta, x:\tau; \Pi \vdash \delta \rrbracket s'_{[x \rightarrow s'(x)]},$$

which shows the desired result. \square

5.5 A Derived Rule

There is an important rule for abstract function definitions that is derivable from the rules in Fig. 2. The rule is

$$\frac{\begin{array}{c} \Delta \vdash \hat{P}:\tau \\ \Delta; \Gamma \vdash \{P_1[\hat{P}/x]\} c_1 \{Q_1[\hat{P}/x]\} \\ \vdots \\ \Delta; \Gamma \vdash \{P_n[\hat{P}/x]\} c_n \{Q_n[\hat{P}/x]\} \\ \Delta; \Gamma, \exists x:\tau. (\{P_1\}k_1\{Q_1\} \wedge \dots \wedge \{P_n\}k_n\{Q_n\}) \vdash \{P\} c \{Q\} \\ \Delta; \Gamma \vdash \{P\} \text{ let } k_1(\vec{x}_1) = c_1, \dots, k_n(\vec{x}_n) = c_n \text{ in } c \text{ end } \{Q\} \end{array}}{\Delta; \Gamma \vdash \{P\} c \{Q\}} \quad x \notin \text{FV}(\{P\} c \{Q\}). \quad (14)$$

Here one may think of x as a predicate describing a resource invariant used by an abstract data type with operations k_1, \dots, k_n .

We show how this rule can be derived; for simplicity, we assume $n = 1$ and that there are no parameters. The proof of the more general case is essentially the same as for this case. First, by the function definition rule,

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \{P_1[\hat{P}/x]\} c_1 \{Q_1[\hat{P}/x]\} \\ \Delta; \Gamma, \{P_1[\hat{P}/x]\} k_1 \{Q_1[\hat{P}/x]\} \vdash \{P\} c \{Q\} \end{array}}{\Delta; \Gamma \vdash \{P\} \text{ let } k_1 = c_1 \text{ in } c \{Q\}}.$$

The rule for existentials gives us

$$\frac{\Delta; \Gamma, \exists x:\tau. \{P_1\} k_1 \{Q_1\} \vdash \{P\} c \{Q\}}{\Delta, x:\tau; \Gamma, \{P_1\} k_1 \{Q_1\} \vdash \{P\} c \{Q\}},$$

so we need to establish

$$\Delta; \Gamma, \{P_1[\hat{P}/x]\} k_1 \{Q_1[\hat{P}/x]\} \vdash \{P\} c \{Q\}$$

given

$$\Delta, x:\tau; \Gamma, \{P_1\} k_1 \{Q_1\} \vdash \{P\} c \{Q\}.$$

But this follows from Lemma 5.4, since x is not free in $\{P\} c \{Q\}$.

6. DATA ABSTRACTION VIA EXISTENTIAL QUANTIFICATION

We present an example that demonstrates how one may use the program logic for reasoning about data abstraction. The example involves two implementations of a priority queue, and the intention is, of course, that client programs which use these implementations should be unaware of and unable to exploit details of the particular implementation used. Data abstraction is modeled via existential quantification over predicates, corresponding to the slogan “abstract types have existential type” [Mitchell and Plotkin 1985].

We first define an abstract priority queue, and use abstract operations in several client programs to demonstrate uses of abstract operations and their specifications. We then show two *implementations* of the abstract module, and prove that these have specifications which make application of the abstract function definition rule (14) possible.

6.1 Reasoning about Abstract Priority Queues

Priority queues are used frequently in programming, for example in scheduling algorithms for processes in operating systems [Silberschatz and Galvin 1998]. They consist of pairs (p, v) , where v is a stored value, and p is the *priority* associated with v . In such a structure, one can then enqueue such pairs and extract an element with the highest priority. For simplicity, assume that the values are integers. Here is a simple grammar for such a structure.

$$Q ::= \varepsilon \mid (p, v) \cup Q.$$

Some operations on such queues are needed. They use the axiom of choice, and are defined by

$$\begin{aligned} \text{MaxPri}(\varepsilon) &= -1 \\ \text{MaxPri}((p, v) \cup Q) &= \text{Max}(p, \text{MaxPri}(Q)) \\ \text{MaxPair}(Q) &= \mathbf{choose}(\{(p, v) \in Q \mid p = \text{MaxPri}(Q)\}). \end{aligned}$$

Note that MaxPair is a nondeterministic operation. We assume a base type PriQ whose values are priority queues, and an operation Set which, given a priority queue, returns the multiset of pairs occurring in it. These types and operations are only used in the logic, *not* in programs. Observe that the type PriQ is, of course, definable in the higher-order logic.

We now discuss how to reason about client code which uses an abstract priority queue. First, since client programs cannot modify abstract values, there should be a predicate stating that there is a “handle” to a priority queue. Hence, we introduce the predicate

$$\text{repr}(q, Q),$$

which asserts that the integer denoted by q is a handle to the priority queue Q – but it does *not* say anything about how Q is represented. Note that the type of repr is $(\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}$, a type of predicates.

This will be used as an abstract predicate in our proofs (and thus plays the role of x in the abstract function definition rule (14)). Given this predicate, the following are reasonable specifications for the various operations on a priority queue.

Creating a Queue. There should be an operation which enables a client program to create a priority queue. Its specification is

$$\{\text{emp}\} \text{createqueue}() \{\text{repr}(ret, \varepsilon)\},$$

which merely states that upon creation of a queue, a handle to an empty priority queue is returned.

Enqueing. There should be an operation for storing elements in a queue. The specification is

$$\{\text{repr}(q, Q)\} \text{enqueue}(q, (p, v)) \{\text{repr}(q, (p, v) \cup Q)\}.$$

Dequeing. There should be an operation for dequeing. We make sure not to dequeue from an empty queue via the specification

$$\begin{aligned} & \{\text{repr}(q, Q) \wedge Q \neq \varepsilon\} \\ & \text{dequeue}(q) \\ & \{\exists Q', p, v. \text{repr}(q, Q') \wedge Q = (p, v) \cup Q' \wedge (p, v) = \text{MaxPair}(Q) \wedge ret = v\}. \end{aligned}$$

Disposing a Queue. When done with a priority queue, we should be able to dispose it. Hence the specification

$$\{\text{repr}(q, Q)\} \text{disposequeue}(q) \{\text{emp}\}.$$

6.2 Sample Programs

We consider examples of client programs which use priority queues, along with their specifications.

