Verifying Generics and Delegates

Kasper Svendsen[1], Lars Birkedal[1], and Matthew Parkinson[2]

[1] IT University of Copenhagen, {kasv,birkedal}@itu.dk
[2] University of Cambridge, Matthew.Parkinson@cl.cam.ac.uk

Abstract. Recently, object-oriented languages, such as C♯, have been extended with language features prevalent in most functional languages: parametric polymorphism and higher-order functions. In the OO world these are called generics and delegates, respectively. These features allow for greater code reuse and reduce the possibilities for runtime errors. However, the combination of these features pushes the language beyond current object-oriented verification techniques.
In this paper, we address this by extending a higher-order separation logic with new assertions for reasoning about delegates and variables. We faithfully capture the semantics of C♯ delegates including their capture of the l-value of a variable, and that "stack" variables can live beyond their "scope". We demonstrate that our logic is sound and illustrate its use by specifying and verifying a series of interesting and challenging examples.

1 Introduction

There has been a recent trend for object-oriented languages, like C♯, to adopt features such as generics (parametric polymorphism) and delegates (first class functions). These features help the programmer improve code reuseability and reliability by providing greater abstraction at the level of types. However, C♯ type safety falls short of proving that programs do the right thing, it simply prevents certain classes of errors.

Program verification enables proofs that programs do the right thing, and there has been a lot of work on verifying object-oriented languages, for example [6, 8, 2, 1]. However, the majority of this work falls short of reasoning about features such as generics or delegates.

Higher-order separation logic (HOSL) [3, 9] and Hoare Type Theory (HTT) [11, 18] have both been developed to reason about higher-order functional languages. HOSL and HTT uses quantification over propositions to allow reasoning about parametric polymorphism, and HTT uses nested Hoare triples to allow reasoning about first class functions.

We borrow ideas from HOSL to extend a separation logic for object-oriented programs [16, 14, 15] to reason about generics and delegates. Unfortunately, the combination is not straightforward. Neither, HTT nor HOSL deal with the combination of mutable variables and first class functions that we have in C♯. In particular, anonymous C♯ delegates have a surprising behaviour for capturing

variables. First, they capture the l-value, i.e., the location of the variable, rather than just its value, and secondly, captured variables can live beyond their static scope, if they have been captured, and can thus not always be allocated on the stack. These two aspects complicate the logic.

Just as C♯ hides these implementation details from the programmer, so should the program logic and in particular one should be able to reason about captured and escaped variables as if they were on the stack. Our key aims for reasoning about delegates and captured/escaped variables are thus that (1) Hoare's assignment rule should remain valid; and (2) reasoning should be standard for programs that do not use these complicated features.

Simply throwing all variables onto the heap would violate both (1) and (2). Likewise, treating variables as resource [5, 17] would also complicate proofs of programs that do not take advantage of delegates with captured variables. Instead, we will use an operational semantics where local variables do reside on the stack and extend the assertion logic with new assertions for reasoning about the location and value of stack variables.

For escaped variables we need to be able to assert the existence of an escaped variable on the stack and reason about its value, and for a delegate with captured variables we need to be able to reason about the captured variables' current values as well as their value at the call- and exit-site of the delegate. To accomplish this, we introduce a new assertion, `lookup L as x in P`. Here `L` is a term denoting a location on the stack and `lookup` simply binds the contents of this stack location to `x` in the assertion `P`. Of course, we need to ensure that this does not introduce any aliasing in reasoning about the stack, as this would invalidate Hoare's assignment rule.

Our resulting system has the following properties:

1. Programs that use generics (but do not use delegates) can be verified using higher-order separation logic reasoning.
2. Programs that use named delegates can be verified using nested triples [19].
3. Programs that use anonymous delegates with escaping local variables can be verified using `lookup` to refer to escaped variables.

In summary, the key contributions of the paper are:

- Application of higher-order separation logic, in particular quantification over predicates in the assertion logic, to allow reasoning about programs that use generics.
- The first logic to deal with C♯ 2.0 style delegates, involving anonymous methods and variable capture.
- Illustration of the utility of our logic with a series of examples.
- Formal semantics and soundness of our logic.

We stress that the soundness of the logic is non-trivial; indeed, we develop a new model of separation logic in order to reason about delegates that can capture mutable variables.

In this paper, we do not address inheritance. However, we believe this is an orthogonal issue that can be dealt with in the same way as Parkinson and Bierman [16, 15].

The rest of the paper is structured as follows. We begin, in §2, by showing how a higher-order separation logic can be used to reason about object-oriented programs using generics. Then, in §3, we carefully explain the intricacies of C♯ delegates and how they can capture variables, and extend the logic to reason about delegates. In §4, we present a short case study using the logic. We then, §5, rigorously define the formal semantics of the logic and demonstrate its soundness. Finally, we close by discussing related work, §6 and conclusions and future work, §7.

See [20] for an extended technical appendix to the present paper with more details and proofs.

2 Generics

Generics allow for greater code reuse, by allowing the programmer to abstract over types. In this section we illustrate how to reason about generic methods using higher-order features of the logic.

Consider a simple example of a Node class, that stores items of a generic type:

```
public class Node⟨X⟩ {
  Node⟨X⟩ next;
  X item;
}
```

Here we see the Node class takes a type parameter X that is the type of the elements it stores.

Typically, in reasoning about Java or C♯ programs the representation of classes in separation logic is done through predicates. Generics extends the notion of a class to enable it to take other classes as arguments, hence we must similarly extend the logical representation to predicates that can take predicates as arguments. Thus we leave first-order separation logic and move to higher-order separation logic.

Our assertion logic is an intuitionistic higher-order logic over a simply typed term language, derived from [3]. We use ω to range over types, which are generated by the following grammar:

$$\omega ::= \omega \times \omega \mid \omega \to \omega \mid \texttt{Val} \mid \texttt{Class} \mid \texttt{Prop} \mid \texttt{Int}.$$

The set of types is closed under products and function spaces. `Val` is the type of mathematical values; it includes all $C^\sharp$ values and strings, and is closed under formation of pairs, such that mathematical sequences and other mathematical objects can be conveniently represented.[3] The type `Class` represents the set of

[3] We use a single universe `Val` for the universe of mathematical values to avoid also having to quantify over types in the logic and because such a single universe is also used in the jStar tool [7].

all syntactic generic class identifiers as generated by the grammar presented in Section 5. Assertions are terms of type `Prop`.

Terms are typed with the typing judgment

$$\phi; \psi \vdash \mathsf{M} : \omega,$$

where ϕ is a program variable context, and ψ a logical variable context. Throughout the paper we assume the two contexts to be disjoint and that variables are not repeated in either context. (In the extended version [20] of this paper, there is also a context Δ of type variables; but since we do not need that for the examples in this paper we have omitted it, to simplify the notation.)

The usual specification style of separation logic for describing data structures is to define a predicate at the meta-level which describes how a mathematical structure is represented on the heap. Using higher-order logic allows us to define these representation predicates inside the logic [3], and to define them abstractly in terms of representation predicates for abstracted types.

For the generic Node class defined above we can, for instance, define a generic representation predicate of the following type:

$$\vdash \mathit{list} : \mathtt{Val} \times \mathtt{Val} \times (\mathtt{Val} \times \mathtt{Val} \to \mathtt{Prop}) \to \mathtt{Prop}.$$

The predicate is defined as follows:

$$\begin{aligned} \mathit{list}(n, [], \mathsf{P}) &\stackrel{\text{def}}{=} n =_{\mathtt{Val}} \mathtt{null} \\ \mathit{list}(n, v :: xs, \mathsf{P}) &\stackrel{\text{def}}{=} \exists n', \mathsf{x} : \mathtt{Val}.\ n.\mathit{next} \mapsto n' * n.\mathit{item} \mapsto \mathsf{x} \\ &\qquad * \mathsf{P}(\mathsf{x}, v) * \mathit{list}(n', xs, \mathsf{P}). \end{aligned}$$

To simplify notation, we use standard notation for mathematical sequences ([] for the empty sequence, and $v :: xs$ for the sequence with head element v and tail xs) that are officially represented as elements in the type `Val`.

Thus $\mathit{list}(n, xs, \mathsf{P})$ expresses that n is a representation of a list of objects, where each object represents the corresponding value in the sequence xs, as described by representation predicate P. A reference n represents the sequence xs if n is null and xs is the empty sequence, or if n is a reference to an object o such that o's item field is a representation of the head of xs and o's next field is a representation of the tail of the sequence.

We can now give the constructor the following specification:

```
public Node(X item, Node next) { ... }
∀v : Val, ys : Val, P : Val × Val →Prop,
  { P(item, v) ∗ list(next, ys, P) }
  { r. list(r, v::ys, P) }
```

where the r in the post-condition is a binder for the return value. So the constructor takes a reference to an X object, which represents the value v and a reference to a Node⟨X⟩ object, representing the list of values ys and produces a Node⟨X⟩ object which represents the list of values $v :: ys$.

