

BI-Hyperdoctrines, Higher-Order Separation Logic, and Abstraction

BODIL BIERING, LARS BIRKEDAL, and NOAH TORP-SMITH
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, 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, Maintenance, and Enhancement—*Documentation*; F.3.1 [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

ACM Reference Format:

Biering, B., Birkedal, L., and Torp-Smith, N. 2007. BI-Hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 24 (August 2007), 34 pages. DOI = 10.1145/1275497.1275499 <http://doi.acm.org/10.1145/1275497.1275499>

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

An extended abstract of the present article appeared in the 2005 *Proceedings of the European Symposium on Programming (ESOP)*. The research of L. Birkedal and N. Torp-Smith was partially supported by Danish Natural Science Research Council Grant 51-00-0315 and Danish Technical Research Council Grant 56-00-0309.

Authors’ addresses: B. Biering, L. Birkedal (contact author), N. Torp-Smith, Department of Theoretical Computer Science, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark; email: birkedal@itu.dk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2007 ACM 0164-0925/2007/08-ART24 \$5.00 DOI 10.1145/1275497.1275499 <http://doi.acm.org/10.1145/1275497.1275499>

ACM Transactions on Programming Languages and Systems, Vol. 29, No. 5, Article 24, Publication date: August 2007.

program logic, and its main advantage over traditional program logics is that it facilitates modular reasoning, that is, *local reasoning*, about programs with shared mutable data. Different extensions of core separation logic [Reynolds 2002] have been used to prove correct various algorithms. For example, 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 article 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, i.e., local variables), has been related to *propositional* BI, the logic of bunched implications introduced by O’Hearn and Pym [1999]. In this article 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 [2004, 2002], which has a bunched structure on variable 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 work, 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 expressiveness of higher-order separation logic and argue, with the use of examples, that it is quite expressive. 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. The main formal rule in this development is

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

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 . If a client c has then been proved correct under the assumption that such a predicate exists, it is possible to use the client with any concrete resource invariant \hat{P} and implementations c_1, \dots, c_n .

Moreover, we show that, using universal quantification over predicates, we can prove correct polymorphic operations on polymorphic data types, for example, 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*, that is, entailment between propositions, and it works because the lattice of subobjects of the terminal object in such functor categories forms a BI algebra (a doubly Cartesian closed preorder). In the second sense, one models *proofs*, and this 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 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 2.7). 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 at BI hyperdoctrines work is that predicates are not required to be modeled by subobjects, but 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 2.6.

The rest of the article 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 work. 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, using existential quantification over predicates. In Section 7 we present some simple applications of universal quantification over predicates in program proving. In the last section we discuss related and future work.

This article is the full version of an extended abstract presented at the ESOP 2005 conference. Compared to the conference version, this work 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, e.g., the handbook chapter [Pitts 2001] and 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. Recall that a Heyting algebra is a bi-Cartesian closed partial order, that is, a partial order which, when considered as a category, is Cartesian closed (\top , \wedge , \rightarrow) and has finite coproducts (\perp , \vee).

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, and has 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 $(\forall X)_\Gamma$.

$$A \leq \mathcal{P}(\pi)(A') \quad \text{if and only if} \quad (\exists X)_\Gamma(A) \leq A'$$

$$\mathcal{P}(\pi)(A') \leq A \quad \text{if and only if} \quad A' \leq (\forall X)_\Gamma(A)$$

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

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

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,

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

where \wedge_H, \vee_H , etc., is the Heyting algebra structure on $\mathcal{P}(\llbracket \Gamma \rrbracket)$. Finally, one may show that $\llbracket \varphi[f(x)/y] \rrbracket$ is interpreted by $\mathcal{P}(\llbracket f \rrbracket)(\llbracket \varphi \rrbracket)$, so one should think of $\mathcal{P}(g)$ as the interpretation of substitution.

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 fibers $\mathcal{P}(X)$ are Boolean algebras.

Definition 2.2 (Hyperdoctrine). A (general) *hyperdoctrine* is a first-order hyperdoctrine with the following additional properties: \mathcal{C} is Cartesian closed, and there is an internal Heyting algebra H (for the definition of internal Heyting algebra, see e.g., MacLane and Moerdijk [1994]) and a natural bijection $\Theta_X : \text{Obj}(\mathcal{P}(X)) \simeq \mathcal{C}(X, H)$.

Higher-order intuitionistic predicate logic is first-order intuitionistic predicate logic extended with a type Prop of propositions and with higher types. See, for instance Jacobs [1999] for a formal presentation. 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., 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 $\mathcal{P}(\text{id}, \Theta(\llbracket \psi \rrbracket))(\llbracket \varphi \rrbracket)$. For more details see, for instance 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 fibers $\mathcal{P}(X)$ are Boolean algebras.

Example 2.3 (Canonical Hyperdoctrine Over a Topos). Let \mathcal{E} be a topos. It is well-known that \mathcal{E} models higher-order, intuitionistic predicate logic. In addition, a topos also models full subset types and extensionality (see, e.g., Jacobs [1999]). The interpretation is given 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 Sub_ε . 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 do toposes.

2.2 BI Hyperdoctrines

We now present a straightforward extension of first-order hyperdoctrines which models first and higher-order predicate BI. Recall that a *BI algebra* is a Heyting algebra which has an additional symmetric monoidal closed structure $(I, *, -*)$ [Pym 2002].

Definition 2.4 (Bi Hyperdoctrine).

- A first-order hyperdoctrine \mathcal{P} over \mathcal{C} is a *first-order BI hyperdoctrine* in the case where all the fibers $\mathcal{P}(X)$ are BI algebras and 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 bijection $\Theta_X : \text{Obj}(\mathcal{P}(X)) \simeq \mathcal{C}(X, B)$, natural in X . In other words, $\text{Obj}(\mathcal{P}(-))$ and $\mathcal{C}(-, B)$ are isomorphic as objects in the functor category $\text{Set}^{\mathcal{C}^{\text{op}}}$.

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

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

Our notion of predicate BI should not be confused with the one presented in 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 the first-order hyperdoctrine defined before by

$$\begin{aligned} \llbracket I \rrbracket &= I_B \\ \llbracket \varphi * \psi \rrbracket &= \llbracket \varphi \rrbracket *_B \llbracket \psi \rrbracket \\ \llbracket \varphi -*\psi \rrbracket &= \llbracket \varphi \rrbracket -*_B \llbracket \psi \rrbracket, \end{aligned}$$

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

THEOREM 2.5.

