When separation logic met Java (by example)

Matthew Parkinson

Middlesex University

Abstract. Separation logic is a promising new approach to modular reasoning, but so far it has primarily been applied to low-level C-like languages. To extend separation logic to allow modular reasoning about object-oriented languages like Java, we must add behavioural subtyping to the logic. However, a naïve integration of behavioural subtyping and separation logic is too restrictive. In this paper we demonstrate how abstract predicate families provide an abstraction mechanism that addresses these restrictions, by mirroring dynamic dispatch in the logic. We demonstrate the utility of our approach with a series of examples, including the Visitor pattern.

1 Introduction

Modularity is key to building large software systems. Object-oriented languages like Java provide support for modularity through encapsulation and inheritance. This modularity allows portions of object-oriented programs to be modified without rewriting, or even recompiling, all the code. We need formal methods that reflect this modularity.

Separation logic [10, 14, 18] is a promising new approach to modular reasoning. It provides local reasoning for low level C-like languages. The essence of "local reasoning" is that in order to understand how a piece of code works it should only be necessary to reason about the memory the code actually accesses (its so-called "footprint"). The pre-condition of a method specifies its footprint: all the state (the fields of objects) the method touches during its execution. This contrasts standard Hoare logic: in Hoare logic $\{y = 7\}\, x = 3;\, \{y = 7\}$ is a valid judgement, but in separation logic $\{y.f \mapsto 7\}\, x.f = 3;\, \{y.f \mapsto 7\}$ is not a valid judgement, because the pre-condition does say the heap contains $x.f$. This constraint on pre-conditions means that anything disjoint from the pre-condition is guaranteed not to be changed. This allows method specifications to be adapted to different calling contexts, and avoids the need to reverify methods at every call site. This is essential for modular reasoning.

To achieve modular verification in object-oriented languages with dynamic dispatch, we must also enforce *behavioural subtyping* [2, 12]. Dynamic dispatch means that many methods can be potentially called at a single call site. Behavioural subtyping allows us to consider just a single call by enforcing compatibility between subtypes' methods. We use a slightly generalised notion called *specification compatibility* [16], which allows the manipulation of logical variables[1] and intersection of specifications. Our generalisation is similar to *specification inheritance* [7].

[1] Sometimes called ghost or auxiliary variables.

However, a naïve combination of behavioural subtyping and separation logic does not work. The footprint of methods in subtypes are larger, in general than at supertypes, but behavioural subtyping requires that footprints are the same (or smaller). This is the *extended state problem* [11]. To address this problem, we use an abstraction called *abstract predicate families* [16].

An abstract predicate family is a predicate with a set of definitions indexed by class. In object-oriented programming a method body is selected based on the dynamic type of the first argument, the receiver. We reflect this in the logic: an abstract predicate family definition is selected based on the dynamic type of the first argument.

In this paper we show that abstract predicate families provide enough abstraction to introduce subtypes that are typically not considered to be *behavioural* subtypes. We demonstrate the utility of this approach with a series of examples. In particular, we present an example using the Visitor pattern [9]. The visitor pattern is an important test for program verification as it uses call-backs: the visitor's `visit` method calls the data structure's `accept` methods, and these `accept` methods call-back into the visitor's `visit` methods. Call-backs can cause problems when reasoning using class invariants [4]. By using abstract predicate families, we do not encounter these complications.

In previous work with Bierman [16], we defined abstract predicate families and specification compatibility. This paper elaborates on the intuitions behind abstract predicate families and uses examples to demonstrate their utility. We also present more details on the integration with behavioural subtyping. In particular, we show how specification compatibility generalises behavioural subtyping to allow the intersection of specifications.

2 Separation logic for Java

Separation logic is an extension to Hoare logic that permits reasoning about shared mutable state. It extends Hoare logic by adding spatial connectives to the assertion language, which allow us to assert that two portions of the heap are disjoint. This separation provides the key feature of separation logic—*local reasoning*—specifications need only mention the state they access [14].

Previous works on separation logic have dealt with a low-level C-like language. In this section, we present our separation logic for a simple subset of Java. Space prevents us from giving the complete description of the separation logic, but it may be found in Parkinson's thesis [15].

We use a simple subset of Java based on Middleweight Java (MJ) [5, 15]. We restrict MJ's expressions to be stack variables and `null`,[2] and remove constructors. We present the full syntax in Figure 1. We write f and m to range over field and method names respectively. We use C, D to range over class names, and x, y to range over variable names. We will assume two functions: $\mathit{fields}(\mathrm{C})$, which returns the set of field names for the class, C; and $\mathit{method}(\mathrm{C}, m)$, which returns a four tuple of (1) the argument types, (2) the return type, (3) argument names, and (4) method body.

[2] This restriction simplifies the presentation of the rules.

Program

$\text{prog} ::= \text{cldef}_1 \ldots \text{cldef}_n; \bar{\text{s}}$

Class definition

$\text{cldef} ::= \texttt{class}\ \text{C}\ \texttt{extends}\ \text{C}'\{\overline{\text{fdef}}\ \overline{\text{mdef}}\}$

Method definition

$\text{mdef} ::= \text{C}m(\text{C}_1\text{x}_1, ..., \text{C}_n\text{x}_n)\{\bar{\text{s}}\ \texttt{return}\ \text{x};\}$

Field definition

$\text{fdef} ::= \text{C}f;$

Expressions

$\text{e} ::= \text{x} \mid \texttt{null}$

Statements

$\text{s} ::= \text{x=y}.f; \mid \text{x=(C)y}; \mid \text{x=}\texttt{new}\ \text{C}();$
$\mid\ \text{x}.f\text{=e}; \mid \text{x=y}.m(\bar{\text{e}}); \mid \text{C x};$
$\mid\ \{\bar{\text{s}}\} \mid ; \mid \texttt{if}\ (\text{e == e})\text{s}\ \texttt{else}\ \text{s}$

Fig. 1. Syntax of MJ subset

2.1 The storage model

Separation logic is a logic of partial heaps, that is, heaps with (potentially) dangling pointers. The spatial connectives are based on the composition of disjoint partial heaps, which allows the logic to specify the splitting, or separation, of the heap.

To reason about Java, we must extend this partial heap model to represent the structure of objects. In addition to the consideration of dangling pointers, we also consider heaps with partial objects, that is objects with fields missing. A method might not modify all the fields of an object. We consider the heap to store two forms of information: (1) the values of fields; and (2) the type of objects. Thus a heap, h, is a pair of functions: (1) a partial function from object identifier and field names to values (for simplicity we take Values to be the object identifiers and null); and (2) a partial function from object identifier to class.

$$\mathsf{Heaps} \stackrel{\text{def}}{=} (\mathsf{OIDS} \times \mathsf{FieldNames} \rightharpoonup_{\text{fin}} \mathsf{Values}) \times (\mathsf{OIDS} \rightharpoonup_{\text{fin}} \mathsf{Class})$$

We use h to range over heaps, and h_v and h_t for the first and second components of a heap respectively. We write $\text{h} * \text{h}$ for the composition of two disjoint heaps:

$$(\text{h}_v, \text{h}_t) * (\text{h}'_v, \text{h}'_t) \stackrel{\text{def}}{=} \begin{cases} (\text{h}_v \cup \text{h}'_v, \text{h}_t) & (dom(\text{h}_v) \cap dom(\text{h}'_v) = \emptyset) \wedge (\text{h}_t = \text{h}'_t) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We only consider the composition where the value heaps, h_v, contain disjoint pairs of object identifiers and field names. We do not split the typing information, because it is immutable and hence valid to share.

