Higher-Order Separation Logic and Abstraction

Lars Birkedal⋆ and Noah Torp-Smith⋆

Department of Theoretical Computer Science, IT University of Copenhagen
{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 force of this extension is illustrated via examples; specifically we demonstrate that existential and universal quantification correspond to abstract data types and parametric data types, respectively. Moreover, we show how to express *invariance* of programs in our framework.

1 Introduction

Variants of the recent formalism of *separation logic* [16, 7] have been used to prove correct many interesting algorithms involving pointers, both in sequential and concurrent settings [11, 17, 3]. It is a Hoare-style program logic, and its main advantage over traditional program logics is that it facilitates *local reasoning*.

In a recent paper [2] with B. Biering we present a precise correspondence between separation logic and a simple notion of *predicate* BI, extending the earlier correspondence given between part of separation logic and *propositional* BI [14]. Moreover, we introduce the notion of a BI hyperdoctrine and show that it soundly models classical and intuitionistic first- and higher-order predicate BI, and use it to show that we may easily extend the assertion language of separation logic to *higher-order*. Being a program logic, the real force of separation logic comes not only from its language of assertions, which we focused our attention on in [2], but also from its language of *specifications* (Hoare triples). In the present paper we extend the higher-order model of assertions from *loc. cit.* to a model of higher-order specifications, and provide a sound collection of inference rules for deriving valid specifications. It turns out that it is technically straightforward to do so; and we believe this serves to emphasize that our model of higher-order separation logic is very natural indeed. For concreteness we choose to consider specifications for a language that has been considered often in recent papers on separation logic, namely a language with first-order functions.

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. This shows that there is no need to extend separation logic with special syntax, such as the recently suggested abstract predicates of

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

Parkinson and Bierman [13], 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 very 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 first-order language here to communicate more easily the ideas of higher-order separation logic. However, in Sec. 4.3, we sketch how to extend our ideas to a small object-oriented language. In future work, we will show how to also make use of higher-order separation logic for a higher-order language, specifically by extending joint work with Yang on separation logic typing for idealized algol [4]. Here we also show how to use higher-order separation lgoic for defining assertions by means of fixed points which *exist in the logic* because of higher-orderness.

Before proceeding with the technical development we give an intuitive description thereof. In Section 2 we first present the model of the assertions of higher-order separation logic in concrete terms; it corresponds to the internal logic of the BI-hyperdoctrine in [2, Section 3]) but this paper does not presuppose familiarity with [2] — we present the model in concrete terms without using hyperdoctrines. Next we present higher-order separation logic specifications in Section 3. The idea is simply to consider a standard kind of predicate logic with atomic predicates corresponding to Hoare triples, in the spirit of Reynold's specification logic [15]. Since we have chosen to focus on a relatively simple programming language (with only first-order functions), the logic of specifications will also be fairly simple, e.g., we do not include implications between specifications, and our main focus will be on existentially and universally quantified specifications. The logic allows quantification over variables as in the assertion language, including quantification over variables ranging over functions and predicates. In Section 4 we argue that the higher-order logic is effective; in particular we show that it can be used to prove correct programs operating on abstract data types. Finally, we conclude in Section 5.

2 Assertions of Higher-Order Separation Logic

In this section we briefly recall the syntax and semantics of the assertion language of higher-order separation logic as introduced in [2]. It is a higher-order logic over a typed term language; formulas correspond to terms of types Prop and there are operations on Prop corresponding to the usual connectives of propositional separation logic.

The set of types of the term language is generated by the following grammer.

$$\tau ::= \mathsf{Prop} \mid \mathsf{Int} \mid \tau \times \tau \mid \tau^{\tau}$$

The semantics of these types is the standard semantics in the cartesian closed category **Set** of sets and functions; we just need to give the interpretation of the basic types. The interpretation of the type Int is given by $[\![\mathsf{Int}]\!] = \mathbb{Z}$, and the interpretation of the type Prop is given by $[\![\mathsf{Prop}]\!] = \mathcal{P}(H)$, where H is the usual set of *heaps*, that is, finite maps from natural numbers to integers:

$$H \stackrel{\text{def}}{=} \mathbb{N} \rightharpoonup_{\text{fin}} \mathbb{Z}.$$

