When Separation Logic met Java (Work in progress)

Matthew Parkinson

University of Cambridge, Computer Laboratory, Cambridge, CB3 0FD, UK

Matthew.Parkinson@cl.cam.ac.uk

Developing a logic for object-oriented programs is complicated by both encapsulation and inheritance. Encapsulation requires a logic that abstracts the actual representation, and inheritance requires a notion of "behaves the same." While contending with these two beasts one must remember that aliasing is just around the corner ready to change our perception of the entire world. Fortunately separation logic [3] saves us from the problems of aliasing, so we are just left with two beasts to slay. In this abstract we propose an extension to separation logic that facilitates reasoning about both inheritance and encapsulation.

Let us consider a naïve approach with a simple example. In the end this will not succeed, but the reason should be instructive. Take a (separation) logic with predicates of the form $x.f \mapsto v$, which means the field f in the object x has the value v. These can be combined using the connectives of classical logic and the separating conjuction, $*$. We use $P * Q$ to mean both P and Q are true in disjoint parts of the heap.

The standard subtyping example of Cell/Recell [1] would be written

```
class Cell {
  Object contents;

  /*@ pre:  this.contents |-> _ /\ newValue != null@*/
  void set(Object newValue) {...}
  /*@ post: this.contents |-> newValue /\ newValue != null@*/
}

class Recell extends Cell {
  Object backup;

  /*@ pre:  (this.contents |-> X /\ newValue != null)
      * (this.backup |-> _ /\ X != null) @*/
  void set(Object newValue) {...}
  /*@ post: (this.contents |-> newValue /\ newValue != null)
      * (this.backup |-> X /\ X != null) @*/
}
```

To allow for dynamic dispatch the `Recell`'s specification needs to be compatible with the `Cell`'s. Compatibility is captured by the following two implications.[1]

$$\textit{precond}(\texttt{Cell}, \texttt{set}) \wedge \textit{this} : \texttt{Recell} \Rightarrow \textit{precond}(\texttt{Recell}, \texttt{set}) \quad (1)$$

$$\textit{postcond}(\texttt{Recell}, \texttt{set}) \Rightarrow \textit{postcond}(\texttt{Cell}, \texttt{set}) \quad (2)$$

where $\textit{precond}(C, m)$ denotes the precondition for the method m in class C, and $\textit{postcond}$ denotes the postcondition. We use the predicate $\textit{this} : \texttt{Recell}$ to mean that $\textit{this}$ points to an object of type `Recell`.

In separation logic these implications can never hold as they require a one element heap to be the same size as a two element heap. A greater level of abstraction is needed to allow changes to the implementation. To achieve this we propose the notion of an *abstract predicate family*. A predicate family is a set of formulae indexed by class, where each formula describes how that class represents the abstraction. A predicate family is written by placing the first character in blackboard bold font, e.g. $\mathbb{P}red$, and is defined by a finite partial function from class names, $\mathbb{C}$, to the arities and formulae Ψ of the class's abstraction, e.g. $\mathbb{P}red : \mathbb{C} \rightharpoonup (\mathbb{N} \times \Psi)$. The family is well formed, written $\vdash \mathbb{P}red$ **ok**, if the following three constraints are meet.

- $\exists C \in \mathbb{C}.dom(\mathbb{P}red) = (\downarrow C)$ (where $(\downarrow C)$ is a down-closed subset of class names with respect to the subtyping relation)
- $\forall (n, \psi) \in cod(\mathbb{P}red). \mid FV(\psi) \mid \leq n$
- $\forall C, D. \quad D \prec C \wedge \mathbb{P}red(C) = (n, \psi) \wedge \mathbb{P}red(D) = (n', \psi') \Rightarrow n \leq n'$

The constraints ensure the family is defined for all subclasses, that the formula does not have more free variables than the specified arity and that the definition in any subclass can not have a smaller arity. We assume an ordering on variables that is used to bind predicate arguments to free variables, but will elide such details in this document. We define the *predicate family environment*, Ξ, as a set of predicate definitions indexed by predicate name.

We extend separation logic with *abstract predicates*, written $Pred_C(X; \vec{v})$ to indicate that the object X satisfies an element from the class C or below of the predicate family $\mathbb{P}$red with arguments $\vec{v}$. In object-oriented programming an object could be from one of many classes; abstract predicates mirror this by asserting that an element from a set of properties holds.

