BI Hyperdoctrines, Higher-Order Separation Logic, and Abstraction

Bodil Biering
Lars Birkedal
Noah Torp-Smith

IT University Technical Report Series ITU-TR-2005-69

July 2005

ISSN 1600–6100

ISBN 87-7949-099-9

EAN Barcode 9788779490994

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaardsvej 7
DK-2300 Copenhagen S
Denmark

Telephone:	+45 7218 5000
Telefax:	+45 7218 5001
Web	www.itu.dk

BI Hyperdoctrines, Higher-Order Separation Logic, and Abstraction

Bodil Biering, Lars Birkedal*, and Noah Torp-Smith*

Department of Theoretical Computer Science, IT University of Copenhagen
{biering, birkedal, noah}@itu.dk

Abstract. We present a simple extension of separation logic which makes the specification language *higher-order*, in the sense that quantification over predicates and higher types is possible. The fact that this is a useful extension is illustrated via examples; specifically we demonstrate that existential and universal quantification correspond to abstract data types and parametric data types, respectively. We also illustrate that the semantics we give is an instance of a general notion, namely that of a *BI hyperdoctrine*, of models for predicate BI.

1 Introduction

Variants of the recent formalism of *separation logic* [26, 11] have been used to prove correct many interesting algorithms involving pointers, both in sequential and concurrent settings [16, 28, 4]. It is a Hoare-style program logic, and its main advantage over traditional program logics is that it facilitates *local reasoning*.

The force of separation logic comes from both its language of assertions – which is a variant of propositional BI [22] – and its language of specifications, or Hoare triples. In the present paper, we extend both of these. First, we introduce an assertion language which is a variant of *higher-order predicate* BI. The extension from the traditional assertion language of separation logic is simply that we allow function types and have a type Prop of proposition, and we allow quantification over variables of this type. Thus the assertion language is higher-order, in the usual sense that it allows quantification over predicates. Next, we give a specification logic for a simple second-order programming language in which it is also possible to quantify over variables of any type. We provide models for both the new assertion language and the specification logic, and provide inference rules for deriving valid specifications. It turns out that it is technically straightforward to do so; we believe that this serves to emphasize that our notion of higher-order predicate BI is the correct one for separation logic.

Next we consider the effectiveness of higher-order separation logic and argue, with the use of several examples, that it is quite effective. In particular, we show that higher-order separation logic can be used in a natural way to model data abstraction, via existential quantification over predicates corresponding to abstract resource invariants; we do so by means of a worked example, which involves two

* Partially supported by Danish Technical Research Council Grant 56-00-0309.

implementations of abstract priority queues. This way of reasoning about data abstraction is more natural than the recently suggested abstract predicates of Parkinson and Bierman [20], for providing modular proofs of programs using abstract data types. Moreover, we show that, using universal quantification over predicates, we may prove correct polymorphic operations on polymorphic data types, e.g., reversing a list of elements described by some 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.

Having introduced higher-order separation logic and a semantics thereof, we show that our semantics is in fact an instance of a general concept. Part of the pointer model of separation logic, namely that given by heaps (but not stacks) has been related to *propositional* BI. We show how the correspondence may be taken further, in the sense that our notion of *predicate* BI corresponds to all of the pointer model (including stacks). We also introduce the notion of a *BI hyperdoctrine*, a simple extension of Lawvere's notion of a hyperdoctrine [13] and show that it soundly models predicate BI. We also show that our semantics is an instance of this general semantics.

It should be noted that we consider a different notion of higher-order predicate BI than that of [22, 23], which has a BI structure on contexts. However, we believe that our notion of higher-order predicate BI with its class of BI hyperdoctrine models is the right one for separation logic (Pym aimed to model mulitiplicative quantifiers; separation logic only uses additive quantifiers); the correspondence mentioned above serves to illustrate this claim. We also give some applications of the extension of the assertion language of separation language to higher-order.

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 [24]. Such functor categories can be used to model *propositional* BI in two different senses: In the first sense, one models *provability*, entailment between propositions, and it works because the lattice of subobjects of the terminal object in such functor categories form a BI algebra (a doubly cartesian closed preorder). In the second sense, one models *proofs*, and it works because the whole functor category is doubly cartesian closed. Here we seek models of provability of *predicate* BI. Since the considered functor categories are toposes and hence model higher-order predicate logic, one might think that a straightforward extension is possible. But, alas, it is not the case. In general, for this to work, *every* lattice of subobjects (for any object, not only for the terminal object) should be a BI algebra and, moreover, to model substitution correctly the BI algebra structure should be preserved by pulling back along any morphism. We show that this can only be the case if the BI algebra structure is trivial, that is, coincides with the cartesian structure (see Theorem 3). Our theorem holds for any topos, not just for the functor categories considered earlier. Hence we need to consider a wider class of models for predicate BI than just toposes and this justifies the notion of a BI hyperdoctrine. The intuitive reason that BI hyperdoctrines work, is that predicates are not required to be modeled by subobjects, they can be something more general. Another important point of BI hyperdoctrines is that they are easy to come by: given any complete BI algebra B, we can define a canonical BI hyperdoctrine in which predicates are modeled as B-valued functions; we explain this in detail in Example 2.

The rest of the paper is organized as follows. In Section 2 we give the syntax and semantics of both the assertion language and the specification language of higher-order separation logic. This includes the definition of a simple programming language which has heap manipulation constructs and simple procedures, and its operational semantics. In Section 3, we give rules for deriving sound specifications, which constitue a specification logic for our programming language. We explain these rules at an intuitive level and show soundness of them with respect to the semantics we gave in Section 2. In Section 4 we give several examples which illustrate that higher-order separation logic is indeed useful. In particular, we show how existential quantification may be used to reason about data abstraction inasmuch as it can be used to show representation independence of two implementations of an abstract priority queue. We also illustrate how universal quantification of the specification language can be used to model polymorphic types. In Section 5, we first recall Lawvere's notion of a hyperdoctrine [13] and straightforwardly extend it to the notion of BI hyperdoctrines, and we show that this soundly models predicate BI and that the semantics of assertions we have given is an instance of a BI hyperdoctrine. In Section 6 we discuss applications of the extension of the assertion language to higher-order, and in particular we show how one can use the higher-order logic to give logical characterizations of interesting classes of assertions. In the last sections we give pointers to related and future work, and conclude.

2 Syntax and Semantics

We give syntax for several judgements, and semantics for most of them.

2.1 Syntax

Types are generated by the grammar

$$\tau ::= \mathsf{Int} \mid \mathsf{Prop} \mid \tau \times \tau \mid \tau \to \tau \mid \cdots .$$

The "$\cdots$" is used to indicate that we may add more base types to the system without complications. For now, the core system is the one indicated.

Terms There is a judgment $\Delta \vdash t{:}\tau$, where Δ is a list of variable assignments $x{:}\tau$ to distinct variables, and the judgment states that the free variables of t are included in Δ, and that the term t is well-formed and of type τ in Δ. The judgment is defined by

$$\vdash n{:}\mathsf{Int}$$

$$\Delta, x{:}\tau \vdash x{:}\tau$$

$$\frac{\Delta \vdash t{:}\mathsf{Int} \quad \Delta \vdash t'{:}\mathsf{Int}}{\Delta \vdash t \otimes t'{:}\mathsf{Int}} \text{ where } \otimes \in \{+, -, \times\}$$

$$\frac{\Delta \vdash t{:}\mathsf{Int} \quad \Delta \vdash t'{:}\mathsf{Int}}{\Delta \vdash t \lhd t'{:}\mathsf{Prop}} \text{ where } \lhd \in \{=, \leq\}$$

$$\vdash \top{:}\mathsf{Prop}$$

$$\vdash \bot{:}\mathsf{Prop}$$

$$\vdash \mathsf{emp}{:}\mathsf{Prop}$$

$$\frac{\Delta \vdash t{:}\mathsf{Int} \quad \Delta \vdash t'{:}\mathsf{Int}}{\Delta \vdash t \mapsto t'{:}\mathsf{Prop}}$$

$$\frac{\Delta \vdash \varphi{:}\mathsf{Prop} \quad \Delta \vdash \varphi'{:}\mathsf{Prop}}{\Delta \vdash \varphi \diamond \varphi'{:}\mathsf{Prop}} \text{ where } \diamond \in \{\vee, \wedge, \rightarrow, -\!\!*, *\}$$

$$\frac{\Delta, x{:}\tau \vdash \varphi{:}\mathsf{Prop}}{\Delta \vdash \natural x.\ \varphi{:}\mathsf{Prop}} \text{ where } \natural \in \{\forall, \exists\}$$

$$\frac{\Delta, x{:}\tau \vdash t{:}\tau'}{\Delta \vdash (\lambda x{:}\tau.\ t){:}\tau \rightarrow \tau'}$$

$$\frac{\Delta \vdash t{:}\tau' \rightarrow \tau \quad \Delta \vdash t'{:}\tau'}{\Delta \vdash tt'{:}\tau}$$

We also allow weakening and exchange in contexts. We assume that all the contexts are well-formed, so, for example, in the second rule above, we implicitly assume that $x \notin \Delta$.

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

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

Formally, booleans have type Prop in our system, but we will sometimes write $B : \mathsf{Bool}$ if they can be generated from this grammar.

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

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

There are some restrictions on the programs, and we call a program *well-formed* if it meets these restrictions. We could express the restrictions formally with a bunch of auxiliary grammars, but we will refrain from that here. The restrictions include:

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

Function Specifications There is a judgment

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

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

$$\frac{\Delta \vdash P{:}\mathsf{Prop} \quad \Delta \vdash Q{:}\mathsf{Prop}}{\Delta \vdash \{P\}\ k\ \{Q\}{:}\mathsf{FSpec}}$$

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

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

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

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

$$\Delta; \Pi \vdash c{:}\mathsf{comm}, \tag{1}$$

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

$$\Pi \ \mathbf{ok} \text{ iff } \forall (x, c) \in \mathit{cod}(\Pi).\ \text{Modifies}(c) = \varnothing.$$

We omit the definition of the judgment (1) here.

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

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

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

$$\frac{\Delta; \Pi \vdash c{:}\mathsf{comm} \quad \Delta \vdash P{:}\mathsf{Prop} \quad \Delta \vdash Q{:}\mathsf{Prop}}{\Delta; \Pi \vdash \{P\}\ c\ \{Q\}{:}\mathsf{Spec}}$$

$$\frac{\Delta; \Pi \vdash \delta{:}\mathsf{Spec} \quad \Delta; \Pi \vdash \delta'{:}\mathsf{Spec}}{\Delta; \Pi \vdash \delta \wedge \delta'{:}\mathsf{Spec}}$$

$$\frac{\Delta, x{:}\tau; \Pi \vdash \delta{:}\mathsf{Spec}}{\Delta; \Pi \vdash \natural x{:}\tau.\ \delta{:}\mathsf{Spec}}\ \natural \in \{\exists, \forall\}$$

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

2.2 Semantics

Semantics of Types The semantics of types is a set, and it is given by

$$\begin{aligned} [\![\mathsf{Prop}]\!] &= \mathcal{P}(H) \\ [\![\mathsf{Int}]\!] &= \mathbb{Z} \\ [\![\tau \times \tau']\!] &= [\![\tau]\!] \times [\![\tau']\!] \\ [\![\tau \to \tau']\!] &= [\![\tau]\!] \Rightarrow [\![\tau']\!] \end{aligned}$$

When more base types are added, one of course has to specify the semantics of them.

Semantics of Terms The semantics $[\![\Delta \vdash t{:}\tau]\!]$ is a map

$$[\![\Delta]\!] \xrightarrow{[\![t]\!]} [\![\tau]\!],$$

where $[\![\Delta]\!]$ is the product of the τ_i for $x_i{:}\tau_i \in \Delta$. Although elements of $[\![\Delta]\!]$ are tuples $(v_1, \ldots, v_n)$, we will treat them as maps from variables in Δ to values. Hence, if $\Delta = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ and $\eta = (v_1, \ldots, v_n) \in [\![\Delta]\!]$, we will write $\eta(x_i)$ instead of $\pi_i(\eta)$ for the value v_i. For this correspondence, we will also use other notation. If $x{:}\tau \notin \Delta$, $v \in [\![\tau]\!]$, and $\eta = (v_1, \ldots, v_n) \in [\![\Delta]\!]$, we write $\eta_{[x \to v]}$ for the tuple $(v_1, \ldots, v_n, v) \in [\![\Delta, x{:}\tau]\!]$. We also use the notation for updates in η. If $x_i{:}\tau_i \in \Delta$, $v_i \in [\![\tau_i]\!]$, and $\eta = (v_1, \ldots, v_n) \in [\![\Delta]\!]$, we write $\eta_{[x_i \to v_i]}$ for the tuple $(v_1, \ldots, v_i, \ldots, v_n, v) \in [\![\Delta]\!]$. Finally we use the notation $\eta - x$ to "remove a component from a tuple": If $x_i{:}\tau_i \in \Delta = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ and $\eta = (v_1, \ldots, v_i, \ldots, v_n) \in [\![\Delta]\!]$, we write $\eta - x_i$ for the tuple $(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \in [\![x_1{:}\tau_1, \ldots x_{i-1}{:}\tau_{i-1}, x_{i+1}{:}\tau_{i+1}, \ldots x_n{:}\tau_n]\!]$.