The first program creates a queue, enqueues some elements, and dequeues again. Here, Q'' is used as a shorthand for $\langle(2, 42), (4, 17)\rangle$.

$$\begin{aligned} & \{\text{emp}\} \\ & \quad q = \text{createqueue}(); \\ & \{\text{repr}(q, \varepsilon)\} \\ & \quad \text{enqueue}(q, (4, 17)); \\ & \{\text{repr}(q, (4, 17) \cup \varepsilon)\} \\ & \quad \text{enqueue}(q, (2, 42)); \\ & \{\text{repr}(q, Q'')\} \\ & \{\text{repr}(q, Q'') \wedge Q'' \neq \varepsilon\} \\ & \quad y := \text{dequeue}(); \\ & \{\exists Q', p, v. \text{repr}(q, Q') \wedge Q'' = Q' \cup (p, v) \wedge (p, v) = \text{MaxElt}(Q'') \wedge y = v\} \\ \Downarrow & \\ & \{\text{repr}(q, \langle(2, 42)\rangle) \wedge y = 17\} \\ & \quad \text{disposequeue}(q) \\ & \{\text{emp} \wedge y = 17\}. \end{aligned}$$

The implication in the middle of this derivation uses the rule of consequence. Henceforth, this kind of implications is used implicitly in proofs. That is, given a nonempty abstract priority queue Q , if $\text{MaxPair}(Q) = (p, v)$ and $Q' = Q \setminus (p, v)$ (such that $Q = Q' \cup (p, v)$), we use the following specification for **dequeue**:

$$\begin{aligned} & \{\text{repr}(q, Q) \wedge Q \neq \varepsilon\} \\ & \text{dequeue}() \\ & \{\text{repr}(q, Q') \wedge ret = v\}. \end{aligned}$$

In summary, we have shown that

$$\begin{array}{l}
\exists \text{repr} : (\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}. \\
\{ \text{emp} \} \text{createqueue}() \{ \text{repr}(ret, \varepsilon) \}, \\
\dots \\
\{ \text{repr}(q, Q) \} \text{disposequeue}(q) \{ \text{emp} \} \\
\vdash \\
\{ \text{emp} \} c \{ \text{emp} \wedge y = 17 \},
\end{array}$$

where c is $q = \text{createqueue}(); \text{enqueue}(q, (4, 17)); \text{enqueue}(q, (2, 42)); y := \text{dequeue}(); \text{disposequeue}(q)$. Observe that the verification of c was done without knowledge of an actual implementation of the abstract type of priority queues.

Here is another basic program (and its specification) which, however, uses two queues. In the third step, the specification for **createqueue** is used along with the frame rule, and the frame rule is used again in all subsequent steps.

$$\begin{array}{l}
\{ \text{emp} \} \\
q_1 = \text{createqueue}(); \\
\{ \text{repr}(q_1, \varepsilon) \} \\
\text{enqueue}(q_1, (3, 7)); \\
\{ \text{repr}(q_1, \langle (3, 7) \rangle) \} \\
q_2 = \text{createqueue}(); \\
\{ \text{repr}(q_1, \langle (3, 7) \rangle) * \text{repr}(q_2, \varepsilon) \} \\
\text{enqueue}(q_1, (7, 3)); \\
\{ \text{repr}(q_1, \langle (3, 7), (7, 3) \rangle) * \text{repr}(q_2, \varepsilon) \} \\
\text{enqueue}(q_2, (4, 13)); \\
\{ \text{repr}(q_1, \langle (3, 7), (7, 3) \rangle) * \text{repr}(q_2, \langle (4, 13) \rangle) \} \\
y_1 := \text{dequeue}(q_1); \\
\{ \text{repr}(q_1, \langle (3, 7) \rangle) * \text{repr}(q_2, \langle (4, 13) \rangle) \wedge y_1 = 3 \} \\
y_2 := \text{dequeue}(q_2); \\
\{ \text{repr}(q_1, \langle (3, 7) \rangle) * \text{repr}(q_2, \varepsilon) \wedge y_1 = 3 \wedge y_2 = 13 \} \\
\text{disposequeue}(q_1); \\
\text{disposequeue}(q_2); \\
\{ \text{emp} \wedge y_1 = 3 \wedge y_2 = 13 \}
\end{array}$$

This example illustrates that our notion of modularity is not static, as is the case in the paper [O’Hearn et al. 2004]. In that setting, it is not possible to give arguments to function calls – one has to initialize variables which are laid out in the specification of each module. This makes it impossible to have multiple instances of the same data structure, as also noted by Parkinson and Bierman [Parkinson and Bierman 2005].

It is illustrative to demonstrate that erroneous programs cannot have a specification in our system. We show two such examples. In the first example, we call **dequeue** on an empty queue.

$$\begin{array}{l}
\{ \text{emp} \} \\
q = \text{createqueue}() \\
\{ \text{repr}(q, \varepsilon) \} \\
y = \text{dequeue}(q) \\
\{ ??? \}
\end{array}$$

Since the precondition for **dequeue** is that there is a non-empty queue represented, there is no assertion which can be put in place of ??? in this derivation.

In the second erroneous program, we dequeue from a disposed queue.

```

{emp}
  q = createqueue();
{repr(q, ε)}
  enqueue(q, (4, 42));
{repr(q, ⟨4, 42⟩)}
  disposequeue(q);
{emp}
  y = dequeue(q)
{???}

```

Since the precondition for **dequeue** is that there is a non-empty queue represented (and this is not implied by **emp**), there is no assertion which can be put in place of ??? in this derivation. In some implementations of a priority queue, the **disposequeue** operation might be a no-op, and thus the **dequeue** would make sense operationally. But allowing this in our system would amount to exposing the implementation to the client program, contradicting the principle of *encapsulation* in data abstraction. This is even more evident in the program

```

{emp}
  q = createqueue();
{repr(q, ε)}
  newvar x; x := [q]
{???}.

```

In most implementations of priority queues, q denotes a pointer to a data structure in the heap which in some sense “represents” the priority queue Q , and therefore, it *would* make operational sense to dereference q . But, the abstract interface to the queue does not mention any heap cells, and therefore, there is no assertion we can fill in for ???. This is fine, since disallowing client programs to dereference handles is in harmony with the principles of data abstraction, which is addressed here.

