Cheaper Reasoning with Ownership Types – Work in Progress –

Matthew Smith and Sophia Drossopoulou
{mjs198,sd}@doc.ic.ac.uk

Imperial College London

Abstract. We use ownership types to facilitate program verification. Namely, an assertion that has been established for a part of the heap which is unaffected by some execution will still hold after this execution. We use ownership and effects, and extend them to assertions to be able to make the judgement as to which executions do not affect which assertions. We describe the ideas in terms of an example, and outline the formal system.

1 Introduction

We outline ways in which ownership types can aid program verification in the context of a Hoare logic for a language with a type and effects ownership system. Previous work [4] describes Joe_1, a Java-like language with such a type system, where *disjoint* types characterise disjoint regions of the heap. Importantly, heap disjointness can be deduced statically.

In this paper we make use of these results to enhance a system for program reasoning: We assume a (sound) underlying Hoare logic for a language like Joe_1. We then describe "read effects" for assertions in terms of the effects from [4]. We can then extend the Hoare logic with a further rule, which states that assertions are preserved by executions whose write effect is disjoint from the assertions' read effect. We argue that the extended Hoare logic is still sound. In this paper we are assuming an underlying Hoare logic with non-standard assertions which allow predicate definitions. We are making that assumption because we believe that this allows for a natural way of reasoning about programs, and also because it allows us to express a compelling example. However our approach is orthogonal to the choice of underlying Hoare logic.

Section 2 discusses alternative approaches to dealing with aliases in program verification and the logics of separation. Section 3 introduces our example. Section 4 introduces Joe_1 through extension of our example with ownership types and discusses the features and properties of Joe_1 essential to our work. Section 5 discusses our contribution in terms of additions to a Hoare Logic system. Section 6 revisits our example in terms of the formal system. Section 7 discusses further directions for this work and concludes.

2 Related Work

Aliasing is a fundamental part of the OO paradigm but it makes reasoning about programs more cumbersome [9]. In a setting with aliasing, changing one object may affect properties of another. Several existing works on reasoning in the presence of aliasing, for OO languages at least, focus on reasoning safely in a language with pointers. [3] describes a system whereby one can safely reason about programs with pointers. The Hoare systems of [14] and [13] deal with a Java subset and so naturally deal with aliasing.

Our work differs from these in that we do not attempt to develop a Hoare logic for reasoning in the presence of aliasing, rather we *extend* such a Hoare logic. We use knowledge provided from static typing about the areas affected by execution, and the areas affecting assertions, in order to be able to deduce *preservation* of assertions in a simple way.

Attacking similar problems from a different direction (and at a different level of abstraction) is work on separation logics [15, 11]. Separation logics introduce logical primitives (* for example) which state that two areas of a heap are separate (disjoint). Assertions about one area of the heap remain unchanged when a separate area is modified. The frame rule of separation logics states that a Hoare sentence {P*R}C{Q*R} is valid if the sentence {P}C{Q} is valid and the computation C does not modify any of the free variables of R. The rule we introduce in Sec. 5.5 bears close resemblance to this. We achieve these goals through static typing and at a higher level.

3 Example

Figure 1 shows a simple project management example, written in a Java-like language . An `Employee` has a `Timetable`, a linked-list of `Project`/`Duration` pairs. We propose two properties for the `Timetable`s of an `Employee`, `e`:

`nonO(e)` The entries in `e.t` are non-overlapping (with respect to their start and finish dates)
`durInP(e)` The duration of each entry in `e.t` is contained within the duration of the project for that entry

We allow `Timetable`s to change and consider a simple mutation whereby all an employee's `Timetable` entries are delayed by some number of days i.e. the method `delay` in class `Employee`.

Suppose we wish to prove that the properties `nonO` and `durInP`, of an object `e`, are preserved when executing the `delay` method of `Employee` on `e`.