A stack is a function from (program) variables to values. In addition to program variables, we also use logical variables,[3] which are variables that can be quantified over and used to relate values between pre- and post-conditions. In general, we will use upper case letters for logical variables to distinguish them from program variables. Unlike other presentations [10], we do not interpret logical variables using the stack because the fragment of Java does not have globally scoped variables. Instead, we define a logical interpretation that is a function from logical variable names to values

$$\mathsf{Stacks} \stackrel{\text{def}}{=} \mathsf{ProgVarNames} \rightarrow \mathsf{Values}$$
$$\mathsf{Interpretations} \stackrel{\text{def}}{=} \mathsf{LogVarNames} \rightarrow \mathsf{Values}$$

[3] Sometimes called ghost or auxiliary variables.

We define a state as a triple consisting of a stack, a heap and an interpretation.

$$\mathsf{States} \stackrel{\text{def}}{=} \mathsf{Heaps} \times \mathsf{Stacks} \times \mathsf{Interpretations}$$

2.2 The assertion language

A predicate is interpreted as a set of states, and formulae are given by the following grammar where e ranges over variables and null.

$$\begin{aligned} P, Q ::= {} & \mathrm{true} \mid \mathrm{false} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \\ & \mid \; P * Q \mid \mathrm{e}.f \mapsto \mathrm{e}' \mid \mathrm{e} : \mathrm{C} \mid \exists \mathrm{X} \cdot P \mid \forall \mathrm{X} \cdot P \end{aligned}$$

The intuitionistic[4] connectives ($\neg, \vee, \wedge, \Rightarrow$) and quantifiers ($\forall$,$\exists$) are interpreted in the usual way [10]. In addition to the intuitionistic connectives, we have the new spatial connective $*$, along with the predicate $\mapsto$, and a predicate for Java's dynamic type information, $\mathrm{e} : \mathrm{C}$. Taking these in reverse order:

$\mathrm{e}.f \mapsto \mathrm{e}'$ consists of all the states where the heap consists of at least the single mapping from the field f of the object given by the meaning of e to the value given by the meaning of e'.

$P * Q$ means the heap can be split into two disjoint parts in which P and Q hold respectively. Heaps of more than one element are specified by using the $*$ to join smaller heaps.

$\mathrm{e} : \mathrm{C}$ means that the result of e is an object whose dynamic type is precisely C.

We use $P(\_)$ as a shorthand for $\exists X \cdot P(X)$, for example, $e \mapsto \_$ means $\exists X \cdot e \mapsto X$.

2.3 The rules

The essence of "local reasoning" is that to understand how a piece of code works it should only be necessary to reason about the memory the code actually accesses (its so-called "footprint"). Ordinarily, aliasing precludes such a principle but the separation enforced by the $*$ connective allows this intuition to be captured formally by the following rule.

L-FRAME

$$\frac{\vdash \{P\}\ \bar{\mathrm{s}}\ \{Q\}}{\vdash \{P * R\}\ \bar{\mathrm{s}}\ \{Q * R\}}$$

where $\bar{\mathrm{s}}$ does not modify the free variables of R, i.e. $\mathrm{mods}(\bar{\mathrm{s}}) \cap FV(R) = \emptyset$.

Note 1. $\mathrm{mods}(\bar{\mathrm{s}})$ denotes the set of stack variables assigned by a given command, $\bar{\mathrm{s}}$, e.g. $\mathrm{mods}(\mathrm{x} = \mathrm{e}) = \{\mathrm{x}\}$. However assignment through a stack variable to a field is not counted: $\mathrm{mods}(\mathrm{x}.f = \mathrm{e}) = \emptyset$. See [15] for the full definition.

[4] Intuitionistic separation logic [10] allows memory leaks; it admits weakening, $P * Q \Rightarrow Q$ holds. This is required for Java because it lacks an explicit "dispose" primitive. Note: we do not have to worry about memory leaks, because Java is a garbage collected language

The frame rule's side-condition is inherited from Hoare logic. It is required because $*$ only describes the separation of heap locations and not variables; see [6, 17] for more details. Without the side-condition we could make the following incorrect derivation:

$$\frac{\{true\}\texttt{x = new C()}\{\texttt{x}.f \mapsto \_\}}{\{true * \texttt{x}.f \mapsto \_\}\texttt{x = new C()}\{\texttt{x}.f \mapsto \_ * \texttt{x}.f \mapsto \_\}}$$

The two fields in the post-condition of the conclusion are of different objects, but without the side-condition we can wrongly infer them to be the same field.

By using the frame rule, a local specification concerning only the variables and parts of the heap that are used by $\overline{\mathrm{s}}$ can be arbitrarily extended as long as the extension's free variables are not modified by $\overline{\mathrm{s}}$. Thus, from a local specification we can infer a global specification that is appropriate to the larger footprint of an enclosing program.

A judgement in our assertion language is written as follows:

$$\Gamma \vdash \{P\}\,\overline{\mathrm{s}}\,\{Q\}$$

This is read as the statement, $\overline{\mathrm{s}}$, satisfies the specification $\{P\}\_\{Q\}$, given the method hypotheses, Γ. These hypotheses are given by the following grammar:

$$\Gamma := \epsilon \mid \{P\}\mathrm{C}.m(\overline{\mathrm{x}})\{Q\}, \Gamma$$

However, when it simplifies the presentation, we will treat Γ as a partial function from method and defining class name, $\mathrm{C}.m$, to specifications. For the hypotheses, Γ, to be well-formed each method, $\mathrm{C}.m$, should appear at most once; and each specification's free program variables should be contained in the method's arguments and ret, the variable used for returning values from methods. We will only consider well-formed Γ.

We present the rules and axioms for Java in Figure 2. The top section presents the axioms associated to particular Java statements. We give small axioms for field manipulation in the style of O'Hearn, Reynolds, and Yang [14]. Field write, L-FWRITE, requires the heap to contain at least the single field being written to, and the post-condition specifies it has the updated value. Implicitly the axiom specifies that no other fields are modified, and hence it can be extended by the frame rule to talk about additional state. The field access axiom, L-FREAD, simply requires the field's state to be known and sets the variable equal to its contents. The logical variables X and Y are used to allow x and y to be syntactically the same variable without needing a case split or substitution in the rule definition. The cast axiom, L-CAST, ensures that a cast will complete successfully: the type of the object identifier being cast must be a subtype of the target type or null. The rule for constructing an object simply constructs all the fields of the class. The final axiom of the section, L-CALL, allows the call of a dynamically dispatched method without case analysis. In the next section we will place a constraint on method environments to make this axiom sound. The logical variable Y is used so that x and $\overline{\mathrm{e}}$ can mention y. If x and $\overline{\mathrm{e}}$ do not mention y, then we can remove the substitution $[Y/\mathrm{y}]$ and equality $Y = \mathrm{y}$ from the axiom.