Recall that $\mathcal{P}(H)$, ordered by inclusion, is a complete Boolean BI-algebra [2]. Terms in context are given in a standard way by a judgement $\Delta \vdash t : \tau$, where Δ is a list of assignments of types to variables, and the free variables in t are included in Δ. The semantics of such a term is defined in the standard manner as a function from the semantics of Δ, denoted $[\![\Delta]\!]$, to $[\![\tau]\!]$, the semantics of the type τ. The semantics $[\![\Delta]\!]$ of Δ is the set of functions η with domain the domain of Δ that assigns a value in $[\![\tau]\!]$ to each variable $x : \tau$ in Δ.

In particular, the semantics of formulas in context $\Delta \vdash \varphi : \mathsf{Prop}$, corresponding to terms of type Prop, is given by a function

$$[\![\Delta \vdash \varphi : \mathsf{Prop}]\!] : [\![\Delta]\!] \to \mathcal{P}(H),$$

defined by induction on φ. Formulas are generated by:

$$\begin{aligned}\varphi ::= {} & t = t \mid \top \mid \varphi \wedge \varphi \mid \bot \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid \mathsf{emp} \mid \varphi * \varphi \mid \varphi \mathrel{-\!\!*} \varphi \\ \mid {} & e \mapsto e \mid \forall x : \tau.\varphi \mid \exists x : \tau.\varphi,\end{aligned}$$

where e ranges over a term of type Int, and τ can be *any* type, including a type of predicates such as $\mathsf{Prop}^{\mathsf{Int}\times\mathsf{Int}}$.

The semantics of all the connectives besides quantification is given pointwise using the BI-structure on $\mathcal{P}(H)$. The quantifiers are interpreted using corresponding set-theoretic quantification. For example,

$$[\![\Delta \vdash \forall x : \tau.\varphi : \mathsf{Prop}]\!] = \eta \mapsto \bigcap_{v \in [\![\tau]\!]} ([\![\Delta, x : \tau \vdash \varphi : \mathsf{Prop}]\!](\eta[x \mapsto v])).$$

The logic for assertions allows one derive conclusions of the form $\varphi \vdash_\Delta \varphi'$. Such a judgement is *valid* if, for all $\eta \in [\![\Delta]\!]$, we have that $[\![\Delta \vdash \varphi : \mathsf{Prop}]\!]\eta \subseteq [\![\Delta \vdash \varphi' : \mathsf{Prop}]\!]\eta$. The rules from standard predicate logic plus the standard rules from first-order separation logic for emp, $e \mapsto e'$, $\varphi * \varphi$, and $\varphi \mathrel{-\!\!*} \varphi$ are all sound.

3 Specifications of Higher-Order Separation Logic

In this section we define the syntax and semantics of specifications of higher-order separation logic. There can, of course, be several variants, depending on the programming language considered. For concreteness and because it serves well to illustrate the effectiveness of higher-order separation logic in a simple way, we choose to use a simple imperative langauge with first-order functions that has been used in several recent papers and has been precisely defined in [13].

We will not repeat the full definition of the language here, but just recall the following. The operational semantics of the language includes a *semantic* *function environment* Π, which 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. The operational semantics is then specified by a judgment

$$(\Pi, C, \eta, h) \Downarrow (\eta', h'),$$

where Modifies$(C) \subseteq dom(\eta)$. The details are given in [13], but note that they have two separate stacks for program variables and so-called abstract variables. We just use one stack for variables of all types here. As in standard separation logic, there is a **wrong** state which denotes memory fault and this induces a notion of a *safe* state.

With assertions and the programming language in place, we turn to specifications, i.e., to the program logic. One way to think about our approach is that we basically seek to define a standard logic, where predicates are specifications, in particular basic predicates are simple Hoare-triples, and where variables range over terms as in the previous section, in the spirit of Reynold's specification logic [15]. Since we have chosen to focus on a relatively simple programming language with only first-order functions, the logic of specifications will also be fairly simple, e.g., it does not include implications between specifications. In future work, we will extend the ideas presented here to a higher-order programming language with a separation-logic typing discipline [4].