The possible side effects of performing `delay` on `e` make such assertions difficult to prove. `Employee`s might share `Timetable` objects and thus performing delay on `e` might affect the properties of any number of other `Employee`s. To ensure the properties hold through execution of `delay` we would need to directly reason about employees which might alias `Timetable`s of `e`. For example, assume two `Employee`s, `alice` and `bob`, which are guaranteed not to be

```
class Duration{
    int start;
    int end;

    void shunt(int period){
        start += period;
        end += period;
    }
}

class Employee{
    Timetable t;
    void delay(int period){
        // invokes delay on t
    }
}
```

```
class Timetable{
    Project p;
    Duration d;
    Timetable next;

    void delay(int period){
        // shunts d by period
        // and call on next
    }
}

class Project{
    Duration d;
}
```

Fig. 1. Project Example in almost-Java

aliases. Assume also, that for some program point, `p1` we have established that `nonO(alice.t)`, `nonO(bobt.t)`, `durInP(alice.t)` and `durInP(bob.t)`. Then we execute `alice.delay(23)`. Because of the potential for aliasing, all these assertions need to be re-established not just those pertaining to `alice`. However, if we knew that `bob`'s `Timetable` was not accessible through `alice`, we would argue that `alice.delay(23)` cannot affect `bob`'s timetable and expect that the assertions `nonO(bob.t)` and `durInP(bob.t)` should be preserved through execution of `alice.delay(23)`.

Ownership types [6, 4] give the machinery to restrict, through the type system, which objects may be accessible from which other objects. Joe$_1$, the extension of ownership to effects and disjointness ([4]), gives the machinery to deduce when certain expressions will not affect certain areas of the heap. In this paper we use the machinery from [4] to extend a Hoare logic.

4 Joe$_1$

Joe$_1$ is a Java-like language with ownership types and effects [4]. We give a brief outline here, which we hope allows an intuitive understanding of our current work.

In Joe$_1$ types are parameterised with ownership *contexts*(p,q etc.). Classes are defined using context variables which are bound to objects or the global context, `world`, when types are instantiated.

In Fig. 2 we add ownership types (and effects) to the example of Fig. 1. Figure 3 shows a possible instantiation of classes from Fig. 2. The rounded boxes represent objects in a UML-like notation. For example `alice:Employee` represents the object `alice` of class `Employee`. The square boxes in the diagram represent the ownership hierarchy, the box on the border being the owner of those immediately inside. Arrows are references i.e. fields. Note that as one expects for ownership systems, arrows can break out of boxes but not into them[1]. All the employees and projects are owned by some higher object, the company or department perhaps. Each `Employee` owns their `Timetable` objects, the `Timetable`s in turn own the underlying duration objects.

```
class Duration<owner>{
    int start;
    int end;

    void shunt(int period)
            wr this {
        start += period;
        end += period;
    }
}

class Employee<owner, proj>{
    Timetable<this, proj> t;
    void delay(int period)
            wr under(this) {
        // invokes delay on t
    }
}
```

```
class Timetable<owner, proj>{
    Project<proj> p;
    Duration<this> d;
    Timetable<owner,proj> next;

    void delay(int period)
            wr under(this){
        // shunts d by period
        // and call on next
    }
}

class Project<owner>{
    Duration<this> d;
}
```

Fig. 2. Project Example in with ownership types in Joe$_1$

Joe$_1$ differs from the original ownership types systems [6] as it allows (in controlled circumstances) access to objects across ownership boundaries i.e. breaking into boxes. `let` statements in Joe$_1$ allow assignment of any reachable statement to a local variable. The variable name can then be used as an ownership context and replaces `this` in types of objects from inside an ownership box. This is safe as the variable is a "short lived dynamic alias"–to borrow the terminology of [4]. The variable is visible only inside the `let` statement and the type system will not allow the referenced object to 'leak' out of the let through methods calls etc. We shall also be making use of these `let` statements in our assertions and type system though at some times we shall omit them in either context for brevity.