The semantics is defined in a standard way; we give the most important clauses here.

$$[\![\Delta \vdash n{:}\mathsf{Int}]\!]\eta = n$$

$$[\![\Delta, x{:}\tau \vdash x{:}\tau]\!]\eta = \eta(x)$$

$$[\![\Delta \vdash t \lhd t'{:}\mathsf{Prop}]\!]\eta = \begin{cases} H \text{ if } [\![\Delta \vdash t{:}\mathsf{Int}]\!]\eta \lhd [\![\Delta \vdash t'{:}\mathsf{Int}]\!]\eta \\ \varnothing \text{ if } \neg([\![\Delta \vdash t{:}\mathsf{Int}]\!]\eta \lhd [\![\Delta \vdash t'{:}\mathsf{Int}]\!]\eta) \end{cases} \text{, where } \lhd \in \{=, \leq\}$$

$$[\![\Delta \vdash \mathsf{emp}{:}\mathsf{Prop}]\!]\eta = \{[]\}$$

$$[\![\Delta \vdash \varphi * \varphi'{:}\mathsf{Prop}]\!]\eta = \left\{ h \mid \begin{array}{l} \exists h_0, h_1.\ h_0 \# h_1 \wedge h = h_0 \cup h_1 \wedge \\ h_0 \in [\![\Delta \vdash \varphi]\!]\eta \wedge h_1 \in [\![\Delta \vdash \varphi']\!]\eta \end{array} \right\}$$

$$[\![\Delta \vdash \varphi \mathrel{-\!\!*} \varphi'{:}\mathsf{Prop}]\!]\eta = \bigcup W, \text{ with } W * [\![\Delta \vdash \varphi{:}\mathsf{Prop}]\!]\eta \subseteq [\![\Delta \vdash \varphi'{:}\mathsf{Prop}]\!]\eta$$

$$[\![\Delta \vdash \exists x{:}\tau.\ \varphi{:}\mathsf{Prop}]\!]\eta = \textstyle\bigcup_{v \in [\![\tau]\!]} [\![\Delta, x{:}\tau \vdash \varphi{:}\mathsf{Prop}]\!]\eta_{[x \to v]}$$

$$[\![\Delta \vdash \forall x{:}\tau.\ \varphi{:}\mathsf{Prop}]\!]\eta = \textstyle\bigcap_{v \in [\![\tau]\!]} [\![\Delta, x{:}\tau \vdash \varphi{:}\mathsf{Prop}]\!]\eta_{[x \to v]}$$

This semantics uses the fact that $\mathcal{P}(H)$ is a boolean BI-agebra [3] to give the semantics of the BI connectives $(*, \mathrel{-\!\!*})$ of the logic. See Sec. 5.3 for more details.

The expected substitution lemma holds:

Lemma 1. *Suppose $\Delta \vdash t{:}\tau$ and $\Delta, x{:}\tau \vdash t'{:}\tau'$. Then for all $\eta \in [\![\Delta]\!]$,*

$$[\![\Delta \vdash t'[t/x]]\!]\eta = [\![\Delta, x{:}\tau \vdash t']\!]\eta_{[x \to v]},$$

where $v = [\![\Delta \vdash t{:}\tau]\!]\eta$.

Remark 1. The traditional way of giving semantics of assertions is via a forcing relation

$$s, h \Vdash \varphi, \tag{2}$$

which asserts that the assertion φ holds in the state (s, h) (where the free variables of φ are included in the domain of the stack s). The grammar of assertions and the definition of the forcing relation (2) is completely standard, and different variants may be found in numerous papers, *e.g.*, [4, 17, 8]. There is a tight connection between these two forms of semantics, since it is not hard to see that if φ has free variables $x_1, \ldots, x_n$, and if $v_1, \ldots, v_n$ are values of the corresponding types (in “traditional” separation logic, there is only one type), then

$$\begin{array}{c} h \in [\![\varphi]\!](v_1, \ldots, v_n) \text{ in our semantics} \\ \text{iff} \\ [x_1 \to v_1, \ldots, x_n \to v_n], h \Vdash \varphi \text{ in the traditional semantics.} \end{array}$$

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

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

The proviso here is that $\eta \in \Delta$ for some Δ in which $\Delta; \Pi \vdash c{:}\mathsf{comm}$ holds, and it intuitively says that the state (η, h) is transformed to the state (η', h') by the program c. The judgment is the same as in [20] and it is given by the clauses in Fig. 1. We have occasionally used Δ for the domain of η below. For example, in the second rule (for assignment), the precondition is that $[\![\Delta \vdash E{:}\mathsf{Int}]\!]\eta = n$. What is meant is just that E is a term of type Int in any context that contains the variables in η.

We say that (Π, c, η, h) is *safe* if $(\Pi, c, \eta, h) \not\Downarrow$ **wrong**. A configuration may either terminate in a state (η', h'), *diverge*, or go wrong.

Note that, since this semantics is the same as the operational semantics of the language in [20], the properties needed to prove the frame rule, namely safety monotonicity and the frame property, are valid for all programs of the language. As a reminder, we state these properties here.

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

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

2.3 Program Logic Judgments

A list Γ of function specifications where the function names are distinct, is called an *environment*. We will give the definition of the jugdment

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

which states that in the context Δ, given the assumptions about functions recorded in Γ, the specification δ holds. This judgment is defined in several straightforward steps, and it is basically the same as the corresponding judgment in [20].

As the first step, we give the semantics of a triple, relative to a context and a semantic function environment. The semantics of $[\![\Delta, \Pi \vdash \delta{:}\mathsf{Spec}]\!]$ is a map

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

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n}{(\mathit{\Pi}, x := E, \eta, h) \Downarrow (\eta_{[x \to n]}, h)}$$

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n}{(\mathit{\Pi}, \mathbf{return}\ E, \eta, h) \Downarrow (\eta_{[ret \to n]}, h)}$$

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n \quad n \in dom(h) \quad h(n) = n'}{(\mathit{\Pi}, x := [E], \eta, h) \Downarrow (\eta_{[x \to n']}, h)}$$

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n \quad [\![\mathit{\Delta} \vdash E'{:}\mathsf{Int}]\!]\eta = n' \quad n \in dom(h)}{(\mathit{\Pi}, [E] := E', \eta, h) \Downarrow (\eta, h_{[n \to n']})}$$

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n \quad n \in dom(h)}{(\mathit{\Pi}, \mathbf{dispose}(E), \eta, h) \Downarrow (\eta, h - \{n\})}$$

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n \quad n \notin dom(h)}{(\mathit{\Pi}, x := [E], \eta, h) \Downarrow \mathbf{wrong}}$$

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n \quad n \notin dom(h)}{(\mathit{\Pi}, [E] := E', \eta, h) \Downarrow \mathbf{wrong}}$$

$$\frac{[\![\mathit{\Delta} \vdash E{:}\mathsf{Int}]\!]\eta = n \quad n \notin dom(h)}{(\mathit{\Pi}, \mathbf{dispose}(E), \eta, h) \Downarrow \mathbf{wrong}}$$

$$\frac{\{n, n+1, \ldots, n+m\} \bot dom(h) \quad ([\![\mathit{\Delta} \vdash E_i{:}\mathsf{Int}]\!]\eta = n_i)_{i=0,\ldots,m}}{(\mathit{\Pi}, x := \mathbf{cons}(E_0, \ldots, E_m), \eta, h) \Downarrow (\eta_{[x \to n]}, h_{[n+i \to n_i]_{i=0,\ldots,m}})}$$

$$\frac{(\mathit{\Pi}, c_1, \eta, h) \Downarrow (\eta', h') \quad (\mathit{\Pi}, c_2, \eta', h') \Downarrow (\eta'', h'')}{(\mathit{\Pi}, c_1; c_2, \eta, h) \Downarrow (\eta'', h'')}$$

$$\frac{[\![\mathit{\Delta} \vdash B{:}\mathsf{Bool}]\!]\eta = \mathbf{false} \quad (\mathit{\Pi}, c_1, \eta, h) \Downarrow (\eta', h')}{(\mathit{\Pi}, \mathbf{if}\ B\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\ \mathbf{fi}) \Downarrow (\eta', h')}$$

$$\frac{[\![\mathit{\Delta} \vdash B{:}\mathsf{Bool}]\!]\eta = \mathbf{true} \quad (\mathit{\Pi}, c_0, \eta, h) \Downarrow (\eta', h')}{(\mathit{\Pi}, \mathbf{if}\ B\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\ \mathbf{fi}) \Downarrow (\eta', h')}$$

$$\frac{[\![\mathit{\Delta} \vdash B{:}\mathsf{Bool}]\!]\eta = \mathbf{false}}{(\mathit{\Pi}, \mathbf{while}\ B\ \mathbf{do}\ c\ \mathbf{od}, \eta, h) \Downarrow (\eta, h)}$$

$$\frac{[\![\mathit{\Delta} \vdash B{:}\mathsf{Bool}]\!]\eta = \mathbf{true} \quad (\mathit{\Pi}, c; \mathbf{while}\ B\ \mathbf{do}\ c\ \mathbf{od}, \eta, h) \Downarrow (\eta', h')}{(\mathit{\Pi}, \mathbf{while}\ B\ \mathbf{do}\ c\ \mathbf{od}, \eta, h) \Downarrow (\eta', h')}$$

$$\frac{\begin{array}{c}\mathit{\Pi}(k) = ((x_1, \ldots, x_m), c_k) \\ ([\![\mathit{\Delta} \vdash E_i{:}\mathsf{Int}]\!]\eta = n_i)_{i=1,\ldots,m}\end{array} \quad (\mathit{\Pi}, c_k, [x_i \to n_i], h) \Downarrow (\eta', h')}{(\mathit{\Pi}, x = k(E_1, \ldots, E_m), \eta, h) \Downarrow (\eta_{[x \to \eta'(ret)]}, h')}$$

$$\frac{(\mathit{\Pi}, c, \eta_{[x \to null]}, h) \Downarrow (\eta', h') \quad \eta(x) = v}{(\mathit{\Pi}, \mathbf{newvar}\ x; c, \eta, h) \Downarrow (\eta'_{[x \to v]}, h')}$$

$$\frac{(\mathit{\Pi} \cup (k_1 \to ((x_1, \ldots, x_{n_1}), c_1), \ldots, k_n \to ((x_1, \ldots, x_{n_k}), c_n)), c, \eta, h) \Downarrow (\eta', h')}{(\mathit{\Pi}, \mathbf{let}\ k_1(x_1, \ldots, x_{n_1}) = c_1, \ldots, k_n(x_1, \ldots, x_{n_n}) = c_n\ \mathbf{in}\ c, \eta, h) \Downarrow (\eta', h')}$$

Fig. 1. Operational Semantics of the Programming Language

from $[\![\Delta]\!]$ to the domain $\{\mathbf{true}, \mathbf{false}\}$, and it is given by (we omit some obvious type annotations here):

$$\begin{aligned}
[\![\Delta, \Pi \vdash \{P\}\ c\ \{Q\}]\!] &\text{ iff } \forall h \in [\![\Delta \vdash P]\!]\eta. \\
&\qquad - (\Pi, c, \eta, h) \text{ is safe, and} \\
&\qquad - (\Pi, c, \eta, h) \Downarrow (\eta', h') \text{ implies } h' \in [\![\Delta \vdash Q]\!]\eta' \\
[\![\Delta, \Pi \vdash \delta \wedge \delta']\!]\eta &\text{ iff } [\![\Delta, \Pi \vdash \delta]\!]\eta \text{ and } [\![\Delta, \Pi \vdash \delta']\!]\eta \\
[\![\Delta, \Pi \vdash \exists x{:}\tau.\ \delta]\!]\eta &\text{ iff } [\![\Delta, \Pi \vdash \delta]\!]\eta_{[x \to v]} \text{ for some } v \in [\![\tau]\!] \\
[\![\Delta, \Pi \vdash \forall x{:}\tau.\ \delta]\!]\eta &\text{ iff } [\![\Delta, \Pi \vdash \delta]\!]\eta_{[x \to v]} \text{ for all } v \in [\![\tau]\!].
\end{aligned}$$

We call $\Delta, \Pi \vdash \delta$ *valid* and write $\Delta, \Pi \models \delta$ iff $[\![\Delta, \Pi \vdash \delta]\!]\eta = \mathbf{true}$ for all $\eta \in [\![\Delta]\!]$.