Here is a simple example of how we can specify a generic append operation:

```
class List {
  public static Node⟨X⟩ append⟨X⟩(Node⟨X⟩ front,
                                  Node⟨X⟩ tail) {
    if(front == null)
      return tail;
    else {
      Node⟨X⟩ tmp = append⟨X⟩(front.next, tail);
      front.next = tmp;
      return front;
    }
  } ∀P : Val × Val →Prop, xs, ys : Val.
      { list(front, xs, P) ∗ list(tail, ys, P) }
      { r. list(r, xs@ys, P) }
}
```

The proof is straightforward, and we provide a detailed proof outline in the extended version [20].

Next we consider a client of this class that appends two lists of numbers.

```
{ list(xs,[1,2,3],Int) ∗ list(ys,[4,5,6],Int) }
   zs = List.append⟨Integer⟩(xs,ys);
{ list(zs,[1,2,3,4,5,6],Int) }
```

Here $Int : \texttt{Val} \times \texttt{Val} \to \texttt{Prop}$ is a predicate such that $Int(i, v)$ holds if i points to an Integer object representing the number v.

Thus we see that it is very simple to specify generic programs by being correspondingly generic in the logic, via quantification over predicates. This is as one would hope, given the earlier work on HOSL and HTT mentioned in the introduction. We now turn to the more challenging issue of delegates.

3 Delegates

In this section we recall the semantics of C♯ delegates (Subsection 3.1); show how to reason about methods using delegates via the delegate call rule (Subsection 3.2); and then show how to specify named delegates (Subsection 3.3) and anonymous delegates (Subsection 3.4).

3.1 Understanding C♯ delegates

A C♯ delegate type describes the parameter types and return type of a method, and is thus very similar to an ML function type. An instance of a C♯ delegate type refers to a method with a compatible signature, and is thus very similar to an ML function (in this paper we ignore that a delegate can refer to multiple such methods, all of which get invoked when the delegate is invoked).

We will use the following delegate types in our examples in this paper:

```
public delegate void Action();
public delegate void Action⟨X⟩ (X x);
public delegate Y Func⟨Y⟩ ();
public delegate Y Func⟨X,Y⟩ (X x);
```

The first two are used for delegates that do not return a result, and the last two for delegates that return a result of type Y. The first and third do not take an argument and the second and fourth take an argument of type X. Overloading will resolve which Action or Func type is meant.

Methods can then take delegates as arguments; for instance, here is how one writes an apply method that takes a delegate with no formal parameters as argument and calls it:

```
public void apply(Action f) { f(); }
```

This might look very "functional" and simple, but, of course, the argument delegate may have lots of effects, using local state on both the stack and the heap.

As an example of the kind of programs we are interested in verifying, consider the following imperative fold method:

```
public static void fold⟨X⟩(Node⟨X⟩ lst, Action⟨Node⟨X⟩⟩ f) {
  if(lst != null) { Node⟨X⟩ tmp = lst.next; f(lst); fold(tmp, f); }
}
```

This method takes a list and a delegate as arguments and applies the delegate to each element in the list, from left to right. An important feature of this implementation is that it remembers the value of the next pointer before calling the delegate f. This allows the delegate to update the next pointer of the current node while preserving that fold applies the delegate to all the nodes of the list. No accumulator value is passed explicitly as an argument to the delegate, since the delegate can maintain an accumulator value itself using local state, as in the following example:

```
class Reverse⟨X⟩ {
  Node⟨X⟩ head;
  public void flip(Node⟨X⟩ x) { x.next = head; head = x; }
  public Node⟨X⟩ reverse(Node⟨X⟩ lst)
     { head = null; fold⟨X⟩(lst, flip); return head; }
}
```

This method uses the fold method to reverse a list in-place. It uses the head field to point to the head of the part of the list reversed so far and folds the flip method, which adds a node to the front of head, over the given list, thus ending up with a reversed list. In Section 3.3, we show how to reason about delegates referring to named methods.

Anonymous methods introduce new complications in the form of captured and potentially escaping variables, because they are declared inline and are allowed to refer to local variables from enclosing scopes. To illustrate, consider the following example:

```
public Func⟨int⟩ counter() {
  int x=0;
  return delegate () { return ++x; };
}
```

This method returns a delegate that has captured the l-value (the location) of the local variable x. Calling the returned delegate will return the number of times the delegate has been called. The strange aspect to this code is that x is scoped to the body of the method counter and thus escapes its scope when the delegate is returned.

One can thus also use captured variables to associate local state with a delegate; for instance, one can implement reverse using a captured variable in place of the head field as follows:

```
public static Node⟨X⟩ reverse⟨X⟩(Node⟨X⟩ lst) {
  Node⟨X⟩ head = null;
  fold⟨X⟩(lst, delegate (Node⟨X⟩ x) { x.next = head; head = x; });
  return head;
}
```

C♯ compilers compile such programs by rewriting them into equivalent C♯ programs without the inline delegate or the captured variable, by introducing a new class with a field corresponding to the captured variable and a method corresponding to the inline delegate. So, while head appears to be a local variable on the stack to the programmer, in fact it ends up on the heap, after the rewriting done by the C♯ compiler.

One could verify the rewritten program, without local state on the stack, with a higher-order separation logic with nested Hoare triples. However, this would mean explicitly reasoning about aliasing of captured local variables in the logic, even though there is no aliasing of captured local variables in the original program. Alternatively, one could devise a storage model for the program logic with no stack at all, but only a heap and an environment, so that all mutation would happen in the heap (as for ML-like languages). For imperative languages, such as Java and C♯, such an approach would, however, lead to proof rules that are more complicated to use than the standard Hoare / Separation logic rules. Instead, we define a storage model where all local variables do reside on the stack and a program logic that treats captured variables as normal stack variables. The operational semantics thus never pops values from the stack, to allow for references to escaped variables.

3.2 Reasoning about Methods that use Delegates

To reason about delegates we extend our higher-order assertion logic with nested Hoare triples [19], written

$$\mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle,$$

for asserting that M denotes a reference to a method satisfying the given specification. The specification includes a context, $\bar{u}$, that specifies the delegate's formal parameters. Variable d binds the return value in Q. When d does not occur in Q we omit d. The typing rule for the new assertion is given below.

$$\frac{\phi;\psi \vdash \mathsf{M} : \texttt{Val} \qquad \phi;\psi,\bar{u} \vdash \mathsf{P} : \texttt{Prop} \qquad \phi;\psi,\bar{u},\mathsf{d} : \texttt{Val} \vdash \mathsf{Q} : \texttt{Prop}}{\phi;\psi \vdash \mathsf{M} \mapsto \langle(\bar{u}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \texttt{Prop}} \tag{1}$$

As the typing rule shows, we allow P and Q to refer to program variables and logical variables from the context. In the case of program variables, references to $x \in \phi$ in P and Q refer to x's current value on the stack.

Using this nested triple assertion, we can specify apply as follows:

```
public void apply(Action f) { f(); }
∀P,Q : Prop.{ P ∗ f ↦ ⟨{P}_{Q}⟩ } { Q ∗ f ↦ ⟨{P}_{Q}⟩ }
```

The specification universally quantifies over the delegate's pre and post-condition, to allow apply to be called with any delegate. For modularity, it is essential that this specification is strong enough for verifying calls with delegates with local state on the heap and/or stack — in the following sections we show that this is indeed the case by demonstrating how to verify calls to apply with named delegates (with local state on the heap) and anonymous delegates (with local state on both the heap and the stack).

The pre and post-condition both contain the delegate assertion $f \mapsto \langle\{P\}\_\{Q\}\rangle$. This specifies what the delegate parameter will do. The pre-condition additionally contains the delegate's pre-condition P, which enables the body to call the delegate. The post-condition of apply contains the post-condition of the delegate. We can see the verification outline as

```
{ P ∗ f ↦ ⟨{P}_{Q}⟩ }
  f()
{ Q ∗ f ↦ ⟨{P}_{Q}⟩ }
```

To call a delegate we require two things: an assertion about the delegate being called, here $f \mapsto \langle\{P\}\_\{Q\}\rangle$, and the pre-condition specified in that assertion, here P. The post-condition of the call is simply the pre-condition with the delegate's pre-condition replaced by the delegate's post-condition, Q.

The apply method did not deal with parameters. We can adapt the apply method for delegates with a single argument as follows

```
public void apply⟨X⟩(Action⟨X⟩ f, X x) { f(x); }
∀P,Q : Val →Prop.{ P(x) ∗ f ↦ ⟨(v).{P(v)}_{Q(v)}⟩ }
                 { Q(x) ∗ f ↦ ⟨(v).{P(v)}_{Q(v)}⟩ }
```

The logical variables P and Q take a parameter for the argument given to the delegate. In the pre-condition of apply, we have P(x), which is the pre-condition of the delegate instantiated with the argument with which it will be called. We give an outline of the verification of the body below.