Note that the priority queues here do not involve any *ownership transfer* [O’Hearn et al. 2004], since the priority queues hold simple data values (integers) only. We could have implemented queues where ownership of heap cells transfers back and forth between clients and modules. Although this would certainly be interesting, the core ideas of our approach to data abstraction in separation logic are presented via the examples at hand.

6.3 Implementations of Priority Queues

In this section, we first present two implementations and then discuss how these can be used to reason about data abstraction in our framework. One implementation uses a sorted, singly-linked list, whereas the other uses an unsorted doubly-linked list.

6.3.1 Sorted, Singly-linked Lists. Singly-linked lists (or rather, singly-linked list segments) are introduced in Reynolds’ introductory paper on separation logic [Reynolds

2002]. The idea is that the predicate

$$\text{slist}(\alpha, i, j)$$

asserts that the finite sequence $\alpha = (p_0, v_0), \dots, (p_n, v_n)$ of priority / value pairs (Reynolds just uses sequences of integers) is represented in the heap in a singly linked list. It is defined by induction on the length of α .

$$\begin{aligned} \text{slist}(\varepsilon, i, j) &\stackrel{\text{def}}{=} \text{emp} \wedge i = j \\ \text{slist}((p, v) \cdot \alpha, i, j) &\stackrel{\text{def}}{=} \exists k. i \mapsto p, v, k * \text{slist}(\alpha, k, j). \end{aligned}$$

We show programs which implement the abstract priority queue with sorted singly linked lists in Appendix B (we have overloaded the **dispose** operation, to dispose several heap cells). With a sorted list, the only non-trivial operations are to enqueue an element and to dispose a queue (since we do not require the queue to be empty when it is disposed). Here is a brief explanation of each of the programs.

The **slistcreate** program simply initializes an empty **slist** (it needs only return *null*). The implementation of **slistEnque** first finds the appropriate position in the list to insert the new element to keep the list sorted (this is what happens in the **while** loop), and then inserts the element. Since the list is sorted according to priorities, we always dequeue the first element of the list, so it is easy to implement **slistDeque**. Finally, to dispose a queue amounts to dispose an **slist**.

One can prove the specifications shown in Figure 3 for these implementations, using the proof rules from Section 5.2 plus lemmas from Reynolds' paper [Reynolds 2002] about the **slist** predicate, and some obvious implications regarding sorted sequences. Our proofs use certain assertions and operations on sequences, which are easy to encode in higher-order logic. The predicate **sorted**(α) is true iff the sequence α is sorted (in decreasing order) according to priorities. For a nonempty sequence $\alpha = (p_0, v_0), \dots, (p_n, v_n)$, **Max**(α) is a pair from α with the highest priority – note that this is non-deterministic. Finally, for a sequence α , **Set**(α) is the multiset of pairs occurring in α .

The hard part is to show the specification for **slistEnque**, which is the most complicated program in this implementation. For the diligent reader who wants to verify the specifications, there are some basic facts and the invariant of the **while** loop of **slistEnque** that was used in our proof in Appendix D.

6.3.2 Doubly-linked Lists. Doubly linked lists are also introduced by Reynolds in the paper [Reynolds 2002]. The predicate

$$\text{dlist}(\alpha, i, i', j, j')$$

asserts that the sequence α is represented in the heap by a doubly linked list segment from i to j' . It is defined by induction on the length of α :

$$\begin{aligned} \text{dlist}(\varepsilon, i, i', j, j') &\stackrel{\text{def}}{=} \text{emp} \wedge i = j \wedge i' = j' \\ \text{dlist}((p, v) \cdot \alpha, i, j, i', j') &\stackrel{\text{def}}{=} \exists k. i \mapsto (p, v, k, i') * \text{dlist}(\alpha, k, i, j, j') \end{aligned}$$

In addition to this implementation with singly linked lists, we implement priority queues by doubly linked lists, but in contrast to our implementation with singly linked lists, this implementation does not keep the list sorted. Consequently, the

$$\begin{array}{l}
\{\text{emp}\} \\
\text{slistcreate} \\
\{\text{slist}(\varepsilon, \text{ret}, \text{null})\} \\
\\
\{\text{slist}(\alpha, q, \text{null}) \wedge \text{sorted}(\alpha)\} \\
\text{slistEnque}(q, (p, v)) \\
\{\exists \alpha'. \text{slist}(\alpha', q, \text{null}) \wedge \text{sorted}(\alpha') \wedge \text{Set}(\alpha') = \text{Set}(\alpha) \uplus (p, v)\} \\
\\
\{\text{slist}(\alpha, q, \text{null}) \wedge \text{sorted}(\alpha) \wedge \alpha \neq \varepsilon\} \\
\text{slistDeque}(q) \\
\left. \begin{array}{l}
\{\exists p, v, \alpha'. \text{slist}(\alpha', q, \text{null}) \wedge \text{Set}(\alpha) = \text{Set}(\alpha') \uplus (p, v) \wedge \\
(p, v) = \text{Max}(\alpha) \wedge \text{ret} = v\}
\end{array} \right\} \\
\\
\{\text{slist}(\alpha, q, \text{null})\} \\
\text{slistDispose}(q) \\
\{\text{emp}\}
\end{array}$$
Fig. 3. Specifications for `slist`-implementation of priority queues

tricky part is to dequeue, whereas enqueueing is easy. The implementations are shown in Appendix C. The handle q is used to point to a cell containing the values i, i', j, j' that determine the “boundaries” of the list, for technical reasons.

Here is a brief explanation of each of the programs in this implementation. The `dlistcreate` program just initializes an empty doubly linked list by storing *nulls* in the “handle cell” in the heap. To enqueue, just insert at the front of the doubly linked list, since the list is not kept sorted. To dequeue we have to find the correct element of the list and take it out of the list. This is done in `dlistDeque`; in the **while** loop we traverse the list and find an element with the highest priority, and then take it out of the list, in a way according to the position of the element in the list.

One can prove the specifications shown in Fig. 4 for these implementations of the priority queue operations. The proofs of these specifications use the same assertions and operations regarding sequences as the proofs about the `slist` representation. Most of the proofs are straightforward. As mentioned, the tricky part of this implementation is `dlistDeque`, and the proof of this program is correspondingly the trickiest one. The diligent reader who wishes to verify the specifications in Fig. 4 may find hints in Appendix E.