Two contexts are *disjoint* when they refer to different objects or one is `world` and the other isn't. Disjointness can be reasoned from type/ownership information stored in the typing environment. Disjointness of types is based on the class hierarchy and the disjointness of contexts. Types are disjoint when i) one is not a subclass of the other, or ii) any of their ownership parameters are disjoint, or iii) one is a subtype of some type which is disjoint from the other.

Since we know that each Employee owns its `Timetable` and each `Timetable` owns its durations we also know that no `Timetable` object can be shared between two `Employee`s (they have disjoint types). Thus, changes which only affect

[1] Therefore `alice` may <u>not</u> point to `t3` nor to `d1`.

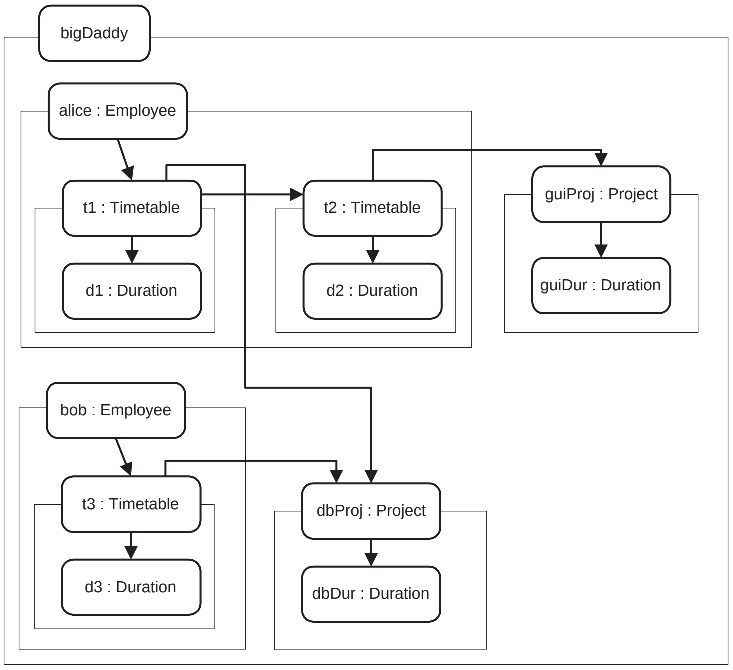

Fig. 3. Possible instantiation of code of Fig. 2

`alice`'s `Timetable` leave `bob`'s `Timetable` unaffected. Hence the properties `nonO` and `durInP` would be preserved for `bob`.

Types in $\texttt{Joe}_1$ come paired with an *effect* which describes where, in the heap, the typed expression will read or write. The read/write components are described by *shapes* (ϕ, ϕ' etc.) which characterise the affected portions of the heap. Thus effects have the form $\texttt{rd}\,\phi\ \texttt{wr}\,\phi'$ ranged over by ψ. The typing judgement of $\texttt{Joe}_1$ has the form $E\ \vdash e:t\ !\psi$ meaning that with respect to some type environment, E, the expression e has type t and produces at most the effect ψ when executed. For example:

$\texttt{alice}$:$\texttt{Employee}$<x,x >,$x \prec^* x\ \vdash \texttt{alice.delay(23)}$:$\texttt{void}$!$\texttt{wr}\ \texttt{under(alice)}$

for some x. Type environments store information of two kinds. The first is the conventional binding of program variables (including `this`) to types. The second kind of information is in the form of ordering of ownership contexts. $\texttt{Joe}_1$ puts certain restrictions on ownership parameters, specifically that the owner of an object should be inside all the other contexts. Along with additional rules this information enables us to reason properties such as disjointness.

Shapes describe sections of the heap in terms of contexts and are based on the ownership hierarchy. A shape could be just the object bound to a context parameter p, say. The shape $\texttt{under}(p)$ refers to the union of all the objects directly or indirectly owned by p. $\phi \cup \phi'$ is the shape derived from set union of ϕ and ϕ'.