$\{\ \mathsf{P(x)} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$
$\quad\{\ \mathsf{P(v)[x/v]} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$
$\quad\mathsf{f(x)}$
$\quad\{\ \mathsf{Q(v)[x/v]} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$
$\{\ \mathsf{Q(x)} * \mathsf{f} \mapsto \langle(\mathsf{v}).\{\mathsf{P(v)}\}\_\{\mathsf{Q(v)}\}\rangle\ \}$

Here we see that we must provide the pre-condition of the delegate with the argument substituted into the parameter, $\mathsf{P(v)[x/v]}$, and similarly for the post-condition.

Now we have presented informally how we can call delegates; next, we present the formal details. We begin by presenting the overall form of our proof system and then the general proof rule for calling a delegate. Hoare triples in our specification logic take the form:

$$\Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}$$

where Γ is a method context, assigning specifications to methods, and ϕ and ψ are as for assertions;[4] P and Q are assertions, both well-typed in ϕ, ψ; and M is a finite set of variables (an over-approximation of the set of variables in ϕ that s might modify, explained in Section 3.4.)

The proof rule for calling a delegate is shown below.

$$\frac{\mathsf{r} \notin \mathrm{FV}(\mathsf{R}) \qquad \mathsf{R} = \mathsf{y} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d.Q}\}\rangle}{\Gamma; \mathsf{r}, \mathsf{y}, \bar{\mathsf{x}}; \psi \vdash \{\mathsf{R} * \mathsf{P}[\bar{\mathsf{x}}/\bar{\mathsf{u}}]\}\mathsf{r} = \mathsf{y}(\bar{\mathsf{x}})\{\mathsf{R} * \mathsf{Q}[\bar{\mathsf{x}}/\bar{\mathsf{u}}, \mathsf{r}/\mathsf{d}]\} \triangleleft \{\mathsf{r}\}}$$

The rule expresses that if y points to a delegate, then we can call the delegate if its precondition P holds, with actual arguments $\bar{\mathsf{x}}$ substituted in for the formal arguments $\bar{\mathsf{u}}$. To simplify the presentation of the proof system, we do not allow delegates to modify their formal arguments (this is formalized in the delegate definition rule below). Furthermore, the rule only applies to delegates that have not captured r, y or $\bar{\mathsf{x}}$ (we will see why in Section 3.4). Hence, neither the formal nor the actual parameters will be modified by the call, so we can also substitute the actual arguments for the formal arguments in the post-condition along with r for the return-value binder d.

3.3 Reasoning about Named Delegates

A delegate referring to a named method can refer to a static method, an open instance method or a closed instance method; in this section we will consider delegates referring to closed instance methods (i.e., a reference to a method and a target object) as we can associate local state with such delegates through the target object. Furthermore, we will only consider delegates referring to exactly one method.

The difficulty in reasoning about delegates is that they can maintain local state, however, since C♯ lacks global variables, methods and named delegates can

[4] Again, in the extended version [20] there is also a context, Δ, of type variables.

only maintain local state across calls on the heap. We can reason about methods with local state on the heap by referring to the fields with the local state in the pre- and post-condition of the method. For instance, an increment method implemented with a count field would get the following specification:

```
public class Counter {
  public int count;
  int increment() { return this.count++;}
  ∀n. { this.count ↦ n } { this.count ↦ n + 1 }
}
```

Note, we could abstract the details of the field name using standard techniques for abstraction in separation logic [16]. For simplicity, we leave it concrete here.

We can specify an increment delegate as follows: (the assertion x : Counter holds iff x has dynamic type Counter)

```
x = new Counter();
{ x.count ↦ 0 ∗ x : Counter }
  { x : Counter }
f = x.increment;
  { ∀n. f ↦ ⟨{(this.count ↦ n)}_{(this.count ↦ n + 1)}⟩[x/this]}
{ x.count ↦ 0 ∗ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n + 1}⟩ }
```

To create a named delegate, we take the original method specification and replace **this** with the target object x. Since delegate specifications can refer to logical variables from the context, universal quantification in the assertion logic can be used to introduce the logical variables used in the method's specification, here n.

We can duplicate delegate assertions, which enables the logical variables in a delegate specification to be instantiated without losing the general specification. For example, the following holds in the logic:

```
∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n+1}⟩
  ⊢ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n+1}⟩
      ∗ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n+1}⟩
```

This is due to the storage model defined in Section 5: the heap is split into a field heap and a closure heap and since the closure heap is never modified and only ever extended, it does not have to be separated by separating conjunction.

The proof rule for creating a delegate referring to a closed instance method is given below:

$$\frac{\Gamma(\mathsf{C},\mathsf{m}) = \langle(\bar{\mathsf{u}};\psi).\{\mathsf{P}\}\_\{\mathsf{d.Q}\}\rangle}{\Gamma;\mathsf{x},\mathsf{y};- \vdash \{\mathsf{y}:\mathsf{C}\}\mathsf{x} = \mathsf{y.m}\{\forall\psi.\ \mathsf{x} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P[y/this]}\}\_\{\mathsf{d.Q[y/this]}\}\rangle\} \triangleleft \{\mathsf{x}\}}$$

where $\Gamma(\mathsf{C},\mathsf{m}) = \langle(\bar{\mathsf{u}};\psi).\{\mathsf{P}\}\_\{\mathsf{d.Q}\}\rangle$ looks up the specification of the method to which the delegate is being created, and ψ are the logical variables for the specification.

To illustrate that the specification given to the previously verified apply method is strong enough for verifying calls with delegates with local state on

the heap, consider calling apply with an increment counter. In a state where the current count is m one can verify a call to apply with delegate f from above by instantiating P and Q with the terms x.count ↦m and x.count ↦m+1.

```
{ x.count ↦ m ∗ ∀n. f ↦ ⟨{x.count ↦ n}_{x.count ↦ n + 1}⟩ }
  { x.count ↦ m ∗ f ↦ ⟨{x.count ↦ m}_{x.count ↦ m+1}⟩ }
    { (P ∗ f ↦ ⟨{P}_{Q}⟩)[x.count ↦ m/P, x.count ↦ m+1/Q] }
  apply(f);
    { (Q ∗ f ↦ ⟨{P}_{Q}⟩)[x.count ↦ m/P, x.count ↦ m+1/Q] }
  { x.count ↦ m+1 ∗ f ↦ ⟨{x.count ↦ m}_{x.count ↦ m+1}⟩ }
{ x.count ↦ m+1 }
```

3.4 Reasoning about Anonymous Delegates

In this section we extend our treatment of delegates to cover anonymous methods with captured variables that can escape their scope. First, we consider reasoning about delegates with captured variables while the captured variables are still in scope, followed by delegates with captured variables that have gone out of scope.

Reasoning about captured variables still in scope Anonymous delegates can have local state, not only on the heap as we saw with named delegates above, but also on the stack. For example, the following code snippet

```
int x = 0;
Func⟨int⟩ f = delegate () { x++; };
apply(f);
assert(x==1);
```

binds a delegate to f, which captures the local stack variable x. The call to apply causes a call to the delegate, which increments the x variable from the enclosing scope. Hence, the assertion will succeed.

Now let us consider the specification of the delegate. Intuitively, the delegate specification should express that for any stack and heap where x has the value n, the delegate is safe to execute and if it terminates, x will have the value $n+1$ on the terminal stack. That is, in the pre- and postcondition of the delegate, we want to refer to the value of x on the stack upon entry to, and exit from, the delegate, respectively.

To express this we extend the logic with two new terms, written

$$\mathsf{L} \overset{s}{\mapsto} \mathsf{N} \quad \text{and} \quad \&\mathsf{x}$$

and a new type Loc of stack locations. The & operator can only be applied to stack variables, and &x denotes the location of x on the stack. The $\mathsf{L} \overset{s}{\mapsto} \mathsf{N}$ term is a points-to predicate for the stack: it asserts that the location denoted by L is allocated on the stack and that this location contains the value denoted by N. The typing rules for $\mathsf{L} \overset{s}{\mapsto} \mathsf{N}$ and & are given below.

$$\frac{\phi;\psi \vdash \mathsf{L} : \mathtt{Loc} \qquad \phi;\psi \vdash \mathsf{N} : \mathtt{Val}}{\phi;\psi \vdash \mathsf{L} \overset{s}{\mapsto} \mathsf{N} : \mathtt{Prop}} \qquad \frac{\mathsf{x} \in \phi}{\phi;\psi \vdash \&\mathsf{x} : \mathtt{Loc}} \tag{2}$$

With these extensions, the specification can be expressed as follows:

$\mathsf{Func}\langle\textbf{int}\rangle\ \mathsf{f} = \textbf{delegate}\ ()\ \{\ \mathsf{x}{+}{+};\ \};$
$\{\ \forall \mathsf{n}.\ \mathsf{f} \mapsto \langle \{\&\mathsf{x} \overset{s}{\mapsto} \mathsf{n}\}\_\{\&\mathsf{x} \overset{s}{\mapsto} \mathsf{n}{+}1\}\rangle\ \}$

Note that if we simply referred directly to x in the the specification then this would refer to x's current value. Whereas, when used in a pre/post-condition of a delegate assertion, the stack points-to predicate refers to the stack upon entry/exit of a call to the delegate. In effect, the stack points-to predicate thus allows us to delay the evaluation of a variable, and the & term allows us to specify the variable to evaluate.