The second section of Figure 2 defines the rules for introducing assumptions about methods. We define a new judgement $\Gamma_1 \Vdash \Gamma_2$ to allow the introduction of mutually recursive methods. $\Gamma_1 \Vdash \Gamma_2$ means that the bodies of the methods in Γ_2 can be verified

Java Commands

L-FWRITE $\quad \Gamma \vdash \{\mathrm{x}.f \mapsto \_\}\, \mathrm{x}.f\texttt{=}\mathrm{e}; \{\mathrm{x}.f \mapsto \mathrm{e}\}$

L-FREAD $\quad \Gamma \vdash \{\mathrm{X} = \mathrm{x} \wedge \mathrm{X}.f \mapsto \mathrm{Y}\}\, \mathrm{y}\texttt{=}\mathrm{x}.f; \{\mathrm{X}.f \mapsto \mathrm{Y} \wedge \mathrm{y} = \mathrm{Y}\}$

L-UPCAST $\quad \Gamma \vdash \{P[\mathrm{x}/\mathrm{y}] \wedge (\mathrm{x} : \mathrm{C} \vee \mathrm{x} = \texttt{null})\}\, \mathrm{y}\texttt{=}(\mathrm{C}')\mathrm{x}; \{P\}$ provided $\mathrm{C} \prec \mathrm{C}'$.

L-NEW $\quad \Gamma \vdash \{\mathrm{true}\}\mathrm{x}\texttt{=new}\ \mathrm{C}(); \{\mathrm{x}{:}\mathrm{C} * \mathrm{x}.f_1 \mapsto \_ * \ldots * \mathrm{x}.f_n \mapsto \_\}$
provided $\mathit{fields}(\mathrm{C}) = f_1, \ldots, f_n$

L-CALL $\quad \{P\}\mathrm{C}.m(\bar{w})\{Q\}, \Gamma \vdash \{\theta(P) \wedge Y = y \wedge x \neq \texttt{null}\}\, y\texttt{=}x.m(\bar{\mathrm{e}}); \{\theta(Q)\}$
provided $\theta = [\bar{\mathrm{e}}[Y/y], x[Y/y], y/\bar{w}, \texttt{this}, \mathrm{ret}]$, and x has static type C

Method introduction

L-DMETHOD $\quad \dfrac{\Gamma \vdash \{P \wedge \texttt{this} : \mathrm{C}\}\, \overline{\mathrm{C}}\, \overline{y}; \overline{y} = \overline{x}; \bar{\mathrm{s}}[\overline{y}/\overline{x}]\, \{Q[\mathrm{x}/\mathrm{ret}]\}}{\Gamma \Vdash \{P\}\, \mathrm{C}.m(\overline{x})\, \{Q\}}$
provided $\mathit{method}(\mathrm{C}, m) = (\overline{\mathrm{C}}, \_, \overline{x}, \bar{\mathrm{s}}\ \texttt{return}\ \mathrm{x})$ and $\overline{y}$ is fresh.

L-DSPLIT $\quad \dfrac{\Gamma \Vdash \Gamma_1 \quad \Gamma \Vdash \Gamma_2}{\Gamma \Vdash \Gamma_1, \Gamma_2}$

L-DINTRO $\quad \dfrac{\Gamma, \Gamma' \Vdash \Gamma' \quad \Gamma, \Gamma' \vdash \{P\}\bar{\mathrm{s}}\{Q\}}{\Gamma \vdash \{P\}\bar{\mathrm{s}}\{Q\}}$

Fig. 2. Java rules and axioms

assuming all the method calls satisify the specifications in Γ_1. The method introduction rule, L-DMETHOD, checks that the body, $\bar{\mathrm{s}}$, meets the specification assuming it is invoked on the correct class. The additional variable declarations are used to ensure that the call-by-value semantics is respected. The remaining two rules are used to introduce and manipulate these definitions. These rules are similar in style to those used by von Oheimb [19] to reason about mutually recursive procedures.

For completeness we present the standard rules from Hoare and separation logic in Figure 3.

3 Behavioural subtyping

Next we present *specification compatibility*: a generalised notion of behavioural subtyping. For the purpose of this exposition we focus on method specifications and ignore class invariants and abstraction functions [2, 12] as we, later, use abstract predicate families to deal with these issues.

A method specification, $\{P_D\}\,\_\,\{Q_D\}$, is said to be a subtype of another method specification, $\{P_C\}\,\_\,\{Q_C\}$, iff the pre-condition of the supertype implies the pre-condition of the subtype, $P_C \Rightarrow P_D$, and the subtype's post-condition implies the supertype's, $Q_D \Rightarrow Q_C$. We can see behavioural subtyping of specifications as simply the rule of consequence in Hoare logic, L-CONSEQUENCE in Figure 3. We can generalise this notion to allow the other structural rules: L-FRAME and L-VARELIM in Figure 3.

Standard commands

L-ASSIGN
$$\Gamma \vdash \{P[\mathrm{e}/\mathrm{x}]\}\ \mathrm{x{=}e};\{P\}$$

L-BLOCK
$$\frac{\Gamma \vdash \{P\}\,\overline{\mathrm{s}}\,\{Q\}}{\Gamma \vdash \{P\}\,\{\overline{\mathrm{s}}\}\,\{Q\}}$$

L-SKIP
$$\Gamma \vdash \{P\}\,;\{P\}$$

L-IF
$$\frac{\Gamma \vdash \{P \wedge \mathrm{x}{=}\mathrm{y}\}\,\mathrm{s}_1\,\{Q\} \quad \Gamma \vdash \{P \wedge \mathrm{x}{\neq}\mathrm{y}\}\,\mathrm{s}_2\,\{Q\}}{\Gamma \vdash \{P\}\,\texttt{if}\ (\mathrm{x}\ \texttt{==}\ \mathrm{y})\ \mathrm{s}_1\ \texttt{else}\ \mathrm{s}_2\,\{Q\}}$$

L-SEQ1
$$\frac{\Gamma \vdash \{P\}\,\mathrm{s}_1\,\{R\} \quad \Gamma \vdash \{R\}\,\mathrm{s}_2 ... \mathrm{s}_n\,\{Q\} \quad \mathrm{s}_1 \neq \mathrm{C\ x};}{\Gamma \vdash \{P\}\,\mathrm{s}_1\mathrm{s}_2 ... \mathrm{s}_n\,\{Q\}}$$