6.4 Representation Independence

We argue that our program logic may be used to reason about independence of the representation of data using the examples from Sec. 6.3. Intuitively, a client program should not be able to distinguish between the two implementations of a priority queue we have presented; this intuition is justified here.

$$\begin{array}{l}
\{\text{emp}\} \\
\text{dlistcreate} \\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\varepsilon, i, i', j, j')\} \\
\\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j')\} \\
\text{dlistEnqueue}(q, (p, v)) \\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}((p, v) \cdot \alpha, i, i', j, j')\} \\
\\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j') \wedge \alpha \neq \varepsilon\} \\
\text{dlistDequeue}(q) \\
\left. \begin{array}{l}
\{\exists i, i', j, j', p, v, \alpha'. \text{dlist}(\alpha', i, i', j, j') \wedge \text{Set}(\alpha) = \text{Set}(\alpha') \uplus (p, v) \wedge \\
(p, v) = \text{Max}(\alpha) \wedge \text{ret} = v\}
\end{array} \right\} \\
\\
\{\exists i, i', j, j'. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j')\} \\
\text{dlistDispose}(q) \\
\{\text{emp}\}
\end{array}$$

Fig. 4. Specifications for dlist-implementation of Priority Queues

Consider the two programs

$$\begin{array}{l}
c_1 \equiv \\
\mathbf{let} \\
\quad \mathbf{createqueue}() = \text{slistcreate} \\
\quad \mathbf{enqueue}(q, (p, v)) = \text{slistEnqueue}(q, (p, v)) \\
\quad \mathbf{dequeue}(q) = \text{slistDequeue}(q) \\
\quad \mathbf{disposequeue}(q) = \text{slistDispose}(q) \\
\mathbf{in } c \mathbf{ end}
\end{array}$$

and

$$\begin{array}{l}
c_2 \equiv \\
\mathbf{let} \\
\quad \mathbf{createqueue}() = \text{dlistcreate} \\
\quad \mathbf{enqueue}(q, (p, v)) = \text{dlistEnqueue}(q, (p, v)) \\
\quad \mathbf{dequeue}(q) = \text{dlistDequeue}(q) \\
\quad \mathbf{disposequeue}(q) = \text{dlistDispose}(q) \\
\mathbf{in } c \mathbf{ end},
\end{array}$$

where c is a program which uses priority queues, for example one of those in Section 6.2. Intuitively, it should not matter which implementation is used, so that any specification we can show for one program, we should be able to show for the other. Consider the abstract function definition rule (14) from Section 5.5, which can be used to verify such programs. The work we have already done can be used to apply this rule.

In the setting with singly linked lists, \hat{P} in rule (14) can be instantiated with the predicate $\hat{P}_{\text{slist}}: (\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}$ defined by

$$\hat{P}_{\text{slist}} \equiv \lambda(Q, q). \exists \alpha. \text{sorted}(\alpha) \wedge \text{Set}(\alpha) = \text{Set}(Q) \wedge \text{slist}(\alpha, q, \text{null}).$$

Using the specifications mentioned in Section 6.3, it is not hard to show the following specifications.

$$\begin{array}{l} \{\mathbf{emp}\} \\ \text{slistcreate} \\ \{\hat{P}_{\text{slist}}(q, \varepsilon)\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{slist}}(q, Q)\} \\ \text{slistEnqueue}(q, (p, v)) \\ \{\hat{P}_{\text{slist}}(q, Q \uplus (p, v))\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{slist}}(q, Q) \wedge Q \neq \varepsilon\} \\ \text{slistDeque}(q) \\ \{\exists Q', p, v. \hat{P}_{\text{slist}}(q, Q') \wedge \text{Set}(Q) = \text{Set}(Q') \uplus (p, v) \wedge (p, v) = \text{MaxElt}(Q) \wedge \text{ret} = v\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{slist}}(q, Q)\} \\ \text{slistDeque}(q) \\ \{\mathbf{emp}\}. \end{array}$$

Similarly, if we define the predicate $\hat{P}_{\text{dlist}}: (\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}$ by

$$\hat{P}_{\text{dlist}}(Q, q) \equiv \lambda(Q, q). \exists i, i', j, j', \alpha. q \mapsto i, i', j, j' * \text{dlist}(\alpha, i, i', j, j') \wedge \text{Set}(\alpha) = \text{Set}(Q),$$

the work in Section 6.3 entails the specifications

$$\begin{array}{l} \{\mathbf{emp}\} \\ \text{dlistcreate}() \\ \{\hat{P}_{\text{dlist}}(\varepsilon, q)\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{dlist}}(Q, q)\} \\ \text{dlistEnqueue}(q, (p, v)) \\ \{\hat{P}_{\text{dlist}}(Q \cup (p, v), q)\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{dlist}}(q, Q) \wedge Q \neq \varepsilon\} \\ \text{dlistDeque}(q) \\ \{\exists Q', p, v, \alpha. \hat{P}_{\text{dlist}}(q, Q') \wedge Q = Q' \cup (p, v) \wedge (p, v) = \text{MaxElt}(Q) \wedge \text{ret} = v\} \end{array}$$

$$\begin{array}{l} \{\hat{P}_{\text{dlist}}(q, Q)\} \\ \text{dlistDispose}(q) \\ \{\mathbf{emp}\}. \end{array}$$

The proofs of the programs in Section 6.2 which use the abstract predicate can be fitted into our framework. For example, the first of those programs has the specification

$$\Delta; \Gamma \vdash \{\mathbf{emp}\} c \{\mathbf{emp} \wedge y = 2\},$$

where Δ is the context $q:\text{Int}, y:\text{Int}, \text{repr}:(\text{PriQ} \times \text{Int}) \Rightarrow \text{Prop}$ and Γ is the environment containing the abstract functions listed at the end of Section 6.1:

$$\begin{aligned} \Gamma = & \{ \text{emp} \} \text{createqueue}() \{ \text{repr}(ret, \varepsilon) \}, \\ & \{ \text{repr}(q, Q) \} \text{enqueue}(q, (v, p)) \{ \text{repr}(q, (v, p) \cup Q) \}, \\ & \{ \text{repr}(q, Q) \wedge Q \neq \varepsilon \} \\ & \text{dequeue}(q) \\ & \{ \exists Q', p, v. \text{repr}(q, Q') \wedge Q = (p, v) \cup Q' \wedge (p, v) = \text{MaxElt}(Q) \wedge ret = v \}, \\ & \{ \text{repr}(q, Q) \} \text{disposequeue}(q) \{ \text{emp} \}. \end{aligned}$$

According to rule (14) in Section 5.5, we can then derive the specification

$$q, : \text{Int}, y:\text{Int}; \emptyset \vdash \{ \text{emp} \} \tilde{c} \{ \text{emp} \wedge y = 2 \},$$

where \tilde{c} is replaced by *either* c_1 or c_2 . Note again that the proof of the client program is independent of the implementation of the module it uses. This is the sense in which our system can be used to reason about data abstraction and representation independence.

