Verifying Generics and Delegates

Kasper Svendsen
Lars Birkedal
Matthew Parkinson

ECOOP 2010

June 23, 2010

Introduction

HO separation logic for C# subset with generics and delegates

- Hoare-style program logic for proving functional correctness
- Separation logic for modular reasoning about state

In HOSL & Hoare Type Theory one can reason about

- polymorphism using universal quantification over predicates
- first-class functions using nested Hoare triples

We extend these techniques to reason about C#

- Main challenge: C# variable capture

C# variable capture

Example

```
public delegate Y Func⟨Y⟩ ();

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

C# semantics

- Inline delegate captures the location of x
- Lifetime of captured x extended to lifetime of delegate

Outline

Setup

C# subset

- Basic imperative features + generics + delegates

Assertion logic

- Logic for reasoning about computational states
- Higher-order separation logic
- Spacial connectives, $\mathit{emp}, *, -\!*$, for controlling aliasing
- State assertions, $\mathrm{M}.f \mapsto \mathrm{N}, \ldots$, for describing state

Specification logic

- Logic for relating initial and terminal state

Example – Integer list

```
class Node {
  Node next;
  Integer val;
}
```

Example – Integer list

```
class Node {
  Node next;
  Integer val;
}
```

- Representation predicate

$$\begin{aligned} list(x, \varepsilon) &\stackrel{def}{=} x = null \\ list(x, a \cdot \alpha) &\stackrel{def}{=} \exists n, \exists v.\ x.next \mapsto n * x.val \mapsto v * \\ &\qquad\qquad \underline{Int(v, a)} * list(n, \alpha) \end{aligned}$$

- $Int(v, a)$: v is a representation of the integer a

Example – Generic list