First we define the syntax of our program logic, and subsequently we give the semantics of several judgments. The programming language includes calls to functions, and as in [13, 12], our logic includes assumptions about these functions, formulated as specifications. Thus, we will have triples in assumptions as well as in conclusions, but the triples will be slightly different, inasmuch as the basic specifications for functions will have the form $\{P\}\ k\ \{Q\}$ (where k is a function name), whereas those in conclusions will have form $\{P\}\ c\ \{Q\}$, for c a command. We will allow conjunction and quantifications for both kinds of specifications.

The specifications that involve function names are called *function specifications*, and they are generated by the grammar (given a context Δ)

$$\gamma ::= \{P\}\ k\ \{Q\} \mid \gamma \wedge \gamma \mid \exists x : \tau.\ \gamma \mid \forall x : \tau.\ \gamma,$$

where P, Q have type Prop in the context Δ. (Formally, we write $\Delta \vdash \gamma : \mathsf{FSpec}$ to state that γ is well-formed in Δ, and this judgement is defined in the obvious way.)

The specifications involving the actual programs are simply called *specifications*. Given a context Δ and a semantic function environment Π in which the program c is well-formed, the grammar for specifications is

$$\delta ::= \{P\}\ c\ \{Q\} \mid \delta \wedge \delta \mid \exists x : \tau.\ \delta \mid \forall x : \tau.\ \delta,$$

where again, P, Q must be well-formed in Δ, all functions called in c must be defined in Π, and all variables in Modifies(c) must be defined to have type Int in Δ. (Formally, we write $\Delta, \Pi \vdash \delta : \mathsf{Spec}$ to state that δ is well-formed in Δ, and this judgement is also defined in the obvious way.)

The semantics of the program logic is given in a couple of straightforward steps. First, the semantics of $\Delta, \Pi \vdash \delta$ is a map from $[\![\Delta]\!]$ to **true** or **false**, and it is defined by induction:

$$
\begin{array}{lll}
[\![\Delta, \Pi \vdash \{P\}\ c\ \{Q\}]\!]\eta & \text{iff} & \text{for all } h \in [\![P]\!]\eta, \\
& & \quad - (\Pi, C, \eta, h) \text{ is safe, and} \\
& & \quad - \text{if } (\Pi, C, \eta, h) \Downarrow (\eta', h'), \text{ then } h' \in [\![Q]\!]\eta'; \\
[\![\Delta, \Pi \vdash \delta_1 \wedge \delta_2]\!]\eta & \text{iff} & [\![\Delta, \Pi \vdash \delta_1]\!]\eta \text{ and } [\![\Delta, \Pi \vdash \delta_2]\!]\eta; \\
[\![\Delta, \Pi \vdash \exists x : \tau.\ \delta]\!]\eta & \text{iff} & [\![\Delta, \Pi \vdash \delta]\!]\eta[x \to v] \text{ for some } v \in [\![\tau]\!]; \\
[\![\Delta, \Pi \vdash \forall x : \tau.\ \delta]\!]\eta & \text{iff} & [\![\Delta, \Pi \vdash \delta]\!]\eta[x \to v] \text{ for all } v \in [\![\tau]\!].
\end{array}
\tag{1}
$$

Next, we say that a specification $\Delta, \Pi \vdash \delta$ is *valid*, denoted $\Delta, \Pi \models \delta$, if $[\![\Delta, \Pi \vdash \delta]\!]\eta$ is **true** for all $\eta \in [\![\Delta]\!]$.

The semantics of function specifications is defined similarly, the only difference being the base case:

$$[\![\Delta, \Pi \vdash \{P\}\ k\ \{Q\}]\!]\eta \text{ iff } [\![\Delta, \Pi \vdash \{P\}\ c\ \{Q\}]\!]\eta, \text{ where } \Pi(k) = (\bar{x}, c).$$

An *environment* is a list of function triples. The meaning of such an environment is similar to the one presented in [13]: for $\eta \in [\![\Delta]\!]$, we simply say that $[\![\Delta; \Pi \vdash \Gamma]\!]\eta$ if $[\![\Delta; \Pi \vdash \gamma]\!]\eta$, for all $\gamma \in \Gamma$.