7. SOME APPLICATIONS OF UNIVERSAL QUANTIFICATION

In the previous section we saw how to use existential quantification over predicates to reason about data abstraction. In this section we present two examples of how to apply *universal* quantification over predicates.

7.1 Polymorphic Types via Universal Quantification

We show that universally quantified predicates may be used to prove correct polymorphic operations on polymorphic data types.

The queue module example from [O'Hearn et al. 2004] is parametric in a predicate P at the meta-level. We show that in higher-order separation logic, the parameterization may be expressed *in the logic*. To that end, consider the following version of the parametric list predicate from [O'Hearn et al. 2004].

$$\text{list}(P, \beta, i) = \begin{cases} i = \text{null} \wedge \text{emp} & \text{if } \beta = \varepsilon \\ \exists j. i \mapsto x, j * P(x) * \text{list}(P, \beta', j) & \text{if } \beta = \langle x \rangle \cdot \beta' \end{cases}$$

The predicate P is required to hold for each element of the sequence β involved. Different instantiations of P yields different versions of the list, with different amounts of data stored in the list. If $P \equiv \text{emp}$, then plain values are stored (no ownership transfer to the queue module in [O'Hearn et al. 2004]), and if $P \equiv x \mapsto -, -$, then addresses of cells are stored in the queue (ownership of the cells is transferred in and out of the queue [O'Hearn et al. 2004]).

Returning to our higher-order separation logic, the definition of `list` may be formalized with

$$i : \text{Int}, \beta : \text{seqInt}, P : \text{Prop}^{\text{Int}} \vdash \text{list}(P, \beta, i) : \text{Prop}.$$

Here we have used a type `seqInt` of sequences of integers, which is easily definable in higher-order separation logic, and the definition of `list(P, β, i)` can be given by induction on β *in the logic*.

Suppose **listRev** is the list reversal program given in the introduction of [Reynolds 2002]. Then one can easily show the specification

$$\{\text{list}(P, \beta, i)\} \mathbf{listRev} \{\text{list}(P, \beta^\dagger, j)\}.$$

By the introduction rule for universal quantification we obtain the specification

$$\beta : \text{seqInt} \vdash \forall P : \text{Prop}^{\text{Int}}. \{\text{list}(P, \beta, i)\} \mathbf{listRev} \{\text{list}(P, \beta^\dagger, j)\},$$

which expresses that **listRev** is *parametric* in the sense that it, roughly speaking, reverses singly-linked lists of any type.

Thus we have *one* parametric correctness proof of a specification for **listRev**, which may then be used to prove correct different applications of **listRev** (to lists of different types).

For such parametric operations on polymorphic data types to be really useful, one would of course prefer a higher-order programming language instead of the first-order language considered here. Then one could, e.g., program the usual **map** function on lists, and provide a single parametric correctness proof for it. See our joint with Yang [Birkedal et al. 2005] for a proposal of separation logic for a higher-order language.

7.2 Invariance

In this subsection we briefly consider an example, suggested to us by John Reynolds, which demonstrates that one may use universal quantification to specify that a command does not modify its input state. We disregard stacks here since they are not important for the argument.

Suppose that our intention is to specify that some command c takes any heap h described by a predicate q , and produces a heap (we assume for simplicity that c terminates), which is an extension of h . We might attempt to use a specification of the form:

$$\{q\} c \{q' * q\}. \quad (15)$$

This does not work, however, unless q is *strictly exact* [Reynolds 2002] (for example, if q is $\exists \beta : \text{seqInt}. \text{list}(\beta, i)$, then c may delete some elements from the list in the input heap h).

Instead, we may use the following specification

$$\forall p : \text{Prop}. \{q \wedge p\} c \{q' * p\}, \quad (16)$$

as we see by the following argument. Predicate q describes a set of heaps $\llbracket q \rrbracket$. For each $h \in \llbracket q \rrbracket$, let $p_h = \{h\}$. Suppose c terminates in heap h' . Then $h' = h_1 * h$, for some h_1 . That is, the heap h is *invariant* under the execution of c , as intended.

Note that (16) is stronger than (15): by instantiation p with q in (16) we get (15). Thus if we wish to prove (15), then we may prove something stronger (16), which may be easier to prove (c.f., strengthening an induction hypothesis), and then derive the desired.

Thus we can use universal quantification to express invariance of commands.

8. RELATED WORK

There are references to related work throughout the paper. We give pointers to some more related work here.

As mentioned, Parkinson and Bierman proposed a system for reasoning about data abstraction [Parkinson and Bierman 2005], in which they introduce a notion of “abstract predicates”. Our approach is more straightforward and natural, since abstraction is here modeled using existential quantification over higher-types (no special syntax for abstract predicates is needed). Prior to that, Reddy gave a semantics for objects and classes [Reddy 1998], in which he also models data abstraction via existential quantification. The main difference between Reddy’s work and the present work is that Reddy considers a programming language *without* heap manipulating constructs, but which is higher-order. Furthermore, Reddy uses a sophisticated type system, and types are interpreted relationally. We are currently working on a relational interpretation of a higher-order programming language with heap-manipulating constructs, based on our analysis in [Birkedal et al. 2005].

Kohei Honda’s group has several papers [Berger et al. 2005; Honda et al. 2005] on higher-order imperative languages. The similarities between their work and the present is that both seek to reason about equivalence of programs in programming languages with pointers. Their work is, however, not restricted to our programming language with simple procedures. The main differences include: (i) Their logic does not include a $*$ connective; instead they use predicate logic with equality to keep track of aliasing. This appears to make local reasoning harder. (ii) The interpretation of triples is not “tight” in their work. For example, the triple

$$\{\mathbf{true}\} [x] := 4 \{\mathbf{true}\}$$

is valid in the setting of Honda *et al.*, but not in separation logic. (iii) One of the goal of Honda *et al.*’s work is to show observational equivalence. Although intriguing, we do not aim to answer such questions in the present work. See the long version of the paper [Berger et al. 2005] for an extensive comparison of their work to separation logic.