```
class Node⟨X⟩ {
  Node⟨X⟩ next;
  X val;
}
```

- Representation predicate

$$
\begin{aligned}
list(x, \varepsilon, P) &\stackrel{def}{=} x = null \\
list(x, a \cdot \alpha, P) &\stackrel{def}{=} \exists n, \exists v.\ x.next \mapsto n * x.val \mapsto v * \\
&\qquad\qquad\qquad \underline{P(v, a)} * list(n, \alpha, P)
\end{aligned}
$$

- $P(v, a)$: v is a representation of a

Example – Fold

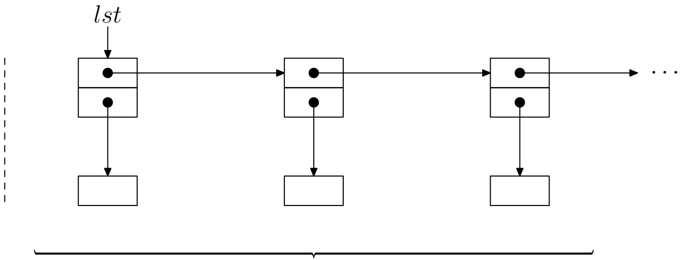
```
void fold⟨X⟩(Node⟨X⟩ lst, Action⟨Node⟨X⟩⟩ f) {
  if(lst != null) {
    Node⟨X⟩ next = lst.next;
    f(lst);
    fold(next, f);
  }
}
```

Example – Fold

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

- foreach
- Stateful fold-right – accumulator maintained by delegate

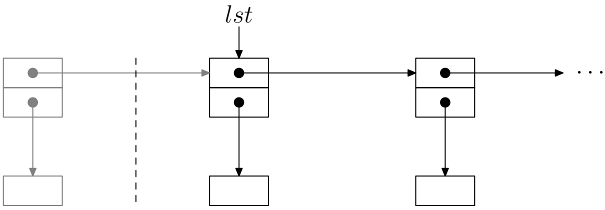
Example – Fold



$$Q(\varepsilon) \quad * \quad \overbrace{\hphantom{xxxxxxxxxxxx}}^{}\!\! list(lst, a_1 \cdot a_2 \cdot a_3 \cdot \alpha, P)$$

- $Q(\alpha)$: accumulator predicate; state after having folded over α

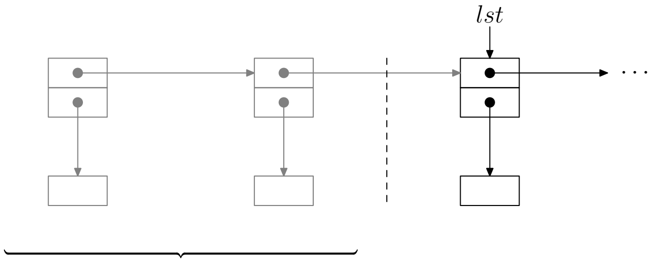
Example – Fold



$$Q(a_1 \cdot \varepsilon) \quad * \quad \underbrace{}_{list(lst, a_2 \cdot a_3 \cdot \alpha, P)}$$

- $Q(\alpha)$: accumulator predicate; state after having folded over α

Example – Fold



$$Q(a_2 \cdot a_1 \cdot \varepsilon) \quad * \, list(lst, a_3 \cdot \alpha, P)$$

- $Q(\alpha)$: accumulator predicate; state after having folded over α

Example – Fold

Specification

$$
\begin{array}{l}
\textbf{void}\ \mathsf{fold}\langle \mathsf{X}\rangle(\mathsf{Node}\langle \mathsf{X}\rangle\ \mathsf{lst},\ \mathsf{Action}\langle \mathsf{Node}\langle \mathsf{X}\rangle\rangle\ \mathsf{f})\ \{\ \ldots\ \} \\
\quad \forall \alpha : \textbf{Val}.\ \forall \mathrm{P} : \textbf{Val} \times \textbf{Val} \to \textbf{Prop}.\ \forall \mathrm{Q} : \textbf{Val} \to \textbf{Prop}. \\
\qquad \{ list(lst, \alpha, \mathrm{P}) * \mathrm{Q}(\varepsilon) * \forall a, \beta, y : \textbf{Val}. \\
\qquad\quad f \mapsto \langle (x).\{x.next \mapsto \_ * x.val \mapsto y * \mathrm{P}(y, a) * \mathrm{Q}(\beta)\} \\
\qquad\qquad\qquad \{Q(a \cdot \beta)\}\rangle \} \\
\qquad \{\mathrm{Q}(rev(\alpha))\}
\end{array}
$$

- $Q(\alpha)$: accumulator predicate; state after having folded over α
- $rev(\alpha)$: mathematical reverse function
- $\mathrm{M} \mapsto \langle(\phi).\{\mathrm{P}\}\_\{\mathrm{Q}\}\rangle$: M denotes delegate satisfying spec

Example – Fold

Specification

$$
\begin{array}{l}
\textbf{void}\ \mathsf{fold}\langle \mathsf{X}\rangle(\mathsf{Node}\langle \mathsf{X}\rangle\ \mathsf{lst},\ \mathsf{Action}\langle \mathsf{Node}\langle \mathsf{X}\rangle\rangle\ \mathsf{f})\ \{\ \dots\ \}\\
\quad \forall \alpha : \textbf{Val}.\ \forall \mathrm{P} : \textbf{Val} \times \textbf{Val} \to \textbf{Prop}.\ \forall \mathrm{Q} : \textbf{Val} \to \textbf{Prop}.\\
\qquad \{ \mathit{list}(\mathit{lst}, \alpha, \mathrm{P}) * \mathrm{Q}(\varepsilon) * \forall a, \beta, y : \textbf{Val}.\\
\qquad\quad f \mapsto \langle (x).\{x.\mathit{next} \mapsto \_ * x.\mathit{val} \mapsto y * \mathrm{P}(y, a) * \mathrm{Q}(\beta)\}\\
\qquad\qquad\qquad \{Q(a \cdot \beta)\}\rangle\}\\
\qquad \{\mathrm{Q}(\mathit{rev}(\alpha))\}
\end{array}
$$

- $Q(\alpha)$: accumulator predicate; state after having folded over α
- $\mathit{rev}(\alpha)$: mathematical reverse function
- $\mathrm{M} \mapsto \langle(\phi).\{\mathrm{P}\}_{\_}\{\mathrm{Q}\}\rangle$: M denotes delegate satisfying spec

Variable capture

Increment example

```
int x = 0;
Action f = delegate () {
    x++;
};
f();
```

Variable capture

Increment example

int x = 0;
Action f = **delegate** () {
$\{ x = n \}$
x++;
$\{ x = n + 1 \}$
};
$\{ x = 0 * \forall n.\ f \mapsto \langle \{ x = n \}\_\{ x = n + 1 \} \rangle \}$
f();

Variable capture

Increment example

int $x = 0$;
Action f = **delegate** () {
    $\{ x = n \}$
  x++;
    $\{ x = n + 1 \}$
};
 $\{ x = 0 * \forall n.\ f \mapsto \langle \{ x = n \}\_\{ x = n + 1 \} \rangle \}$
f(); $\Rightarrow f \mapsto \langle \{ 0 = 0 \}\_\{ 0 = 1 \} \rangle$

Variable capture

Increment example

int x = 0;
Action f = **delegate** () {
$\{\ \mathsf{emp}\ \}$
x++;
$\{\ \mathsf{emp}\ \}$
};
$\{\ x = 0 * f \mapsto \langle\{\ \mathsf{emp}\ \}\_\{\ \mathsf{emp}\ \}\rangle\ \}$
f();

Variable capture

Increment example