Finally, under a given context, a specification is valid in an environment if truth of the environment entails truth of the specification, for any well-formed semantic function environment:

$$\Delta; \Gamma \models \delta \quad \text{iff} \quad \forall \Pi. \forall \eta \in [\![\Delta]\!].\ [\![\Delta; \Pi \models \Gamma]\!]\eta \Rightarrow [\![\Delta; \Pi \models \delta]\!]\eta. \tag{2}$$

This definition gives rise to a program logic, using which we can derive valid triples of the form $\Delta; \Gamma \vdash \delta$. The rules are listed in Fig. 1 (we have omitted some obvious side-conditions of, e.g., the form $\Delta; \Gamma \vdash P : \mathsf{Prop}$). Most of the rules are as for standard (first-order) separation logic. The new rules are the rules for $\exists$ and $\forall$, which are really the standard rules for existential and universal quantification. They are formulated as "double-rules," that is, they may be applied both downwards and upwards. As is standard (see, e.g., [8, Ch. 4]), the rules for $\exists$ and $\forall$ are equivalent to the pairs of rules

$$\frac{\Delta \vdash M{:}\tau \quad \Delta; \Gamma \vdash \gamma[M/x]}{\Delta; \Gamma \vdash \exists x : \tau.\ \gamma} \qquad \frac{\Delta; \Gamma \vdash \exists x{:}\tau.\ \gamma \quad \Delta, x{:}\tau; \Gamma', \gamma \vdash \gamma'}{\Delta; \Gamma, \Gamma' \vdash \gamma'}\ x \notin FV(\Gamma', \gamma')$$

and

$$\frac{\Delta, x{:}\tau; \Gamma \vdash \gamma}{\Delta; \Gamma \vdash \forall x{:}\tau.\ \gamma}\ x \notin FV(\Gamma) \qquad \frac{\Delta \vdash M{:}\tau \quad \Delta; \Gamma \vdash \forall x{:}\tau.\ \gamma}{\Delta; \Gamma \vdash \gamma[M/x]}\ ,$$

respectively.

Theorem 1 (Soundness). *If a triple $\Delta; \Gamma \vdash \delta$ can be derived from the rules in Fig. 1, then it is valid in the sense of* (2).

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

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

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

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

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

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

$$\frac{\Delta;\Gamma \vdash \{P\}\ c_1\ \{P'\} \quad \Delta;\Gamma \vdash \{P'\}\ c_2\ \{Q\}}{\Delta;\Gamma \vdash \{P\}\ c_1;c_2\ \{Q\}} \quad \frac{\Delta, x : \mathsf{Int};\Gamma \vdash \{P \wedge x = nil\}\ c\ \{Q\}}{\Delta;\Gamma \vdash \{P\}\ \mathbf{newvar}\ x\ \mathbf{in}\ C\ \mathbf{end}\ \{Q\}}$$

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

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

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

$$\frac{\Delta, x : \tau;\Gamma,\gamma \vdash \delta'}{\Delta;\Gamma,\exists x : \tau.\ \gamma \vdash \delta'}\ x \notin FV(\delta',\Gamma) \quad \frac{\Delta;\Gamma \vdash \{P\}\ c\ \{Q\}}{\Delta;\Gamma \vdash \{P * P'\}\ c\ \{Q * P'\}}\ \mathrm{Mod}(c) \cap FV(P') = \varnothing$$

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

Fig. 1. Program Logic

The proof of the theorem is straightforward, but there is nothing wrong with that. It is in part due the fact that we use standard logical rules for quantification (a good thing) and in part due to our choice of language, a simple first-order language, which we have chosen to illustrate the ideas of higher-order separation logic most clearly. However, we emphasize that one can use the same ideas also for more complicated langauges, e.g., for a language with class-based objects (see Section 4.5) and in connection with separation-logic typing for idealized algol [4], as we will show elsewhere. The effectiveness of the rules (see the following section) stems from the fact that we allow higher-order quantification.