L-SEQ2
$$\frac{\Gamma \vdash \{P\}\,\mathrm{s}_1 ... \mathrm{s}_n\,\{Q\} \quad \mathrm{x} \notin FV(P) \cup FV(Q)}{\Gamma \vdash \{P\}\,\mathrm{C\ x};\mathrm{s}_1 ... \mathrm{s}_n\,\{Q\}}$$

Structural rules

L-CONSEQUENCE
$$\frac{P \Rightarrow P' \quad \Gamma \vdash \{P'\}\,\overline{\mathrm{s}}\,\{Q'\} \quad Q' \Rightarrow Q}{\Gamma \vdash \{P\}\,\overline{\mathrm{s}}\,\{Q\}}$$

L-VARELIM
$$\frac{\Gamma \vdash \{P\}\,\overline{\mathrm{s}}\,\{Q\}}{\Gamma \vdash \{\exists \mathrm{X} \cdot P\}\,\overline{\mathrm{s}}\,\{\exists \mathrm{X} \cdot Q\}}$$
provided X is not free in $\overline{\mathrm{s}}$.

L-FRAME
$$\frac{\Gamma \vdash \{P\}\,\overline{\mathrm{s}}\,\{Q\}}{\Gamma \vdash \{P * R\}\,\overline{\mathrm{s}}\,\{Q * R\}}$$
provided $\mathrm{mods}(\overline{\mathrm{s}}) \cap FV(R) = \emptyset$

Fig. 3. Standard rules

Specification compatibility generalises behavioural subtyping in two important ways: it allows the introduction of new logical variables in subclasses' specifications; and it allows the intersection of specifications. We demonstrate these at the end of the section.

Definition 1 (Specification compatibility). *We define specification compatibility,* $\vdash \{P_{\mathrm{D}}\}\_\{Q_{\mathrm{D}}\} \Longrightarrow \{P_{\mathrm{C}}\}\_\{Q_{\mathrm{C}}\}$*, iff*

$$\forall \overline{\mathrm{s}}, \Gamma \cdot\ (\mathit{mods}(\overline{\mathrm{s}}) \subseteq \{\mathit{ret}\}) \Rightarrow \frac{\dfrac{}{\Gamma \vdash \{P_{\mathrm{D}}\}\overline{\mathrm{s}}\{Q_{\mathrm{D}}\}}}{\Gamma \vdash \{P_{\mathrm{C}}\}\overline{\mathrm{s}}\{Q_{\mathrm{C}}\}}$$

That is, a proof of $\Gamma \vdash \{P_{\mathrm{C}}\}\overline{\mathrm{s}}\{Q_{\mathrm{C}}\}$ *exists whose only assumption is* $\Gamma \vdash \{P_{\mathrm{D}}\}\overline{\mathrm{s}}\{Q_{\mathrm{D}}\}$.

The quantification over $\overline{\mathrm{s}}$ and Γ in the definition restricts the derivation to the structural rules in Figure 3. The frame rule, L-FRAME, can only introduce formulae that do not mention ret: all methods potentially alter this variable.

We will restrict method environments to *well-behaved environments*, where methods in subtypes have compatible specifications with their supertypes' specification.

Definition 2 (Well-behaved environments). *Γ is well-behaved iff for each specification $\{P_{\mathrm{C}}\}\mathrm{C}.m(\overline{\mathrm{x}})\{Q_{\mathrm{C}}\}$ in Γ, if $\mathrm{D} \prec \mathrm{C}$, then $\{P_{\mathrm{D}}\}\mathrm{D}.m(\overline{\mathrm{x}})\{Q_{\mathrm{D}}\}$ is in Γ and $\vdash \{P_{\mathrm{D}}\}\_\{Q_{\mathrm{D}}\} \Longrightarrow \{P_{\mathrm{C}}\}\_\{Q_{\mathrm{C}}\}$*

For simplicity, we assume parameter names are not changed in subclasses.

As mentioned earlier, our generalised notion of behavioural subtyping allows the following additional ideas:

3.1 Logical Variable Manipulation

Consider the following specifications

$$\{\texttt{this}.f \mapsto \_\}\_\{\texttt{this}.f \mapsto \_\} \tag{1}$$
$$\{\texttt{this}.f \mapsto X\}\_\{\texttt{this}.f \mapsto X\} \tag{2}$$

If code expects the behaviour of (1), then clearly it would be satisfied with (2), after all (2) is a refinement of (1). However, the standard idea of behavioural subtyping would not permit this refinement as $\texttt{this}.f \mapsto \_ \Rightarrow \texttt{this}.f \mapsto X$ does not hold. This is valid in specification compatibility as it is a use of the variable elimination rule, L-VARELIM.

3.2 Specification intersection

Consider two specifications:

$$\{P_{\mathtt{C}}\}\_\{Q_{\mathtt{C}}\} \tag{3}$$
$$\{P_{\mathtt{D}}\}\_\{Q_{\mathtt{D}}\} \tag{4}$$

How can a single specification capture both of the above? Consider

$$\left\{\begin{array}{l}(P_{\mathtt{C}} \wedge X = 1)\\ \vee\, (P_{\mathtt{D}} \wedge X = 2)\end{array}\right\}\_\left\{\begin{array}{l}(Q_{\mathtt{C}} \wedge X = 1)\\ \vee\, (Q_{\mathtt{D}} \wedge X = 2)\end{array}\right\} \tag{5}$$

We use a fresh logical variable X to allow us to encode the intersection of the specification. Semantically, logical variables are universally quantified at the level of specifications, i.e.

$$\forall X \cdot \left\{\begin{array}{l}(P_{\mathtt{C}} \wedge X = 1)\\ \vee\, (P_{\mathtt{D}} \wedge X = 2)\end{array}\right\}\_\left\{\begin{array}{l}(Q_{\mathtt{C}} \wedge X = 1)\\ \vee\, (Q_{\mathtt{D}} \wedge X = 2)\end{array}\right\}$$

Hence, we can understand the specification as

$$(\{P_{\mathtt{C}}\}\_\{Q_{\mathtt{C}}\}) \quad \bigwedge \quad (\{P_{\mathtt{D}}\}\_\{Q_{\mathtt{D}}\})$$

We use this notation as a shorthand in the examples later. We can show that (5) is specification compatible with both (3) and (4). We present the derivation for (3).