9. CONCLUSION

We introduced the notion of a BI hyperdoctrine and showed that it soundly and completely models intuitionistic and classical first- and higher-order BI. We showed that the standard semantics for BI provided by traditional separation logic is an instance of our class of models, and that interesting models for higher-order predicate BI cannot exist in toposes. Several applications of higher-order BI in program proving, and in particular separation logic, were illustrated. In particular, we introduced higher-order separation logic, and gave sound reasoning principles for data abstraction and parametric data types in the presence of aliasing, using existential and universal quantification over predicates, respectively.

APPENDIX

A. PROOF OF THEOREM 2.1

For a term t with $y:Y \vdash t(y):X$ we add the following abbreviation

$$\exists_t. \varphi(y) \stackrel{def}{=} \exists y:Y. t(y) = x \wedge \varphi(y)$$

The following rule can be deduced

$$\frac{x:X \mid \exists_t. \varphi(y) \vdash \psi(x)}{y:Y \mid \varphi(y) \vdash \psi[t(y)/x]}.$$

In particular for $y:\{x:X \mid \varphi\} \vdash o(y):X$ we have

$$\frac{x:X \mid \exists_o. \theta(y) \vdash \psi(x)}{y:\{x:X \mid \varphi\} \mid \theta(y) \vdash \varphi[o(y)/x]}.$$

Let $\varphi, \psi, \psi', \chi$ be formulas in a context $\{x:X\}$ (for simplicity we just assume one free variable, the general case is similar). First we show that

$$x:X \mid \varphi \wedge \psi \dashv\vdash \exists_o. \psi[o(y)/x]. \quad (17)$$

This is done by

$$\frac{\frac{x:X \mid \exists_o. \psi[o(y)/x] \vdash \exists_o. \psi[o(y)/x]}{y:\{x:X \mid \varphi\} \mid \psi[o(y)/x] \vdash (\exists_o. \psi[o(y)/x])[o(y)/x]}}{x:X \mid \psi \wedge \varphi \vdash \exists_o. \psi[o(y)/x]},$$

where the last derivation is the rule for full subset types. For the other direction, consider

$$\frac{y:\{x:X \mid \varphi\} \mid \psi[o(y)/x] \vdash \psi[o(y)/x]}{x:X \mid \exists_o. \psi[o(y)/x] \vdash \psi}$$

and

$$\frac{\frac{x:X \mid \varphi \wedge \psi \vdash \varphi}{y:\{x:X \mid \varphi\} \mid \psi[o(y)/x] \vdash \varphi[o(y)/x]}}{x:X \mid \exists_o. \psi[o(y)/x] \vdash \varphi},$$

which imply $x:X \mid \exists_o. \psi[o(y)/x] \vdash \varphi \wedge \psi$. We also need the following

$$\frac{y:\{x:X \mid \varphi\} \mid \chi[o(y)/x] \vdash \psi[o(y)/x]}{x:X \mid \exists_o. \chi[o(y)/x] \vdash \exists_o. \psi[o(y)/x]}. \quad (18)$$

which is shown by

$$\frac{\frac{y:\{x:X \mid \varphi\} \mid \chi[o(y)/x] \vdash \psi[o(y)/x]}{x:X \mid \chi \wedge \varphi \vdash \psi}}{x:X \mid \chi \wedge \varphi \vdash \psi \wedge \varphi},$$

where the last derivation follows from (17). We then have

$$\frac{y:\{x:X \mid \varphi\} \mid \psi[o(y)/x] * \psi'[o(y)/x] \vdash \chi[o(y)/x]}{y:\{x:X \mid \varphi\} \mid \psi[o(y)/x] \vdash \psi'[o(y)/x] \multimap \chi[o(y)/x]},$$

i.e.,

$$\frac{y:\{x:X \mid \varphi\} \mid (\psi * \psi')[o(y)/x] \vdash \chi[o(y)/x]}{y:\{x:X \mid \varphi\} \mid \psi[o(y)/x] \vdash (\psi' \multimap \chi)[o(y)/x]}.$$

By (18) we then get

$$\frac{x:X \mid \exists_o. (\psi * \psi')[o(y)/x] \vdash \exists_o. \chi[o(y)/x]}{x:X \mid \exists_o. \psi[o(y)/x] \vdash \exists_o. (\psi' \multimap \chi)[o(y)/x]},$$

which by 17 gives us

$$\frac{x:X \mid \varphi \wedge (\psi * \psi') \vdash \varphi \wedge \chi}{x:X \mid \varphi \wedge \psi \vdash \varphi \wedge (\psi' \multimap \chi)}.$$

This entails the following

$$\frac{\frac{\frac{x:X \mid \varphi \wedge (\psi * \psi') \vdash \chi}{x:X \mid \varphi \wedge (\psi * \psi') \vdash \chi \wedge \varphi}}{x:X \mid \varphi \wedge \psi \vdash \varphi \wedge (\psi' \multimap \chi)}}{x:X \mid \varphi \wedge \psi \vdash \psi' \multimap \chi}}{x:X \mid (\varphi \wedge \psi) * \psi' \vdash \chi}.$$

Letting χ be $(\varphi \wedge \psi) * \psi'$ respectively $\varphi \wedge (\psi * \psi')$ we read off the equivalence $x:X \mid \varphi \wedge (\psi * \psi') \dashv\vdash (\varphi \wedge \psi) * \psi'$. Now, let φ and ψ be \top and ψ' be \top ; this gives $\top \wedge (\top * \top) \dashv\vdash (\top \wedge \top) * \top$, that is, $\top \dashv\vdash \top$, which in return yields $\varphi \wedge (\top * \psi') \dashv\vdash (\varphi \wedge \top) * \psi'$, i.e., $\varphi \wedge \psi' \dashv\vdash \varphi * \psi'$. \square

B. IMPLEMENTATION USING SLISTS

Creating a queue with slists:

```
slistcreate  $\stackrel{\text{def}}{=} \mathbf{return\ null}$ 
```