In the above specification, we only need to refer to the value of the captured variable once in the pre- and post-condition and can thus use the stack points-to predicate directly. However, in general, it is more convenient with a term $\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P}$ for asserting that the location denoted by L is allocated and that P holds with the contents of this stack location bound to x. We can define such a term in terms of the stack points-to predicate as follows:[5]

$$\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P} \stackrel{\text{def}}{=} \exists \mathsf{x} : \texttt{Val}.\ (\mathsf{L} \overset{s}{\mapsto} \mathsf{x} * \mathsf{P})$$

The creation of an anonymous delegate is given by the following rule:

$$\frac{\bar{\mathsf{u}} \cap \mathsf{M} = \emptyset \qquad \bar{\mathsf{y}}, \bar{\mathsf{u}} \cap \mathrm{FVA}(\mathsf{P}, \mathsf{Q}) = \emptyset \qquad \bar{\mathsf{y}} \subseteq \mathrm{FV}(\mathsf{B}) \qquad \Gamma; \bar{\mathsf{y}}, \bar{\mathsf{u}}; \psi \vdash \{\mathsf{P}\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}}{\begin{array}{l}\Gamma; \bar{\mathsf{y}}, \mathsf{x}; \psi \vdash \{\texttt{emp}\} \\ \qquad \mathsf{x} = \texttt{delegate}(\bar{\mathsf{G}}\bar{\mathsf{u}})\ \{\mathsf{B}\} \\ \qquad \{\mathsf{x} \mapsto \langle(\bar{\mathsf{u}}).\{\texttt{lookup}\ \&\bar{\mathsf{y}}\ \texttt{as}\ \bar{\mathsf{y}}\ \texttt{in}\ \mathsf{P}\}\_\{\mathsf{d}.\texttt{lookup}\ \&\bar{\mathsf{y}}\ \texttt{as}\ \bar{\mathsf{y}}\ \texttt{in}\ \mathsf{Q}\}\rangle\} \triangleleft \{\mathsf{x}\}\end{array}}$$

where $\mathrm{FVA}(\mathsf{P})$ denotes the set of variables x, such that $\&\mathsf{x}$ occurs in P and B is a method body,

$$\mathsf{B} ::= \mathsf{G}\,\bar{\mathsf{x}}; \mathsf{s}; \texttt{return}\ \mathsf{x};$$

Note that, implicitly $\bar{\mathsf{y}}$ and $\bar{\mathsf{u}}$ are disjoint.

To ensure that substituting the actual arguments for the formal arguments in the delegate call rule correctly captures C♯'s calling convention, we prevent the delegate from modifying or capturing its arguments, hence the premises $\bar{\mathsf{u}} \notin \mathsf{M}$ and $\bar{\mathsf{u}} \notin \mathrm{FVA}(\mathsf{P}, \mathsf{Q})$. The captured variables used in the body from the context, $\bar{\mathsf{y}}$, are bound in the specification using a `lookup` term to allow their evaluation to be postponed until the point of call, and return. The premise $\bar{\mathsf{y}} \notin \mathrm{FVA}(\mathsf{P}, \mathsf{Q})$, which prevents nested delegates from capturing variables from any but the innermost enclosing scope, is just to simplify the rule slightly. The technical report [20] contains a generalized rule without this restriction.

It might seem like $\mathsf{L} \overset{s}{\mapsto} \mathsf{N}$ and & opens up the possibility of aliasing stack variables in the logic, which would invalidate Hoare's assignment rule. This is not the case, as we will explain in Section 5.1 (basically, variables in the program

[5] Whenever we use this abbreviation we will implicitly assume that $x \notin \mathrm{FV}(\mathsf{L})$.

variable context are implicitly separated from stack variables mentioned in the pre- and post-condition of a Hoare triple, outside of a delegate assertion.)

When we come to calling the delegate, we need to be able to evaluate lookup in the current state. We use a structural rule to enable this evaluation:

$$\frac{\Gamma; \phi; \psi \vdash \{\texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P}\}\mathsf{s}\{\texttt{lookup}\ l\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{Q}\} \lhd \mathsf{M}}{\Gamma; \phi, \mathsf{x}; \psi \vdash \{\&\mathsf{x} = l \wedge \mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \lhd \mathsf{M} \cup \{\mathsf{x}\}} \tag{3}$$

The rule is hiding two different evaluations of x, one at the beginning and one at the end. These may evaluate to different values, so we must ensure that we have not framed any facts about x that could be invalidated, hence this rule includes x in the modified set. Note that the assertions P and Q cannot refer to the address of x, since `lookup L as x in P` binds x as a *logical* variable in P. Furthermore, by our implicit assumption about the program variable context, this rule only applies if $\mathsf{x} \notin \phi$. The $\mathsf{x} \notin \phi$ restriction is vital to the soundness of the rule, as will be explained in Section 5.2.

Using (3) we can verify the call of the anonymous increment delegate given above:

```
{ x = 0 * f ↦ ⟨{lookup &x as y in y=0}_{lookup &x as y in y=1}⟩ }
{ &x = l * x = 0 * f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
   { lookup l as x in (x = 0 * f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩) }
   { (lookup l as x in x = 0) * f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
      apply(f);
   { (lookup l as x in x = 1) * f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
   { lookup l as x in (x = 1 * f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩) }
{ x = 1 * f ↦ ⟨{lookup l as y in y=0}_{lookup l as y in y=1}⟩ }
```

Here we instantiate P and Q from the apply specification as lookup l **as** y **in** y=0, and lookup l **as** y **in** y=1, respectively. This proof outline uses the following key property of the lookup binder to rearrange the assertions:

$$(\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ \mathsf{P}) * \mathsf{Q} \quad \Leftrightarrow \quad \texttt{lookup}\ \mathsf{L}\ \texttt{as}\ \mathsf{x}\ \texttt{in}\ (\mathsf{P} * \mathsf{Q}) \qquad (\mathsf{x} \notin \mathrm{FV}(\mathsf{Q}))$$

Reasoning about captured variables that have escaped their scope Consider the following counter method from the introduction:

```
public Func⟨int⟩ counter() {
  int x = 0;
  return delegate () { return ++x; };
}
```

The interesting aspect is how to verify the body of counter.

```
{emp}
  int x = 0;
  {x=0}
    Func⟨int⟩ f = delegate () { return ++x; };
  {x=0 * ∀n. S(f,&x,n)) }
```

```
    return f;
  {d. x=0 * ∀n. S(d,&x,n)) }
{???}
```

where

$\mathsf{S} = \lambda(\mathsf{f},\mathsf{l},\mathsf{n}).\ \mathsf{f} \mapsto \{\ \mathsf{lookup\ l\ as\ x\ in\ x} = \mathsf{n}\ \}\ \{\ \mathsf{d.\ lookup\ l\ as\ x\ in\ x} = \mathsf{d} = \mathsf{n} + 1\ \}$

The final step of the proof, marked ???, deals with leaving the scope of the variable x. Typically, this would be dealt with by existential quantification. However, since the variable x is still accessible through the delegate and its value may change, existential quantification does not suffice. Instead, we assert that there exists a stack location:

{d. ∃l. lookup l **as** x **in** x=0 * ∀n. S(d,l,n)) }

The existential is used to quantify over the location of the variable, and lookup allows us to bind its value. Importantly, lookup asserts that there is a stack location disjoint from those already in scope. Indeed we have the following implication in the logic:

$$(\texttt{lookup}\ \mathsf{L}\ \texttt{as}\ x\ \texttt{in}\ \mathsf{P}) * (\texttt{lookup}\ \mathsf{L}'\ \texttt{as}\ x\ \texttt{in}\ \mathsf{Q}) \quad \Rightarrow \quad \mathsf{L} \neq \mathsf{L}' \tag{4}$$

Thus, if we call counter twice, we will reason correctly about the different stack locations for their internal state.

The general case is given by the variable declaration rule:

$$\frac{\Gamma; \phi, \mathsf{z}; \psi \vdash \{\mathsf{P} \wedge \mathsf{z} = \texttt{null}\}\mathsf{B}\{\mathsf{d}.\mathsf{Q}\} \triangleleft \mathsf{M}}{\Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{Gz}; \mathsf{B}\{\mathsf{d}.\exists l : \texttt{Loc}.\ \texttt{lookup}\ l\ \texttt{as}\ \mathsf{z}\ \texttt{in}\ \mathsf{Q}[l/\&\mathsf{z}]\} \triangleleft \mathsf{M} \setminus \mathsf{z}}$$

This degenerates into the standard rule if $\mathsf{z} \notin FV(\mathsf{Q})$.

The frame rule With the introduction of captured variables, it is no longer possible to determine syntactically from a statement which stack variables it might modify. Hence, we have extended our Hoare triples with a finite set M of variables:

$$\Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}$$

to give an explicit over-approximation of which variables in ϕ, s modifies. We restrict M to variables in scope, since we have no way of referring to variables not in scope, nor any useful syntactic approximation of which assertions a given assertion makes about out-of-scope variables.