- (1) *The interpretation of first-order predicate BI given previously is sound and complete.*

(2) *The interpretation of higher-order predicate BI given previously is sound and complete.*

PROOF. Soundness is proved by straightforward induction and completeness is proved by forming the Lindenbaum-Tarski algebra over each context Γ of variables, and showing that this gives a first-order BI hyperdoctrine in the first case, and a BI hyperdoctrine in the second. The proof is a simple extension of the proof of the corresponding result for intuitionistic predicate logic given in Jacobs [1999]. \square

Of course, a first-order BI hyperdoctrine is sound for classical BI in the case where all the fibers $\mathcal{P}(X)$ are Boolean BI algebras and all the reindexing functions $\mathcal{P}(f)$ are Boolean BI algebra homomorphisms. The following is a canonical example of a BI hyperdoctrine, which we will use later in Section 3.2 to show that the pointer model is actually an instance of a BI hyperdoctrine.

Example 2.6 (Bi Hyperdoctrine Over a Complete Bi Algebra). Let B be a complete BI algebra, namely, it has all joins and meets. It determines a BI hyperdoctrine over the category **Set** as follows. For each set X , let $\mathcal{P}(X) = B^X$, that is, the set of functions from X to B , be 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 $s, t \in \mathcal{P}(Y)$, that is, $s, t : Y \rightarrow B$, 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).

This example can be stated more generally by replacing **Set** with any Cartesian closed category \mathcal{C} and let B be an internal, complete BI algebra, that is, B is a BI algebra object in \mathcal{C} which is complete as an internal Heyting algebra. 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 \mathcal{J} , $\text{Sub}_{\text{Sh}(\mathcal{C}, \mathcal{J})}(1)$ is a complete BI algebra [Biering 2004; Pym et al. 2004]. For a different kind of example based on realizability, see Biering et al. [2006].

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 2.3). Indeed, this is the reason for introducing the more general BI hyperdoctrines.

THEOREM 2.7. *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, that is, 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

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

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

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

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

$$(\varphi \wedge \psi) -*_Y (\varphi \wedge \psi') \leftrightarrow \varphi \wedge (\psi -*_X \psi') \quad (4)$$

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

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

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

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

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

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

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 preceding the line in Eq. (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) in bottom-up fashion 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 Eqs. (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 $\top *_X \mathbb{I}_X \vdash_X \mathbb{I}_X$, which entails $\top \leftrightarrow \mathbb{I}_X$. 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 slightly strengthen Theorem 2.7. 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

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

is valid. Here $\Gamma \mid \varphi \vdash \psi$ is an alternative notation for $\varphi \vdash_{\Gamma} \psi$ to make the previous formula more readable.

One can then show

PROPOSITION 2.8. *Adding the aforementioned rules for full subset types to our notion of predicate BI yields a logic where 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 2.9. *Any BI hyperdoctrine which satisfies the rules for full subset types is trivial.*

The BI hyperdoctrine S , which we define next and which corresponds to the standard pointer model of separation logic, satisfies all of the preceding 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]. In fact, any BI hyperdoctrine over a complete BI algebra, that is, following the recipe of Example 2.6, has subset types but not necessarily full subset types.

3. SEPARATION LOGIC MODELED BY BI-HYPERDOCTRINES

We briefly recall the standard pointer model of separation logic (for a more thorough presentation, see, e.g., 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

$$\varphi ::= \top \mid \perp \mid t = t \mid t \mapsto t \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi * \varphi \mid \varphi \text{ -* } \varphi \mid \text{emp} \mid \forall x. \varphi \mid \exists x. \varphi.$$

The symbol `emp` is used in separation logic for the unit of BI.

Note that the aforementioned is just another way of defining a signature (i.e., 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 $\mapsto \subseteq \text{Val} \times \text{Val}$.

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`, 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 $H = \llbracket \text{Loc} \rrbracket \rightarrow_{fin} \llbracket \text{Val} \rrbracket$ 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 is the function $\llbracket \text{Val} \rrbracket \times \llbracket \text{Val} \rrbracket \rightarrow P(H)$ given by $h \in \llbracket v_1 \mapsto v_2 \rrbracket$ iff $\text{dom}(h) = \{v_1\}$ and $h(v_1) = v_2$. To define the standard interpretation of terms and formulas, one assumes a partial function $s : \text{Var} \rightarrow_{fin} \llbracket \text{Val} \rrbracket$, 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 $s, h \models \varphi$, where $\text{FV}(\varphi) \subseteq \text{dom}(s)$, as follows:

$$\begin{aligned} s, h \models t_1 = t_2 &\text{ iff } \llbracket t_1 \rrbracket s = \llbracket t_2 \rrbracket s \\ s, h \models t_1 \mapsto t_2 &\text{ iff } \text{dom}(h) = \{\llbracket t_1 \rrbracket s\} \text{ and } h(\llbracket t_1 \rrbracket s) = \llbracket t_2 \rrbracket s \\ s, h \models \text{emp} &\text{ iff } h = \emptyset \\ s, h \models \top &\text{ always} \\ s, h \models \perp &\text{ never} \end{aligned}$$

$$\begin{aligned}
s, h \models \varphi * \psi & \text{ iff there exists } h_1, h_2 \in H \text{ such that } h_1 * h_2 = h, \text{ and} \\
& s, h_1 \models \varphi, \text{ and } s, h_2 \models \psi \\
s, h \models \varphi \text{ } * \psi & \text{ iff for all } h', h' \# h, \text{ and } s, h' \models \varphi \text{ implies } s, h * h' \models \psi \\
s, h \models \varphi \vee \psi & \text{ iff } s, h \models \varphi \text{ or } s, h \models \psi \\
s, h \models \varphi \wedge \psi & \text{ iff } s, h \models \varphi \text{ and } s, h \models \psi \\
s, h \models \varphi \rightarrow \psi & \text{ iff } s, h \models \varphi \text{ implies } s, h \models \psi \\
s, h \models \forall x. \varphi & \text{ iff for all } v \in \llbracket \text{Val} \rrbracket, s[x \mapsto v], h \models \varphi \\
s, h \models \exists x. \varphi & \text{ iff there exists } v \in \llbracket \text{Val} \rrbracket, \text{ such that } s[x \mapsto v], h \models \varphi
\end{aligned}$$

Remark 3.1. The pointer model has a single-sorted signature (the only type is Val), and to get a many-sorted or higher-order version of the pointer model, we add appropriate types to the signature. Variables come with a type $x : X$, and we require that $s(x : X) \in \llbracket X \rrbracket$ for all variables $x \in \text{dom } s$. The last two rules of the forcing relation, become typed.

$$s, h \models \forall x : X. \varphi \text{ iff for all } v \in \llbracket X \rrbracket, s[x \mapsto v], h \models \varphi$$

and similar for the exists rule.