It is possible to reason not only disjointness of types but also of shapes based on some type environment E. Two shapes are disjoint just when they have no objects in common. Read $E \vdash \phi \# \phi'$ as ϕ and ϕ' are disjoint. One can deduce some disjointness of context from the type environment and further disjointness of shapes follows intuition from set theory. For example, if two contexts, p, q, are disjoint and have a common owner then the shapes $\texttt{under}(p)$ and $\texttt{under}(q)$ are also disjoint.

Soundness of the shapes of expressions is an important result of [4]. Lemma 6.1 of that paper states that if an expression e having type t and effect $\texttt{rd}\,\phi\ \texttt{wr}\,\phi'$ is executed in the context of some heap H and stack S, then the areas of the heap not contained in ϕ' are unchanged in execution[2]. This result is important for this work as we apply the effects of $\texttt{Joe}_1$ to assertions.

5 Hoare Logics with Ownership

5.1 Assertions

We assume a logic with assertions, based on first order logic with predicates. Π, a predicate declaration, consists of a name, p, a list of arguments with their types, a shape ϕ, and a predicate body, B. ϕ is the declared shape of a predicate, in terms of the ownership parameters of the arguments and the arguments

[2] [4] includes a method of projecting an effect from its abstract contexts onto its underlying heap locations

themselves. Assertions and predicates are as follows:

$$
\begin{array}{lll}
\Pi & ::= & p\ (arg^*)\phi\ (B) \\
\Pi s & ::= & \Pi^* \\
arg & ::= & t\ z \\
z & ::= & \texttt{this}\ \mid x \\
B & ::= & \texttt{let}\, x\ = a\ \ \texttt{in}\, B\ \mid A \\
A & ::= & A_0\ \wedge A_1\ \mid \neg(A)\ \mid \texttt{ff} \mid a_0\ = a_1\ \mid\ p\ (z^*) \\
a & ::= & z \mid z.f\ \mid \texttt{null}
\end{array}
$$

where `this` is the current receiver, x ranges over local programme variables and predicate parameters and f ranges over field identifiers. Assertion expressions, a, are a restricted form of program expressions without side-effects.

$a_0 = a_1$ states that the two expressions are equal i.e. they refer to the same heap location. Conjunction and negation are as one expects. p ranges over predicate names and $p\ (\overline{z})$ is the application of a predicate.

We also employ the `let` statements of $\texttt{Joe}_1$ to allow predicates to look inside ownership boundaries. This is safe not only by virtue of safety of $\texttt{Joe}_1$ but intuitively as assertions (particularly equality) have no computational effect, as no reassignment is occurring no ownership properties can be compromised.

We shall adopt the $\overline{overscored}$ notation observed in [10] meaning an indexed sequence, for example $\overline{t\ x}$ stands for $t_1\ x_1, \ldots, t_n\ x_n$ and $\overline{a}$ stands for $a_1, \ldots, a_m$.

Note that we allow a very powerful language for predicates - much more powerful than [14, 11] as we allow predicate definitions. We need such predicates in order to express the example of Sec. 3. We omit quantification since it is not useful for our example. As we said in the Introduction, the main thrust of this work is orthogonal to the choice of Hoare logic. If it should prove difficult to find such a logic we are confident that we shall be able to apply our work to a more usual one.

Example The following predicate definition, declared with respect to the code of Fig. 2 captures `non0` from Sec. 3. It exploits recursion to traverse the list of `Timetable`. For convenience we assumed that the entries in `t` are ordered. We omit the details of the shape ϕ for now. In this example we write out `let` statements in full, in future we shall omit them for brevity's sake.

```
non0(Timetable<o, po> t)ϕ (
   let dur = t.d in
      let next = t.next in
         let ndur = next.dur in
            let start = ndur.start in
               end≤start ∧ non0(next)
)
```