To ensure that one cannot frame on assertions about potentially modified out-of-scope variables, we let separating conjunction separate out of scope variables (as expressed by (4)). Our language satisfies the standard heap frame property and heap safety monotonicity (used for showing the standard frame rule sound), and it also satisfies a corresponding stack frame property and stack safety monotonicity. We can thus prove the soundness of the following frame rule:

$$\frac{\Gamma; \phi; \psi \vdash \mathsf{R} : \texttt{Prop} \qquad \Gamma; \phi; \psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} \qquad \mathsf{M} \cap \mathrm{FVV}(\mathsf{R}) = \emptyset}{\Gamma; \phi; \psi \vdash \{\mathsf{P} * \mathsf{R}\}\mathsf{s}\{\mathsf{Q} * \mathsf{R}\} \triangleleft \mathsf{M}}$$

since P has to assert the existence of any out-of-scope variables potentially modified by s. Here FVV(R) denotes the set of free value variables, which is defined as follows for variables and &:

$$\mathrm{FVV}(x) = \{x\} \qquad\qquad \mathrm{FVV}(\&x) = \emptyset$$

and like FV(R) for every other case. Hence, one can frame on assertions that refer to the address of a variable; intuitively, since the address cannot be modified, only the contents stored at that address.

4 Case study

In this section we return to the fold/reverse example from Section 3. We show how to specify fold, such that one can verify calls with delegates with local state and use it to verify the second reverse method, which maintains its local state on the stack.

We can give fold the following specification:

```
public void fold⟨X⟩(Node⟨X⟩ lst, Action⟨Node⟨X⟩⟩ f) {...}
  ∀xs,ys : Val. ∀P : Val × Val →Prop.
  ∀Q : Val × Val →Prop.
    { list(lst,xs,P) ∗ Q(lst,ys) ∗ ∀v,ys : Val, ∀n', x : Val,
        f ↦ ⟨(n).{n.next ↦ n' ∗ n.item ↦ x ∗ P(x, v) ∗ Q(n, ys)}_{Q(n', v::ys)}⟩ }
    { Q(null, rev(xs) @ ys) }
```

The code and specification is easiest to understand as an imperative form of a fold-left function, where the accumulator is not passed around explicitly, but rather maintained as local state by the delegate. Then Q(n,ys) is an accumulator predicate, intended to describe the current state, after having folded over ys and where the next node to be visited is n. In the pre-condition of the folding-delegate we explicitly mention n.next and n.item, to allow the delegate to modify these fields. The rev in the post-condition is a function for reversing a list in the logic.

Define Q' as follows:

$$Q' = \lambda l : \mathrm{Loc}.\ \lambda n : \mathrm{Val}.\ \lambda ys : \mathrm{Val}.\ \mathsf{lookup}\ l\ \mathbf{as}\ \mathsf{head}\ \mathbf{in}\ \mathsf{list}(\mathsf{head}, ys, P)$$

Then we can verify the reverse method as follows by instantiating fold's accumulator predicate Q with Q'(l), where l is a logical variable introduced to refer to the location of the captured head variable:

```
public static Node⟨X⟩ reverse⟨X⟩(Node⟨X⟩ lst) {
{ list(lst, xs, P) }
  Node⟨X⟩ head = null;
{ list(lst, xs, P) ∗ head = null }
  Action⟨Node⟨X⟩⟩ f = delegate (Node⟨X⟩ n) {
  { n.next ↦ n' ∗ n.item ↦ x ∗ P(x, v) ∗ list(head, ys, P) }
    n.next = head; head = n;
  { list(head, v::ys, P) }
```

```
  };
{ l = &head ∧ list(lst, xs, P) * head = null * ∀v,ys,x,n' : Val,
     f ↦ ⟨(n).{lookup l as h in n.next ↦ n' * n.item ↦ x * P(x, v) * list(h, ys, P)}
              {lookup l as h in list(h, v::ys, P)}⟩ }
  { lookup l as head in list(lst, xs, P) * head = null * ∀v,ys,x,n' : Val,
     f ↦ ⟨(n).{n.next ↦ n' * n.item ↦ x * P(x, v) * Q'(l, n, ys)}
              {Q'(l, n', v::ys)}⟩ }
  { list(lst, xs, P) * Q'(l, lst, []) * ∀v,ys,x,n' : Val,
     f ↦ ⟨(n).{n.next ↦ n' * n.item ↦ x * P(x, v) * Q'(l, n, ys)}
              {Q'(l, n', v::ys)}⟩ }
  fold⟨X⟩(lst, f);
  { Q'(l, null, rev(xs)@[]) }
  { lookup l as head in list(head, rev(xs)@[], P) }
{ list(head, rev(xs), P) }
  return head;
{ r. list(r, rev(xs), P) }
}
```

In this example, we did not use the first parameter of Q, which is a reference to the next node in the list. This field is primarily useful for delegates which do not modify the node's next field, to express that there is a list segment ending with next-pointer n. Consider, for instance a map method which takes a delegate whose action on a list element is described by a function f in the logic. Then we could take the accumulator predicate to be:

$$\mathsf{Q(n,\ ys) = list{-}segment(lst,\ n,\ map(f,\ ys),\ P)}$$

where the second argument to list−segment is the value of the next field of the last node in the list, if the list is non-empty.

Filter As a second example we consider a generic filter method for the List class, that takes as argument a delegate which is called on each element of the list to determine whether or not it should be included in the filtered list:

```
class List {
  public static Node⟨X⟩ filter⟨X⟩(Node⟨X⟩ lst, Func⟨X,bool⟩ f) {
    if(lst == null) return null;
    else {
      Node⟨X⟩ tmp = filter(lst.next, f);
      if(f(lst.item)) { lst.next = tmp; return lst; } else return tmp;
    }
  } ∀xs : Val, p : Val →Val, P : Val →Prop,
      { list(lst, xs, P) * ∀v : Val. f ↦ ⟨(x).{P(x,v)}_{r. P(x,v) * r = p(v) }⟩ }
      { r. list(r, filter(xs, p), P) }
}
```

The filter(xs, p) in the post-condition is a mathematical function, which filters the sequence xs using the predicate p. The precondition asserts that f should be

a reference to a delegate, which, when called with a reference x representing the value v, should return $p(v)$.

We can verify a simple client of the filter class that filters the even numbers in a list of integers.

```
{ m : Math ∗ list(x,[1,2,3,4],Int) }
 d = m.isEven;
{ list(x,[1,2,3,4],Int) ∗ ∀v : Val. d ↦ ⟨(i).{Int(i,v)}_{r. Int(i,v) ∗ r=even(v)}⟩ }
 List⟨Integer⟩.filter(x,d);
{ list(x,filter([1,2,3,4],even),Int) }
{ list(x,[2,4],Int) }
```

where we assume the Math class has the following isEven method:

```
public boolean isEven(Integer i) { return i.intValue()%2==0; }
∀v : Val. { Int(i,v) } { r. Int(i, v) ∗ r = even(v) }
```

5 Semantics and Soundness of Proof Rules

In this section we formalize the syntax and operational semantics of the fragment of the C♯ programming language that we consider and the semantics of our logic. For reasons of space, many details and proofs have been omitted, please see [20] for details.

The language that we consider is a subset of C♯ with the most basic object-oriented and imperative features of C♯ and with a restricted syntax to simplify the presentation of the proof system. The syntax of the language is given in Figure 1. In the syntax we use the following metavariables: f ranges over field names, m over method names, C over class names, x, y and z over program variables, and T over type variables. We denote the set of field names by $\mathbb{F}$, the set of method names by $\mathbb{M}$, the set of class names by $\mathbb{C}$, the set of generic class names by $\mathbb{T}$, and the set of variables by $\mathbb{A}_p$. We use an overbar for sequences.