We may derive an important rule for abstract function definitions from the rules in Fig. 1; it corresponds to a rule in [13]. In the following section we show

how to use it for data abstraction.

$$\frac{\begin{array}{c}\Delta \vdash \hat{P} : \tau \\ \Delta; \Gamma \vdash \{P_1[\hat{P}/x]\}\ c_1\ \{Q_1[\hat{P}/x]\} \\ \vdots \\ \Delta; \Gamma \vdash \{P_n[\hat{P}/x]\}\ c_n\ \{Q_n[\hat{P}/x]\} \\ \Delta; \Gamma, \exists x{:}\tau.(\{P_1\}k_1\{Q_1\} \wedge \cdots \wedge \{P_n\}k_n\{Q_n\}) \vdash \{P\}\ c\ \{Q\}\end{array}}{\Delta; \Gamma \vdash \{P\}\ \mathbf{let}\ k_1(\bar{x}_1) = c_1, \ldots, k_1(\bar{x}_n) = c_n\ \mathbf{in}\ c\ \mathbf{end}\ \{Q\}}\ x \notin FV(P, Q) \quad (3)$$

The rule could easily be generalized to a vector $x_1, \ldots, x_j$ of variables and j predicates $\hat{P}_1, \ldots \hat{P}_j$, but for presentational purposes, we just present it with one variable. The rule is easily seen to be derivable by instantiating the function definition rule with $P_1[\hat{P}/x]/Q_1[\hat{P}/x]$, and applying the double rule for $\exists$ to the premise about the client code.

The advantage of this rule is, of course, that the reasoning about the client code C does not depend upon the specific $\hat{P}$, so, in a sense, the reasoning is "abstract". This point will be illustrated by the examples in Section 4.1.

4 On the Effectiveness of Higher-order Separation Logic

In this section we argue that higher-order separation logic is very effective. We do so by explaining how it can be used to model data abstraction via existential quantification over predicates. The slogan is, of course, that abstract predicates are existential predicates, corresponding to the slogan that "abstract types have existential types" [10]. Thus we show that when we use higher-order separation logic, then there is no need to extend separation logic with special syntax, such as the syntax for abstract predicates in the recent work of Parkinson and Bierman [13] — the abstract predicates are directly definable *in the logic*. As also argued in [2], this is of course important for machine-verifiable formalizations of separation logic. We further show that universal quantification over predicates can be used to prove the correctness of polymorphic operations on data types (such as the reversal of a list), and that certain useful recursive predicates may be defined by fixed points definable *in the logic*.

4.1 Abstract Data Types via Existential Quantification

We now show how to prove correct programs using abstract data types with internal hidden (abstract) resource invariants. We do so by considering some examples from a recent paper [13], in which Parkinson and Bierman nicely explains that one needs some kind of abstract predicates for making modular proofs of correctness for abstract data types that may be instantiated by several clients. Here we show that we may simply take abstract predicates to be existentially quantified predicates.

We begin by calling to mind the connection pool example of Parkinson and Bierman [13]. When programming with databases, the object pool design pattern is often used to keep the number of constructed connections low. In the simple implementation illustrated here, a *pool* of connections are kept in a linked list, and the pool makes sure that connections to the database are not used after they are returned to the pool, via ownership transfer. We assume two system routines, **consConn** and **dispConn**, to construct and dispose a database connection, with the following specifications:

$$\{\mathsf{emp}\}\ \mathbf{consConn}(s)\ \{\mathsf{conn}(ret, s)\}$$
$$\{\mathsf{conn}(x, s)\}\ \mathbf{dispConn}(x)\ \{\mathsf{emp}\}$$

The specifications use a predicate $\mathsf{conn}(x, s)$ to express that x holds a handle to a connection to the database s. As mentioned, the pool will be represented in the heap by a linked list. Using the same implementations of operations as in [13] (call them $c_{\mathbf{consPool}}$, etc.), we can show the following specifications using our program logic (for the definition of clist, see Section 4.4 — for now, just assume that it abbreviates a linked list predicate as in [13]):