We add four axioms to separation logic, given in Figure 1. The first rule, Open, allows a predicate to be opened if the class of the object is known. The second allows an abstract predicate to be constructed given knowledge of the class the object belongs to, providing that the formula for the abstraction holds. The remaining two rules al-

Open
$\Xi \Vdash Pred_C(X; \vec{v}) \wedge X : C \Rightarrow \psi(X; \vec{v})$
where $\Xi(Pred_C) = (n, \psi)$ and $\vdash \Xi$ **ok**

Close
$\Xi \Vdash \psi(X; \vec{v}) \wedge X : C \Rightarrow Pred_C(X; \vec{v})$
where $\Xi(Pred_C) = (n, \psi)$ and $\vdash \Xi$ **ok**

Upcast
$\Xi \Vdash Pred_D(X; \vec{v}, \vec{v'}) \Rightarrow Pred_C(X; \vec{v})$
where $\vdash \Xi$ **ok**, $D \prec C$, $\Xi(Pred_C) = (n, \psi)$ and $\mid \vec{v} \mid = n$

Downcast
$\Xi \Vdash Pred_C(X; \vec{v}) \wedge X : E \Rightarrow \exists \vec{Y}.Pred_D(X; \vec{v}, \vec{Y})$
where $\vdash \Xi$ **ok**, $E \prec D \prec C$, $\Xi(Pred_D) = (n, \psi)$ and $\mid \vec{v}, \vec{Y} \mid = n$

Figure 1. Predicate Family Axioms

[1](1) is weaker than the usual constraint as it includes $this : Recell$, but it is sound with respect to the semantics of dynamic dispatch.

```
class Cell {
  Object contents;
  /*@ predicate
      Value(this;X) = this.contents |-> X /\ X != null@*/

  /*@ pre:  Value(this;X) @*/
  Object get() {...}
  /*@ post: Value(this;X) /\ ret = X  /\ X != null @*/

  /*@ pre:  Value(this;_) /\ newValue != null@*/
  void set(Object newValue) {...}
  /*@ post: Value(this;newValue) @*/
}

class Recell extends Cell {
  Object backup;
  /*@ predicate
      Value(this;X,Y)
           = (this.contents |-> X /\ X != null)
             * (this.backup |-> Y /\ Y != null) @*/

  /*@ pre:  Value(this; X,_) /\ newValue != null@*/
  void set(Object newValue) {...}
  /*@ post: Value(this; newValue,X) @*/
}
```

Figure 2. Example using predicate families

low abstract predicates to be cast up and down the inheritance hierarchy.

Figure 2 provides an example using abstract predicate families. The predicate family $\mathbb{V}alue$ is used to abstract away the fields from the pre- and post-conditions of the methods. We omit the subscript from an abstract predicate if it is the same as the enclosing class. An additional annotation is used to specify the *local* definition; how this class represents the abstraction. For the `Cell` class the local definition is

```
Value(this;X) = this.contents |-> X /\ X != null
```

In this example the axioms upcast and downcast prove the following two implications.

$$Value_{Recell}(X; v, v') \Rightarrow Value_{Cell}(X; v) \qquad \text{(upcast)}$$

$$Value_{Cell}(X; v) \wedge X : \mathit{Recell} \Rightarrow \exists v'.Value_{Recell}(X; v, v') \qquad \text{(downcast)}$$

Using these implications the `Recell`'s specification is compatible with the `Cell`'s, because we can use (downcast) to satisfy (1) and (upcast) for (2). We require the open and close axioms to prove that the method bodies meet the specifications given with respect to the predicate families.

So far we have only considered overridden methods; however, the `get` method is inherited by the `Recell` class. We must check the body of the method is correct with respect to both the `Cell`'s and `Recell`'s local definition of the `Value` predicate. Hence when validating the `Recell` we must also have the `Cell`'s code available.

We can summarise the class writer's burden as the following tasks.

- Give a local definition for all the required predicates.
- Give a pre- and post-condition for each method.
- Check method bodies satisfy the pre- and post-conditions.
- Check overridden methods are compatible with the superclass's specification.
- Check bodies of inherited methods against new local definitions of predicates.

Conclusions and Future work

The notion of *abstract predicate families* outlined in this document shows promise in providing a usable logic for reasoning about Java like languages. The predicate definitions have been treated in a global way in this document; however by scoping the definitions reasoning about encapulsation is greatly improved. This reasoning can also be used to prove properties about ownership transfer such as O'Hearn at al's example of a memory manager [2].

1 References

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

[2] P.W. O'Hearn, H.Yang, and J.C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, 2004.

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