For the operational semantics we assume countable disjoint infinite sets $\mathbb{O}$, $\mathbb{L}_s$, and $\mathbb{L}_h$ of object identifiers, stack locations and heap locations, respectively. We take values to be object identifiers, heap locations, and null. An environment is a finite function from variables to stack locations and a stack is a finite function from stack locations to values:

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{O} \uplus \mathbb{L}_h \uplus \{\text{null}\} \qquad \mathbb{E}_p \stackrel{\text{def}}{=} \mathbb{A}_p \stackrel{\text{fin}}{\to} \mathbb{L}_s \qquad \mathbb{S} \stackrel{\text{def}}{=} \mathbb{L}_s \stackrel{\text{fin}}{\to} \mathbb{V}$$

A heap is a tuple (h_v, h_t, h_c) of finite functions, where h_v is a field heap, mapping object identifiers and field names to values; h_t is a type heap, mapping object identifiers to class names; and h_c is a closure heap, mapping heap locations to delegates. A delegate is either an object identifier and a method name or an environment and an anonymous method body.

$$\mathbb{H} \stackrel{\text{def}}{=} (\mathbb{O} \times \mathbb{F} \stackrel{\text{fin}}{\to} \mathbb{V}) \times (\mathbb{O} \stackrel{\text{fin}}{\to} \mathbb{T}) \times (\mathbb{L}_h \stackrel{\text{fin}}{\to} \mathbb{D})$$

$$\mathbb{D} \stackrel{\text{def}}{=} (\mathbb{O} \times \mathbb{M}) \uplus (\mathbb{E}_p \times \mathbb{A}_p^* \times \mathbb{A}_p^* \times \mathbb{P} \times \mathbb{A}_p)$$

G	$::= \mathsf{C}\langle\overline{\mathsf{G}}\rangle \mid \mathsf{T}$	Generic class
L	$::= \texttt{class}\ \mathsf{C}\langle\overline{\mathsf{T}}\rangle : \mathsf{G}\ \{\bar{\mathsf{G}}\,\bar{\mathsf{f}}; \bar{\mathsf{M}}\}$	Class definition
M	$::= \mathsf{G}\,\mathsf{m}(\bar{\mathsf{G}}\,\bar{\mathsf{z}})\ \{\mathsf{B}\}$	Method definition
B	$::= \bar{\mathsf{G}}\,\bar{\mathsf{x}}; \mathsf{s}; \texttt{return}\ \mathsf{x};$	Method body
s	$::=$	Statement
	x = y	assignment
	x = null	initialization
	x = y.f	field access
	x.f = y	field update
	x = y.m($\bar{\mathsf{z}}$)	method invocation
	x = (G)y	cast
	if (x == y) {s_1} else {s_2}	conditional
	x = new $\mathsf{C}\langle\bar{\mathsf{G}}\rangle()$	object creation
	x = delegate ($\bar{\mathsf{G}}\,\bar{\mathsf{z}}$) {B}	anonymous delegate
	x = y.m	named delegate
	x = y($\bar{\mathsf{z}}$)	delegate application
	$\mathsf{s}_1; \mathsf{s}_2$	sequential composition

Fig. 1. Syntax of a simplified C♯

The operational semantics is defined as a big-step semantics with step-indices corresponding to a small-step semantics. It takes configurations, (P, E, S, H, s), consisting of a program P, mapping class and method names to method bodies, an environment E, a stack S, a heap H, and a statement s to err or a terminal stack and heap. The operational semantics of the fragment of the language without generics and delegates is standard and omitted. Instead, we just give the following two rules for constructing and invoking an anonymous delegate. When constructing an anonymous delegate we store the relevant part of the current stack environment along with the delegate; we restore it again when invoking the delegate:

$$\frac{\begin{array}{c} l \notin \mathrm{Dom}(H_c) \qquad S' = S[E(\mathsf{x}) \mapsto l] \\ E_c = E|_{\mathrm{FV}(\mathsf{s},\mathsf{r})\setminus(\bar{\mathsf{x}}\cup\bar{\mathsf{z}})} \qquad H' = H_c[l \mapsto (E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r})] \end{array}}{(P, E, S, H, \mathsf{x}\ \texttt{=}\ \texttt{delegate}\ (\bar{\mathsf{G}}\bar{\mathsf{x}})\ \{\bar{\mathsf{G}}\bar{\mathsf{z}};\ \mathsf{s}; \texttt{return}\ \mathsf{r}\}) \Downarrow_1 (S', H')}$$

$$\frac{\begin{array}{c} \bar{l}_x, \bar{l}_z \notin \mathrm{Dom}(S) \qquad H_c(S(E(\mathsf{y}))) = (E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}) \\ (P, E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z], S[\bar{l}_x \mapsto S(E(\bar{\mathsf{u}})), \bar{l}_z \mapsto \mathrm{null}], H, \mathsf{s}) \Downarrow_n (S', H') \end{array}}{(P, E, S, H, \mathsf{x} = \mathsf{y}(\bar{\mathsf{u}})) \Downarrow_{n+1} (S'[E(\mathsf{x}) \mapsto S'(E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z](\mathsf{r}))], H')}$$

We use the notation H_c to refer to the closure heap of H and $H_c[x \mapsto y]$ for the heap H where the closure heap has been extended to map x to y. For each rule we implicitly assume all applications with finite functions to be defined.

A configuration is safe for n steps if it cannot fault in n steps or less:

Definition 1. $(P, E, S, H, \mathsf{s}) : \mathit{safe}_n \quad \mathit{iff} \quad \forall m \leq n.\ (P, E, S, H, \mathsf{s}) \not\Downarrow_m \text{err}$

In addition to the usual safety monotonicity and frame property the language satisfies the following stack monotonicity and stack frame properties:

Lemma 1. *If* $(P, E, S_1, H, \mathsf{s}) : \mathit{safe}_n$ *and* $S_1 \# S_2$ *then* $(P, E, S_1 \cup S_2, H, \mathsf{s}) : \mathit{safe}_n$.

Lemma 2. *If* $(P, E, S_1, H, \mathsf{s}) : \mathit{safe}_n$, $S_1 \# S_2$, *and* $(P, E, S_1 \cup S_2, H, \mathsf{s}) \Downarrow_n (S', H')$ *then there exists an* S_1' *such that* $S' = S_1' \cup S_2$, $S_1' \# S_2$, *and* $(P, E, S_1, H, \mathsf{s}) \Downarrow_n (S_1', H')$.

5.1 Assertion logic

The proof system consists of two layers: an assertion logic for reasoning about program states and a specification logic for reasoning about the effects of programs. In this subsection we formalize the syntax and semantics of the assertion logic. The assertion logic is an intuitionistic higher-order logic over a typed term language. The terms of the language are generated by the following grammar:

$$\begin{aligned}\mathsf{P}, \mathsf{Q}, \mathsf{L}, \mathsf{M}, \mathsf{N} ::= {} & x \mid \lambda x : \omega.\mathsf{M} \mid \mathsf{M}\ \mathsf{N} \mid (\mathsf{M}, \mathsf{N}) \mid \mathtt{fst}\ \mathsf{M} \mid \mathtt{snd}\ \mathsf{M} \\ \mid {} & \bot \mid \top \mid \mathsf{P} \vee \mathsf{Q} \mid \mathsf{P} \wedge \mathsf{Q} \mid \mathsf{P} \Rightarrow \mathsf{Q} \mid \forall x : \omega.\mathsf{M} \mid \exists x : \omega.\mathsf{M} \mid \mathsf{M} =_\omega \mathsf{N} \\ \mid {} & \mathsf{P} * \mathsf{Q} \mid \mathsf{P} \mathrel{-\!\!*} \mathsf{Q} \mid \mathtt{emp} \mid \mathsf{M}.\mathsf{f} \mapsto \mathsf{N} \mid \mathsf{M} : \mathsf{N} \mid \mathtt{null} \\ \mid {} & \mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \mid \mathsf{L} \overset{s}{\mapsto} \mathsf{N} \mid \&\mathsf{x}\end{aligned}$$

where ω ranges over types generated by the following grammar:

$$\omega ::= \omega \to \omega \mid \omega \times \omega \mid \mathtt{Prop} \mid \mathtt{Loc} \mid \mathtt{Val} \mid \mathtt{Int} \mid \mathtt{Class}$$

Assertions are terms of type `Prop`. We will follow the convention of using P and Q for assertions and predicates, L for terms of type `Loc`, and M and N for general terms. The terms are typed with the typing judgment, $\phi; \psi \vdash \mathsf{M} : \omega$, where ϕ is a program variable context, and ψ a logical variable context. The typing rules include all the usual rules of higher-order separation logic [3], extended with the typing rule for delegate assertions, $\overset{s}{\mapsto}$, and the address-of operator (rules (1) and (2)).

To ensure the soundness of Hoare's assignment rule, we need to ensure that $\overset{s}{\mapsto}$ does not introduce any aliasing of stack variables into the logic. Intuitively, we achieve this by only allowing $\overset{s}{\mapsto}$ to assert the existence and value of out-of-scope stack locations. More formally, the meaning of an assertion, $\phi; \psi \vdash \mathsf{P} : \mathtt{Prop}$, is given in terms of the meaning of the program variables in ϕ, i.e., the part of the stack in scope. To reason about out-of-scope stack locations, we interpret assertions as sets of stacks and heaps. Intuitively, this second stack is the "rest of the stack" in the *specification logic*; see the definition of the semantics of triples in Section 5.2. Aliasing is thus avoided, by restricting $\overset{s}{\mapsto}$ to this second stack (see the semantics of $\overset{s}{\mapsto}$ below), which is disjoint from the part of the stack in scope.

We follow [15] in indexing the interpretation of the *specification logic* with step-indices, to allow verification of mutually recursive methods. Furthermore, since we now have specifications in the assertion logic, we also step-index the interpretation of assertions. This idea comes from current work by the second author jointly with Thamsborg and Støvring.