$$\begin{array}{l}
\Delta; \Gamma \vdash \{\mathsf{emp}\}\ c_{\mathbf{consPool}}(s)\ \{\exists i.x \mapsto i, s * \mathsf{clist}(i, s)\} \\
\Delta; \Gamma \vdash \{\exists i.x \mapsto i, s * \mathsf{clist}(i, s)\}\ c_{\mathbf{dispPool}}(x)\ \{\mathsf{emp}\} \\
\Delta; \Gamma \vdash \{\exists i.x \mapsto i, s * \mathsf{clist}(i, s)\}\ c_{\mathbf{getCon}}(x) \\
\qquad\qquad\qquad\qquad \{(\exists i.x \mapsto i, s * \mathsf{clist}(i, s)) * \mathsf{conn}(ret, s)\} \\
\Delta; \Gamma \vdash \{(\exists i.x \mapsto i, s * \mathsf{clist}(i, s)) * \mathsf{conn}(y, s)\}\ c_{\mathbf{freeCon}}(x, y) \\
\qquad\qquad\qquad\qquad \{(\exists i.x \mapsto i, s * \mathsf{clist}(i, s))\}
\end{array} \tag{4}$$

As in [13], we note that a client program c using the connection pool, should not rely on the implementation of the connection pool. Thus we verify clients c as follows:

$$\begin{array}{l}
\Delta, \mathsf{cpool} : \mathsf{Prop}^{\mathsf{Int}\times\mathsf{Int}}, \mathsf{conn} : \mathsf{Prop}^{\mathsf{Int}\times\mathsf{Int}}; \\
\Gamma, \{\mathsf{emp}\}\ \mathbf{consPool}(s)\ \{\mathsf{cpool}(ret, s)\}, \\
\{\mathsf{cpool}(x, s)\}\ \mathbf{dispPool}(x)\ \{\mathsf{emp}\}, \\
\{\mathsf{cpool}(x, s)\}\ \mathbf{getCon}(x)\ \{\mathsf{cpool}(x, s) * \mathsf{conn}(ret, s)\}, \\
\{\mathsf{cpool}(x, s) * \mathsf{conn}(y, s)\}\ \mathbf{freeCon}(x, y)\ \{\mathsf{cpool}(x, s)\} \vdash \{P\}\ c\ \{Q\},
\end{array} \tag{5}$$

where the $\{P\}\ c\ \{Q\}$ on the right of the $\vdash$ are the derivations from [13].[1] By rule (3) we obtain

$$\dfrac{\mathcal{D}_{c_{\mathbf{consPool}}} \quad \mathcal{D}_{c_{\mathbf{dispPool}}} \quad \mathcal{D}_{c_{\mathbf{getCon}}} \quad \mathcal{D}_{c_{\mathbf{freeCon}}} \quad \mathcal{D}_{c}}{\begin{array}{l}
\Delta; \Gamma \vdash \{P\} \\
\mathbf{let} \\
\qquad \mathbf{consPool}(x) = c_{\mathbf{consPool}} \\
\qquad \mathbf{dispPool}(x) = c_{\mathbf{dispPool}} \\
\qquad \mathbf{getCon}(x, s) = c_{\mathbf{getCon}} \\
\qquad \mathbf{freeCon}(x, s) = c_{\mathbf{freeCon}} \\
\mathbf{in}\ c\ \{Q\}
\end{array}},$$

[1] Note that according to the abstract function definition rule in [13], cpool is not allowed to occur in the specification of the conclusion (i.e,. in P and Q), so the examples in *loc.cit.* should dispose the pool using a **dispPool** operation.

where in the premise, the first four derivations are obtained by substitution (we just show one of them) from the derivations in (4):

$$\frac{\Delta; \Gamma \vdash \{\mathsf{emp}\}\ c_{\mathbf{consPool}}(s)\ \{\exists i.x \mapsto i, s * \mathsf{clist}(i, s)\}}{\Delta; \Gamma \vdash \begin{array}{l}\{\mathsf{emp}[\lambda x{:}\mathsf{Int}, s{:}\mathsf{Int}.\exists i.x \mapsto i, s * \mathsf{clist}(i, s)/\mathsf{cpool}]\} \\ \quad c_{\mathbf{consPool}}(s) \\ \{\mathsf{cpool}(ret, s)[\lambda x{:}\mathsf{Int}, s{:}\mathsf{Int}.\exists i.x \mapsto i, s * \mathsf{clist}(i, s)/\mathsf{cpool}]\}\end{array}}$$