```
int x = 0;
Action f = delegate () {
      { emp }
    x++;
      { emp }
};
  { x = 0 * f ↦ ⟨{ emp }_{ emp }⟩ }
f();
```

Issues

- How to refer to captured variables in nested specs
- How to keep track of potentially modified variables

Variable capture

Variable assertions

- Extend assertion logic with variable assertions $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}, \&x$
  - $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}$: location M is allocated on the stack and contains N
  - $\&x$: denotes location of program variable x

Variable capture

Variable assertions

- Extend assertion logic with variable assertions $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}, \&x$
  - $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}$: location M is allocated on the stack and contains N
  - $\&x$: denotes location of program variable x

Inline delegate

```
int x = 0;
Action f = delegate () {
    { P }
  B
    { Q }
};
```

$\{\ \mathsf{x} = 0 * \mathsf{f} \mapsto \langle\{\ \exists \mathsf{y}.\ \&\mathsf{x} \overset{s}{\mapsto} \mathsf{y} * [\mathsf{y}/\mathsf{x}]\mathsf{P}\ \}\_\{\ \exists \mathsf{y}.\ \&\mathsf{x} \overset{s}{\mapsto} \mathsf{y} * [\mathsf{y}/\mathsf{x}]\mathsf{Q}\ \}\rangle\ \}$

Variable capture

Variable assertions

- Extend assertion logic with variable assertions $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}, \&x$
  - $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}$: location M is allocated on the stack and contains N
  - $\&x$: denotes location of program variable x

Inline delegate

int x = 0;
Action f = **delegate** () {
  $\{\ \mathsf{x} = \mathsf{n}\ \}$
 x++;
  $\{\ \mathsf{x} = \mathsf{n} + 1\ \}$
};
$\{\ \mathsf{x} = 0 * \forall \mathsf{n}.\ \mathsf{f} \mapsto \langle\{\ \&\mathsf{x} \overset{s}{\mapsto} \mathsf{n}\ \}\_\{\ \&\mathsf{x} \overset{s}{\mapsto} \mathsf{n} + 1\ \}\rangle\ \}$

Variable capture

Variable assertions

- Extend assertion logic with variable assertions $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}, \&x$
  - $\mathrm{M} \overset{s}{\mapsto} \mathrm{N}$: location M is allocated on the stack and contains N
  - $\&x$: denotes location of program variable x

Inline delegate

```
int x = 0;
Action f = delegate () {
    { emp }
  x++;
    { emp }
};
```

$\{\ \mathsf{x} = 0 * \mathsf{f} \mapsto \langle\{\ \exists \mathsf{y}.\ \&\mathsf{x} \overset{s}{\mapsto} \mathsf{y}\ \}\_\{\ \exists \mathsf{y}.\ \&\mathsf{x} \overset{s}{\mapsto} \mathsf{y}\ \}\rangle\ \}$

Variable capture

Aliasing

- Var. assertions introduce aliasing in reasoning about variables:
- Build separation into specification logic:
  - Can either reason directly or indirectly, but not both
  - Reason directly about variables in the program var. ctx. ϕ

Variable capture

Aliasing

- Var. assertions introduce aliasing in reasoning about variables:
- Build separation into specification logic:
    - Can either reason directly or indirectly, but not both
    - Reason directly about variables in the program var. ctx. ϕ
    - Hoare's assignment rule thus stil sound

$$\frac{\phi; \psi \vdash \mathrm{P} : \textbf{Prop} \qquad x, y \in \phi}{\phi; \psi \vdash \{\mathrm{P}[y/x]\} x := y \{\mathrm{P}\}}$$

Variable capture

Capturing delegates

- Verify body using Hoare treatment of variables
- Switch to SL treatment of captured variables in nested spec
- Switch back to Hoare treatment of variables to verify calls

Variable capture

Capturing delegates

- Verify body using Hoare treatment of variables
- Switch to SL treatment of captured variables in nested spec
- Switch back to Hoare treatment of variables to verify calls

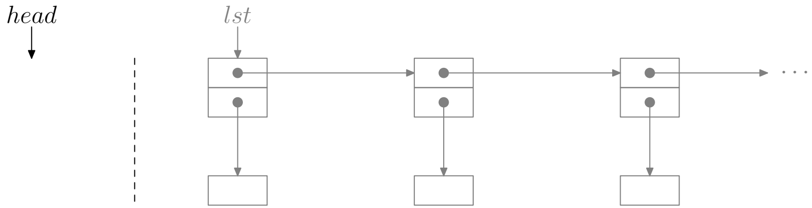
$$\frac{x \notin FV(s) \qquad \phi; \psi \vdash \{\exists x : \mathbf{Val}.\ l \overset{s}{\mapsto} x * \mathrm{P}\} s \{\exists x : \mathbf{Val}.\ l \overset{s}{\mapsto} x * \mathrm{Q}\}}{\phi, x; \psi \vdash \{\&x = l \wedge \mathrm{P}\} s \{\mathrm{Q}\}}$$

Example – In-place reverse