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \left\{\begin{array}{l}(P_{\mathtt{C}} \wedge X = 1)\\ \vee\,(P_{\mathtt{D}} \wedge X = 2)\end{array}\right\} \text{-} \left\{\begin{array}{l}(Q_{\mathtt{C}} \wedge X = 1)\\ \vee\,(Q_{\mathtt{D}} \wedge X = 2)\end{array}\right\}}{\vdash \left\{\begin{array}{l}X = 1 \;*\\ \left(\begin{array}{l}(P_{\mathtt{C}} \wedge X = 1)\\ \vee\,(P_{\mathtt{D}} \wedge X = 2)\end{array}\right)\end{array}\right\} \text{-} \left\{\begin{array}{l}X = 1 \;*\\ \left(\begin{array}{l}(Q_{\mathtt{C}} \wedge X = 1)\\ \vee\,(Q_{\mathtt{D}} \wedge X = 2)\end{array}\right)\end{array}\right\}}\,\text{L-FRAME}}{\vdash \{P_{\mathtt{C}} \wedge X = 1\} \text{-} \{Q_{\mathtt{C}} \wedge X = 1\}}\,\text{L-CONSEQUENCE}}{\vdash \{\exists X \cdot\; P_{\mathtt{C}} \wedge X = 1\} \text{-} \{\exists X \cdot\; Q_{\mathtt{C}} \wedge X = 1\}}\,\text{L-VARELIM}}{\vdash \{P_{\mathtt{C}}\} \text{-} \{Q_{\mathtt{C}}\}}\,\text{L-CONSEQUENCE}$$

The key to the proof is the L-FRAME rule, which lets us preserve the value of the logical variable across the call. In standard Hoare logic the invariance rule [3] could be used.

Specification inheritance of Dhara and Leavens [7] also allows the intersection of specifications.

4 Abstract predicate families

So far, we have presented a separation logic for Java and enforced behavioural subtyping. As it stands, it is very hard to create any valid subtypes, because the specifications are too concrete. In this section, we define our abstraction mechanism, *abstract predicate families*.

We use α to range over abstract predicate family names. One can think of an abstract predicate family as a predicate that existentially hides the type:

$$\mathrm{e}.\alpha(\overline{\mathrm{e}}) \quad \approx \quad \exists \mathrm{C} \cdot \;\; \alpha_{\mathrm{C}}(\mathrm{e}; \overline{\mathrm{e}}) \wedge \mathrm{e} : \mathrm{C} \tag{6}$$

where α_{C} is the definition of α for class C. This intuition mirrors dynamic dispatch in object-oriented languages, where `e.m(...)` is a call to a method `m`, but the body is selected by the dynamic type of the receiver, `e`. Abstract predicate families reflect this in the logic, by allowing predicates whose definition is selected by the dynamic type of the "receiver".

We define abstract predicate family definitions, Λ, with the following syntax.

$$\Lambda := \epsilon \mid (\alpha_{\mathrm{C}}(x; \overline{x}) \stackrel{\text{def}}{=} P), \Lambda$$

Λ is well-formed if it has at most one entry for each predicate and class name pair, and the free variables of the body, P, are in its argument list, $x; \overline{x}$. The first argument is distinguished, as it is used to index by class. We use a semi-colon, ;, to separate this distinguished argument, and commas to separate the remaining arguments. We treat Λ as a function from predicate and class name pairs to predicate definitions. Each entry corresponds to the definition of an abstract predicate family for a particular class.

This semantic intuition, (6), leads to the following pair of axioms:

OPEN	$\Lambda \models (\mathrm{x} : \mathrm{C} \wedge \mathrm{x}.\alpha(\overline{\mathrm{x}})) \Rightarrow \Lambda(\alpha, \mathrm{C})[\mathrm{x}; \overline{\mathrm{x}}]$
CLOSE	$\Lambda \models (\mathrm{x} : \mathrm{C} \wedge \Lambda(\alpha, \mathrm{C})[\mathrm{x}; \overline{\mathrm{x}}]) \Rightarrow \mathrm{x}.\alpha(\overline{\mathrm{x}})$

where $\alpha, \mathrm{C} \in \mathit{dom}(\Lambda)$.

To OPEN or CLOSE a predicate we must know which class contains the definition, and must have that definition in scope.

In our examples later, we find it useful to consider the predicate to have many different arities. We add the following pair of axioms to alter a predicate's arity.

WIDEN	$\Lambda \models x.\alpha(\overline{x}) \;\Rightarrow\; \exists \overline{y} \cdot \;\; x.\alpha(\overline{x}, \overline{y})$
NARROW	$\Lambda \models \exists \overline{y} \cdot \;\; x.\alpha(\overline{x}, \overline{y}) \;\Rightarrow\; x.\alpha(\overline{x})$

As we allow changes in arity, we must provide an operation to provide fewer or more arguments than expected. We use square brackets to denote this unusual argument application operation.

$$(\alpha_{\mathrm{C}}(x;\overline{x}) \stackrel{\text{def}}{=} P)[\mathrm{e};\overline{\mathrm{e}}] \stackrel{\text{def}}{=} \begin{cases} P[\mathrm{e}/\mathrm{x}, \overline{\mathrm{e_1}}/\overline{\mathrm{x}}] & |\overline{\mathrm{e_1}}| = |\overline{\mathrm{x}}| \text{ and } \overline{\mathrm{e}} = \overline{\mathrm{e_1}}, \overline{\mathrm{e_2}} \\ \exists \overline{\mathrm{x'}} \cdot P[\mathrm{e}/\mathrm{x}, (\overline{\mathrm{e}}, \overline{\mathrm{x'}})/\overline{\mathrm{x}}] & |\overline{\mathrm{e}}, \overline{\mathrm{x'}}| = |\overline{\mathrm{x}}| \end{cases}$$

If we give a predicate more arguments than its definition requires, it ignores them. If too few, it existentially quantifies the missing arguments. This definition of substitution is used in the definition of OPEN and CLOSE.

We present the following rule for introducing abstract predicate families

ABSTRACT WEAKENING
$$\frac{\Lambda; \Gamma \vdash \{P\}\overline{\mathrm{s}}\{Q\}}{\Lambda, \Lambda'; \Gamma \vdash \{P\}\overline{\mathrm{s}}\{Q\}}$$
provided $dom(\Lambda')$ and $dom(\Lambda)$ are disjoint.

This rule allows us to add more predicate definition without affecting a proofs validity. For details of the modularity and scoping of abstract predicate families see [15, 16].

5 Examples

For legibility, in the following examples we use full Java statements and expressions and a primitive type of integer, `int`. These can be encoded in the obvious way [15].

5.1 Cell/Recell

We begin with a simple example of subtyping: `Cell` and `Recell` [1]. We give the source code in Figure 4. We ignore `NullCell` until the next subsection.

If we give the obvious specification to `Cell`

$$\{\texttt{this.cnts} \mapsto \_\}\texttt{Cell.set(n)}\{\texttt{this.cnts} \mapsto \texttt{n}\}$$

then there is no specification that `Recell` satisfies and is a behavioural subtype. This is the extended state problem: the subclass is modifying more state. Therefore, we need to introduce an abstract predicate family into the specification. We present the correct specifications in Figure 5.