We now show how this pointer model is an instance of a BI-hyperdoctrine of a complete Boolean BI algebra (compare with Example 2.6).

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 $h \in H_\perp$, and $h * h' = \perp$ if h and h' are not disjoint. This defines an ordered, commutative monoid with the empty heap \emptyset 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 $U, V \in \mathcal{P}(H)$):

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

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] and Biering [2004].

Let S be the BI hyperdoctrine induced by the complete Boolean BI algebra $\mathcal{P}(H)$, as in Example 2.6. To show that the interpretation of separation logic in this BI hyperdoctrine exactly corresponds to the standard pointer model presented earlier, we spell out the interpretation of separation logic in S .

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

- $\llbracket x_i : \text{Val} \rrbracket = \pi_i$, where $\pi_i : \text{Val}^n \rightarrow \text{Val}$ is the i th projection;
- $\llbracket n \rrbracket$ is the map $\llbracket n \rrbracket : \llbracket \Gamma \rrbracket \rightarrow 1 \rightarrow \llbracket \text{Val} \rrbracket$ which sends the unique element of the one-point set 1 to $\llbracket n \rrbracket$; and

— $\llbracket t_1 \pm t_2 \rrbracket = \llbracket t_1 \rrbracket \pm \llbracket t_2 \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \text{Val} \rrbracket \times \llbracket \text{Val} \rrbracket \rightarrow \llbracket \text{Val} \rrbracket$, where $\llbracket t_i \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket \text{Val} \rrbracket$, for $i = 1, 2$.

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

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

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

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 aforementioned 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 before. 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 , that is, downwards closed subsets of H , ordered by inclusion. This is a complete BI algebra, as can be verified directly or by 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 2.6. 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 $*$, that makes $(H_{\perp}, *)$ a partially ordered commutative monoid.

Remark 3.3. The correspondences between separation logic and BI hyperdoctrines given previously 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 earlier 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 usefulness of separation logic has been demonstrated in numerous papers before. It has been shown that it can 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 nontrivial 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 [2001] used lists and binary trees as parts of his term language, and Birkedal et al. [2004] introduced expression forms for finite sets and relations. It would seem 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 , namely, 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 sublogic of pure predicates is simply the standard classical higher-order logic of \mathbf{Set} , and thus 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 can also prove the standard properties of these structures, as needed for the proofs presented in the aforementioned papers. 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, as detailed in Section 4.3. For machine verification one thus need only formalize the same exact 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” (e.g., it can have class-many types and function symbols); 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 [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 allow a logical characterization of the classes of assertions, as well as 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 begin straightforward. Recall that closed assertions are interpreted in S as functions from 1 to $\mathcal{P}(H)$, that is, as subsets of H .

In the proofs to follow, 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 $(s, h_0) \models q$. 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 4.2. *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 semantical 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 4.3. *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 before 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 4.4. *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. Suppose (11) holds for the assertion q , and that $h \in \llbracket q \rrbracket$. For any heap h_0 , we must 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 by 11, $h \in \llbracket (q \wedge p_1) * p_2 \rrbracket$. 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 by 11, $h' \in \llbracket q \wedge (p_1 * p_2) \rrbracket$, and in particular, $h' \in \llbracket q \rrbracket$, as desired.

Using Facts 1 and 2, we deduce that $h \in \llbracket q \rrbracket \Rightarrow \text{emp} \in \llbracket q \rrbracket \Rightarrow h_0 \in \llbracket q \rrbracket$. \square

4.3 Predicates via Fixed Points

Consider the following predicate clist taken from Parkinson and Bierman [2005]. It is required to satisfy the 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 (due to Prawitz and Scott, independently). 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. Note that the latter is the corresponding induction principle. 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 it. The programming language is a simple extension of that of standard separation logic with simple call-by-value procedures, and the program logic includes standard rules for these. The logic is for partial correctness and absence of pointer errors.

Programming language. The programming language uses a restricted set of terms of type `Int`, referred to as *expressions*, and uses *Booleans*, which consist of a restricted (heap-independent) set of terms of type `Prop`. E ranges over the set of terms of type `Int`, and B ranges over the Boolean terms. They are generated by the grammar.

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

Formally, Booleans have type `Prop` in our system, but we sometimes write $B : \text{Bool}$ if they can be generated from this grammar (i.e., Boolean expressions are pure assertions). 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, respectively. A context Δ is a pair $\Delta_l; \Delta_p$ of contexts for logical and program variables (i.e., finite maps from variables to types).

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.

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

There are some restrictions on the programs, and a program is called *well formed* if it meets them. 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 modify neither nonlocal variables nor parameters.

The semantics is mostly standard; we specify it formally in the following. Note that the language includes a declaration of new local variables, as well as operations for reading from the heap ($x := [E]$), updating the heap $[E] := E'$, allocating new cells in the heap ($x := \mathbf{cons}(E_1, \dots, E_m)$), and disposing cells in the heap (**dispose**(E)). Functions are first order and call-by-value.

Function specifications. There is a judgment

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

stating that γ is a well-formed *function specification* in context Δ .

Function specifications are used to record assumptions about the functions used in programs. The judgment is given by

$$\frac{\Delta \vdash P:\text{Prop} \quad \Delta \vdash Q:\text{Prop}}{\Delta \vdash \{P\} k(x_1, \dots, x_n) \{Q\}:\text{FSpec}}$$

$$\frac{\Delta \vdash \gamma:\text{FSpec} \quad \Delta \vdash \gamma':\text{FSpec}}{\Delta \vdash \gamma \wedge \gamma':\text{FSpec}}$$

$$\frac{\Delta, x:\tau \vdash \gamma:\text{FSpec}}{\Delta \vdash \lambda x:\tau. \gamma:\text{FSpec}} \text{ where } \lambda \in \{\exists, \forall\}.$$

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 $\Delta \vdash c:\text{comm}$ which asserts that the program c is well formed in the context Δ . We omit the formal definition here.