There is a substitution lemma for this semantics, which we will need later.

Lemma 2. *Let δ be a specification, $x{:}\tau$ a variable, and $\Delta \vdash t{:}\tau$ a term. Further, let $\eta \in [\![\Delta]\!]$, and Π be well-formed. Then,*

$$[\![\Delta; \Pi \vdash \delta[t/x]]\!]\eta \textit{ iff } [\![\Delta, x{:}\tau; \Pi \vdash \delta]\!]\eta_{[x \to v]},$$

where $v = [\![\Delta \vdash t{:}\tau]\!]\eta$.

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

$$[\![\Delta, \Pi \vdash \gamma{:}\mathsf{FSpec}]\!] : [\![\Delta]\!] \to \{\mathbf{true}, \mathbf{false}\},$$

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

$$\begin{aligned}
[\![\Delta; \Pi \vdash \{P\}\ k\ \{Q\}]\!]\eta &\text{ iff } [\![\Delta'; \Pi \vdash \{P\}\ c_m\ \{Q\}]\!]\eta \\
&\qquad \text{where } \Pi(k) = ((x_1, \ldots, x_{n_m}), c_m),
\end{aligned}$$

where Δ' is Δ with those x_i added (with type Int) that are not there.

For this semantics, we also have a substitution lemma, which resembles that of Lemma 2.

Lemma 3. *Let γ be a specification, $x{:}\tau$ a variable, and $\Delta \vdash t{:}\tau$ a term. Further, let $\eta \in [\![\Delta]\!]$, and Π be well-formed. Then,*

$$[\![\Delta; \Pi \vdash \gamma[t/x]]\!]\eta \textit{ iff } [\![\Delta, x{:}\tau; \Pi \vdash \gamma]\!]\eta_{[x \to v]},$$

where $v = [\![\Delta \vdash t{:}\tau]\!]\eta$.

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

$$[\![\Delta, \Pi \vdash \Gamma]\!]\eta \text{ iff } [\![\Delta, \Pi \vdash \gamma]\!]\eta \text{ for all } \gamma \in \Gamma.$$

Lemma 4. *Let $\Delta \vdash t{:}\tau$ be a term, $\eta \in [\![\Delta]\!]$, and Γ an environment. Then,*

$$[\![\Delta; \Pi \vdash \Gamma[t/x]]\!]\eta \textit{ iff } [\![\Delta, x{:}\tau; \Pi \vdash \Gamma]\!]\eta_{[x \to v]},$$

where $v = [\![\Delta \vdash t : \tau]\!]\eta$.

Finally, we arrive at the semantics of specifications, relative to a context and an environment.

$$\begin{aligned}\Delta; \Gamma \models \delta \text{ iff for all well-formed } \Pi \text{ and all } \eta \in [\![\Delta]\!],\\ [\![\Delta; \Pi \vdash \Gamma]\!]\eta \text{ implies } [\![\Delta; \Pi \vdash \delta]\!]\eta.\end{aligned}$$

The relevant substitution lemma for this semantics is:

Lemma 5. *Let $\Delta \vdash t{:}\tau$ be a term. Then*

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

3 Program Logic

We define a judgment

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

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

3.1 Explanation of Rules

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

The rule

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

for function call says that in order to call a function, we have to make sure that the precondition for the function is satisfied. This precondition is recorded in the environment, along with the corresponding postcondition.

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

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

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

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

called *the frame rule*, implies that one can infer a *global* specification from a *local* specification like the one for **cons**. Hence, we can execute **cons** in *any*

$$\frac{}{\Delta; \Gamma \vdash \{P\}\mathbf{skip}\{P\}} \qquad \frac{}{\Delta; \Gamma \vdash \{P[E/x]\}\ x := E\ \{P\}}\ x \notin FV(E)$$

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

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

$$\frac{}{\Delta; \Gamma \vdash \{E \mapsto -\}\mathbf{dispose}(E)\{\mathsf{emp}\}}$$

$$\frac{}{\Delta; \Gamma \vdash \{E \mapsto n \wedge x = m\}x := [E]\{E[m/x] \mapsto n \wedge x = n\}}$$

$$\frac{}{\Delta; \Gamma \vdash \{\Delta; \Gamma \vdash E \mapsto -\}[E] := E'\{E \mapsto E'\}}$$

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

$$\frac{\Delta; \Gamma \vdash \{P\}\ c_1\ \{P'\} \quad \Delta; \Gamma \vdash \{P'\}\ c_2\ \{Q\}}{\Delta; \Gamma \vdash \{P\}\ c_1; c_2\ \{Q\}}$$

$$\frac{\Delta, x{:}\mathsf{Int}; \Gamma \vdash \{P \wedge x = nil\}\ c\ \{Q\}}{\Delta; \Gamma \vdash \{P\}\ \mathbf{newvar}\ x\ \mathbf{in}\ c\ \mathbf{end}\ \{Q\}}\ x \notin FV(P)$$

$$\frac{\Delta; \Gamma \vdash \{P \wedge B\}\ c_1\{Q\} \quad \Delta; \Gamma \vdash \{P \wedge \neg B\}\ c_2\{Q\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\ \{Q\}}$$

$$\frac{\Delta; \Gamma \vdash \{P \wedge B\}\ c\ \{P\}}{\Delta; \Gamma \vdash \{P\}\ \mathbf{while}\ B\ \mathbf{do}\ c\ \mathbf{od}\ \{P \wedge \neg B\}}$$

$$\frac{\Delta; \Gamma \vdash \{P\}\ c\ \{Q\}}{\Delta, \Delta'; \Gamma \vdash \{P\}\ c\ \{Q\}}\ \Delta \cap \Delta' = \varnothing$$

$$\frac{\Delta, \Delta'; \Gamma \vdash \{P\}\ c\ \{Q\}}{\Delta; \Gamma \vdash \{P\}\ c\ \{Q\}}\ \Delta' \text{ disjoint from } \Delta, P, c, Q, \Gamma$$

$$\frac{\Delta \vdash P \Rightarrow P' \quad \Delta; \Gamma \vdash \{P'\}\ c\ \{Q'\} \quad \Delta \vdash Q' \Rightarrow Q}{\Delta; \Gamma \vdash \{P\}\ c\ \{Q\}}$$

$$\begin{array}{c} \Delta, x{:}\tau; \Gamma, \gamma \vdash \delta \\ \hline\hline \Delta; \Gamma, \exists x{:}\tau.\ \gamma \vdash \delta \end{array}\ x \notin FV(\Gamma) \cup \mathrm{Mod}(\delta)$$

$$\begin{array}{c} \Delta, x{:}\tau; \Gamma \vdash \delta \\ \hline\hline \Delta; \Gamma \vdash \forall x{:}\tau.\ \delta \end{array}\ x \notin FV(\Gamma) \cup \mathrm{Mod}(\delta)$$

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

Fig. 2. Program Logic

heap, described by the predicate P (not containing x), by the following instance of the frame rule:

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

The rule

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

for function definitions is the usual one from Hoare logic with procedures [9]. The rules for **while** and **if-then-else** are also standard. The next two rules are structural and allow certain straightforward manipulations of contexts. The rule of consequence is also standard. Finally, the rules

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

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

are straightforward adaptations of standard rules of higher-order logic. They will be used later for reasoning about data abstraction and parametricity. Note that in both of the rules, $x \notin \mathrm{Mod}(\delta)$. Often, x will denote an “abstract value”, which should not be part of the specification one wants to show in the end, as we shall see later.

3.2 Soundness

Theorem 1. *If a specification*

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

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

Proof. For each rule of form

$$\frac{\Delta; \Gamma \vdash \delta}{\Delta'; \Gamma' \vdash \delta'} \ , \qquad (3)$$

this is checked by showing that $\Delta'; \Gamma' \models \delta'$, under the assumption that $\Delta; \Gamma \models \delta$. For axioms of the form

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

the proof obligation is to show that $\Delta; \Gamma \models \delta$. We only show soundness for some of the rules here.

Consider the rule for **skip**:

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

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

$$[\![\Delta; \Pi \vdash \{P\}\ \mathbf{skip}\ \{P\}]\!]\eta$$

for all $\eta \in [\![\Delta]\!]$. Let $h \in [\![P]\!]\eta$. Then,

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

and clearly, $h \in [\![P]\!]\eta$, so this rule is sound.

Soundness of the rule for assignment

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

depends, as usual, on the substitution Lemma 1.

The rule for **return**

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

is essentially just an instance of the assignment rule.

Now consider the rule for function call:

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

To show soundness, suppose $\{P\}\ k_i(x_1, \dots, x_{n_i})\ \{Q\} \in \Gamma$. Let $\eta \in [\![\Delta]\!]$, and let Π be a well-formed semantic function environment such that $[\![\Delta; \Pi \models \Gamma]\!]\eta$. In particular,

$$[\![\Delta; \Pi \vdash \{P\}\ k_i(x_1, \dots, x_{n_i})\ \{Q\}]\!]\eta,$$

so if $\Pi(k_i) = ((x_1, \dots, x_{n_i}), c_i)$, then $[\![\Delta; \Pi \vdash \{P\}\ c_i\ \{Q\}]\!]\eta$. Now, suppose

$$h \in [\![P[E_1/x_1 \cdots E_{n_i}/x_{n_i}]]\!]\eta = [\![P]\!]\eta_{[x_1 \to v_1, \cdots, x_{n_i} \to v_{n_i}]},$$

where $v_j = [\![\Delta \vdash E_j{:}\mathsf{Int}]\!]\eta$ for $j = 1, \dots, n_i$ (we've used the substitution lemma here). This means that if

$$(\Pi, c_i, \eta_{[x_1 \to v_1, \cdots, x_{n_i} \to v_{n_i}]}, h) \Downarrow (\eta', h'),$$

then $h' \in [\![Q]\!]\eta'$. Since Π is well-formed, c_i does not modify any variables, so η' is of the form

$$\eta' = \eta_{[x_1 \to v_1, \cdots, x_{n_i} \to v_{n_i}, ret \to \eta'(ret)]},$$

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

$$(\Pi, y = k_i(E_1, \dots, E_{n_i}), \eta, h) \Downarrow (\eta_{[}y \to \eta'(ret)], h'),$$

and thus, the rule holds.

We now turn to the first rule for existentials:

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

Suppose that for all well-formed Π and $\eta \in [\![\Delta, x{:}\tau]\!]$,

$$[\![\Delta, x{:}\tau; \Pi \vdash \Gamma, \gamma]\!]\eta \text{ implies} [\![\Delta, x{:}\tau; \Pi \vdash \delta]\!]\eta,$$

and let $[\![\Delta; \Pi \vdash \Gamma]\!]\eta$ and $[\![\Delta; \Pi \vdash \exists x{:}\tau.\ \gamma]\!]\eta$. This means $[\![\Delta; \Pi \vdash \gamma]\!]\eta_{[x \to v]}$ for some $v \in [\![\tau]\!]$. Since $x \notin FV(\Gamma)$, we also have $[\![\Delta; \Pi \vdash \Gamma]\!]\eta_{[x \to v]}$. This implies $[\![\Delta; \Pi \vdash \delta]\!]\eta_{[x \to v]}$, and since $x \notin Mod(\delta)$, we have $[\![\Delta; \Pi \vdash \delta]\!]\eta$ (we use an obvious property of specifications that do not contain variables).