The derivation $\mathcal{D}_c$ is just the following instantiation of the rule for $\exists$ in Fig. 1, and the derivation (5):

$$\frac{\begin{array}{l}\Delta, \mathsf{cpool} : \mathsf{Prop}^{\mathsf{Int}\times\mathsf{Int}}, \mathsf{conn} : \mathsf{Prop}^{\mathsf{Int}\times\mathsf{Int}}; \\ \Gamma, \{\mathsf{emp}\}\ \mathbf{consPool}(s)\ \{\mathsf{cpool}(ret, s)\}, \\ \{\mathsf{cpool}(x, s)\}\ \mathbf{dispPool}(x)\ \{\mathsf{emp}\}, \\ \{\mathsf{cpool}(x, s)\}\ \mathbf{getCon}(x)\ \{\mathsf{cpool}(x, s) * \mathsf{conn}(ret, s)\}, \\ \{\mathsf{cpool}(x, s) * \mathsf{conn}(y, s)\}\ \mathbf{freeCon}(x, y)\ \{\mathsf{cpool}(x, s)\} \vdash \{P\}\ c\ \{Q\}\end{array}}{\begin{array}{l}\Delta; \\ \Gamma, \exists \mathsf{cpool} : \mathsf{Prop}^{\mathsf{Int}\times\mathsf{Int}}, \mathsf{conn} : \mathsf{Prop}^{\mathsf{Int}\times\mathsf{Int}}. \\ (\{\mathsf{emp}\}\ \mathbf{consPool}(s)\ \{\mathsf{cpool}(ret, s)\}, \\ \{\mathsf{cpool}(x, s)\}\ \mathbf{dispPool}(x)\ \{\mathsf{emp}\}, \\ \{\mathsf{cpool}(x, s)\}\ \mathbf{getCon}(x)\ \{\mathsf{cpool}(x, s) * \mathsf{conn}(ret, s)\}, \\ \{\mathsf{cpool}(x, s) * \mathsf{conn}(y, s)\}\ \mathbf{freeCon}(x, y)\ \{\mathsf{cpool}(x, s)\}) \vdash \{P\}\ c\ \{Q\}\end{array}}$$

Note the straightforward use of existentially quantified predicates for expressing the abstract predicates.

An important point to note here is that rule (3) allows us to interchange implementations, *without effecting the reasoning about client programs*. Indeed, suppose we for some reason wanted to represent the connection pool by a doubly-linked list, and that predicate dclist expresses a doubly-linked list in the same way as clist represents a singly-linked structure (see [16] for a doubly-linked list predicate). Then rule (3) says that all we need to show are the specifications from (4) with clist replaced by dclist. The argument for the client is completely independent of the choice of representation, as long as there is some cpool predicate with associated operations with the right properties. Obviously, this argument of representation independence uses no concept of "logical relation" or such as one does when proving representation independence results for abstract types, since we are here dealing with specifications rather than types.

4.2 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 [12] is parametric in a predicate P *at the meta-level*. We show that in higher-order separation logic, the parameterization may be expressed *in the logic*. To that end, consider the following version of the parametric list predicate from [12].

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

The predicate P is required to hold for each element of the sequence β involved. Different instantiations of P yields different versions of the list, with different amounts of data stored in the list. If $P \equiv \mathsf{emp}$, then plain values are stored (no ownership transfer to the queue module in [12]), and if $P \equiv x \mapsto -,-$, then addresses of cells are stored in the queue (ownership of the cells is tranferred in and out of the queue [12]).

Returning to our 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*.

Suppose **listRev** is the list reversal program given in the introduction of [16]. Then one can easily show the specification

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

By the introduction rule for universal quantification we obtain the specification

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

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

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

For such parametric operations on polymorphic data types to be really useful, one would of course prefer a higher-order programming language instead of the first-order language considered here. Then one could, e.g., program the usual **map** function on lists, and provide a single parametric correctness proof for it. 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 [4].