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

$$\Delta \vdash \delta:\text{Spec}$$

which asserts that δ is a well-formed specification in the context Δ . This judgment is given by

$$\frac{\Delta \vdash c:\text{comm} \quad \Delta \vdash P:\text{Prop} \quad \Delta \vdash Q:\text{Prop}}{\Delta \vdash \{P\} c \{Q\}:\text{Spec}}$$

$$\frac{\Delta \vdash \delta:\text{Spec} \quad \Delta \vdash \delta':\text{Spec}}{\Delta \vdash \delta \wedge \delta':\text{Spec}}$$

$$\frac{\Delta, x:\tau \vdash \delta:\text{Spec}}{\Delta \vdash \lambda x:\tau. \delta:\text{Spec}} \lambda \in \{\exists, \forall\}.$$

The set $\text{FV}(\delta)$ of free variables of a specification δ is the set of free variables in the assertions and 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

$$(\Pi, c, s, h) \Downarrow (s', h'), \tag{12}$$

where Π is a *well-formed semantic function environment*. A semantic function environment maps function names k to pairs (\vec{x}, c) , where \vec{x} is a vector of integer variables and c 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).

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

We write SemFunEnv for the set of all well-formed semantic function environments.

Intuitively, the judgment (12) says that the state (s, h) is transformed to the state (s', h') by the program c . The judgment is given by the clauses in Figure 1. We occasionally use Δ_p 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. In the evaluation of a function call $x = k(E)$, a designated variable ret is used to transfer the return value of the function call via the stack to x .

The configuration (Π, c, s, h) is called *safe* if $(\Pi, c, s, h) \not\Downarrow \text{wrong}$. A configuration may 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 [2005], the properties needed to prove the frame rule, namely, safety monotonicity and the frame property [Yang and O’Hearn 2002], 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 , $(\Pi, c, s, h * h')$ 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 $(\Pi, c, s, h * h') \Downarrow (s', h'')$ implies that there is h_0 disjoint from h' such that $h'' = h_0 * h'$ and $(\Pi, c, s, h) \Downarrow (s', h_0)$.

5.1 Program Logic Judgments

A list Γ of function specifications is called an *environment*. We shall define the judgment

$$\Delta_l; \Delta_p; \Gamma \models \delta : \text{Spec}$$

which states that in the context Δ_l used for logical variables, and context Δ_p used for program variables, given the assumptions about functions recorded in Γ , the specification δ holds. This judgment is defined in several straightforward steps. First, we give the semantics of a triple, relative to a context. The semantics of $\llbracket \Delta_l; \Delta_p \vdash \delta : \text{Spec} \rrbracket$ is a map from $\text{SemFunEnv} \times \llbracket \Delta_l \rrbracket$ to the domain $\{\mathbf{true}, \mathbf{false}\}$, and given by (some obvious type annotations are omitted):

$$\begin{aligned} \llbracket \Delta_l; \Delta_p \vdash \{P\} c \{Q\} \rrbracket(\Pi, s_l) &\text{ iff } \forall s_p \in \llbracket \Delta_p \rrbracket. \forall h \in \llbracket \Delta_l, \Delta_p \vdash P \rrbracket(s_l, s_p). \\ &\quad - (\Pi, c, s_p, h) \text{ is safe, and} \\ &\quad - (\Pi, c, s_p, h) \Downarrow (s'_p, h') \text{ implies} \\ &\quad \quad h' \in \llbracket \Delta \vdash Q \rrbracket(s_l, s'_p) \\ \llbracket \Delta_l; \Delta_p \vdash \delta \wedge \delta' \rrbracket(\Pi, s_l) &\text{ iff } \llbracket \Delta_l; \Delta_p \vdash \delta \rrbracket(\Pi, s_l) \text{ and } \llbracket \Delta_l; \Delta_p \vdash \delta' \rrbracket(\Pi, s_l) \\ \llbracket \Delta_l; \Delta_p \vdash \exists x : \tau. \delta \rrbracket(\Pi, s_l) &\text{ iff } \llbracket \Delta_l x : t; \Delta_p \vdash \delta \rrbracket(\Pi, (s_l)_{[x \mapsto v]}) \text{ for some } v \in \llbracket \tau \rrbracket \\ \llbracket \Delta_l; \Delta_p \vdash \forall x : \tau. \delta \rrbracket(\Pi, s_l) &\text{ iff } \llbracket \Delta_l x : t; \Delta_p \vdash \delta \rrbracket(\Pi, (s_l)_{[x \mapsto v]}) \text{ for all } v \in \llbracket \tau \rrbracket. \end{aligned}$$

We call $\Delta_l; \Delta_p \vdash \delta$ *valid* and write $\Delta_l; \Delta_p \models \delta$ iff $\llbracket \Delta_l; \Delta_p \vdash \delta \rrbracket(\Pi, s_l) = \mathbf{true}$ for all Π and all $s_l \in \llbracket \Delta_l \rrbracket$.