We have to validate four methods: the `set` and `get` methods of both `Cell` and `Recell`. Details of these proofs can be found in [15, 16]. Even though the bodies of

```
class Cell extends Object {
  Object cnts;
  void set(Object x)
  {this.cnts = x;}
  Object get()
  {return this.cnts;}
}
class NullCell extends Cell {
  Object get() {return null;}
}
```

```
class Recell extends Cell {
  Object bk;
  void set(Object o) {
    temp = this.cnts;
    this.bk = temp;
    this.cnts = x;
  }
}
```

Fig. 4. Source code for Cell classes

Method	Specification
`Cell.set(y)`	$\{\texttt{this}.Val(\_)\}\ \_\ \{\texttt{this}.Val(\texttt{y})\}$
`Cell.get()`	$\{\texttt{this}.Val(X)\}\ \_\ \{\texttt{this}.Val(X) \wedge \text{ret} = X\}$
`Recell.set(y)`	$\{\texttt{this}.Val(X,\_)\}\ \_\ \{\texttt{this}.Val(\texttt{y},X)\}$
`Recell.get()`	$\{\texttt{this}.Val(X,Y)\}\ \_\ \{\texttt{this}.Val(X,Y) \wedge \text{ret} = X\}$
`Nullcell.set(y)`	$(\{\texttt{this}.Val(\_)\}\ \_\ \{\texttt{this}.Val(\texttt{y})\}) \bigwedge (\{\texttt{this}.NVal()\}\ \_\ \{\texttt{this}.NVal()\})$
`Nullcell.get()`	$(\{\texttt{this}.Val(X)\}\ \_\ \{\texttt{this}.Val(X) \wedge \text{ret}=X\}) \bigwedge (\{\texttt{this}.NVal()\}\ \_\ \{\texttt{this}.NVal() \wedge \text{ret}=\texttt{null}\})$

$$Val_{\texttt{Cell}}(x;y) \stackrel{\text{def}}{=} x.\texttt{cnts} \mapsto y \qquad Val_{\texttt{Recell}}(x;y,z) \stackrel{\text{def}}{=} x.\texttt{cnts} \mapsto y * x.\texttt{bk} \mapsto z$$
$$Val_{\texttt{Nullcell}}(x;) \stackrel{\text{def}}{=} \text{false} \qquad NVal_{\texttt{Nullcell}}(x;) \stackrel{\text{def}}{=} x.\texttt{cnts} \mapsto \_$$

Fig. 5. Specifications and predicate definitions for Cell classes

`Cell.get` and `Recell.get` are the same, we must validate both, because they have different predicate definitions. We will return to this point in §6.

We must prove that the method specifications are compatible, defined in the sense of definition 2. The two `get` methods have the same specification, so they are compatible. The compatibility of the `set` method follows as

$$\dfrac{\dfrac{\vdash \{\texttt{this}.Val(X,\_)\}\_\{\texttt{this}.Val(n,X)\}}{\vdash \{\texttt{this}.Val(\_,\_)\}\_\{\texttt{this}.Val(n,\_)\}}\ \text{L-VarElim}}{\vdash \{\texttt{this}.Val(\_)\}\_\{\texttt{this}.Val(n)\}}\ \text{L-Consequence}$$

Above L-Consequence uses WIDEN on the pre-condition, and NARROW on the post-condition.

5.2 NullCell

Normally, we would not consider the cell that always returns null, `NullCell`, to be a behavioural subtype of `Cell`. However, we may wish it to inherit from `Cell`. Abstract

```
class Ast extends Object {
 void accept(Visitor x) {;}
}
class Const extends Ast {
 int v;
 void accept(Visitor x)
 { x.visitC(this); }
}
class Plus extends Ast {
 Ast l; Ast r;
 void accept(Visitor x)
 { x.visitP(this); }
}
```

```
class Visitor {
 void visitC(Const x) {;}
 void visitP(Plus x) {;}
}
class Calc extends Visitor {
 int amount;
 void visitC(Const x)
 { this.amount += x.n; }
 void visitP(Plus x) {
   x.l.accept(this);
   x.r.accept(this);
 }
}
```

Fig. 6. Source code for formulae and visitor classes.

predicate families allow us to see `NullCell` as a behavioural subtype (using specification compatibility), while preventing it being used in place of a `Cell`.

In Figure 5, we define the predicate family Val for `NullCell` to be false. This means that it is never possible to get a satisfied instance of Val where its first parameter is a `NullCell`, and hence we cannot use a `NullCell` in place of a `Cell`. We give the `NullCell` a new predicate family, $NVal$, to represent its internals, and specify its methods as the intersection, as defined in section 3.2, of its supertype's method's specification and its own specification with $NVal$.

Abstract predicate families allow implementations to be inherited, while not preserving behavioural subtyping. They distinguish implementation inheritance from specification inheritance.

5.3 Visitor pattern

Now let us consider an extended example using the visitor design pattern [9]. We consider a visitor over a very simple syntax of formulae with just constants and addition of two terms. We present the source code in Figure 6. The `Visitor` class has two methods: the first, `visitC` is invoked when the visitor visits a `Const` node; and the second, `visitP` is for a `Plus` node. The `Visitor` class is a template that should be overridden to produce more interesting visitors. We define three classes to represent an abstract syntax tree of these formulae: `Ast` is used as a common parent for the term constructors; `Const` represents constant terms; and `Plus` represents the addition of two terms. We define a single method, `accept`, in `Ast` that is overridden in each of the subclasses. The `Const` class represents an integer constant, and calls `visitC` when it accepts a visitor. The `Plus` class represents an addition of two formulae and calls `visitP` when it accepts a visitor.

Before we can formally specify the visitor, we must extend the logic with the formulae's syntax:

$$\tau ::= n \mid \tau \oplus \tau$$
$$\mu ::= \bullet \mid \tau \oplus \mu \mid \mu \oplus \tau$$

Method	Specification
`visitC(x)`	$\{\mathtt{x} : \mathtt{Const} \wedge \mathtt{x}.\mathit{Ast}(\tau) * \mathtt{this}.\mathit{Visitor}(\mu)\}\ \_\ \{\mathtt{this}.\mathit{Visited}(\mathtt{x}, \tau, \mu)\}$
`visitP(x)`	$\{\mathtt{x} : \mathtt{Plus} \wedge \mathtt{x}.\mathit{Ast}(\tau) * \mathtt{this}.\mathit{Visitor}(\mu)\}\ \_\ \{\mathtt{this}.\mathit{Visited}(\mathtt{x}, \tau, \mu)\}$
`accept(x)`	$\{\mathtt{this}.\mathit{Ast}(\tau) * \mathtt{x}.\mathit{Visitor}(\mu)\}\ \_\ \{\mathtt{x}.\mathit{Visited}(\mathtt{this}, \tau, \mu)\}$

$$\mathit{Visitor}_{\mathtt{Calc}}(x;\mu) \stackrel{\text{def}}{=} x.\mathtt{amount} \mapsto \mathit{lcalc}(\mu)$$
$$\mathit{Visited}_{\mathtt{Calc}}(x;y,\tau,\mu) \stackrel{\text{def}}{=} x.\mathtt{amount} \mapsto (\mathit{lcalc}(\mu) + \mathit{calc}(\tau)) * \mathit{Ast}(y;\tau)$$

where

$$\mathit{calc}(n) \stackrel{\text{def}}{=} n \qquad \mathit{calc}(\tau_1 \oplus \tau_2) \stackrel{\text{def}}{=} \mathit{calc}(\tau_1) + \mathit{calc}(\tau_2)$$

$$\mathit{lcalc}(\bullet) \stackrel{\text{def}}{=} 0 \qquad \mathit{lcalc}(\mu_1 \oplus \tau) \stackrel{\text{def}}{=} \mathit{lcalc}(\mu_1) \qquad \mathit{lcalc}(\tau \oplus \mu_1) \stackrel{\text{def}}{=} \mathit{calc}(\tau) + \mathit{lcalc}(\mu_1)$$

Fig. 7. Specifications for accept and visit methods.