The other rule for existentials is

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

For soundness, first suppose that τ is inhabited and that for all well-formed Π and $\eta \in [\![\Delta]\!]$,

$$[\![\Delta; \Pi \vdash \Gamma, \exists x{:}\tau.\ \gamma]\!]\eta \text{ implies } [\![\Delta; \Pi \vdash \delta]\!]\eta,$$

and suppose $[\![\Delta, x{:}\tau; \Pi \vdash \Gamma, \gamma]\!]\eta$. Since τ is inhabited, this means

$$[\![\Delta, x{:}\tau; \Pi \vdash \Gamma, \gamma]\!]\eta_{[x \to \eta(x)]},$$

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

$$[\![\Delta, x{:}\tau; \Pi \vdash \Gamma, \exists x{:}\tau.\ \gamma]\!]\eta,$$

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

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

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

suppose that for all wellformed Π and $\eta \in [\![\Delta]\!]$,

$$[\![\Delta; \Pi \vdash \Gamma]\!]\eta \text{ implies } [\![\Delta; \pi \vdash \forall x{:}\tau.\ \delta]\!]\eta,$$

and let $\eta' \in [\![\Delta, x{:}\tau]\!]$. Suppose $[\![\Delta, x{:}\tau; \Pi \vdash \Gamma]\!]\eta'$. Since $\notin FV(\Gamma)$, we get

$$[\![\Delta; \Pi \vdash \Gamma]\!]\eta' - x,$$

and this implies

$$[\![\Delta, x{:}\tau; \Pi \vdash \delta]\!](\eta' - x)_{[x \to v]} \text{ for all } v \in [\![\tau]\!].$$

This means, in particular, that

$$[\![\Delta, x{:}\tau; \Pi \vdash \delta]\!]\eta'_{[x \to \eta'(x)]},$$

which shows the desired result. □

3.3 A Derived Rule

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

$$\frac{\begin{array}{c} \Delta \vdash \hat{P}{:}\tau \\ \Delta; \Gamma \vdash \{P_1[\hat{P}/x]\}\ c_1\ \{Q_1[\hat{P}/x]\} \\ \vdots \\ \Delta; \Gamma \vdash \{P_n[\hat{P}/x]\}\ c_n\ \{Q_n[\hat{P}/x]\} \\ \Delta; \Gamma, \exists x{:}\tau.(\{P_1\}k_1\{Q_1\} \wedge \cdots \wedge \{P_n\}k_n\{Q_n\}) \vdash \{P\}\ c\ \{Q\} \end{array}}{\Delta; \Gamma \vdash \{P\}\ \mathbf{let}\ k_1(\bar{x}_1) = c_1, \ldots, k_1(\bar{x}_n) = c_n\ \mathbf{in}\ c\ \mathbf{end}\ \{Q\}}\ x \notin FV(\{P\}\ c\ \{Q\}), \tag{4}$$

and it could easily be generalized to more variables than x, and more predicates $\hat{P}$, but for presentational purposes, we have just presented it with one variable.

We show how this rule can be derived; for brevity, we just do it for $n = 1$, and we suppose there are no parameters. The proof of the more general case is essentially the same as for this case. First, we have the following instance of the function definition rule

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

The rule for existentials gives us that

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

so we need to establish

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

given that

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

But this follows from Lemma 5, since x is not free in $\{P\}\ c\ \{Q\}$.

4 Examples

We show an example of data abstraction and how it can be handled using our program logic. The example involves two implementations of a priority queue, and the intention is, of course, that client programs which use these implementations should be unaware of and unable to exploit details of which implementation is used. Data abstraction is modelled via existential quantification over predicates, corresponding to the slogan that "abstract types have existential type" [14].

In this section, we first define an abstract priority queue, and use abstract operations in several client programs to demonstrate uses of abstract operations and their specifications. We then show two *implementations* of the abstract module, and prove that have specifications that make application of the abstract function definition rule (4) possible.

4.1 Reasoning with an Abstract Priority Queue

Priority queues are used frequently in programming, for example in scheduling algorithms for processes in operating systems [27]. They consist of pairs (p, v), where v is a value that is stored, and p is the *priority* associated with v. In such a structure, one can then enqueue such pairs and extract an element with the highest priority. For simplicity, we assume that the values are integers. Here is a simple grammar for such a structure.

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

There are some operations we will need on such queues. They use the axiom of choice, and are defined by

$$\begin{array}{l}\mathsf{MaxPri}(\epsilon) = -1\\ \mathsf{MaxPri}((p,v) \cup Q) = \mathsf{Max}(p, \mathsf{MaxPri}(Q))\\ \mathsf{MaxPair}(Q) = \mathbf{choose}(\{(p,v) \in Q \mid p = \mathsf{MaxPri}(Q)\}).\end{array}$$

Note that $\mathsf{MaxPair}$ is a nondeterministic operation. We will assume a base type PriQ whose values are priority queues, and an operation Set which, given a priority queue, returns the multiset of pairs occuring in it. These types and operations will only be used in the logic, *not* in programs. Also, we could have encoded the type PriQ in our higher-order logic, but for simplicity, we just introduce it in the logic here.

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

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

which asserts that the integer denoted by q is a handle to the priority queue Q – but it does *not* say anything about Q is represented. This will be used as an abstract predicate in our proofs (and thus play the role of $\hat{P}$ when we apply the abstract function definition rule (4)). 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

$$\{\mathsf{emp}\}\ \mathbf{createqueue}()\ \{\mathsf{repr}(ret, \epsilon)\},$$

which merely says that upon creation of a queue, we return a handle to an empty priority queue.

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

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

Dequeing There should be an operation for dequeing. We need to take care that we do not dequeue from an empty queue, and hence the specification is

$$\begin{array}{l}\{\mathsf{repr}(q,Q)\wedge Q\neq\epsilon\}\\ \quad\mathbf{dequeue}(q)\\ \{\exists Q',p,v.\ \mathsf{repr}(q,Q')\wedge Q=(p,v)\cup Q'\wedge(p,v)=\mathsf{MaxPair}(Q)\wedge ret=v\}.\end{array}$$

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

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

4.2 Sample Programs

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

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

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

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

$$\begin{array}{l}\{\mathsf{repr}(q,Q)\wedge Q\neq\epsilon\}\\ \quad\mathbf{dequeue}()\\ \{\mathsf{repr}(q,Q')\wedge ret=v\}.\end{array}$$

Here is another basic program (and its specification) which, however, uses two queues. In the second step, we use the specification for **createqueue** and the frame rule.

```
{$\mathsf{emp}$}
  $q_1 = \mathbf{createqueue}()$;
{$\mathsf{repr}(q_1, \epsilon)$}
  $\mathbf{enqueue}(q_1, (3,7))$;
{$\mathsf{repr}(q_1, \langle(3,7)\rangle)$}
  $q_2 = \mathbf{createqueue}()$;
{$\mathsf{repr}(q_1, \langle(3,7)\rangle) * \mathsf{repr}(q_2, \epsilon)$}
  $\mathbf{enqueue}(q_1, (7,3))$;
{$\mathsf{repr}(q_1, \langle(3,7),(7,3)\rangle) * \mathsf{repr}(q_2, \epsilon)$}
  $\mathbf{enqueue}(q_2, (4,13))$;
{$\mathsf{repr}(q_1, \langle(3,7),(7,3)\rangle) * \mathsf{repr}(q_2, \langle(4,13)\rangle)$}
  $y_1 := \mathbf{dequeue}(q_1)$;
{$\mathsf{repr}(q_1, \langle(3,7)\rangle) * \mathsf{repr}(q_2, \langle(4,13)\rangle) \wedge y_1 = 3$}
  $y_2 := \mathbf{dequeue}(q_2)$;
{$\mathsf{repr}(q_1, \langle(3,7)\rangle) * \mathsf{repr}(q_2, \epsilon) \wedge y_1 = 3 \wedge y_2 = 13$}
  $\mathbf{disposequeue}(q_1)$;
  $\mathbf{disposequeue}(q_2)$;
{$\mathsf{emp} \wedge y_1 = 3 \wedge y_2 = 13$}
```

This example serves to illustrate that our notion of modularity is not static, as is the case in [19] In the setting of that paper, it is not possible to give arguments to function calls – one has to initialize variables that are laid out in the specification of each module. This implies that it is not possible have multiple instances of the same data structure (this was also noted in [20]).

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

```
{$\mathsf{emp}$}
  $q = \mathbf{createqueue}()$
{$\mathsf{repr}(q, \epsilon)$}
  $y = \mathbf{dequeue}(q)$
{???}
```

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

In the second erroneous program, we try to dequeue from a queue that has been disposed.

$$
\begin{array}{l}
\{\mathsf{emp}\} \\
\quad q = \mathbf{createqueue}(); \\
\{\mathsf{repr}(q, \epsilon)\} \\
\quad \mathbf{enqueue}(q, (4, 42)); \\
\{\mathsf{repr}(q, \langle 4, 42 \rangle)\} \\
\quad \mathbf{disposequeue}(q); \\
\{\mathsf{emp}\} \\
\quad y = \mathbf{dequeue}(q) \\
\{???\}
\end{array}
$$

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

$$
\begin{array}{l}
\{\mathsf{emp}\} \\
\quad q = \mathbf{createqueue}(); \\
\{\mathsf{repr}(q, \epsilon)\} \\
\quad \mathbf{newvar}\ x; x := [q] \\
\{???\}.
\end{array}
$$

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

Note that the priority queues we use here do not involve any *ownership transfer* [19], since the priority queues only hold simple data values (integers). We could have chosen to implement queues where ownership of heap cells transfer back and forth between clients and modules. Although this would certainly be interesting, we believe that the core ideas are presented via the examples at hand.

4.3 Implementations of Priority Queues

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

Sorted, Singly-linked Lists Singly-linked lists (or rather, singly-linked list segments) are introduced in Reynolds' introductory paper on separation logic [26]. The idea is that the predicate

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

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

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

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

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

One can prove the specifications shown in Figure 3 for these implementations, using the proof rules from Section 3 plus the lemmas from [26] about the slist predicate, and some obvious implications regarding sorted sequences. In our proofs, we use certain assertions and operations on sequences, which are easy to encode in higher-order logic. The predicate $\mathsf{sorted}(\alpha)$ is true iff the sequence α is sorted (in decreasing order) according to priorities. For a nonempty sequence $\alpha = (p_0, v_0), \ldots, (p_n, v_n)$, $\mathsf{Max}(\alpha)$ is a pair from α with the highest priority – note that this is non-deterministic. Finally, for a sequence α, we define $\mathsf{Set}(\alpha)$ to be the multiset of pairs occurring in α.

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

Doubly-linked Lists Doubly linked lists are also introduced in [26]. The predicate

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

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

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

$$
\begin{array}{l}
\{\mathsf{emp}\} \\
\quad \mathsf{slistcreate} \\
\{\mathsf{slist}(\epsilon, ret, null)\}
\end{array}
$$

$$
\begin{array}{l}
\{\mathsf{slist}(\alpha, q, null) \wedge \mathsf{sorted}(\alpha)\} \\
\quad \mathsf{slistEnque}(q, (p, v)) \\
\{\exists \alpha'.\ \mathsf{slist}(\alpha', q, null) \wedge \mathsf{sorted}(\alpha') \wedge \mathsf{Set}(\alpha') = \mathsf{Set}(\alpha) \uplus (p, v)\}
\end{array}
$$

$$
\begin{array}{l}
\{\mathsf{slist}(\alpha, q, null) \wedge \mathsf{sorted}(\alpha) \wedge \alpha \neq \epsilon\} \\
\quad \mathsf{slistDeque}(q) \\
\left\{ \begin{array}{l} \exists p, v, \alpha'.\ \mathsf{slist}(\alpha', q, null) \wedge \mathsf{Set}(\alpha) = \mathsf{Set}(\alpha') \uplus (p, v) \wedge \\ (p, v) = \mathsf{Max}(\alpha) \wedge ret = v \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\{\mathsf{slist}(\alpha, q, null)\} \\
\quad \mathsf{slistDispose}(q) \\
\{\mathsf{emp}\}
\end{array}
$$

Fig. 3. Specifications for slist-implementation of Priority Queues

We implement priority queues by doubly linked lists. In contrast to our implementation with singly linked lists, this implementation does *not* keep the list sorted. Consequently, the tricky part is to dequeue, whereas enqueing is easy. The implementations are shown in Appendix B. We use the handle q to point to a cell containing the values i, i', j, j' that determine the "boundaries" of the list, for technical reasons.

Here is a brief explanation to each of the programs in this implementation. The dlistcreate program just initializes an empty doubly linked list by storing *null*s in the heap. To enqueue, we just insert at the front of the doubly linked list, since we do not keep the list sorted. The tricky part in this implementation is to dequeue, since we have to find the correct element of the list and take it out of the list. This is done in dlistDeque; in the **while** loop we traverse the list and find an element with the highest priority, and then we take it out of the list. This is done a little differently according to the position of the element in the list.