$$\begin{array}{c}
 \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n}{(\Pi, \text{skip}, s, h) \Downarrow (s, h)} \quad \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n}{(\Pi, x := E, s, h) \Downarrow (s_{[x \mapsto n]}, h)} \\
 \\
 \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n}{(\Pi, \text{return } E, s, h) \Downarrow (s_{[\text{ret} \mapsto n]}, h)} \quad \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n \quad n \in \text{dom}(h) \quad h(n) = n'}{(\Pi, x := [E], s, h) \Downarrow (s_{[x \mapsto n']}, h)} \\
 \\
 \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n \quad \llbracket \Delta_p \vdash E':\text{Int} \rrbracket s = n' \quad n \in \text{dom}(h)}{(\Pi, [E] := E', s, h) \Downarrow (s, h_{[n \mapsto n']})} \\
 \\
 \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n \quad n \in \text{dom}(h)}{(\Pi, \text{dispose}(E), s, h) \Downarrow (s, h - \{n\})} \quad \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n \quad n \notin \text{dom}(h)}{(\Pi, \text{dispose}(E), s, h) \Downarrow \text{wrong}} \\
 \\
 \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n \quad n \notin \text{dom}(h)}{(\Pi, x := [E], s, h) \Downarrow \text{wrong}} \quad \frac{\llbracket \Delta_p \vdash E:\text{Int} \rrbracket s = n \quad n \notin \text{dom}(h)}{(\Pi, [E] := E', s, h) \Downarrow \text{wrong}} \\
 \\
 \frac{\{n, n+1, \dots, n+m\} \notin \text{dom}(h) \quad (\llbracket \Delta_p \vdash E_i:\text{Int} \rrbracket s = n_i)_{i=0, \dots, m}}{(\Pi, x := \text{cons}(E_0, \dots, E_m), s, h) \Downarrow (s_{[x \mapsto n]}, h_{[n+i \mapsto n_i]_{i=0, \dots, m}})} \\
 \\
 \frac{(\Pi, c_1, s, h) \Downarrow (s', h') \quad (\Pi, c_2, s', h') \Downarrow (s'', h'')}{(\Pi, c_1; c_2, s, h) \Downarrow (s'', h'')} \\
 \\
 \frac{\llbracket \Delta_p \vdash B:\text{Bool} \rrbracket s = \text{false} \quad (\Pi, c_1, s, h) \Downarrow (s', h')}{(\Pi, \text{if } B \text{ then } c_0 \text{ else } c_1 \text{ fi}) \Downarrow (s', h')} \\
 \\
 \frac{\llbracket \Delta_p \vdash B:\text{Bool} \rrbracket s = \text{true} \quad (\Pi, c_0, s, h) \Downarrow (s', h')}{(\Pi, \text{if } B \text{ then } c_0 \text{ else } c_1 \text{ fi}) \Downarrow (s', h')} \\
 \\
 \frac{\llbracket \Delta_p \vdash B:\text{Bool} \rrbracket s = \text{false}}{(\Pi, \text{while } B \text{ do } c \text{ od}, s, h) \Downarrow (s, h)} \\
 \\
 \frac{\llbracket \Delta_p \vdash B:\text{Bool} \rrbracket s = \text{true} \quad (\Pi, c; \text{while } B \text{ do } c \text{ od}, s, h) \Downarrow (s', h')}{(\Pi, \text{while } B \text{ do } c \text{ od}, s, h) \Downarrow (s', h')} \\
 \\
 \frac{\Pi(k) = ((x_1, \dots, x_m), c_k) \quad (\Pi, c_k, s[x_i \mapsto n_i], h) \Downarrow (s', h')}{(\Pi, x = k(E_1, \dots, E_m), s, h) \Downarrow (s_{[x \mapsto s'(\text{ret})]}, h')} \\
 \\
 \frac{(\Pi, c, s_{[x \mapsto \text{null}]}, h) \Downarrow (s', h') \quad s(x) = v}{(\Pi, \text{newvar } x; c, s, h) \Downarrow (s'_{[x \mapsto v]}, h')} \\
 \\
 \frac{(\Pi \cup (k_1 \mapsto ((x_1, \dots, x_{n_1}), c_1), \dots, k_n \mapsto ((x_1, \dots, x_{n_k}), c_n)), c, s, h) \Downarrow (s', h')}{(\Pi, \text{let } k_1(x_1, \dots, x_{n_1}) = c_1, \dots, k_n(x_1, \dots, x_{n_n}) = c_n \text{ 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 $\Delta_l \vdash t:\tau$ a term such that $(\text{FV}(t) \cup \{x\}) \cap \text{Mod}(\delta) = \emptyset$. Further, let $s_l \in \llbracket \Delta_l \rrbracket$, and Π be well formed. Then,*

$$\llbracket \Delta_l; \Delta_p \vdash \delta[t/x] \rrbracket (\Pi, s_l) \text{ iff } \llbracket \Delta_l; \Delta_p, x:\tau \vdash \delta \rrbracket (\Pi, (s_l)_{[x \mapsto v]}),$$

where $v = \llbracket \Delta_l \vdash t:\tau \rrbracket s_l$.

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

$$\llbracket \Delta_l; \Delta_p \vdash \gamma : \text{FSpec} \rrbracket : \text{SemFunEnv} \times \llbracket \Delta_l \rrbracket \mapsto \{\mathbf{true}, \mathbf{false}\}$$

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

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

where Δ'_p is Δ_p with the x_i 's added (with type `Int`).

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

$$\llbracket \Delta_l; \Delta_p \vdash \Gamma \rrbracket(\Pi, s_l) \text{ iff } \llbracket \Delta_l; \Delta_p \vdash \gamma \rrbracket(\Pi, s_l) \text{ for all } \gamma \in \Gamma$$

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

$$\begin{aligned} \Delta_l; \Delta_p; \Gamma \models \delta \text{ iff for all well-formed } \Pi \text{ and all } s_l \in \llbracket \Delta_l \rrbracket, \\ \llbracket \Delta_l; \Delta_p \vdash \Gamma \rrbracket(\Pi, s_l) \text{ implies } \llbracket \Delta_l; \Delta_p \vdash \delta \rrbracket(\Pi, s_l). \end{aligned}$$

5.2 Inference Rules

We define a judgment

$$\Delta_l; \Delta_p; \Gamma \vdash \delta$$

for deriving valid specifications. The inference rules are given in Figure 2. For brevity, we have omitted obvious rules for conjunctions of specifications and some structural rules for weakening and strengthening of variable contexts. We first explain some of the rules at an intuitive level, and then show soundness.

5.3 Informal Explanation of Rules

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

The rule

$$\frac{\{P\} k(\vec{x}) \{Q\} \in \Gamma}{\Delta_l; \Delta_p; \Gamma \vdash \{P[\vec{E}/\vec{x}]\} y := k(\vec{E}) \{Q[\vec{E}, y/\vec{x}, \text{ret}]\}}, Y \notin \text{FV}(Q) \cup \text{FV}(E)$$

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 those heap cells that are actually manipulated by the commands. For

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

Fig. 2. Program logic.

example, the rule

$$\overline{\Delta_l; \Delta_p; \Gamma \vdash \{\text{emp} \wedge x = m\} x := \text{cons}(\vec{E}) \{x \mapsto \vec{E}[m/x]\}}$$

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

$$\overline{\Delta_l; \Delta_p; \Gamma \vdash \{P\} c \{Q\}} \\
 \overline{\Delta_l; \Delta_p; \Gamma \vdash \{P * P'\} c \{Q * P'\}} \quad \text{Mod}(c) \cap \text{FV}(P') = \text{emp},$$

called the *frame rule*, implies that one can infer a *global* 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.

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

The rule

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

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 rule of consequence is standard, and the rules

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

$$\frac{\Delta_l, x:\tau; \Delta_p; \Gamma \vdash \delta}{\Delta_l; \Delta_p; \Gamma \vdash \forall x:\tau. \delta} x \notin \mathbf{FV}(\Gamma)$$

are straightforward adaptations of standard rules of predicate logic. (Note that by the convention that variables in contexts $\Delta_l; \Delta_p$ are all distinct, $x \notin \mathbf{FV}(\delta)$ in the first rule and $x \notin \mathbf{FV}(\Gamma)$ in the second.) They are used later for reasoning about data abstraction. 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 5.2. *If a specification*

$$\Delta_l; \Delta_p; \Gamma \vdash \delta$$

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

PROOF. By induction. For each rule of form

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

we check that $\Delta'_l; \Delta'_p; \Gamma' \models \delta'$, under the assumption $\Delta_l; \Delta_p; \Gamma \models \delta$. For axioms of the form

$$\frac{}{\Delta_l; \Delta_p; \Gamma \vdash \delta},$$

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

Consider the rule for **skip**:

$$\frac{}{\Delta_l; \Delta_p; \Gamma \vdash \{P\} \mathbf{skip} \{P\}}$$

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

$$\llbracket \Delta_l; \Delta_p \vdash \{P\} \mathbf{skip} \{P\} \rrbracket (\Pi, s_l)$$

for all $s_l \in \llbracket \Delta_l \rrbracket$. Let $s_p \in \llbracket \Delta_p \rrbracket$ and let $h \in \llbracket P \rrbracket (s_l, s_p)$. Then,

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

and clearly, $h \in \llbracket P \rrbracket (s_l, s_p)$, so this rule is sound.