We use τ to represent the structure of an abstract syntax tree, and μ for a context: a tree with a hole, $\bullet$, in it. We use, $\oplus$, to distinguish it from the arithmetic operation of addition, and use $\mu[\mu']$ to mean replace $\bullet$ by μ' in μ. We define the Ast predicate family for the three classes.

$$\mathit{Ast}_{\mathtt{Ast}}(x;\tau) \stackrel{\text{def}}{=} \text{false}$$
$$\mathit{Ast}_{\mathtt{Const}}(x;\tau) \stackrel{\text{def}}{=} \exists n \cdot x.\mathtt{v} \mapsto n \wedge \tau = n$$
$$\mathit{Ast}_{\mathtt{Plus}}(x;\tau) \stackrel{\text{def}}{=} \exists i\, j\, \tau_l\, \tau_r \cdot\ x.\mathtt{l} \mapsto i * x.\mathtt{r} \mapsto j * \mathit{Ast}(i,\tau_l) * \mathit{Ast}(j,\tau_r) \wedge \tau = \tau_l \oplus \tau_r$$

Note 2. Defining the predicate for the `Ast` class as false prevents any invocation of its methods.

We give the specifications for the `visit` and `accept` methods in Figure 7. There are many choices one can make for the specification of a visitor. The μ parameter is used to allow the evaluation to depend on the context. In the example that follows we use the context to accumulate the value of the expression as we traverse it, rather than calculating it in a bottom up fashion. We can remove the μ parameter from the predicates if we do not want context sensitive visitors, that is, if the state of the visitor did not depend on its context. One might have expected the methods all to have post-conditions of the form $\mathtt{this}.\mathit{Visited}(\ldots) * x.\mathit{Ast}(\tau)$. However, this kind of specification prevents us altering the structure of the expression.

We are now in a position to verify the `accept` methods for the `Plus` and `Const` classes. We only present the `Plus` case as the `Const` case is very similar.

$\{\mathtt{this}.\mathit{Ast}(\tau) * \mathtt{x}.\mathit{Visitor}(\mu) \wedge \mathtt{this} : \mathtt{Plus}\}$
$\{(\mathtt{x}.\mathit{Ast}(\tau) * \mathtt{this}.\mathit{Visitor}(\mu) \wedge \mathtt{x} : \mathtt{Plus})[\mathtt{this}, \mathtt{x}/\mathtt{x}, \mathtt{this}]\}$ } L-Call
`x.visitP(this);`
$\{(\mathtt{this}.\mathit{Visited}(\mathtt{x}, \tau, \mu))[\mathtt{this}, \mathtt{x}/\mathtt{x}, \mathtt{this}]\}$ } L-Consequence
$\{\mathtt{x}.\mathit{Visited}(\mathtt{this}, \tau, \mu)\}$

The verification of the method call introduces the current class's type: this is exactly what is required to meet `visitP`'s specification. This method call simply provides information about which type of node this is to the `Visitor`: it is providing a case selection using dynamic dispatch. This proof does not need changing no matter how many subclasses of `Visitor` are added.

Let us consider an actual visitor implementation. The implementation given in Figure 6 calculates the value of the expression. We define the $Visitor$ and $Visited$ predicates for this class in Figure 7. The $calc$ function calculates the value of the tree, and the $lcalc$ function is used to calculate the accumulated total from the context: the sum of everything to the left of the hole, $\bullet$, i.e. the nodes we have already visited.

We verify the `visitP` method as

$$\begin{array}{l}
\{\mathtt{x} : \mathtt{Plus} \wedge \mathtt{x}.Ast(\tau) * \mathtt{this}.Visitor(\mu) \wedge \mathtt{this} : \mathtt{Calc}\} \\
\quad \{\exists i, j, \tau_1, \tau_2 \cdot Q * i.Ast(\tau_1) * j.Ast(\tau_2) * \mathtt{this}.Visitor(\mu)\} \\
\qquad \left.\begin{array}{l} \{Q * i.Ast(\tau_1) * j.Ast(\tau_2) * \mathtt{this}.Visitor(\mu)\} \\ \quad \mathtt{x.l.accept(this);\ x.r.accept(this);} \\ \{Q * i.Ast(\tau_1) * \mathtt{this}.Visited(j, \tau_2, \mu[\tau_1 \oplus \bullet])\} \end{array}\right\} (7) \\
\quad \{\exists i, j, \tau_1, \tau_2 \cdot Q * i.Ast(\tau_1) * \mathtt{this}.Visited(j, \tau_2, \mu[\tau_1 \oplus \bullet])\} \\
\{\mathtt{this}.Visited(x, \tau, \mu)\}
\end{array}
\begin{array}{l} \left.\begin{array}{l} \\ \\ \\ \\ \\ \end{array}\right\} \text{L-VarElim} \\ \left.\begin{array}{l} \\ \\ \\ \\ \\ \end{array}\right\} \text{L-Consequence} \end{array}$$

where $P \stackrel{\text{def}}{=} \mathtt{x} : \mathtt{Plus} \wedge \mathtt{this} : \mathtt{Calc} \wedge \tau = \tau_1 \oplus \tau_2$

and $\quad Q \stackrel{\text{def}}{=} P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j$

Above L-Consequence uses Open with the definition of $Visitor$ for `Calc` in the implication between the pre-conditions, and for the post-conditions uses both Open and Close with the definition of $Visited$ and $lcalc(\mu) + calc(\tau) = lcalc(\mu[\tau \oplus \bullet])$ which can be shown by induction on μ.