Enqueing with slists:

```

slistEnque( $q, (p, v)$ )  $\stackrel{\text{def}}{=}$ 
  newvar ptemp, temp, tempri, found, new;
  ptemp := null;
  if ( $q = \text{null}$ )
     $q := \text{cons}(p, v, \text{null})$ 
  else
    tempri := [ $q$ ];
    if ( $p \geq \text{tempri}$ )
       $q := \text{cons}(p, v, q)$ 
    else
      ptemp :=  $q$ ;
      temp := [ $q + 2$ ];
      found := false;
      while ( $\text{temp} \neq \text{null} \wedge \text{found} = \text{false}$ ) do
        tempri := [ $\text{temp}$ ];
        if ( $p \geq \text{tempri}$ )
          found := true
        else
          ptemp := temp;
          temp := [ $\text{ptemp} + 2$ ]
        fi
      od;
      new :=  $\text{cons}(p, v, \text{temp})$ ;
      [ $\text{ptemp} + 2$ ] := new
    fi
  fi

```

Dequeing with slists:

```

slistDeque( $q$ )  $\stackrel{\text{def}}{=}$ 
  newvar temp, maxVal;
  temp :=  $q$ ;
  maxVal := [ $q + 1$ ];
   $q := [q + 2]$ ;
  dispose(temp, temp + 1, temp + 2);
  return maxVal

```

Disposing a with slist-queue:

```

slistDispose( $q$ )  $\stackrel{\text{def}}{=}$ 
  newvar temp;
  while ( $q \neq \text{null}$ ) do
    temp :=  $q$ ;
     $q := [q + 2]$ ;
    dispose(temp, temp + 1, temp + 2)
  od

```

C. IMPLEMENTATION USING DLISTS

Creating a queue with dlists:

```
dlistcreate  $\stackrel{\text{def}}{=}
q := \mathbf{cons}(null, null, null, null);
\mathbf{return} q$ 
```

Enqueing with dlists:

```
dlistEnque( $q$ )  $\stackrel{\text{def}}{=}
\mathbf{newvar} i, i', j, j', \mathbf{temp};
i := [q]; i' := [q + 1]; j := [q + 2]; j' := [q + 3];
\mathbf{if} (i = j)
  i := \mathbf{cons}(p, v, j, i');
  j' := i
\mathbf{else}
  \mathbf{temp} := \mathbf{cons}(p, v, i, i');
  [i + 3] := \mathbf{temp};
  i := \mathbf{temp}
\mathbf{fi};
[q] := i; [q + 1] := i'; [q + 2] := j; [q + 3] := j'$ 
```

Dequeing with dlists

```

dlistDeque( $q$ )  $\stackrel{\text{def}}{=}$ 
  newvar  $i, i', j, j', \text{max}, \text{maxP}, \text{maxVal}, \text{temp}, \text{tempP}, \text{temp}_0, \text{temp}_1$ ;
   $i := [q]; i' := [q + 1]; j := [q + 2]; j' := [q + 3];$ 
   $\text{max} := i;$ 
   $\text{maxP} := [i];$ 
   $\text{temp} := [i + 2];$ 
  while ( $\text{temp} \neq j$ ) do
     $\text{tempP} := [\text{temp}];$ 
    if ( $\text{tempP} > \text{maxP}$ )
       $\text{max} := \text{tempP};$ 
       $\text{maxP} := \text{tempP}$ 
    else
      skip
    fi;
     $\text{temp} := [\text{temp} + 2]$ 
  od;
   $\text{maxVal} := [\text{max} + 1];$ 
  if ( $\text{max} = i$ )
    if ( $\text{max} = j'$ )
      dispose( $\text{max}$ );
       $i := j; i' := j'$ 
    else
       $\text{temp} := i;$ 
       $i := [i + 2];$ 
       $[i + 3] := i;$ 
      dispose( $\text{max}, \text{max} + 1, \text{max} + 2, \text{max} + 3$ )
    fi
  else
    if ( $\text{max} = j'$ )
       $j' := [j' + 3];$ 
       $[j' + 2] := j;$ 
      dispose( $\text{max}, \text{max} + 1, \text{max} + 2, \text{max} + 3$ )
    else
       $\text{temp}_0 := [\text{max} + 2];$ 
       $\text{temp}_1 := [\text{max} + 3];$ 
       $[\text{temp}_0 + 3] := \text{temp}_1;$ 
       $[\text{temp}_1 + 2] := \text{temp}_0;$ 
      dispose( $\text{max}, \text{max} + 1, \text{max} + 2, \text{max} + 3$ )
    fi;
  fi;
   $[q] := i; [q + 1] := i'; [q + 2] := j; [q + 3] := j'$ 
  return  $\text{maxVal}$ 

```

Disposing with `dlist-queue`:

```

dlistDispose(q)  $\stackrel{\text{def}}{=}
\text{newvar } i, j, \text{temp};
i := [q]; j := [q + 2];
\text{while } (i \neq j) \text{ do}
  \text{temp} := i;
  i := [\text{temp} + 2];
  \text{dispose}(\text{temp}, \text{temp} + 1, \text{temp} + 2, \text{temp} + 3)
\text{od};
\text{dispose}(q, q + 1, q + 2, q + 3)$ 
```

D. HINTS FOR PROOF OF SLISTENQUE

An invariant of the `while` loop in `slistEnque` is

$$\begin{aligned}
& \exists \beta, \beta', p', v'. \\
& \text{slist}(\beta, i, \text{ptemp}) * \text{ptemp} \mapsto (p', v', \text{temp}) * \text{slist}(\beta', \text{temp}, j) \wedge \\
& ((\text{found} \wedge \text{sorted}(\beta \cdot (p', v') \cdot (p, v) \cdot \beta')) \vee p' > p) \wedge \\
& \alpha = \beta \cdot (p', v') \cdot \beta' \wedge \text{sorted}(\alpha)
\end{aligned}$$

Here are some properties about the `slist` predicate that we used in the proof.

$$\begin{aligned}
& \text{slist}(\alpha, i, \text{null}) \wedge i \neq j \Rightarrow \exists \alpha', p, v, k. \alpha = (p, v) \cdot \alpha' \wedge i \mapsto (p, v, k) * \text{slist}(\alpha', k, j) \\
& k \mapsto (p, v, i) * \text{slist}(\alpha, i, j) \Rightarrow \text{slist}((p, v) \cdot \alpha, k, j) \\
& \text{slist}(\alpha, i, k) * k \mapsto (p, v, j) \Rightarrow \text{slist}(\alpha \cdot (p, v), i, j)
\end{aligned}$$