The soundness of the rule for assignment

$$\frac{}{\Delta_l; \Delta_p; \Gamma \vdash \{P[E/x]\} x := E \{P\}}$$

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

Now consider the rule for function calls.

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

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

$$\llbracket \Delta_l; \Delta_p \vdash \{P\} k_i(x_1, \dots, x_{n_i}) \{Q\} \rrbracket (\Pi, s_l),$$

so if $\Pi(k_i) = ((x_1, \dots, x_{n_i}), c_i)$, then $\llbracket \Delta_l; \Delta_p \vdash \{P\} c_i \{Q\} \rrbracket (\Pi, s_l)$. Now, suppose that $s_p \in \llbracket \Delta_p \rrbracket$ and

$$h \in \llbracket P[E_1/x_1 \cdots E_{n_i}/x_{n_i}] \rrbracket (s_l, s_p) = \llbracket P \rrbracket (s_l, (s_p)_{[x_1 \mapsto v_1, \dots, x_{n_i} \mapsto v_{n_i}]}),$$

where $v_j = \llbracket \Delta_l; \Delta_p \vdash E_j; \text{Int} \rrbracket (s_l, s_p)$ and $j \in \{1, \dots, n_i\}$, by the substitution lemma. This means that if

$$(\Pi, c_i, (s_p)_{[x_1 \mapsto v_1, \dots, x_{n_i} \mapsto v_{n_i}]}, h) \Downarrow (s'_p, h'),$$

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

$$s'_p = (s_p)_{[x_1 \mapsto v_1, \dots, x_{n_i} \mapsto v_{n_i}, ret \mapsto s'(ret)]}$$

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

$$(\Pi, y = k_i(E_1, \dots, E_{n_i}), s_p, h) \Downarrow ((s_p)_{[y \mapsto s'_p(ret)]}, h')$$

and thus, the rule holds.

The first rule for existentials is

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

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

$$\llbracket \Delta_l, x:\tau; \Delta_p \vdash \Gamma, \gamma \rrbracket (\Pi, s_l) \text{ implies } \llbracket \Delta_l, x:\tau; \Delta_p \vdash \delta \rrbracket (\Pi, s_l)$$

and let $\llbracket \Delta_l; \Delta_p \vdash \Gamma \rrbracket (\Pi, s_l)$ and $\llbracket \Delta_l; \Delta_p \vdash \exists x:\tau. \gamma \rrbracket (\Pi, s)$. This means that $\llbracket \Delta_l, x:\tau; \Delta_p \vdash \gamma \rrbracket (\Pi, (s_l)_{[x \mapsto v]})$ for some $v \in \llbracket \tau \rrbracket$. Since $x \notin \text{FV}(\Gamma)$, $\llbracket \Delta_l, x:\tau; \Delta_p \vdash$

$\Gamma](\Pi, (s_l)_{[x \mapsto v]})$. This implies $\llbracket \Delta_l, x:\tau; \Delta_p \vdash \delta \rrbracket(\Pi, (s_l)_{[x \mapsto v]})$, and since $x \notin \text{FV}(\delta)$, we have $\llbracket \Delta_l; \Delta_p \vdash \delta \rrbracket(\Pi, s)$.

The other rule for existentials is

$$\frac{\Delta_l; \Delta_p; \Gamma, \exists x:\tau. \gamma \vdash \delta}{\Delta_l, x:\tau; \Delta_p; \Gamma, \gamma \vdash \delta}.$$

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

$$\llbracket \Delta_l; \Delta_p \vdash \Gamma, \exists x:\tau. \gamma \rrbracket(\Pi, s_l) \text{ implies } \llbracket \Delta_l; \Delta_p \vdash \delta \rrbracket(\Pi, s_l)$$

and suppose $\llbracket \Delta_l, x:\tau; \Delta_p \vdash \Gamma, \gamma \rrbracket(\Pi, s_l)$. Since τ is inhabited, this means that

$$\llbracket \Delta_l, x:\tau; \Delta_p \vdash \Gamma, \gamma \rrbracket(\Pi, (s_l)_{[x \mapsto s_l(x)]})$$

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

$$\llbracket \Delta_l, x:\tau; \Delta_p \vdash \Gamma, \exists x:\tau. \gamma \rrbracket(\Pi, s_l)$$

and thus, $\llbracket \Delta_l; \Delta_p \vdash \delta \rrbracket(\Pi, s_l)$, as desired. If τ is an empty type, one can make an easy case analysis on whether x occurs in γ .

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

$$\frac{\Delta_l; \Delta_p; \Gamma \vdash \forall x:\tau. \delta}{\Delta_l, x:\tau; \Delta_p; \Gamma \vdash \delta},$$

suppose that for all well-formed Π and $s_l \in \llbracket \Delta_l \rrbracket$,

$$\llbracket \Delta_l; \Delta_p \vdash \Gamma \rrbracket(\Pi, s_l) \text{ implies } \llbracket \Delta_l; \Delta_p \vdash \forall x:\tau. \delta \rrbracket(\Pi, s_l)$$

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

$$\llbracket \Delta_l; \Delta_p \vdash \Gamma \rrbracket(\Pi, (s'_l - x)),$$

and this implies

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

This means in particular that

$$\llbracket \Delta_l, x:\tau; \Delta_p \vdash \delta \rrbracket(\Pi, (s'_l)_{[x \mapsto s'_l(x)]}),$$

which shows the desired result. \square

5.5 A Derived Rule

There is an important rule *abstract function definition* that is derivable from the rules in Figure 2. The rule is

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

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. First, by the function definition rule,

$$\begin{array}{c}
 \Delta_l; \Delta_p, y; \Gamma \vdash \{P_1[\hat{P}/x]\} c_1 \{Q_1[\hat{P}/x]\} \\
 \Delta_l; \Delta_p; \Gamma, \{P_1[\hat{P}/x]\} k_1(y) \{Q_1[\hat{P}/x]\} \vdash \{P\} c \{Q\} \\
 \hline
 \Delta_l; \Delta_p; \Gamma \vdash \{P\} \mathbf{let} k_1(y) = c_1 \mathbf{in} c \{Q\}
 \end{array}$$