Definition 2. *The types are interpreted as follows:*

$$\begin{array}{ccc} [\![\omega \to \omega']\!] = [\![\omega]\!] \to [\![\omega']\!] & [\![\omega \times \omega']\!] = [\![\omega]\!] \times [\![\omega']\!] & [\![\mathtt{Class}]\!] = \mathbb{T} \\ [\![\mathtt{Val}]\!] = \mathrm{Val} & [\![\mathtt{Loc}]\!] = \mathbb{L}_s & [\![\mathtt{Int}]\!] = \mathbf{Z} \end{array}$$

$$[\![\mathtt{Prop}]\!] = \{U \in \mathcal{P}^{\uparrow}(\mathsf{N} \times \mathbb{S} \times \mathbb{H}) \mid \forall \pi \in \mathrm{Perm}(\mathbb{A}_p).\ \forall a \in U.\ \pi(a) \in U\}$$

where Val *is the least set satisfying,*

$$\mathrm{Val} \cong \mathbb{V} \uplus \mathrm{Strings} \uplus \mathrm{Val} \times \mathrm{Val},$$

and where the ordering on $\mathsf{N} \times \mathbb{S} \times \mathbb{H}$ *is defined as follows*

$$(n, S, H) \leq (m, S', H') \quad \textit{iff} \quad m \leq n \wedge S \sqsubseteq S' \wedge H \sqsubseteq H'$$

with $\sqsubseteq$ *given by the point-wise extension of the following order on finite functions:*

$$f \leq g \quad \textit{iff} \quad \mathrm{Dom}(f) \subseteq \mathrm{Dom}(g) \wedge \forall \mathsf{x} \in \mathrm{Dom}(f).\ f(x) = g(x).$$

and the permutation action is given by atom-permutation on $\mathbb{A}_p$ *and* $\mathbb{P}$ *and the trivial action on* $\mathsf{N}, \mathbb{O}, \mathbb{F}, \mathbb{C}, \mathbb{L}_s, \mathbb{L}_h$, *and* $\mathbb{V}$.

Assertions are thus interpreted as step-indexed subsets of stacks and heaps, downwards-closed in the step-index and upwards closed in extensions of the stack and heap and equivariant under permutations of the closures and environments on the closure heap. Permutations are used to ensure that program and logical variables in the specification logic context are α-convertible (see Rule (α) in [20]).

Lemma 3. *Let* $\mathcal{L} = ([\![\mathtt{Prop}]\!], \subseteq)$, *then* $\mathcal{L}$ *is a complete BI-algebra [3], with BI structure* $(I, *, -\!\!*)$ *given by:*

$$\begin{aligned} I &= \emptyset \\ U * V &= \{(n, C \cup C', (h_v \cup h'_v, h_t, h_c)) \mid C \# C' \wedge h_v \# h'_v \wedge \\ &\qquad (n, C, (h_v, h_t, h_c)) \in U \wedge (n, C', (h'_v, h_t, h_c)) \in V\} \\ U -\!\!* V &= \bigcup \{W \in [\![\mathtt{Prop}]\!] \mid W * U \subseteq V\} \end{aligned}$$

for $U, V \in [\![\mathtt{Prop}]\!]$.

Definition 3. *A term-in-context, $\phi;\psi \vdash \mathsf{M} : \omega$, is interpreted as a set-theoretic function:*

$$[\![\phi;\psi \vdash \mathsf{M} : \omega]\!] : [\![\phi]\!] \times [\![\psi]\!] \to [\![\omega]\!]$$

where

$$[\![\phi]\!] = \{(E,S) \in \mathbb{E}_p \times \mathbb{S} \mid E \textit{ injective} \wedge \phi = \mathrm{Dom}(E) \wedge \mathrm{Rng}(E) = \mathrm{Dom}(S)\}$$
$$[\![\psi]\!] = \Pi x \in \mathrm{Dom}(\psi).\, [\![\psi(x)]\!].$$

The standard part of the assertion logic is interpreted using a BI-hyperdoctrine over the category **Set**, *induced by the complete BI algebra $\mathcal{L}$ (as in Example 6 in [3]). The interpretation is written out in full in [20].*

The new assertion forms are interpreted as follows:

$$[\![\phi;\psi \vdash \&\mathsf{x} : \mathtt{Loc}]\!]((E,S),\vartheta) = E(\mathsf{x})$$
$$[\![\phi;\psi \vdash \mathsf{L} \overset{s}{\mapsto} \mathsf{N} : \mathtt{Prop}]\!]((E,S);\vartheta) =$$
$$\{(n,C,H) \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid l \in \mathrm{Dom}(C) \wedge C(l) = [\![\phi;\psi \vdash \mathsf{N} : \mathtt{Val}]\!]((E,S);\vartheta)\}$$

where $l = [\![\phi;\psi \vdash \mathsf{L} : \mathtt{Loc}]\!]((E,S),\vartheta)$. For delegate assertions, we just show the interpretation of anonymous delegate assertions:

$$[\![\phi;\psi \vdash \mathsf{R} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle : \mathtt{Prop}]\!]((E,S),\vartheta) =$$
$$\{(n,\_,(h_v,h_t,h_c)) \in \mathsf{N} \times \mathbb{S} \times \mathbb{H} \mid \exists E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}.$$
$$(h_c([\![\phi;\psi \vdash \mathsf{R} : \mathtt{Val}]\!]((E,S),\vartheta) = (E_c, \bar{\mathsf{x}}, \bar{\mathsf{z}}, \mathsf{s}, \mathsf{r}) \wedge$$
$$\forall m \le n.\ \forall k \le m.\ \forall C \in \mathbb{S}.\ \forall H \in \mathbb{H}.\ \forall \bar{l}_x, \bar{l}_z\ \in \mathbb{L}_s \setminus (\mathrm{Dom}(C) \cup \mathrm{Rng}(E_c)).\ \forall \bar{v}_x \in \mathbb{V}.$$
$$(m-1,C,H) \in [\![\phi;\psi,\bar{\mathsf{u}} \vdash \mathsf{P} : \mathtt{Prop}]\!]((E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x])\ \Rightarrow$$
$$(E', C[\bar{l}_x \mapsto \bar{v}_x, \bar{l}_z \mapsto \mathtt{null}], H, \mathsf{s}) : \mathtt{safe}_k \wedge$$
$$(E', C[\bar{l}_x \mapsto \bar{v}_x, \bar{l}_z \mapsto \mathtt{null}], H, \mathsf{s}) \Downarrow_k (C', H')\ \Rightarrow$$
$$(m-k, C' \setminus \bar{l}_x; H') \in [\![\phi;\psi,\bar{\mathsf{u}},\mathsf{d} \vdash \mathsf{Q} : \mathtt{Prop}]\!]((E,S),\vartheta[\bar{\mathsf{u}} \mapsto \bar{v}_x, \mathsf{d} \mapsto C'(E'(\mathsf{r}))])$$

where $E' = E_c[\bar{\mathsf{x}} \mapsto \bar{l}_x, \bar{\mathsf{z}} \mapsto \bar{l}_z]$ and $f \setminus U$ is shorthand for $f|_{\mathrm{Dom}(f)\setminus U}$.

Note that in the interpretation of the delegate assertion, we use the current stack to give meaning to the program variables in ϕ in the pre- and postcondition, but we do not use this stack when running the body. This allows us to refer to the value of captured variables upon entry to and exit from a delegate call using $\overset{s}{\mapsto}$, even for variables currently in scope.

Theorem 1. *The standard part of the higher-order assertion logic is sound.*

Proof. As in [3].

Theorem 2. *The following rules are sound.*

$$\frac{\phi;\psi \vdash \mathsf{M} : \mathtt{Val} \qquad \phi;\psi,\bar{\mathsf{u}} \mid \mathsf{P}' \vdash \mathsf{P} \qquad \phi;\psi,\bar{\mathsf{u}},\mathsf{d} \mid \mathsf{Q} \vdash \mathsf{Q}'}{\phi;\psi \mid \mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}\}\_\{\mathsf{d}.\mathsf{Q}\}\rangle \vdash \mathsf{M} \mapsto \langle(\bar{\mathsf{u}}).\{\mathsf{P}'\}\_\{\mathsf{d}.\mathsf{Q}'\}\rangle}$$

$$\frac{\phi;\psi \vdash \mathsf{L}, \mathsf{L}' : \mathit{Loc} \qquad \phi;\psi,\mathsf{x} \vdash \mathsf{P},\mathsf{Q} : \mathit{Prop}}{\phi;\psi \mid \mathit{lookup}\ \mathsf{L}\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{P} * \mathit{lookup}\ \mathsf{L}'\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{Q} \vdash \mathsf{L} \neq \mathsf{L}'}$$

$$\frac{\phi;\psi \vdash \mathsf{L} : \mathit{Loc} \qquad \phi;\psi,\mathsf{x} \vdash \mathsf{P} : \mathit{Prop} \qquad \phi;\psi \vdash \mathsf{Q} : \mathit{Prop}}{\phi;\psi \mid (\mathit{lookup}\ \mathsf{L}\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ \mathsf{P}) * \mathsf{Q} \dashv\vdash \mathit{lookup}\ \mathsf{L}\ \mathit{as}\ \mathsf{x}\ \mathit{in}\ (\mathsf{P} * \mathsf{Q})}$$

The soundness of the second rule follows from the BI-structure on $\mathcal{L}$, and the soundness of the third rule follows from the fact that the meaning of a term is independent of the meaning of logical variables that do not appear free in the term.