5.2 Shapes of Assertions

The judgement $E \vdash B{:}\phi$ states that the assertion body, B, depends on the part of the heap characterised by ϕ, with respect to the type environment E. By depends we mean that all the variables, fields etc. referenced are in a part of the heap covered by ϕ. We take the program P and the predicate declarations $\mathit{\Pi s}$ as implicit. The judgement is defined as follows:

$$\frac{E \vdash A_0 : \phi_0 \qquad E \vdash A_1 : \phi_1}{E \vdash A_0 \wedge A_1 : \phi_0 \cup \phi_1}\ \text{ASS-CONJ} \qquad \frac{E \vdash A : \phi}{E \vdash \neg(A) : \phi}\ \text{ASS-NEG}$$

$$\frac{\begin{array}{l} p(\overline{t\ x})\phi(A) \in \mathit{\Pi s} \\ E \vdash \overline{z} : \sigma(\overline{t})!\overline{\psi} \end{array}}{E \vdash p(\overline{z}) : \sigma[\overline{x} \mapsto \overline{z}](\phi)}\ \text{ASS-PRED} \qquad \frac{\begin{array}{l} E \vdash a_0 : t_0!\mathtt{rd}\,\phi_0 \\ E \vdash a_1 : t_1!\mathtt{rd}\,\phi_1 \end{array}}{E \vdash a_0 = a_1 : \phi_0 \cup \phi_1}\ \text{ASS-EQ}$$

$$\frac{\begin{array}{l} E \ \vdash a : t\ !\mathtt{rd}\,\phi_0\ \ \mathtt{wr}\,\phi_1 \\ E, x : t \vdash B : \phi' \\ E, x : t \vdash \phi' \sqsubseteq \phi \\ E \vdash \phi \end{array}}{E \vdash \mathtt{let}\ x = a\ \mathtt{in}\ B : \phi}\ \text{ASS-LET} \qquad \frac{}{E \vdash \mathtt{ff} : \emptyset}\ \text{ASS-FF}$$

The rules for evaluating the shape of a judgement are similar to those of Joe_1. In rules ASS-EQ, ASS-RED and ASS-LET we use the typing judgement of Joe_1 (as described in Sec. 4) to type the assertion expressions, a, since these are valid Joe_1 syntax. Note that we do not include the shape of the parameters (in rule ASS-PRED) or a (in rule ASS-LET) since we are only interested in the shape of the assertions; we do not accumulate computational effect as in Joe_1. The shape of a predicate is declared in terms of the declared contexts of its parameters and the parameters themselves. The shape of a predicate application is the declared shape under the substitution which maps the ownership variables of the predicate declaration to those of the supplied arguments and the formal arguments to the actual parameters.

An assertion, A, and the predicate definitions, $\mathit{\Pi s}$, are well-formed with respect to a type environment (and also implicitly the program and the predicate declarations) if all assertion expressions a, are well-formed, and types of arguments to predicates match the declared types. Using the type system of Joe_1 to find the shape of a_0 and a_1 has the added benefit that assertions which have a shape are 'well-formed' i.e. expressions of the form $z.f$ are type correct and thus we needn't check elsewhere that field, f exists.

Well-formedness of predicate definitions must be checked separately to ensure the declared shape is valid for the body of the predicate.

$$\frac{\overline{t\ x}, \mathit{constr}(\overline{t\ x}) \vdash B : \phi' \qquad \overline{t\ x}, \mathit{contr}(\overline{t\ x}) \vdash \phi' \sqsubseteq \phi}{\vdash p(\overline{t\ x})\phi(B)}$$

A predicate declaration is well-formed if i) the type environment generated from the declared types of its parameters is valid, ii) the shape of its body, in the context of the constructed environment is well-formed with shape ϕ', and iii) the calculated shape of the body is a subshape of the declare shape (which also requires that ϕ is well-formed w.r.t. the environment. The type environment pairs the formal parameters to their declared types and has ownership constraints (generated by *constr*) as required by $\texttt{Joe}_1$ (we omit details).

Note that the body of the predicate, A, is typed in the environment $\overline{t\ x}$, created by the formal parameters of the predicate. Therefore the identifier `this` may not appear in A (nor indeed any variable names other than the formal parameters), thus reflecting the fact that the predicates are not declared within classes.