The rule for existentials gives us

$$\frac{\Delta_l; \Delta_p; \Gamma, \exists x:\tau. \{P_1\} k_1(y) \{Q_1\} \vdash \{P\} c \{Q\}}{\Delta_l, x:\tau; \Delta_p; \Gamma, \{P_1\} k_1(y) \{Q_1\} \vdash \{P\} c \{Q\}},$$

so we need to establish

$$\Delta_l; \Delta_p; \Gamma, \{P_1[\hat{P}/x]\} k_1(y) \{Q_1[\hat{P}/x]\} \vdash \{P\} c \{Q\},$$

given

$$\Delta_l; x:\tau; \Delta_p; \Gamma, \{P_1\} k_1(y) \{Q_1\} \vdash \{P\} c \{Q\}.$$

But this follows from a substitution lemma, 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 using data abstraction. The example involves two implementations of a priority queue, and the intention is, of course, that the 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].

6.1 Reasoning using Abstract Priority Queues

Priority queues are used frequently in programming, for example, in scheduling algorithms for processes in operating systems [Silberschatz and Galvin 1998]. These 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. Some operations and relations on such queues are needed:

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

We assume a base type PriQ , whose values are priority queues. 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, we'll use 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 does *not* say anything about how Q is represented. Note that the type of repr is $(\text{Int} \times \text{PriQ}) \Rightarrow \text{Prop}$, a type of predicate.

This will be used as an abstract predicate in our proofs (thus playing 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

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

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

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

Note that ownership of the cell pointed to by v transfers from the client to the module.

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

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

Note that the ownership of the dequeued cell is now transferred back to the client.

Disposing a queue. The specification for disposing a queue is

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

We can now show a specification for a client program c using the abstract specification of the priority queue.

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

Observe that a client may use multiple instances of priority queues, unlike in O’Hearn et al. [2004], which only considers static modularity.

6.2 Implementations of Priority Queues

One can implement priority queues in many ways. We have verified two implementations: one using sorted linked lists and the other doubly-linked lists. The implementations and proofs make use of some of the properties shown by Reynolds [2002], are fairly standard, and thus omitted. Of course, a client may use either of the two implementations, and we expect that the behavior of a client is independent of which implementation of priority queues is used. The simple model we have devised in this article cannot be used to prove this formally; for that we would need a relationally parametric model.

7. SOME APPLICATIONS OF UNIVERSAL QUANTIFICATION

In the previous section we saw how to use existential quantification over predicates to reason using data abstraction. In this section we present two examples of how to apply *universal* quantification over predicates (in addition to the examples involving fixed points in Section 4.3).

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 metalevel*. 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 yield different versions of the list, with different amounts of data stored in the list. If $P \equiv \text{emp}$, then plain values are stored

(i.e., no ownership transfer to the queue module in O’Hearn et al. [2004]), and if $P \equiv x \mapsto -, -$, then the addresses of cells are stored in the queue (i.e., 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 uniformly, independently of how much heap storage is used for each element of the list.

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, for example, program the usual **map** function on lists, and provide a single parametric correctness proof for it. See our joint paper 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], that is, uniquely describes the heaps satisfying q (e.g., if q is $\exists \beta : \text{seqInt}. \text{list}(\text{emp}, \beta, i)$, then c may delete some elements from the list in the input heap h).

Instead, we may use the 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 that c terminates in heap h' . Then $h' = h_1 * h$, for some h_1 . In other words, the heap h is *invariant* under the execution of c , as intended.

8. RELATED AND FUTURE WORK

We have 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 semantics for BI given by 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 particularly separation logic, were illustrated. Specifically, we introduced higher-order separation logic, and gave sound reasoning principles for data abstraction in the presence of mutable pointer structures, using existential quantification over predicates.

The idea of using data abstraction to reason about complex data structures goes back to Hoare [1972], who introduced the idea of using abstraction functions, namely, functions that map object structures to values of an abstract domain. Modifications of object structures can then be described in terms of their abstract values, which makes implementation-independent specifications possible. Hoare's idea has been extended and applied in a variety of contexts (see, e.g., Leavans [1988], Liskow and Guttag [1986], Leino [1995], Müller [2002], Leino and Müller [2004, 2006], Barnett et al. [2003], Barnett and Naumann [2004], and Naumann and Barnett [2006]). In several of these papers, abstraction functions are captured via so-called *model fields* and the data abstraction technique is combined with ownership-based invariants to deal with mutable pointer structures. The model fields correspond very closely to (some of) the arguments of our existentially quantified propositions, for example, the PriQ argument of the repr predicate in Section 6.1. We believe that our approach to data abstraction using standard higher-order existential quantification gives a particularly clear account of data abstraction by employing standard logical notions, rather than introducing additional new logical concepts. One could argue, however, that our logical approach to data abstraction comes at the price that we move to higher-order logic, which poses difficulties for tool support. More research is needed to evaluate how much of an issue this is in practice. More research is also needed to evaluate how useful our approach is for practical verification; the examples we have considered in this article merely serve to show that the approach is viable. In particular, it would be interesting to extend the presented specification logic to richer programming languages with more of the features found in modern programming languages. We are currently investigating extensions to higher-order programming languages [Nanevski et al. 2006; Krishnaswami et al. 2006] and hope in the future to extend it to object-oriented languages.

In other work, we extended separation logic to a higher-order language [Birkedal et al. 2005], a version of Algol with immutable variables and a first-order heap. The system in *loc. cit.* doesn't distinguish the type system

from the specification language: Command types can contain pre- and postconditions written in separation logic in a fashion similar to refinement types. The assertion logic is first order (i.e., no quantification over propositions) but includes a powerful kind of hypothetical frame rule, extending the second-order frame rule of O’Hearn et al. [2004] to higher order. We have worked out a simple translation from hypothetical frame rules to higher-order separation logic, which suggests that all uses of hypothetical frame rules can be represented in higher-order separation logic, but more work is needed to properly analyze this conjecture.

As mentioned in Section 6.2, we expect that one should be able to show that clients cannot detect any differences between different implementations of abstract data types. Such representation independence (i.e., relational parametricity) results have been shown for a Java-like language and for a semantic notion of confinement by Banerjee and Naumann [2005a, b]. It is quite challenging to develop relationally parametric models for separation logic, even for a simple first-order programming language like the one considered in this article. The reason is that standard models of separation logic allow location identities to be observed in the model. This means, in particular, that allocation of new heap cells is not parametric because the location identity of the allocated cell can be observed in the model. In very recent work, the second author and Yang did, however, succeed in defining a relationally parametric model of separation logic [Birkedal and Yang 2006]. However, the model in *loc. cit.* was only developed for a first-order logic with hypothetical frame rules, and thus it is still an open question how to devise a relationally parametric model for higher-order separation logic.