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

$\{\mathsf{emp}\}$
  $\mathsf{dlistcreate}$
$\{\exists i, i', j, j'.\ q \mapsto i, i', j, j' * \mathsf{dlist}(\epsilon, i, i', j, j')\}$

$\{\exists i, i', j, j'.\ q \mapsto i, i', j, j' * \mathsf{dlist}(\alpha, i, i', j, j')\}$
  $\mathsf{dlistEnque}(q, (p, v))$
$\{\exists i, i', j, j'.\ q \mapsto i, i', j, j' * \mathsf{dlist}((p, v) \cdot \alpha, i, i', j, j')\}$

$\{\exists i, i', j, j'.\ q \mapsto i, i', j, j' * \mathsf{dlist}(\alpha, i, i', j, j') \wedge \alpha \neq \epsilon\}$
  $\mathsf{dlistDeque}(q)$
$$\left\{\begin{array}{l}\exists i, i', j, j', p, v, \alpha'.\ \mathsf{dlist}(\alpha', i, i', j, j') \wedge \mathsf{Set}(\alpha) = \mathsf{Set}(\alpha') \uplus (p, v) \wedge \\ (p, v) = \mathsf{Max}(\alpha) \wedge ret = v\end{array}\right\}$$

$\{\exists i, i', j, j'.\ q \mapsto i, i', j, j' * \mathsf{dlist}(\alpha, i, i', j, j')\}$
  $\mathsf{dlistDispose}(q)$
$\{\mathsf{emp}\}$

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

4.4 Representation Independence

We now argue that our system may be used to reason about independence of the representation of data using the examples from Sec. 4.3. Intuitively, a client program should not be able to distinguish between the two implementations of a priority queue we have given; here we justify this intuition.

Consider the two programs

```
c₁ ≡
let
   createqueue() = slistcreate
   enqueue(q, (p, v)) = slistEnque(q, (p, v))
   dequeue(q) = slistDeque(q)
   disposequeue(q) = slistDispose(q)
in c end
```

and

```
c₂ ≡
let
   createqueue() = dlistcreate
   enqueue(q, (p, v)) = dlistEnque(q, (p, v))
   dequeue(q) = dlistDeque(q)
   disposequeue(q) = dlistDispose(q)
in c end,
```

where c is a program that uses priority queues, for example the ones in Section 4.2. Intuitively, it should not matter which implementation we use, so that any specification we can show for one program, we should be able to show for the other. Consider the abstract function definition rule from Section 3.3, which can be used to verify such programs. We spell out how we can use the work we have already done to apply this rule.

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

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

Using the specifications mentioned in Section 4.3, it is not hard to show the following specifications.

$\{\mathsf{emp}\}$
  $\mathsf{slistcreate}$
$\{\hat{P}_{\mathsf{slist}}(q,\epsilon)\}$

$\{\hat{P}\mathsf{slist}(q,Q)\}$
  $\mathsf{slistEnque}(q,(p,v))$
$\{\hat{P}_{\mathsf{slist}}(q, Q \uplus (p,v))\}$

$\{\hat{P}_{\mathsf{slist}}(q,Q) \wedge Q \neq \epsilon\}$
  $\mathsf{slistDeque}(q)$
$\{\exists Q',p,v.\ \hat{P}_{\mathsf{slist}}(q,Q') \wedge \mathsf{Set}(Q) = \mathsf{Set}(Q') \uplus (p,v) \wedge (p,v) = \mathsf{MaxElt}(Q) \wedge ret = v\}$

$\{\hat{P}_{\mathsf{slist}}(q,Q)\}$
  $\mathsf{slistDeque}(q)$
$\{\mathsf{emp}\}$.

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

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

we can use the work in Section 4.3 to prove the specifications

$\{\mathsf{emp}\}$
$\quad\mathsf{dlistcreate}()$
$\{\hat{P}_{\mathsf{dlist}}(\epsilon, q)\}$

$\{\hat{P}_{\mathsf{dlist}}(Q, q)\}$
$\quad\mathsf{dlistEnque}(q, (p, v))$
$\{\hat{P}_{\mathsf{dlist}}(Q \cup (p, v), q)\}$

$\{\hat{P}_{\mathsf{dlist}}(q, Q) \wedge Q \neq \epsilon\}$
$\quad\mathsf{dlistDeque}(q)$
$\{\exists Q', p, v, \alpha.\ \hat{P}_{\mathsf{dlist}}(q, Q') \wedge Q = Q' \cup (p, v) \wedge (p, v) = \mathsf{MaxElt}(Q) \wedge ret = v\}$

$\{\hat{P}_{\mathsf{dlist}}(q, Q)\}$
$\quad\mathsf{dlistDispose}(q)$
$\{\mathsf{emp}\}$.

The proofs of the programs in Section 4.2 which use the abstract predicate can be fitted into our framework. For example, the first of those programs has the specification

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

where Δ is the context $q{:}\mathsf{Int}, y{:}\mathsf{Int}, \mathsf{repr}{:}(\mathsf{PriQ} \times \mathsf{Int}) \Rightarrow \mathsf{Prop}$ and Γ is the environment containing the abstract functions listed at the end of Section 4.1:

$\{\mathsf{emp}\}\ \mathbf{createqueue}()\ \{\mathsf{repr}(ret, \epsilon)\}$,
$\{\mathsf{repr}(q, Q)\}\ \mathbf{enqueue}(q, (v, p))\ \{\mathsf{repr}(q, (v, p) \cup Q)\}$,
$\{\mathsf{repr}(q, Q) \wedge Q \neq \epsilon\}$
$\quad\mathbf{dequeue}(q)$
$\{\exists Q', p, v.\ \mathsf{repr}(q, Q') \wedge Q = (p, v) \cup Q' \wedge (p, v) = \mathsf{MaxElt}(Q) \wedge ret = v\}$,
$\{\mathsf{repr}(q, Q)\}\ \mathbf{disposequeue}(q)\ \{\mathsf{emp}\}$.

According to the rule (4) in Section 3.3, we can then derive the specification

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

where $\tilde{c}$ is replaced by *either* c_1 or c_2. We note that *the proof of the client program is independent of the implementation of the module it uses.* This is the sense in which we can use our system to reason about representation independence.

4.5 Polymorphic Types via Universal Quantification

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

The queue module example from [19] is parametric in a predicate P *at the meta-level.* We show that in higher-order separation logic, the parametricity may be expressed *in the logic*. To this end, consider the following version of the parametric list predicate from [19].

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

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 $P \equiv \mathsf{emp}$, plain values are stored (and no ownership transfer to the queue module in [19]), and if $P \equiv x \mapsto -,-$, addresses of cells are stored in the queue (and ownership of the cells is tranferred in and out of the queue [19]).

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

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

Here we have used a type seqInt of sequences of integers, which is easily definable in higher-order separation logic, and the definition of $\mathsf{list}(P,\beta,i)$ can be given by induction on β *in the logic*, in the same sense as the slist and dlist predicates from Section 4.3.

```
$j :=$ ;
**while** $i \neq$ **do**   $k := [i+1]$;
   $[i+1] := j$;
   $j := i$;
   $i := k$
**od**
```

Fig. 5. The list reversal program **listRev**

Suppose **listRev** is the list reversal program in Fig. 5 (taken from the introduction of [26]). Then one can easily show the specification

$$\{\mathsf{list}(P,\beta,i)\}\ \mathbf{listRev}\ \{\mathsf{list}(P,\beta^{\dagger},i)\},$$

where $\beta^{\dagger}$ is the reverse of the sequence β. By the introduction rule for universal quantification we obtain the specification

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

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

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

For such parametric operations on polymorphic data types to be more useful, one would of course prefer a higher-order programming language instead of the first-order language considered here. Then one could, e.g., program the usual **map** function on lists, and provide a single parametric correctness proof for it. In future work we will show how to make use of higher-order separation logic for a higher-order language, specifically by extending the separation-logic typing discipline for idealized algol recently introduced in joint work with Yang [5].

4.6 Invariance

In this subsection we briefly consider an example, suggested to us by John Reynolds, which demontrates 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 prediate 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:

$$\{q\}\ c\ \{q' * q\}. \tag{5}$$

This does not work, however, unless q is *strictly exact* [26] (for example, if q is $\exists\beta{:}\mathsf{seqInt}.\ \mathsf{list}(\beta, i)$, then c may delete some elements from the list in the input heap h).

Instead, we may use the specification

$$\forall p{:}\mathsf{Prop}.\{q \wedge p\}\ c\ \{q' * p\}, \tag{6}$$

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

Note that (6) is stronger than (5): by instantiation p with q in (6) we get (5). Thus if we wish to prove (5), then we may prove something stronger (6), which may be easier to prove (c.f., strengthening an induction hypothetis), and then derive the desired.

This illustrates that we can use universal quantification to express invariance of commands.

4.7 Predicates via Fixed Points

Recall the slist predicate from the priority queue example above. It is required to satisfy the following recursive equation:

$$\mathsf{slist} = \lambda(x, s).(x = \mathit{null} \wedge \mathsf{emp}) \vee (\exists p, v, k.\ x \mapsto p, v, k * \mathsf{slist}(k, s)).$$

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

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

satisfying that q only occurs positively in φ. Then

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

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

$$\nu q.\varphi(q) = \exists q.(q \to \varphi(q)) \land q$$

is the maximal fixed point for φ.

5 General Semantics for Higher-order BI

The sematics from Section 2.2 of the assertion language (higher-order predicate BI) is an instance of a general concept, which we introduce in this section.

Part of the pointer model of separation logic, namely that given by heaps (but not stacks), has been related to *propositional* BI, the logic of bunched implications introduced by O'Hearn and Pym [15]. Here we show how the correspondence may be extended to a precise correspondence between all of the pointer model (including stacks) and our notion of *predicate* BI. We introduce the notion of a *BI hyperdoctrine*, a simple extension of Lawvere's notion of hyperdoctrine [13], and show that it soundly models predicate BI.

We first introduce Lawvere's notion of a hyperdoctrine [13] and briefly recall how it can be used to model intuitionistic and classical first- and higher-order predicate logic (see, for example, [21] and [12] 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.

5.1 Hyperdoctrines

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

Definition 1 (First-order hyperdoctrines). *Let $\mathcal{C}$ be a category with finite products. A* first-order hyperdoctrine *$\mathcal{P}$ over $\mathcal{C}$ is a contravariant functor $\mathcal{P}{:}\mathcal{C}^{op} \to$ **Poset** from $\mathcal{C}$ into the category of partially ordered sets and monotone functions, with the following properties.*

1. *For each object X, the partially ordered set $\mathcal{P}(X)$ is a Heyting algebra.*