5.2 Specification Logic

The specification logic has Hoare triples as its only propositions (it can straightforwardly be extended to a full first-order logic over Hoare triples as atomic propositions). A triple-in-context is interpreted as a downwards-closed set of step-indices:

$$[\![\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!] : [\![\phi]\!] \times [\![\psi]\!] \to \mathcal{P}^{\downarrow}(\mathsf{N})$$

as follows:

$$\begin{array}{l}
[\![\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M} : \mathtt{Spec}]\!]((E,S);\vartheta) = \{n \in \mathsf{N} \mid \\
\quad \forall m \leq n.\ \forall k \leq m.\ \forall C \in \mathbb{S}.\ \forall H \in \mathbb{H}. \\
\qquad (m-1,C,H) \in [\![\phi;\psi \vdash \mathsf{P} : \mathtt{Prop}]\!]((E,S);\vartheta) \wedge C \# S\ \Rightarrow \\
\qquad\quad (E;C \cup S;H;\mathsf{s}) : \mathtt{safe}_k \wedge \\
\qquad\quad (E;C \cup S;H;\mathsf{s}) \Downarrow_k (S';H') \Rightarrow \\
\qquad\qquad (m-k;S' \setminus E(\phi);H') \in [\![\phi;\psi \vdash \mathsf{Q} : \mathtt{Prop}]\!]((E,S'|_{E(\phi)});\vartheta) \wedge \\
\qquad\qquad \forall \mathsf{x} \in \phi \setminus \mathsf{M}.\ S(E(\mathsf{x})) = S'(E(\mathsf{x}))\}
\end{array}$$

and entailment is interpreted as:

$$\begin{array}{l}
[\![\Gamma;\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!] = \forall n \in \mathsf{N}.\ \forall (E,S) \in [\![\phi]\!].\ \forall \vartheta \in [\![\psi]\!]. \\
\quad n \in [\![\Gamma : \mathtt{MCtx}]\!] \Rightarrow n+1 \in [\![\phi;\psi \vdash \{\mathsf{P}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}]\!]((E,S),\vartheta)
\end{array}$$

where $n \in [\![\Gamma : \mathtt{MCtx}]\!]$ expresses that all the method specifications in Γ hold for at least n steps.

In the interpretation of Hoare triples above, S corresponds to the part of the stack in scope and C to the "rest of the stack". Since we restrict attention to disjoint S and C, the following specification holds trivially for any ϕ, ψ, s, Q, N, and M,

$$\phi,\mathsf{x};\psi \vdash \{\&\mathsf{x} \overset{s}{\mapsto} \mathsf{N}\}\mathsf{s}\{\mathsf{Q}\} \triangleleft \mathsf{M}$$

as x's stack location must be in the domain of S and any stack C in the interpretation of the precondition.

Theorem 3. *The specification logic is sound.*

6 Related and Future Work

Our work has built on Parkinson and Bierman's separation logic for object-oriented programs [16, 14, 15]. Their work did not deal with generics or delegates. In the functional programming world, higher-order separation logic [3, 9] and Hoare type theory [11, 18] have both dealt with parametric polymorphism and first-class functions. We have used ideas from those approaches to extend the separation logic for object-oriented programs.

Our logic uses an assertion that contains a delegate specification. Something similar was used in Hoare Type Theory [11] (using types as specifications), for reasoning about ML-like functions. Recently, a foundational study of the interaction between nested triples and higher-order frame rules have been performed by Schwinghammer *et al.* [19], who restricted attention to an idealized language with immutable variables and storable code (rather than functions / delegates). Here our focus has instead been on treating delegates as they appear in a real language like C♯; future work will show whether we can combine our present logic with more advanced higher-order frame rules.

There have been other approaches to reasoning about delegates in object-oriented languages. Müller and Ruskiewicz [10] have a specification for each delegate type, and then every instance of that type must satisfy the specification. This makes it difficult to specify a generic filter method, as you would require a different type for each semantic filter operation. If the delegate passed to the filter is pure, that is it doesn't modify the heap, then it can be used directly in the specification. However, impure methods cannot be used in specifications.

To address this, Nordio *et al.* [12] add to the assertions the ability to abstractly assert that a delegates' pre- and post-conditions, ($d.pre(x)$ and $d.post(x)$) hold for the argument supplied (x). This means they can express the filter specification, by saying every element of the list satisfies the delegate's post-condition, even if the delegate is not pure. However, more complex examples where the implementation must impose a structure on the delegates' specification, such as the fold method, cannot be handled by [12] as it stands. In the fold example, we require one step's post-condition to tie up with the next steps pre-condition, that is, we accumulate a result. With filter, each delegate call is independent of the others. It is unclear how Nordio *et al.*'s work could be extended to the fold example. Nordio *et al.*'s work has been implemented on top of Spec♯. It remains future work to implement our solution.

Both of these works focus on C♯1.0 style delegates, so they do not need to address the issues of anonymous methods, or the subtleties of variable escape in C♯. This is one of the key contributions of this paper.

Yoshida *et al.* [21] have studied such a language with higher-order functions and local state. Their hiding quantifier has some similarity with our `lookup` assertion. They make a distinction between l-values and r-values of a local variable, unlike Hoare logic, so to regain Hoare's assignment axiom, they extend the definition of substitution. It would be interesting to see if our approach to escaping local state could be used in their language.

Variable side-conditions have been a problem for concurrent separation logic [13]. Bornat *et al.*'s [5, 17] variables as resources is another approach to treating variables by separating them using a separating conjunction. We do not believe our new $\overset{s}{\mapsto}$ assertion can be used to reason about shared variable concurrency, because we cannot split knowledge of a variable using permissions [4]. However, for the sequential setting our approach does not complicate reasoning about standard programs. If we had adapted variables as resources [5, 17] to our setting, proofs not involving escaping variables would have to be altered. Now we can simply treat the majority of variables in the same way as Hoare logic.

In future work we plan to combine the present development with the earlier work of Parkinson and Bierman [16, 15] on inheritance. Since the semantics of our logic is based on the semantics used earlier in [16, 15] we are confident that this will be possible. We also plan to extend jStar [7] to allow for semi-automatic verification of programs with generics.

7 Conclusion

We have shown how to apply higher-order separation logic, in particular quantification over predicates, to reason about generics. Earlier work on separation logic for OO languages [15] used restricted forms of quantification over predicates for reasoning about abstraction; here we show how pleasingly straightforward quantification over predicates applies to reasoning about generics. This is as should be expected from the earlier work on HOSL and HTT mentioned above.

Moreover, we have developed the first logic for reasoning about C♯ 2.0 style delegates, involving anonymous methods and capture of variables. To reason about escaping variables, we introduced the assertion

lookup l **as** x **in** P

and the term &x denoting the address of local variable x, with associated proof rules. Soundness was proved by a new model of separation logic in which the truth value of an assertion, relative to a stack corresponding to the topmost stack frame, is a subset of pairs of heaps and stacks (these stacks containing values for escaped variables). We have demonstrated the applicability of the logic via several small, but non trivial, examples.

8 Acknowledgements

We would like to thank our anonymous reviewers for their comments and in particular, for suggesting that `lookup` could be defined in terms of $\overset{s}{\mapsto}$.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *CASSIS*, pages 49–69, 2005.
3. B. Biering, L. Birkedal, and N. Torp-Smith. Bi hyperdoctrines and higher-order separation logic. In *In Proceedings of European Symposium on Programming 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 233–247, 2005.
4. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, pages 259–270, 2005.
5. R. Bornat, C. Calcagno, and H. Yang. Variables as resources in separation logic. In *Proceedings of MFPS*, pages 125–146, 2005.
6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *FMICS*, pages 73–89, 2003.
7. D. Distefano and M. J. Parkinson. jstar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
9. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *In Proceedings of TLDI'09*, pages 105–116, 2009.
10. P. Müller and J. N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, 2007. To appear.
11. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. In *In Proceedings of ESOP'08*, pages 189–204, 2007.
12. M. Nordio, C. Calcagno, B. Meyer, and P. Müller. Reasoning about function objects. Technical Report 615, ETH Zurich, 2009.
13. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
14. M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, November 2005.
15. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, January 2008.
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*, pages 137–146. IEEE, 2006.
18. R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative hoare type theory. In *In Proceedings of ESOP'08*, pages 337–352, 2008.
19. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL 2009*, Apr. 2009.
20. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates (technical appendix). Technical report, 2009. Available at `http://www.itu.dk/people/kasv/generics-delegates-tr.pdf`.
21. N. Yoshida, K. Honda, and M. Berger. Logical reasoning for higher-order functions with local state. In *FoSSaCS*, pages 361–377, 2007.