Example The shape `under(o)` would be a valid shape for the predicate `non0` defined previously[3].

5.3 Satisfaction of Assertions

We express satisfaction of an assertion with respect to a heap, H, and a stack, S, by the *partial* function $\mathcal{S}$ mapping assertions, heaps and stacks (and implicit predicate declarations) to $\{\texttt{tt}, \texttt{ff}, \perp\}$[4]

$\mathcal{S}$ is defined as follows:

$$
\begin{array}{ll}
\mathcal{S}(H, S, \texttt{let}\ x = a\ \texttt{in}\ B) & = \mathcal{S}(H, S[x \mapsto (H,S)(a)], B) \\
\mathcal{S}(H, S, A_0 \wedge A_1) & = \mathcal{S}(H, S, A_0) \wedge \mathcal{S}(H, S, A_1) \\
 & \quad \text{if } \mathcal{S}(H, S, A_i) \neq \perp \quad i \in \{0, 1\} \\
\mathcal{S}(H, S, \neg(A)) & = \neg(\mathcal{S}(H, S, A)) \text{ if } \mathcal{S}(H, S, A) \neq \perp \\
\mathcal{S}(H, S, \texttt{ff}) & = \texttt{ff} \\
\mathcal{S}(H, S, a_0 = a_1) & = (H,S)(a_0) = (H,S)(a_1) \text{ if } (H,S)(a_i) \neq \perp \quad i \in \{0,1\} \\
\mathcal{S}(H, S, p(\overline{z})) & = \mathcal{S}(H, S, A[\overline{x}/\overline{z}]) \text{ where } p(\overline{t\ x})\phi(B) \in \mathit{\Pi s} \\
\mathcal{S}(H, S, A) & = \perp \ \text{ otherwise}
\end{array}
$$

The auxiliary function $(H,S)(\ldots)$ maps expressions to the appropriate heap location.

[3] Since `int`s are passed by value in Java we assume they are `under` the object referencing them

[4] $\mathcal{S}$ has to be a *partial* function since we can define inconsistent predicates in our system, for example:

$$p\ (\text{c}<\ldots>\ x)\emptyset\ (\neg(p\ (x)))$$

Alternatively we could have defined satisfaction through the interpretation of predicates from $\mathit{\Pi s}$ into H,S as e.g. in [1]. The function $\mathcal{S}$ is undecidable in the general case however this does not affect the applicability of our result.

$$\begin{array}{ll}(H,S)(x) & = S(x)\\ (H,S)(a.f) & = \begin{cases} H((H,S)(a))(f) \text{ if } (H,S)(a) \neq \perp, \texttt{null} \\ \perp \text{ otherwise}\end{cases}\\ (H,S)(\texttt{this}) & = S(\texttt{this})\\ (H,S)(\texttt{null}) & = \texttt{null}\end{array}$$

5.4 The Assumed Hoare Logic

Suppose there exists a Hoare Logic for the language with sentences of the form:

$$E \vdash A_0 \ \{e\ \} A_1$$

where E is a typing environment, binding `this` and local variables to types. The program P, and predicate definitions Πs are taken to be implicit. Soundness of the Hoare logic is defined as follows:

Definition 1. *The Hoare logic is sound if:*
$\forall E, A_0, A_1, e, H, H'\ S, v$:

$$\left.\begin{array}{l} E \vdash H \\ E \ \vdash e : t \ !\psi \\ < H; S; e > \overset{\psi'}{\rightarrow} < H'; v > \\ \mathcal{S}(H, S, A_0) = \texttt{tt} \\ E \vdash A_0 \ \{e\ \} A_1 \end{array}\right\} \Rightarrow \mathcal{S}(H', S, A_1) = \texttt{tt}$$