E. HINTS FOR PROOF OF DLISTDEQUE

An invariant of the `while` loop in this program is

$$\begin{aligned}
& \exists i, i', j, j'. q \mapsto (i, i', j, j') * \\
& (\exists \beta, \beta', \beta'', \text{ptemp}, \text{pmax}. \\
& \text{dlist}(\beta, i, i', \text{max}, \text{pmax}) * \\
& \text{dlist}(\beta', \text{max}, \text{pmax}, \text{temp}, \text{ptemp}) * \\
& \text{dlist}(\beta'', \text{temp}, \text{ptemp}, j, j') \wedge \\
& \alpha = \beta \cdot \beta' \cdot \beta'' \wedge \beta' \neq \varepsilon \wedge \text{maxP} = \text{Max}(\beta \cdot \beta') = \text{head}(\beta')).
\end{aligned}$$

The cell containing the “boundary values” for the `slist` need not appear in most of the proof but can be “framed in” via the frame rule.

The following basic properties about the `dlist` predicate have been used in our proof of the `dlist` implementation of priority queues. Some of these are also listed in [Reynolds 2002].

$$\begin{aligned}
& \text{dlist}(\alpha, i, i', j, j') \wedge i \neq j \Rightarrow \text{dlist}(\alpha, i, i', j, j') \wedge \alpha \neq \varepsilon \\
& \text{dlist}(\alpha, i, i', j, j') \wedge \alpha \neq \varepsilon \Rightarrow \\
& \quad \exists p, v, k, \alpha'. \alpha = (p, v) \cdot \alpha' \wedge i \mapsto (p, v, k, i') * \text{dlist}(\alpha', k, i, j, j') \\
& i = i' \wedge j = j' \wedge \text{emp} \Rightarrow \text{dlist}(\varepsilon, i, i', j, j') \\
& \text{dlist}(\alpha, i, i', k, k') * \text{dlist}(\alpha', k, k', j, j') \Rightarrow \text{dlist}(\alpha \cdot \alpha', i, i', j, j')
\end{aligned}$$

ACKNOWLEDGMENTS

The authors wish to thank Carsten Butz and the anonymous referees of previous versions of the work in this paper for helpful comments and insights.

REFERENCES

- BERGER, M., HONDA, K., AND YOSHIDA, N. 2005. A logical analysis of aliasing in imperative higher-order functions. In *Proc. of The 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallinn, Estonia. ACM Press. Accepted for publication.
- BIERING, B. 2004. On the logic of bunched implications and its relation to separation logic, University of Copenhagen.
- BIRKEDAL, L., TORP-SMITH, N., AND REYNOLDS, J. 2004. Local reasoning about a copying garbage collector. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, Venice, Italy, pp. 220–231.
- BIRKEDAL, L., TORP-SMITH, N., AND YANG, H. 2005. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, Chicago, IL, USA, pp. 260–269. IEEE Press.
- BORNAT, R., CALCAGNO, C., AND O'HEARN, P. 2004. Local reasoning, separation and aliasing. In *Proceedings of SPACE 2004*, Venice, Italy.
- BORNAT, R., CALCAGNO, C., O'HEARN, P., AND PARKINSON, M. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, Long Beach, CA, USA. ACM.
- HOARE, C. A. R. 1971. Procedures and parameters: An axiomatic approach. In E. ENGLER (Ed.), *Symp. on Semantics of Algorithmic Languages*, pp. 102–116. Springer-Verlag, Berlin.
- HONDA, K., YOSHIDA, N., AND BERGER, M. 2005. An observationally complete program logic for imperative higher-order functions. In *Proc. of Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, Chicago, IL, USA, pp. 270–279. IEEE Press.
- ISHTIAQ, S. AND O'HEARN, P. W. 2001. BI as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'01)*, London.
- JACOBS, B. 1999. *Categorical Logic and Type Theory*, Volume 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam.
- LAWVERE, F. 1969. Adjointness in foundations. *Dialectica* 23, 3/4, 281–296.
- MITCHELL, J. C. AND PLOTKIN, G. D. 1985. Abstract types have existential type. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL'85)*, New Orleans, LA, USA, pp. 37–51.
- O'HEARN, P. AND PYM, D. J. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (June).
- O'HEARN, P. W. 2004. Resources, concurrency and local reasoning. In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'04)*, Volume 3170 of *LNCS*, London, England, pp. 49–67.
- O'HEARN, P. W., YANG, H., AND REYNOLDS, J. C. 2003. Separation and information hiding (work in progress). Extended version of [O'Hearn et al. 2004].
- O'HEARN, P. W., YANG, H., AND REYNOLDS, J. C. 2004. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'04)*, Venice, Italy, pp. 268–280.
- PARKINSON, M. AND BIERMAN, G. 2005. Separation logic and abstraction. In *Proceedings of the 32nd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'05)*, Long Beach, CA, USA, pp. 247–258.
- PITTS, A. M. 2001. Categorical logic. In S. ABRAMSKY, D. M. GABBAY, AND T. S. E. MAIBAUM (Eds.), *Handbook of Logic in Computer Science, Volume 5: Algebraic and Logical Structures*, Chapter 2. Oxford: Clarendon Press.
- PYM, D. 2002. *The Semantics and Proof Theory of the Logic of Bunched Implications*, Volume 26 of *Applied Logics Series*. Kluwer.

- PYM, D. J. 2004. Errata and remarks for the semantics and proof theory of the logic of bunched implications. Addendum to [Pym 2002]. Available at <http://www.cs.bath.ac.uk/~pym/>.
- PYM, D. J., O'HEARN, P. W., AND YANG, H. 2004. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science* 315, 1, 257–305.
- REDDY, U. 1998. Objects and classes in Algol-like languages. In *Proc. of The Fifth International Workshop on Foundations of Object-Oriented Languages (FOOL'98)*, San Diego, CA, USA.
- REYNOLDS, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, Copenhagen, Denmark, pp. 55–74. IEEE Press.
- SILBERSCHATZ, A. AND GALVIN, P. 1998. *Operating Systems Concepts* (Fifth ed.). World Student Series. Addison-Wesley, Reading, MA, USA.
- YANG, H. 2001. Local reasoning for stateful programs. Ph. D. thesis, University of Illinois, Urbana-Champaign.