APPENDIX

A. PROOF OF PROPOSITION 2.8

For a term t with $y:Y \vdash t(y):X$, we add the 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 that $x:X \mid \exists_o. \psi[o(y)/x] \vdash \varphi \wedge \psi$. 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

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

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

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

namely,

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

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:

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

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 I, and ψ' be \top ; this gives $I \wedge (I * \top) \dashv\vdash (I \wedge I) * \top$, that is, $I \dashv\vdash \top$, which in return yields $\varphi \wedge (\top * \psi') \dashv\vdash (\varphi \wedge \top) * \psi'$, namely, $\varphi \wedge \psi' \dashv\vdash \varphi * \psi'$. \square

ACKNOWLEDGMENTS

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

REFERENCES

- BANERJEE, A. AND NAUMANN, D. 2005a. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM* 52, 6, 894–960.
- BANERJEE, A. AND NAUMANN, D. 2005b. State based ownership, reentrance and encapsulation. In *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 3586. Springer, 387–411.
- BARNETT, M., DELINE, R., FÄHNDRICH, M., LEINO, K., AND SCHULTE, W. 2003. Verification of object-oriented programs with invariants. In *Proceedings of the Conference on Formal Techniques for Java-Like Programs*.
- BARNETT, M. AND NAUMANN, D. 2004. Friends need a bit more: Maintaining invariants over shared shate. In *Proceedings of the Conference on Mathematics of Program Construction (MPC)*.
- BIERING, B. 2004. On the logic of bunched implications and its relation to separation logic. M.S. thesis, University of Copenhagen.
- BIERING, B., BIRKEDAL, L., BUTZ, C., HYLAND, J., VAN OOSTEN, J., AND STREICHER, P. R. T. 2006. Notes on the dialectica topos. To appear.
- BIRKEDAL, L., TORP-SMITH, N., AND YANG, H. 2005. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Press, Chicago, IL, 260–269.
- BIRKEDAL, L., TORP-SMITH, N., AND REYNOLDS, J. 2004. Local reasoning about a copying garbage collector. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Venice, Italy). 220–231.
- 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)* (Long Beach, CA). ACM, New York.
- BORNAT, R., CALCAGNO, C., AND O'HEARN, P. 2004. Local reasoning, separation and aliasing. In *Proceedings of the SPACE* (Venice, Italy).
- BIRKEDAL, L. AND YANG, H. 2006. Relational parametricity and separation logic. To appear.
- HOARE, C. A. R. 1972. Proof of correctness of data representations. *Acta Inf.* 1, 271–281.
- HOARE, C. A. R. 1971. Procedures and parameters: An axiomatic approach. In *Proceedings of the Symposium on Semantics of Algorithmic Languages*, E. Engler, ed. Springer, 102–116.
- 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)* (London).
- JACOBS, B. 1999. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics, vol. 141. North-Holland, Amsterdam, The Netherlands.
- KRISHNASWAMI, N., BIRKEDAL, L., ALDRICH, J., AND REYNOLDS, J. 2006. Idealized ML and its separation logic. To appear.

- LAWVERE, F. 1969. Adjointness in foundations. *Dialectica* 23, 3-4, 281–296.
- LEAVANS, G. 1988. Verifying object-oriented programs that use subtypes. Ph.D. thesis, MIT. Published as MIT/LCS/TR-439 in February 1989.
- LEINO, K. 1995. Toward reliable modular programs. Ph.D. thesis, California Institute of Technology.
- LEINO, K. R. M. AND MÜLLER, P. 2006. A verification methodology for model fields. In *Proceedings of the European Symposium on Programming (ESOP)*, P. Sestoft, ed. Lecture Notes in Computer Science, vol. 3924. Springer, 115–130.
- LEINO, K. AND MÜLLER, P. 2004. Object invariants in dynamic contexts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.
- LISKOW, B. AND GUTTAG, J. 1986. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA.
- MACLANE, S. AND MOERDLJK, I. 1994. *Sheaves in Geometry and Logic*. Universitext. Springer, New York. A first introduction to topos theory, Corrected reprint of the 1992 edition.
- 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)* (New Orleans, LA), 37–51.
- MÜLLER, P. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, vol. 2262, Springer.
- NANEVSKI, A., AHMED, A., MORRISETT, G., AND BIRKEDAL, L. 2006. Abstract predicates and mutable ADTs in Hoare type theory. Tech. Rep. TR-14-06, Harvard University.
- NAUMANN, D. AND BARNETT, M. 2006. Towards imperative modules: Reasoning about invariants and mutable state. *Theor. Comput. Sci.* 365, 143–168.
- O’HEARN, P. W. 2004. Resources, concurrency and local reasoning. In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR)* (London). Lecture Notes in Computer Science, vol. 3170. Springer, 49–67.
- 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)* (Venice, Italy). 268–280.
- 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. AND PYM, D. J. 1999. The logic of bunched implications. *Bull. Symb. Logic* 5, 2 (Jun.).
- 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)* (Long Beach, CA). 247–258.
- PITTS, A. M. 2001. Categorical logic. In *Handbook of Logic in Computer Science, volume 5: Algebraic and Logical Structures*, S. Abramsky et al., eds. Clarendon Press, Oxford, UK. Chapter 2.
- PYM, D. J. 2004. Errata and remarks for the semantics and proof theory of the logic of bunched implications. Addendum to Pym [2002]. <http://www.cs.bath.ac.uk/~pym/>.
- PYM, D. 2002. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logics Series, vol. 26. Kluwer.
- PYM, D. J., O’HEARN, P. W., AND YANG, H. 2004. Possible worlds and resources: The semantics of BI. *Theor. Comput. Sci.* 315, 1, 257–305.
- REYNOLDS, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)* (Copenhagen, Denmark). IEEE Press 55–74.
- SILBERSCHATZ, A. AND GALVIN, P. 1998. *Operating Systems Concepts*, 5th ed. World Student Series. Addison-Wesley, Reading, MA.
- YANG, H. 2001. Local reasoning for stateful programs. Ph.D. thesis, University of Illinois, Urbana-Champaign.
- YANG, H. AND O’HEARN, P. 2002. A semantic basis for local reasoning. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)* (Grenoble, France). Springer, 402–416.

Received August 2005; revised November 2006; accepted February 2007