5.5 Hoare Logic Extension

Assuming a system as described previously, we define an *ownership extended* Hoare logic as follows:

$$\frac{E \vdash A_0 \ \{e\ \} A_1 \qquad E \vdash A : \phi \qquad E \ \vdash e : t \ !\texttt{rd}\ \phi_0 \ \ \texttt{wr}\ \phi_1 \qquad E \vdash \phi \ \# \ \phi_1}{E \vdash_O A_0 \ \wedge A \ \{e\ \} A_1 \ \wedge A}$$

The first judgement, $E \vdash A_0 \ \{e\ \} A_1$, is a derivation of the assumed Hoare logic. It does not utilise the properties of ownership types. The second judgement, $E \vdash A{:}\phi$, is the calculation of the shape of the assertion, A, as described in Sec. 5.2. $E \ \vdash e : t \ !\texttt{rd}\ \phi_0 \ \ \texttt{wr}\ \phi_1$ is the typing judgement of $\texttt{Joe}_1$ as described in Sec. 4. $E \vdash \phi \ \# \ \phi_1$ is the disjointness judgement described in Sec. 4. Thus, the Hoare logic rule we propose states that any assertion which is true before the execution of an expression and which inhabits a part of the heap which is not written to by the expression, is true after execution of the expression. Note that the subscript on $\vdash$ makes the distinction between judgements of the original and the "ownership-aware" Hoare logic.

The following conjecture states that the extension we suggest preserves soundness of the Hoare logic.

Conjecture 1. If a Hoare logic $\vdash$ is sound, then the ownership aware extension, $\vdash_O$, is also sound.

We prove that conjecture using the following conjecture which states that execution of an expression e with disjoint shape from assertion A preserves the satisfaction of that assertion.

Conjecture 2.

$$\left.\begin{array}{l} E\ \vdash e : t\ !\texttt{rd}\,\phi_0\ \ \texttt{wr}\,\phi_1 \\ E \vdash A : \phi \\ E \vdash \phi \ \#\ \phi_1 \\ E \vdash H \\ < H; S; e > \xrightarrow{\psi} < H'; v > \\ \mathcal{S}(H, S, A) = \texttt{tt} \end{array}\right\} \Rightarrow \mathcal{S}(H', S, A) = \texttt{tt}$$

We expect this to follow as a corollary of Lemma 6.1 in [4] which states that on execution of some instruction the portion of the heap which is not in the shape written by the execution is unchanged.

6 Once More with Formality

We shall now roughly show how one might apply our system in the context of the example of Fig. 2.

Consider the following set of predicate definitions, intended to capture the properties discussed in Sec. 3:

```
Πs =
     within(Duration<o1> d1, Duration<o2> d2) under(o1) ∪under(o2)
         ( d1.start ≥ d2.start ∧ d1.end ≤ d2.end )

     nonO(Timetable<o, po> t) under(o)
         ( t.next ≠ null⇒
             t.d.end ≤ t.next.d.start ∧ nonO(t.next) )

     durInP(Timetable<o,op> t) under(op)
         ( t.next ≠ null⇒ within(t.d,t.p.d) ∧ durInP(t.next) )


     nonO(Employee<o,op> e) under(() o)
         ( nonO(e.t) )

     durInP(Employee<o,op> e) under(op)
         ( durInP(e.t) )
```

Suppose:

$$E = \texttt{alice:Employee<bigDaddy, bigDaddy>},\\ \texttt{bob:Employee<bidDaddy, bigDaddy>}$$

and we also know that $E \vdash \texttt{alice\#bob}$ (which we have if we know `alice` and `bob` are not aliases).

By application of type rules of [4] we obtain that:

$$E \vdash \texttt{alice.delay(23)} : \texttt{void!wr}\,\texttt{under}(\texttt{alice})$$

By further application of the rules of [4] and those of Sec. 5.2 we obtain

$$E \vdash \texttt{nonO(alice)} : \texttt{under}(\texttt{alice})$$
$$E \vdash \texttt{durInP(alice)} : \texttt{under}(\texttt{bigDaddy})$$
$$E \vdash \texttt{nonO(bob)} : \texttt{under}(\texttt{bob})$$
$$E \vdash \texttt{durInP(bob)} : \texttt{under}(\texttt{bigDaddy})$$