2. *For each morphism $f : X \to Y$ in $\mathcal{C}$, the monotone function $\mathcal{P}(f) : \mathcal{P}(Y) \to \mathcal{P}(X)$ is a Heyting algebra homomorphism.*
3. *For each diagonal morphism $\Delta_X : X \to X \times X$ 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 $A \in \mathcal{P}(X \times X)$,*
$$\top \leq \mathcal{P}(\Delta_X)(A) \quad \textit{iff} \quad =_X \leq A.$$
4. *For each product projection $\pi : \Gamma \times X \to \Gamma$ in $\mathcal{C}$, the monotone function $\mathcal{P}(\pi) : \mathcal{P}(\Gamma) \to \mathcal{P}(\Gamma \times X)$ has both a left adjoint $(\exists X)_\Gamma$ and a right adjoint $(\forall X)_\Gamma$:*
$$\begin{array}{lll} A \leq \mathcal{P}(\pi)(A') & \textit{if and only if} & (\exists X)_\Gamma(A) \leq A' \\ \mathcal{P}(\pi)(A') \leq A & \textit{if and only if} & A' \leq (\forall X)_\Gamma(A). \end{array}$$
Moreover, these adjoints are natural in Γ, i.e., given $s : \Gamma \to \Gamma'$ in $\mathcal{C}$, we have
$$\begin{tikzcd} \mathcal{P}(\Gamma' \times X) \arrow[r, "\mathcal{P}(s\times id_X)"] \arrow[d, "(\exists X)_{\Gamma'}"'] & \mathcal{P}(\Gamma \times X) \arrow[d, "(\exists X)_\Gamma"] \\ \mathcal{P}(\Gamma') \arrow[r, "\mathcal{P}(s)"'] & \mathcal{P}(\Gamma) \end{tikzcd} \qquad \begin{tikzcd} \mathcal{P}(\Gamma' \times X) \arrow[r, "\mathcal{P}(s\times id_X)"] \arrow[d, "(\forall X)_{\Gamma'}"'] & \mathcal{P}(\Gamma \times X) \arrow[d, "(\forall X)_\Gamma"] \\ \mathcal{P}(\Gamma') \arrow[r, "\mathcal{P}(s)"'] & \mathcal{P}(\Gamma). \end{tikzcd}$$
*The elements of $\mathcal{P}(X)$, where X ranges over objects of $\mathcal{C}$, will be 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 $f : X_1, \dots, X_n \to X$, 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 $[\![X]\!]$ in $\mathcal{C}$ to each type, a morphism $[\![f]\!] : [\![X_1]\!] \times \cdots \times [\![X_n]\!] \to [\![X]\!]$ to each function symbol, and a $\mathcal{P}$-predicate $[\![R]\!] \in \mathcal{P}([\![X_1]\!] \times \cdots \times [\![X_n]\!])$ 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 $[\![t]\!] : [\![\Gamma]\!] \to [\![X]\!]$, where $[\![\Gamma]\!] = [\![X_1]\!] \times \cdots \times [\![X_n]\!]$, 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 $[\![\varphi]\!] \in \mathcal{P}([\![\Gamma]\!])$ by induction on the structure of φ using the properties given in Definition 1. For atomic formulas $R(t_1, \dots, t_n)$, the interpretation is given by $\mathcal{P}(\langle [\![t_1]\!], \dots, [\![t_n]\!] \rangle)([\![R]\!])$. In particular, the atomic formula $t =_X t'$ is interpreted by the $\mathcal{P}$-predicate $\mathcal{P}(\langle [\![t]\!], [\![t']\!] \rangle)(=_{[\![X]\!]})$. The interpretation of other formulas is given 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,
$$\begin{array}{ll} [\![\top]\!] = \top_H & [\![\varphi \wedge \varphi']\!] = [\![\varphi]\!] \wedge_H [\![\varphi']\!] \\ [\![\bot]\!] = \bot_H & [\![\varphi \vee \varphi']\!] = [\![\varphi]\!] \vee_H [\![\varphi']\!] \\ & [\![\varphi \to \varphi']\!] = [\![\varphi]\!] \to_H [\![\varphi']\!] \end{array}$$
$$\begin{array}{l} [\![\forall x{:}X.\psi]\!] = (\forall [\![X]\!])_{[\![\Gamma]\!]}([\![\psi]\!]) \in \mathcal{P}([\![\Gamma]\!]) \\ [\![\exists x{:}X.\psi]\!] = (\exists [\![X]\!])_{[\![\Gamma]\!]}([\![\psi]\!]) \in \mathcal{P}([\![\Gamma]\!]), \end{array}$$

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

We say that a formula φ with free variables in Γ is *satisfied* if $[\![\varphi]\!]$ is the top element of $\mathcal{P}([\![\Gamma]\!])$. This notion of satisfaction is *sound* for intuitionistic predicate logic, in the sense that all provable formulas are satisfied. Moreover, it is *complete* in the sense that a formula is provable if it is satisfied in all structures in first-order hyperdoctrines. A first-order hyperdoctrine $\mathcal{P}$ is sound for *classical* predicate logic in case all the fibres $\mathcal{P}(X)$ are Boolean algebras and all the reindexing functions $\mathcal{P}(f)$ are Boolean algebra homomorphisms.

Definition 2 (Hyperdoctrines). *A (general)* hyperdoctrine *is a first-order hyperdoctrine with the following additional properties: $\mathcal{C}$ is cartesian closed, and there is a Heyting algebra H and a natural bijection $\Theta_X : Obj(\mathcal{P}(X)) \simeq \mathcal{C}(X, H)$.*

A hyperdoctrine is sound for higher-order intuitionistic predicate logic: the Heyting algebra H is used to interpret the type Prop of propositions and higher types (e.g., 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 $[\![\psi]\!] \in \mathcal{P}([\![\Gamma]\!])$, $[\![\varphi]\!] \in \mathcal{P}([\![\Gamma]\!] \times H)$, and $\varphi[\psi/q]$ (φ with ψ substituted in for q) is interpreted by $\mathcal{P}(\langle \mathrm{id}, \Theta([\![\psi]\!])\rangle)([\![\varphi]\!])$. For more details see, e.g., [21].

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

Example 1 (Canonical hyperdoctrine over a topos). Let $\mathcal{E}$ be a topos. It is well-known that $\mathcal{E}$ models higher-order predicate logic, by interpreting types as objects in $\mathcal{E}$, terms as morphisms in $\mathcal{E}$ and predicates as subobjects in $\mathcal{E}$. The topos $\mathcal{E}$ induces a canonical $\mathcal{E}$-indexed hyperdoctrine $\mathrm{Sub}_{\mathcal{E}} : \mathcal{E}^{op} \to \mathbf{Poset}$, which maps an object X in $\mathcal{E}$ to the poset of subobjects of X in $\mathcal{E}$ and a morphisms $f : X \to Y$ to the pullback functor $f^* : \mathrm{Sub}(Y) \to \mathrm{Sub}(X)$. Then the standard interpretation of predicate logic in $\mathcal{E}$ coincides with the interpretation of predicate logic in the hyperdoctrine $\mathrm{Sub}_{\mathcal{E}}$. Compared to the standard interpretation in toposes, however, hyperdoctrines allow that predicates are not always modeled by subobjects but can come from some other universe. This means that hyperdoctrines describe a wider class of models than toposes do.

5.2 BI Hyperdoctrines

Recall that a Heyting algebra is a bi-cartesian closed partial order, i.e., a partial order, which, when considered as a category, is cartesian closed ($\top$, $\wedge$, $\to$) and has finite coproducts ($\bot$, $\vee$). Further recall that a *BI algebra* is a Heyting algebra, which has an additional symmetric monoidal closed structure (I, $*$, $-\!\!*$) [22].

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

Definition 3 (BI Hyperdoctrines).

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

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

$$(\varphi * \psi) * \theta \vdash_\Gamma \varphi * (\psi * \theta) \qquad \varphi * (\psi * \theta) \vdash_\Gamma (\varphi * \psi) * \theta \qquad \vdash_\Gamma \varphi \leftrightarrow \varphi * \mathrm{I}$$

$$\varphi * \psi \vdash_\Gamma \psi * \varphi \qquad \frac{\varphi \vdash_\Gamma \psi \quad \theta \vdash_\Gamma \omega}{\varphi * \theta \vdash_\Gamma \psi * \omega} \qquad \frac{\varphi * \psi \vdash_\Gamma \theta}{\overline{\varphi \vdash_\Gamma \psi \mathrel{-\!\!*} \theta}}$$

Our notion of predicate BI should not be confused with the one presented in [22]; the latter seeks to also include a BI structure on contexts but we do not attempt to do that here, since that 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 can interpret first-order predicate BI in a first-order BI hyperdoctrine simply by extending the interpretation of first-order logic in first-order hyperdoctrine given above by:

$$\begin{aligned} [\![\mathrm{I}]\!] &= \mathrm{I}_B \\ [\![\varphi * \psi]\!] &= [\![\varphi]\!] *_B [\![\psi]\!] \\ [\![\varphi \mathrel{-\!\!*} \psi]\!] &= [\![\varphi]\!] \mathrel{-\!\!*}_B [\![\psi]\!], \end{aligned}$$

where I_B, $*_B$ and $\mathrel{-\!\!*}_B$ is the monoidal closed structure in the BI algebra $\mathcal{P}([\![\Gamma]\!])$. We then have:

Theorem 2. *The interpretation of first-order predicate BI given above is sound and complete.*

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

Example 2 (BI hyperdoctrine over a complete BI algebra). Let B be a complete BI algebra, i.e., it has all joins and meets. It determines a BI hyperdoctrine over the category **Set** as follows. For each set X, let $\mathcal{P}(X) = B^X$, the set of functions from X to B, ordered pointwise. Given $f : X \to Y$, $\mathcal{P}(f) : B^Y \to B^X$ is the BI algebra homomorphism given by composition with f. For example if $s, t \in \mathcal{P}(Y)$, i.e., $s, t : Y \to B$, then $\mathcal{P}(f)(s) = s \circ f : X \to B$ and $s * t$ is defined pointwise as $(s * t)(y) = s(y) * t(y)$. Equality predicates $=_X$ in $B^{X \times X}$ are given by

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

where $\top$ and $\bot$ 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 are trivially satisfied (Θ is the identity).

There are plenty of examples of complete BI algebras: for any Grothendieck topos $\mathcal{E}$ with an additional symmetric monoidal closed structure, $\mathrm{Sub}_\mathcal{E}(1)$ is a complete BI algebra, and for any monoidal category $\mathcal{C}$ such that the monoid is cover preserving w.r.t. the Grothendieck topology J, $\mathrm{Sub}_{\mathrm{Sh}(\mathcal{C},J)}(1)$ is a complete BI algebra [2, 24].

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 canoncial hyperdoctrine over a topos (as in Example 1). Indeed this is the reason for introducing the more general BI hyperdoctrines.

Theorem 3. *Let $\mathcal{E}$ be a topos and suppose $Sub_\mathcal{E} : \mathcal{E}^{op} \to$ **Poset** is a BI hyperdoctrine. Then the BI structure on each lattice $Sub_\mathcal{E}(X)$ is trivial, i.e., for all $\varphi, \psi \in Sub_\mathcal{E}(X)$, $\varphi * \psi \leftrightarrow \varphi \wedge \psi$.*

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

$$\mathrm{Sub}_\mathcal{E}(Y) = \{\psi \wedge \varphi \mid \psi \in \mathrm{Sub}_\mathcal{E}(X)\}. \tag{7}$$

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

$$\text{For all } \chi, \chi' \in \mathrm{Sub}_\mathcal{E}(Y),\ \chi \vdash_Y \chi' \text{ iff } \chi \vdash_X \chi'. \tag{8}$$

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

$$(\varphi \wedge \psi) *_Y (\varphi \wedge \psi') \leftrightarrow \varphi \wedge (\psi *_X \psi') \tag{9}$$

and

$$(\varphi \wedge \psi) -\!\!*_Y (\varphi \wedge \psi') \leftrightarrow \varphi \wedge (\psi -\!\!*_X \psi'). \tag{10}$$

By assumption, $\mathrm{Sub}_\mathcal{E}(Y)$ forms a BI algebra with connectives $*_Y$, $-\!\!*_Y$ and I_Y, so using the characterization of subobjects of Y given in (7), we get the following rule for each $\chi \in \mathrm{Sub}_\mathcal{E}(X)$:

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

Using (8), (9), and (10) we deduce

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

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

$$\cfrac{\cfrac{\varphi \wedge (\psi *_X \psi') \vdash_X \chi \wedge \varphi}{\varphi \wedge \psi \vdash_X \psi' \mathrel{-\!\!*}_X \chi}}{(\varphi \wedge \psi) *_X \psi' \vdash_X \chi} \tag{11}$$

Inserting $\varphi \wedge (\psi *_X \psi')$ for χ into (11) we get

$$\frac{\varphi \wedge (\psi *_X \psi') \vdash_X \varphi \wedge (\psi *_X \psi')}{(\varphi \wedge \psi) *_X \psi' \vdash_X \varphi \wedge (\psi *_X \psi').} \tag{12}$$

Since the entailment above the line in (12) always holds, we have

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

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 $\chi \leftrightarrow \chi \wedge \varphi$ because we have projections for $*_X$. Using (11) downwards-up, we get

$$\frac{(\varphi \wedge \psi) *_X \psi' \vdash_X (\varphi \wedge \psi) *_X \psi'}{\varphi \wedge (\psi *_X \psi') \vdash_X (\varphi \wedge \psi) *_X \psi'} \tag{13}$$

By (12) and (13) we conclude that for all $\varphi, \psi, \psi' \in \mathrm{Sub}_{\mathcal{E}}(X)$,

$$\varphi \wedge (\psi *_X \psi') \leftrightarrow (\varphi \wedge \psi) *_X \psi'. \tag{14}$$

We already noted that we have projections for $*_X$, so $\top *_X \mathrm{I}_X \vdash_X \mathrm{I}_X$ which means that $\top \leftrightarrow \mathrm{I}_X$. Let ψ be $\top$ in (14), then $\varphi \wedge (\top *_X \psi') \leftrightarrow (\varphi \wedge \top) *_X \psi'$ and so $\varphi \wedge \psi' \leftrightarrow \varphi *_X \psi'$, as claimed. $\square$

In fact, it is possible to make a slight strengthening of Theorem 3. We say that a logic has *full subset types* [12] if the following conditions are satisfied.