4.3 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:

$$\{q\}\ c\ \{q' * q\}. \tag{6}$$

This does not work, however, unless q is *strictly exact* [16] (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 following specification

$$\forall p{:}\mathsf{Prop}.\{q \wedge p\}\ c\ \{q' * p\}, \tag{7}$$

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 (7) is stronger than (6): by instantiation p with q in (7) we get (6). Thus if we wish to prove (6), then we may prove something stronger (7), which may be easier to prove (c.f., strengthening an induction hypothetis), and then derive the desired.

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

4.4 Predicates via Fixed Points

Recall the clist predicate from the connection pool example above. It is required to satisfy the following recursive equation:

$$\mathsf{clist} = \lambda(x, s).x \doteq \mathit{null} \vee (\exists j, k.\ x \mapsto j, k * \mathsf{conn}(j, s) * \mathsf{clist}(k, s)).$$

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,

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

is the maximal fixed point for φ.

4.5 Higher Order Separation Logic for a Class-based Language

Parkinson and Bierman [13] not only give abstraction results for the first-order language considered above, but also treat class-based objects, inasmuch as they give a sound logic for a subset of Java. We now sketch that one can also use a model of higher-order separation logic for modeling abstraction in such class-based languages. The underlying storage model of the class-based language considered in [13] is different from the storage model of the first-order language considered above, and thus we should correspondingly change the model of higher-order separation logic to reflect the change of storage model. Luckily, that is straightforward, by the following result, which follows from the analysis in [2].

Proposition 1 *If $(H, \cdot)$ is a partial commutative monoid, then $\mathcal{P}(H)$ can be viewed as a complete BI-algebra in a canonical way. Hence there is a natural BI-hyperdoctrine $\mathbf{Set}^{op} \to \mathcal{P}(H)$.*

Thus, from the storage model $\mathcal{H}$ in [13, Def'n 4.2], we easily obtain a BI-hyperdoctrine over **Set**, and the basic predicates for the class-based language from [13] can be interpreted in this hyperdoctrine in a natural way. Thus one can use this hyperdoctrine to give a higher-order separation logic model for the class-based langauge in [13] and deal with abstraction and the abstract predicate families of [13] via existential quantification, much as we have done with abstract predicates in this paper.

5 Conclusion and Future Work

We have extended the model of assertions of higher-order separation logic in [2] to a higher-order model of specifications, and argued that the resulting higher-order separation logic is very effective. In particular, we have shown how it, in a straightforward way, can be used to prove correct programs using abstract data types with internal hidden (abstract) resource invariants. We have also considered applications of universal quantification over predicates, e.g., for proving correct polymorphic operations. Finally, we have argued that one may also employ versions of higher-order seperation logic for class-based languages.

For parametric operations on polymorphic data types to be really useful, one would of course prefer a higher-order programming language. 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 [4].

References

1. A. Banerjee and D. A. Naumann. Representation independence, confinement and access control [extended abstract]. In *Proc. POPL'02*, 2002.
2. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP'05*, Edinburgh, Scotland, April 2005. Accepted for publication.

3. L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proc. of POPL'04*, pages 220 – 231, Venice, Italy, 2004.
4. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. Submitted, 2005.
5. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined (resume). In *Proc. of ESOP'86*, pages 187 – 196, 1986.
6. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
7. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of POPL'01*, 2001.
8. B. Jacobs. *Categorical Logic and Type Theory*. North-Holland Publishing Co., Amsterdam, 1999.
9. I. Mijajlovic, N Torp-Smith, and P. W. O'Hearn. Refinement and separation contexts. In *Proc. of FSTTCS'04*, pages 421 – 433, 2004.
10. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Proc. of POPL'85*, pages 37–51, New Orleans, LA, USA, 1985.
11. P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. of CONCUR'04*, pages 49 – 67, London, England, 2004.
12. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of POPL'04*, pages 268 – 280, Venice, Italy, 2004.
13. M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proc. of POPL'05*, Long Beach, CA, USA, January 2005. (To appear).
14. D. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 315(1):257 – 305, May 2004.
15. J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.
16. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55 – 74, Copenhagen, Denmark, 2002.
17. H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.
18. H. Yang. Relational separation logic. *Theoretical Computer Science*, 2004. Accepted for publication.