```
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;
}
```

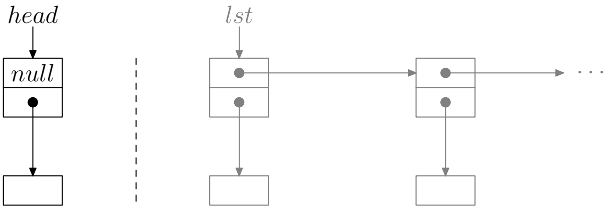
Example – In-place reverse

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;
}



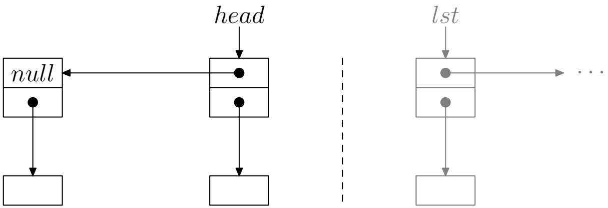
Example – In-place reverse

```
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;
}
```



Example – In-place reverse

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;
}



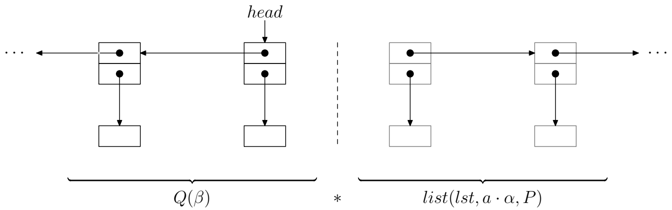
Example – In-place reverse

```
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;
}
```

$$\forall \alpha : \textbf{Val}.\ \forall \mathrm{P} : \textbf{Val} \times \textbf{Val} \to \textbf{Prop}.$$
$$\{\mathit{list}(\mathit{lst}, \alpha, \mathrm{P})\}$$
$$\{r.\mathit{list}(r, \mathit{rev}(\alpha), \mathrm{P})\}$$

Example – In-place reverse

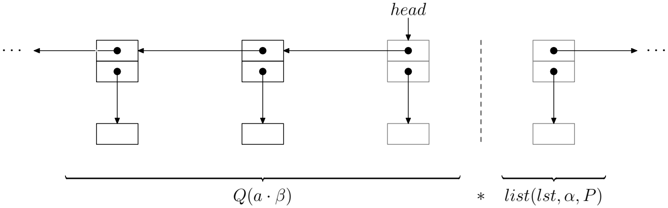
```
Node⟨X⟩ reverse⟨X⟩(Node⟨X⟩ lst) {
  Node⟨X⟩ head = null;
  fold⟨X⟩(lst, delegate (Node⟨X⟩ x) { x.next = head; head = x; });
  return head;
}
```



where $\mathrm{Q}(\alpha) = \exists n : \textbf{Val}.\ \&head \overset{s}{\mapsto} n * list(n, \alpha, \mathrm{P})$

Example – In-place reverse

```
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;
}
```



where $\mathrm{Q}(\alpha) = \exists n : \mathbf{Val}.\ \&head \overset{s}{\mapsto} n * list(n, \alpha, \mathrm{P})$

Conclusion

Generics and non-capturing delegates

- HOL & nested Hoare triples (standard)

Capturing delegates

- Separation logic treatment of variables
- Variable separation build into specification logic

- Unified treatment of local state on the heap and/or stack
- Reasoning standard when there is no capturing