- For each formula $\varphi(x_1, \ldots, x_n)$, there is a type $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mid \varphi(x_1, \ldots, x_n)\}$.
- For a term N of type $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mid \varphi(x_1, \ldots, x_n)\}$, in a context Γ ,there is a term $\mathsf{o}(N)$ of type $\tau_1 \times \cdots \times \tau_n$ in Γ.
- The rule
$$\frac{\Gamma, y{:}\{x{:}X \mid \varphi\} \mid \theta[o(y)/x] \vdash \psi[o(y)/x]}{\Gamma, x{:}X \mid \theta, \varphi \vdash \psi} \tag{15}$$
is valid.

One can then show

Theorem 4. *If our notion of predicate BI has full subset types, then for all formulas φ, ψ in a context Γ, we have*

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

The proof may be found in Appendix E.

Corollary 1. *Any BI hyperdoctrine which satisfies the rules for full subset types is trivial.*

Note that the BI hyperdoctrine we use to model separation logic (the standard pointer model) satisfies all of the above except the downward direction of (15). When this is the case, we say that the logic has subset types, but not *full* subset types [12].

5.3 The Pointer Model as a BI Hyperdoctrine

We now show how the semantics of assertions from Section 2.2 is an instance of the general framework we have described.

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

- I is $\{\varnothing\}$
- $U * V := \{h * h' \mid h \in U \wedge h' \in V\} \setminus \{\perp\}$
- $U \mathrel{-\!\!*} V := \bigcup\{W \subseteq H \mid (W * U) \subseteq V\}$.

It can easily be verified directly that this defines a complete Boolean BI algebra; it also follows from more abstract arguments in [24, 2].

Let S be the BI hyperdoctrine induced by the complete Boolean BI algebra $\mathcal{P}(H)$ as in Example 2. To show that the interpretation of separation logic in this BI hyperdoctrine exactly corresponds to the standard pointer model presented above we spell out the interpretation of separation logic in S.

A term t in a context $\Gamma = \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$ is interpreted as a morphism between sets. For example,

- $[\![x_i{:}\tau_1]\!] = \pi_i$,
- $[\![n]\!]$ is the map $[\![n]\!] : [\![\Gamma]\!] \to \{*\} \to \mathbb{Z}$ which sends the unique element of the one-point set $\{*\}$ to n,
- $[\![t_0 \pm t_1]\!] = [\![t_0]\!] \pm [\![t_1]\!] : [\![\Gamma]\!] \to \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, where $[\![t_i]\!] : [\![\Gamma]\!] \to \mathbb{Z}$, for $i = 0, 1$.

The interpretation of a formula φ in a context $\Gamma = \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$ is given inductively. Let $I = [\![\Gamma]\!]$ and write $\overline{\mathrm{v}}$ for elements of I. Then φ is interpreted as an element of $\mathcal{P}(I)$. The interpretation is given in Fig. 6.

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 semantics of terms of type Prop given in Section 2.2, in the sense given by

$$\begin{array}{ll}
[\![t_1 \mapsto t_2]\!](\overline{v}) & = \{h \mid \mathrm{dom}(h) = \{[\![t_1]\!](\overline{v})\} \text{ and } h([\![t_1]\!](\overline{v})) = [\![t_2]\!](\overline{v})\} \\
[\![t_1 = t_2]\!](\overline{v}) & = H \text{ if } [\![t_1]\!](\overline{v}) = [\![t_2]\!](\overline{v}),\ \varnothing \text{ otherwise} \\
[\![\top]\!](*) & = H \\
[\![\bot]\!](*) & = \varnothing \\
[\![\mathsf{emp}]\!](*) & = \{h \mid \mathrm{dom}(h) = \varnothing\} \\
[\![\varphi \wedge \psi]\!](\overline{v}) & = [\![\varphi]\!](\overline{v}) \cap [\![\psi]\!](\overline{v}) \\
[\![\varphi \vee \psi]\!](\overline{v}) & = [\![\varphi]\!](\overline{v}) \cup [\![\psi]\!](\overline{v}) \\
[\![\varphi \to \psi]\!](\overline{v}) & = \{h \mid h \in [\![\varphi]\!](\overline{v}) \text{ implies } h \in [\![\psi]\!](\overline{v})\} \\
[\![\varphi * \psi]\!](\overline{v}) & = [\![\varphi]\!](\overline{v}) * [\![\psi]\!](\overline{v}) \\
 & = \{h_1 * h_2 \mid h_1 \in [\![\varphi]\!](\overline{v}) \text{ and } h_2 \in [\![\psi]\!](\overline{v})\} \setminus \{\bot\} \\
[\![\varphi \mathrel{-\!\!*} \psi]\!](\overline{v}) & = [\![\varphi]\!](\overline{v}) \mathrel{-\!\!*} [\![\psi]\!](\overline{v}) \\
 & = \{h \mid [\![\varphi]\!](\overline{v}) * \{h\} \subseteq [\![\psi]\!](\overline{v})\} \\
[\![\forall x{:}\tau.\varphi]\!](\overline{v}) & = \bigcap_{v_x \in [\![\tau]\!]}([\![\varphi]\!](v_x, \overline{v})) \\
[\![\exists x{:}\tau.\varphi]\!](\overline{v}) & = \bigcup_{v_x \in [\![\tau]\!]}([\![\varphi]\!](v_x, \overline{v}))
\end{array}$$

Fig. 6. Semantics of Assertions in the Hyperdoctrine S

Theorem 5. *Let the predicate $\Delta \vdash \varphi : \mathsf{Prop}$ have free variables in the context $\Delta = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, and let $(v_1, \ldots, v_n) \in [\![\tau_1 \times \ldots \times \tau_n]\!]$. Then,*

$$[\![\varphi]\!](v_1, \ldots, v_n) = [\![\varphi]\!]_{(x_1 \to v_1, \ldots, x_n \to v_n)},$$

where the semantics on the left is the one in Fig. 6 and the semantics on the right is the semantics from Section 2.2.

As a consequence, we of course obtain the well-known result that separation logic is sound for classical first-order BI. The correspondence also shows that it is sensible to extend separation logic to higher-order since the BI hyperdoctrine S soundly models higher-order BI. One can also obtain a correspondence like that of Theorem 5 for other versions of separation logic.

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

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

Let I be the set of sieves on H, i.e., downwards closed subsets of H, ordered by inclusion. This is a complete BI algebra, as can be verified directly or by an abstract argument [2, 24]. Now let T be the BI hyperdoctrine induced by the complete BI algebra I as in Example 2. The interpretation of predicate BI in this BI hyperdoctrine corresponds exactly to the intuitionistic pointer model of separation logic, which is presented using a forcing style semantics in [11].

The permissions model. It is also possible to fit the permissions model of separation logic from [7] into the framework presented here. The main point is that the set of heaps, which in that model map locations to values and permissions, has a binary operation $*$, which makes $(H_\bot, *)$ a partially ordered commutative monoid.

Remark 2. The correspondences between separation logic and BI hyperdoctrines given above illustrate that what matters for the interpretation of separation logic is the choice of BI algebra. Indeed, the main relevance of the topos-theoretic constructions in [24] for models of separation logic is that they can be used to construct suitable BI-algebras (as subobject lattices in categories of sheaves).

6 Other Applications of Higher-order BI

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

6.1 Formalizing Separation Logic

The strength of separation logic has been demonstrated in numerous papers before. In the early days of separation logic, it was shown that it could handle simple programs for copying trees, deleting lists, etc. The first proof of a more realistic program appeared in Yang's thesis [28], 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 [4], and other examples of correctness proofs of non-trivial algorithms may be found in [6]. In all of these papers, different simple extensions of core separation logic were used. For example, Yang used lists and binary trees as parts of his term language, and Birkedal et. al. introduced expression forms for finite sets and relations. It would seem that it is a weakness of separation logic that one has to come up with suitable extensions of it every time one has to prove a new program correct. In particular, it would make machine-verifiable formalizations of such proofs more burdensome and dubious if one would have to alter the underlying logic for every new proof.

The right way to look at these "extensions" is that they are really trivial definitional extensions of one and the same logic, namely the internal logic of the classical BI hyperdoctrine S presented in Section 5. 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 [12]) 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 Set). There is a canonical map $\iota : 2 \to \mathcal{P}(H)$ that 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. *Let φ be an S-predicate over a set X, i.e., a function $\varphi : X \to \mathcal{P}(H)$. Call φ* pure *if φ factors through ι.*

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

$$\begin{tikzcd} X \arrow[rr, "\varphi"] \arrow[dr, "\chi_\varphi"'] & & \mathcal{P}(H) \\ & 2 \arrow[ur, "\iota"'] & \end{tikzcd}$$

commutes. This corresponds to the notion of pure predicate traditionally used in separation logic [26].

The sub-logic of pure predicates is simply the standard classical higher-order logic of Set, and thus it is sound for classical higher-order logic. Hence one can use classical higher-order logic for defining lists, trees, finite sets and relations in the standard manner using pure predicates and prove the standard properties of these structures, as needed for the proofs presented in the papers referred to above. In particular, notice that recursive definitions of predicates, which in [28, 4, 6] are defined at the meta level, can be defined inside the higher-order logic itself. For machine verification one would thus only need to formalize one and the same logic, namely a sufficient fragment of the internal logic of the BI hyperdoctrine (with obvious syntactic rules for when a formula is pure). The internal logic itself is "too big" (it can have class-many types and function symbols, e.g.); hence the need for a fragment thereof, say classical higher-order logic with natural numbers.

6.2 Logical Characterizations of Classes of Assertions

Different classes of assertions, precise, monotone, and pure, were introduced in [26], and it was noted that special axioms for these classes of assertions are valid. Such special axioms were further exploited in [4], where pure assertions were moved in and out of the scope of iterated separating conjunctions, and in [19], where precise assertions were crucially used to verify soundness of the hypothetical frame rule. The different classes of assertions were defined semantically and the special axioms were also validated using the semantics. We now show how the higher-order features of higher-order separation logic may be used to logically characterize the classes of assertions and logically prove the properties earlier taken as axioms. This is, of course, important for machine verification, since it means that the special classes of assertions and their properties can be expressed *in the logic.*

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

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

Precise assertions. The traditional definition of a precise assertion is semantic, in that an assertion q is precise if, and only if, for all states (s, h), there is at most one subheap h_0 of h such that $(s, h_0) \Vdash q$. The following proposition logically characterizes closed precise assertions (at the semantic level, this characterization of precise predicates was mentioned in [18]).

Proposition 1. *The closed assertion q is precise if, and only if, the assertion*

$$\forall p_1, p_2 : \mathsf{Prop}.\ (p_1 * q) \wedge (p_2 * q)\ \ (p_1 \wedge p_2) * q \tag{16}$$

is valid in the BI hyperdoctrine S.

Proof. The "only-if" direction is trivial, so we focus on the other implication. Thus suppose (16) holds for q, and let h be a heap with two different subheaps h_1, h_2 for which $h_i \in [\![q]\!]$. Let p_1, p_2 be canonical assertions that describe the heaps $h \setminus h_1$ and $h \setminus h_2$, respectively. Then $h \in [\![(p_1 * q) \wedge (p_2 * p)]\!]$, so $h \in [\![(p_1 \wedge p_2) * q]\!]$, whence there is a subheap $h' \subseteq h$ with $h' \in [\![p_1 \wedge p_2]\!]$. This is a contradiction. □

One can verify properties that hold 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 (16) holds for q_1, q_2. Then,

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

as desired.

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

Proposition 2. *The closed assertion q is* monotone *if, and only if, the assertion $\forall p{:}\mathsf{Prop}.\ p * q \to q$ is valid in the BI hyperdoctrine S.*

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

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

Proposition 3. *The closed assertion q is pure if, and only if, the assertion*

$$\forall p_1, p_2{:}\mathsf{Prop}.\ (q \wedge p_1) * p_2 \leftrightarrow q \wedge (p_1 * p_2) \tag{17}$$

is valid in the BI hyperdoctrine S.