Therefore, by application of the Hoare logic extension rule, we obtain that:

$$E \vdash_O \texttt{nonO(bob)}\ \{\ \texttt{alice.delay(23)}\ \}\ \texttt{nonO(bob)}$$

Furthermore, if we were able to derive (the hard way) that:

$$E \vdash \texttt{nonO(alice)}\ \{\ \texttt{alice.delay(23)}\ \}\ \texttt{nonO(alice)}$$

then we could also obtain (the cheap way) through the extension rule that:

$$E \vdash_O \texttt{nonO(bob)} \wedge \texttt{nonO(alice)}\ \{\ \texttt{alice.delay(23)}\ \}\\ \texttt{nonO(bob)} \wedge \texttt{nonO(alice)}$$

So, we were able to obtain the preservation of the assertion `nonO(bob)` after the execution of `alice.delay(23)` through mechanisms of the type system.

Note, however, that we are not able to deduce that:

$$E \vdash_O \texttt{durInP(bob)}\{\texttt{alice.delay(23)}\}\texttt{durInP(bob)}$$

even though we know `alice.delay(23)` only shifts the timetable of `alice` and thus does not affect the validity of `durInP(bob)`. This is so because in our current system the effects/shapes are too coarse. In further work we aim to refine shapes through e.g. data groups [7, 12].

7 Conclusions and Further Work

We have shown how to extend a Hoare logic with ownership awareness and how this will support locality of reasoning, thus allowing the re-establishing of properties after execution of some code in a cheap way.

Our approach is "parametric" in the choice of the underlying Hoare logic. Any logic which supports assertions of the kind described in Sec 5.1 can be extended and soundness will be preserved. Thus we should be able to "plug" and experiment with various such logics.

Our approach should also be extensible to be made "parametric" with respect to the containment system. We have used ownership types in order to be able to isolate the shape of expressions and assertions. We should be able to generalise to any approach which supports the expression of shapes of expressions and notions of disjointness, e.g. datagroups, balloons etc.

The current extension of the Hoare logic is limited in that we may only apply our extension at the last step. The full extension of the the Hoare logic would allow the application of the new rule at any part of the derivation. We have not considered this here as we were not sure how to prove preservation of soundness. It has been recently pointed out to us [5] that if all rules of the underlying system preserve soundness then the extension also preserves soundness. We shall describe the full extension in future work.

As well as the above, further work includes:

- investigation of suitable Hoare logics for application of these results
- details and proofs of the presented work
- further examples which will demonstrate the benefits or point to further extensions
- an extension of Joe_1 to support datagroups [7, 12, 2] so that we can better localise the shapes and effects
- incorporation of further ideas from ownership e.g. uniqueness [5]

8 Acknowledgements

We are grateful to Dave Clarke, John Potter and our colleagues from SLURP for useful discussions. We are particularly grateful to Mariangiola Dezani and Joe Wells for their help and suggestions for the full extensions of Hoare logic. We would also like to thank the anonymous referees for important and valuable comments.

References

[1] Krzysztof R. Apt. Ten years of hoare's logic: A survey.part i. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[2] G.M. Bierman and M.J. Parkinson. Effects and effect inference for a core java calculus. In *WOOD 2003*. ENTCS, 2003.

[3] Richard Bornat. Proving pointer programs in hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] Private communication with Joe Wells and Mariangiola Dezani

[4] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.

[5] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP 2003*, 2003.

[6] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.

[7] Aaron Greenhouse and John Boyland. An object-oriented effects system. *Lecture Notes in Computer Science*, 1628:205–230, 1999.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[9] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[10] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Lecture Notes in Computer Science*, 1850:129–154, 2000.

[11] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[12] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 144–153. ACM Press, 1998.

[13] Cees Pierik and Frank S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. http://www.cs.uu.nl/research/techreps/UU-CS-2003-010.html, 2003.

[14] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symosium un Programming (ESOP '99)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[15] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.