We present an outline of the proof for the two function calls:

$$\begin{array}{l}
\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * i.Ast(\tau_1) * j.Ast(\tau_2) * \mathtt{this}.Visitor(\mu)\} \\
\quad \left.\begin{array}{l} \{\mathtt{x.l} \mapsto i * i.Ast(\tau_1) * \mathtt{this}.Visitor(\mu, \bullet)\} \\ \quad \mathtt{x.l.accept(this);} \\ \{\mathtt{x.l} \mapsto i * \mathtt{this}.Visited(i, \tau_1, \mu)\} \end{array}\right\} (8) \\
\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * j.Ast(\tau_2) * \mathtt{this}.Visited(i, \tau_1, \mu)\} (9)
\end{array}
\left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\} \text{L-Frame}$$

To prove (8), we expand the code and actually prove `t=x.l;t.accept(this);`. We omit the details of the proof.

$$\begin{array}{l}
\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * j.Ast(\tau_2) * i.Ast(\tau_1) * \mathtt{this}.Visitor(\mu[\tau_1 \oplus \bullet])\} (10) \\
\quad \left.\begin{array}{l} \{\mathtt{x.r} \mapsto j * j.Ast(\tau_2) * \mathtt{this}.Visitor(\mu[\tau_1 \oplus \bullet])\} \\ \quad \mathtt{x.r.accept(this);} \\ \{\mathtt{x.r} \mapsto j * \mathtt{this}.Visited(j, \tau_2, \mu[\tau_1 \oplus \bullet])\} \end{array}\right\} (11) \\
\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * i.Ast(\tau_1) * \mathtt{this}.Visited(j, \tau_2, \mu[\tau_1 \oplus \bullet])\}
\end{array}
\left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\} \text{L-Frame}$$

(11) is verified in the same way. We can combine these two proofs to prove 7, provided (9) $\Rightarrow$ (10) holds, which follows from using Close with $Visitor$ and Open with $Visited$, and $lcalc(\mu) + calc(\tau) = lcalc(\mu[\tau \oplus \bullet])$.

We can produce different subclasses of `Visitor` to perform many different functions, such as checking whether an AST contains a particular constant, or to clone an AST. Parkinson's thesis [15] presents another subclass that removes all additions of zero from the term. As the proof is modular, we only need to verify the new classes we write: we know all the other classes interact correctly.

6 Related and future work

In this paper, we have built on our previous work [16]. Our examples have demonstrated the flexibility of separation logic and abstract predicate families. Although we have enforced behavioural subtyping, this has not placed undue constraints on subtypes. Additionally, we have shown that this methodology does not have difficulties with call-backs, because we do not need class invariants.

One drawback of this methodology is that it requires the rechecking of every inherited method. This is because the predicate definitions can be changed completely in subclasses. We are currently investigating ways of structuring abstract predicate families to allow the inheritance of methods without rechecking their specifications.

Our original inspiration for abstract predicate families came from Leino's work on the extended state problem [11]. He proposed *data groups* to abstract modifies clauses, the set of fields modified by each method. The clauses were given in terms of abstract groups of fields. Each class was free to add its new fields to the abstract groups. Abstract predicate families generalise data groups by allowing constraints to be placed on these fields.

Closely related to our work is Boogie [4], which takes a different approach to the class invariant and call-backs problem. Boogie uses an auxiliary field, st, to indicate if the invariant holds (or even how much of the invariant holds). The invariant can be seen as $(this.st = Valid) \Rightarrow P(this)$, where P is the property that should hold of the object. `pack` and `unpack` operations are used to validate and invalidate the st field, respectively. These operations roughly correspond to the OPEN and CLOSE axioms in our logic. Boogie also represents similar encapsulation to abstract predicate families using an auxiliary $owner$ field. Owned objects can only be `unpack`ed, if its owner is unpacked. For comparison, consider an example from [4]:

```
class T {
  rep U f;
  invariant 0≤f.g;
  void m() requires st=Valid
  { unpack this;
    f.n();...
    pack this; }
}
```

```
class U {
  int g;
  void n() requires st=Valid
  { ... }
}
```

In this example `T` owns its `f` field. We could encode this in our methodology using a single predicate family

$$BoogieInv_{\mathsf{T}}(x;) \stackrel{\text{def}}{=} \exists i, g \cdot \ x.\mathsf{f} \mapsto i * i.BoogieInv(g) \wedge g \geq 0$$
$$BoogieInv_{\mathsf{U}}(x;g) \stackrel{\text{def}}{=} x.\mathsf{g} \mapsto g$$

and making the pre- and post-conditions of each method `this`.$BoogieInv$. In the Boogie proof, the `unpack` operation in `T.m()` exposes the object in the `f` field. In our proof, we simply use OPEN on the predicate family to replace `this`.$BoogieInv()$ with $\exists i, g \cdot \ x.\mathsf{f} \mapsto i * i.BoogieInv(g) \wedge g \geq 0$. This allows us to meet the pre-condition

of the call to `f.n()`. In the Boogie proof, the `pack` operation in `T.m()` hides the object in the `f` field, and must also check the invariant holds. Similarly in our proof, we use CLOSE, which requires the predicate family for the field, and the invariant is true. There are many similarities between to the two proof systems. In the future, we plan to explore the correspondence between the two systems in more detail.

Another closely related work is Typestates by Fähndrich and DeLine [8], which allows specification to be given with respect to named externally visible states. These states correspond to abstracted predicates in a similar way to that used here. They impose a strong structure on the predicates, which allows them to inherit methods without reverifying code.

Finally, in this paper we have only considered sequential Java. Recently, O'Hearn has shown how to extend separation logic with rules to reason about concurrency primitives [13]. They allow state to be stored in a semaphore, and by manipulating this semaphore the state can be transferred between threads. Unfortunately the semaphore is statically scoped, which prevents reasoning about heap allocated semaphores including, for example, Java's `synchronised` primitive. We are currently investigating how to extend O'Hearn's system for concurrency to allow for reasoning about semaphores in the heap, and hence Java with threads.

Acknowledgments I should like to thank Gavin Bierman, Sophia Drossopoulou, Alisdair Wren and the anonymous referees for their comments on this work. This work was support by an EPSRC DTA at the University of Cambridge, EPSRC grant EP/C523997/1, and Intel Research Cambridge.

Bibliography

[1] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[2] P. America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, 1991.

[3] K. R. Apt. Ten years of Hoare's logic: A survey: Part I. *ACM TOPLAS*, 3(4):431–483, 1981.

[4] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[5] G. M. Bierman and M. J. Parkinson. Effects and effect inference for a core Java calculus. In *Proceedings of WOOD*, volume 82 of *ENTCS*, 2004.

[6] R. Bornat. Variables as resources in separation logic. In *Proceedings of MFPS*, pages 125–146, 2005.

[7] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of ICSE*, pages 258–267, 1996.

[8] M. Fähndrich and R. DeLine. Typestates for objects. In *Proceedings of ECOOP*, pages 465–490, 2004.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[10] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, pages 14–26, 2001.

[11] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of OOPSLA*, pages 144–153, 1998.

[12] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6): 1811–1841, 1994.

[13] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proceeding of CONCUR*, volume 3170, pages 49–67, 2004.

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

[15] M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

[16] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.

[17] M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logic. In *Proceedings of LICS*, 2006.

[18] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74, 2002.

[19] D. von Oheimb. Hoare logic for mutual recursion and local variables. In *Proceedings of FSTTCS*, pages 168–180, 1999.