Proof. Again, the interesting direction here is the "if" implication. Hence, suppose (17) holds for the assertion q, and that $h \in [\![q]\!]$. For any heap h_0, we must then show that $h_0 \in [\![q]\!]$. This is done via the verification of two claims.
Fact 1: For all $h' \subseteq h$, $h' \in [\![q]\!]$. Proof: Let p_1 be a canonical description of h', and p_2 a canonical description of $h \setminus h'$. Then $h \in [\![q \wedge (p_1 * p_2)]\!]$, so $h \in [\![(q \wedge p_1) * p_2]\!]$. This means that there is a split $h_1 * h_2 = h$ with $h_1 \in [\![q \wedge p_1]\!]$ and $h_2 \in [\![p_2]\!]$. But then, $h_2 = h \setminus h'$, so $h_1 = h'$, and thus, $h' \in [\![q]\!]$.
Fact 2: For all $h' \supseteq h$, $h' \in [\![q]\!]$. Proof: Let p_1 and p_2 be canonical descriptions of h and $h' \setminus h$, respectively. Then, $h' \in [\![(q \wedge p_1) * p_2]\!]$, so $h' \in [\![q \wedge (p_1 * p_2)]\!]$, and in particular, $h' \in [\![q]\!]$, as desired.

Using Facts 1 and 2, we deduce $h \in [\![q]\!] \;\Rightarrow\; \varnothing \in [\![q]\!] \;\Rightarrow\; h_0 \in [\![q]\!]$.

7 Related Work

There are references to related work throughout this paper. Here, we give pointers to some more related work.

As mentioned, Parkinson and Bierman propsed a system for reasoning about data abstraction [20], in which they introduce a notion of "abstract predicates". We believe our approach is more straightforward and natural, as we have already argued. Even before that, Reddy gave a semantics for objects and classes in [25], in which he also uses the slogan that data abstraction should be modeled via existential quantification. The main difference from that work compared to the present is that Reddy does not consider a programming language with heap manipulating constructs, but which is higher-order. Also, Reddy uses a more sophisticated type system than ours, and types are interpreted relationally. It is ongoing work to give a relational interpretation to a higher-order programming language with heap-manipulating constructs.

Kohei Honda's group has numerous papers on higher-order imperative languages [1, 10]. The similarities between their work and the present is that both seek to reason about equivalence of programs in programming languages with pointers. Their work is, however, not restricted to our programming language with simple procedures. The main differences include: (i) Their logic does not include a $*$ connective; instead they use predicate logic with equality to keep track of aliasing. This makes local reasoning harder. (ii) The interpretation of triples is not "tight" in their work. For example, the triple

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

is valid in the setting of Honda *et al.*, but not in separation logic. (iii) One of the goal of Honda *et al.*'s work is to show observational equivalence. Although intriguing, we do not aim to answer such questions in the present work. See the long version of [1] for an extensive comparison of their work to separation logic.

8 Conclusion

We have extended the assertion language and specification language of separation logic to higher-order and given a model for it. Further, we have argued that this is a useful extension. In particular, we have shown that we can prove correct programs that use abstract data types with with internal (hidden) resources, and illustrated that universal quantification over predicates can be used to reason about polymorphic data types. Further, we have introduced the notion of a BI hyperdoctrine and we showed that our semantics is an instance of this general concept, inasmuch as our interpretation of predicates coincides with the standard interpretation of predicates in a hyperdoctrine. We also showed that the general concept of hyperdoctrines is needed, since one cannot hope to get interesting models of predicate BI by extending the standard interpretation of predicate logic in toposes.

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

References

1. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. Technical report, Queen Mary, University of London, 2005. Available at http://www.dcs.qmul.ac.uk/~kohei/logics/.
2. B. Biering. On the logic of bunched implications and its relation to separation logic. Master's thesis, University of Copenhagen, 2004.
3. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP 2005: The European Symposium on Programming*, pages 233–247, Edinburgh, Scotland, April 2005.
4. L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 220 – 231, Venice, Italy, January 2004.
5. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 260–269, Chicago, IL, USA, June 2005. IEEE Publishing Company.
6. R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *Proceedings of SPACE 2004*, Venice, Italy, January 2004.
7. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, Long Beach, CA, USA, January 2005. ACM.
8. C. Calcagno, P. W. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(3):557 – 587, 2003.
9. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engler, editor, *Symp. on Semantics of Algorithmic Languages*, pages 102 – 116. Springer-Verlag, Berlin, 1971.
10. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. of Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, Chicago, IL, USA, June 2005. IEEE Press.
11. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, volume 28, London, 2001. ACM - SIGPLAN.
12. B. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1999.
13. F. Lawvere. Adjointness in foundations. *Dialectica*, 23(3/4):281–296, 1969.
14. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL'85)*, pages 37–51, New Orleans, LA, USA, 1985.
15. P. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), June 1999.
16. P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, London, England, September 2004.
17. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European*

Association for Computer Science Logic (CSL 2001), Berlin, Germany, September 2001.

18. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding (work in progress). Extended version of [19], 2003.
19. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 268 – 280, Venice, Italy, 2004.
20. M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proceedings of the 32nd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 247–258, Long Beach, CA, USA, January 2005.
21. A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5: Algebraic and Logical Structures*, chapter 2. Clarendon Press, Oxford, 2001.
22. D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logics Series.* Kluwer, 2002.
23. D. J. Pym. Errata and remarks for the semantics and proof theory of the logic of bunched implications. 2004. Available at http://www.cs.bath.ac.uk/˜pym/.
24. D. J. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.
25. U. Reddy. Objects and classes in Algol-like languages. In *Proc. of The Fifth International Workshop on Foundations of Object-Oriented Languages (FOOL'98)*, San Diego, CA, USA, January 1998.
26. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55 – 74, Copenhagen, Denmark, July 2002.
27. A. Silberschatz and P. Galvin. *Operating Systems Concepts.* World Student Series. Addison-Wesley, Reading, MA, USA, Fifth edition, 1998.
28. H. Yang. *Local Reasoning for Stateful Programs.* PhD thesis, University of Illinois, Urbana-Champaign, 2001.

A Implementation Using slists

Creating a Queue with slists:

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

Enqueing with slists:

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

Dequeing with slists:

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

Disposing a with slist-queue:

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

B Implementation Using dlists

Creating a Queue with dlists:

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

Enqueing with dlists:

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

Dequeing with dlists

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

Disposing with dlist-queue:

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

C Hints for Proof of slistEnque

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

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

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

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

D Hints for Proof of dlistDeque

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

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

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

The following basic properties about the dlist predicate have been used in our proof of the dlist implementation of priority queues. Some of these are also listed in [26].

$$\begin{array}{l}
\mathsf{dlist}(\alpha,i,i',j,j') \wedge i \neq j \Rightarrow \mathsf{dlist}(\alpha,i,i',j,j') \wedge \alpha \neq \epsilon \\
\mathsf{dlist}(\alpha,i,i',j,j') \wedge \alpha \neq \epsilon \Rightarrow \\
\quad \exists p,v,k,\alpha'.\ \alpha = (p,v)\cdot\alpha' \wedge i \mapsto (p,v,k,i') * \mathsf{dlist}(\alpha',k,i,j,j') \\
i = i' \wedge j = j' \wedge \mathsf{emp} \Rightarrow \mathsf{dlist}(\epsilon,i,i',j,j') \\
\mathsf{dlist}(\alpha,i,i',k,k') * \mathsf{dlist}(\alpha',k,k',j,j') \Rightarrow \mathsf{dlist}(\alpha\cdot\alpha',i,i',j,j')
\end{array}$$

E Proof of Theorem 4

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

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

The following rule can be deduced

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

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

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

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

$$x{:}X \mid \varphi \wedge \psi \dashv\vdash \exists_o.\ \psi[o(y)/x]. \tag{18}$$

This is done by

$$\begin{array}{c}
x{:}X \mid \exists_o.\ \psi[o(y)/x] \vdash \exists_o.\ \psi[o(y)/x] \\
\hline\hline
y{:}\{x{:}X \mid \varphi\} \mid \psi[o(y)/x] \vdash (\exists_o.\ \psi[o(y)/x])[o(y)/x] \\
\hline\hline
x{:}X \mid \psi \wedge \varphi \vdash \exists_o.\ \psi[o(y)/x]
\end{array}\ ,$$

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

$$\begin{array}{c}
y{:}\{x{:}X \mid \varphi\} \mid \psi[o(y)/x] \vdash \psi[o(y)/x] \\
\hline\hline
x{:}X \mid \exists_o.\ \psi[o(y)/x] \vdash \psi
\end{array}$$

and

$$\begin{array}{c}
x{:}X \mid \varphi \wedge \psi \vdash \varphi \\
\hline\hline
y{:}\{x{:}X \mid \varphi\} \mid \psi[o(y)/x] \vdash \varphi[o(y)/x] \\
\hline\hline
x{:}X \mid \exists_o.\ \psi[o(y)/x] \vdash \varphi
\end{array}\ ,$$

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

$$\begin{array}{c}
y{:}\{x{:}X \mid \varphi\} \mid \chi[o(y)/x] \vdash \psi[o(y)/x] \\
\hline\hline
x{:}X \mid \exists_o.\ \chi[o(y)/x] \vdash \exists_o.\ \psi[o(y)/x]
\end{array}\ . \tag{19}$$

which is shown by

$$\begin{array}{c}
y{:}\{x{:}X \mid \varphi\} \mid \chi[o(y)/x] \vdash \psi[o(y)/x] \\ \hline\hline
x{:}X \mid \chi \wedge \varphi \vdash \psi \\ \hline\hline
x{:}X \mid \chi \wedge \varphi \vdash \psi \wedge \varphi \\ \hline\hline
x{:}X \mid \exists_o .\ \chi[o(y)/x] \vdash \exists_o .\ \psi[o(y)/x]
\end{array} \ ,$$

where the last derivation follows from (18). We then have

$$\begin{array}{c}
y{:}\{x{:}X \mid \varphi\} \mid \psi[o(y)/x] * \psi'[o(y)/x] \vdash \chi[o(y)/x] \\ \hline\hline
y{:}\{x{:}X \mid \varphi\} \mid \psi[o(y)/x] \vdash \psi'[o(y)/x] \mathrel{-\!\!*} \chi[o(y)/x]
\end{array} \ ,$$

i.e.,

$$\frac{y{:}\{x{:}X \mid \varphi\} \mid \psi[o(y)/x] \vdash (\psi' \mathrel{-\!\!*} \chi)[o(y)/x]}{=} \ \dot{}\, y{:}\{x{:}X \mid \varphi\} \mid (\psi * \psi')[o(y)/x] \vdash \chi[o(y)/x]$$

By (19) we then get

$$\begin{array}{c}
x{:}X \mid \exists_o .\ (\psi * \psi')[o(y)/x] \vdash \exists_o .\ \chi[o(y)/x] \\ \hline\hline
x{:}X \mid \exists_o .\ \psi[o(y)/x] \vdash \exists_o .\ (\psi' \mathrel{-\!\!*} \chi)[o(y)/x]
\end{array} \ ,$$

which by 18 gives us

$$\begin{array}{c}
x{:}X \mid \varphi \wedge (\psi * \psi') \vdash \varphi \wedge \chi \\ \hline\hline
x{:}X \mid \varphi \wedge \psi \vdash \varphi \wedge (\psi' \mathrel{-\!\!*} \chi)
\end{array} \ .$$

This entails the following

$$\begin{array}{c}
x{:}X \mid \varphi \wedge (\psi * \psi') \vdash \chi \\ \hline\hline
x{:}X \mid \varphi \wedge (\psi * \psi') \vdash \chi \wedge \varphi \\ \hline\hline
x{:}X \mid \varphi \wedge \psi \vdash \varphi \wedge (\psi' \mathrel{-\!\!*} \chi) \\ \hline\hline
x{:}X \mid \varphi \wedge \psi \vdash \psi' \mathrel{-\!\!*} \chi \\ \hline\hline
x{:}X \mid (\varphi \wedge \psi) * \psi' \vdash \chi
\end{array} \ .$$

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 $\mathrm{I} \wedge (\mathrm{I} * \top) \dashv\vdash (\mathrm{I} \wedge \mathrm{I}) * \top$, that is, $\mathrm{I} \dashv\vdash \top$, which in return yields $\varphi \wedge (\top * \psi') \dashv\vdash (\varphi \wedge \top) * \psi'$, i.e., $\varphi \wedge \psi' \dashv\vdash \varphi * \psi'$. □