Specifying and Verifying Concurrent Programs with Ghost State

William Mansky, Princeton University

at DeepSpec 2018

Concurrency in Verifiable C

Verified Software Toolchain

$\{\mathrm{x} \overset{5}{\mapsto} v\}$ (read-only)

```
void f(int *x){
   …
   y = *x + 1;
   …
}
```

$\{\exists p.\, p \mapsto v + 1 * \mathrm{x} \overset{5}{\mapsto} v\}$

Concurrency in Verifiable C

Verified Software Toolchain

$\{\mathrm{lock}(\mathrm{l}, \mathrm{x} \mapsto v)\}$ (writable)

```
void f(int *x){
  acquire(l);
  y = *x + 1;
  release(l);
}
```

$\{\exists p.\, p \mapsto v + 1 * \mathrm{lock}(\mathrm{l}, \mathrm{x} \mapsto v)\}$

Concurrency in Verifiable C

Verified Software Toolchain

- Distribute memory between threads with shares
- Lock-based concurrency library (makelock, freelock, acquire, release, spawn)

- Newest concurrent feature: ghost state

What Is Ghost State?

$\{x = N\}$

```
y = 1;
```

$\{x = N \wedge y = 1\}$

```
while (x > 0){

  y = 3 * y;
  x = x / 2;

}
```

$\{x = 0 \wedge y = 3\text{^}\lg(N)\}$

What Is Ghost State?

$\{x = N\}$

```
y = 1;
```

$\{x = N \wedge y = 1 \wedge z = 0\}$

```
while (x > 0){
```

$\{x = N / (2^z) \wedge y = 3^z\}$

```
  y = 3 * y;
  x = x / 2;
```

$\{x = N / (2^{(z+1)}) \wedge y = 3^{(z+1)}\}$

```
}
```

$\{x = 0 \wedge y = 3^{\lg(N)}\}$

Ghost State in Concurrent Separation Logic

```
*x = 0;

release(l);
```

$\{\mathrm{lock}(\mathrm{l}, R)\}$

```
acquire(l); || acquire(l);
            ||
 x = x + 1; || x = x + 1;
            ||
release(l); || release(l);
```

$\{\mathrm{lock}(\mathrm{l}, R)\}$

```
acquire(l);
```

$\{\mathrm{lock}(\mathrm{l}, R) * R\}$

$R \coloneqq \exists v. \mathrm{x} \mapsto v$

Ghost State in Concurrent Separation Logic

```
*x = 0;
```

$\{\mathrm{lock}(\mathrm{l}, R) * \mathrm{x} \mapsto 0 * g_1 \mapsto 0 * g_2 \mapsto 0\}$

```
release(l);
```

$\{\mathrm{lock}(\mathrm{l}, R) * g_1 \overset{.5}{\mapsto} 0 * g_2 \overset{.5}{\mapsto} 0\}$

```
acquire(l); || acquire(l);
```

$\{\ldots * g_1 \mapsto 0 * \mathrm{x} \mapsto 0 + v_2 * \ \ldots\}$ ||

```
x = x + 1; || x = x + 1;
```

$\{\ldots * g_1 \mapsto 1 * \mathrm{x} \mapsto 1 + v_2 * \ \ldots\}$ ||

```
release(l); || release(l);
```

$\{\mathrm{lock}(\mathrm{l}, R) * g_1 \overset{.5}{\mapsto} 1 * g_2 \overset{.5}{\mapsto} 1\}$

```
acquire(l);
```

$\{\mathrm{lock}(\mathrm{l}, R) * x \mapsto 2 * g_1 \mapsto 1 * g_2 \mapsto 1\}$

$R \ := \exists v_1, v_2. \mathrm{x} \mapsto v_1 + v_2 * g_1 \overset{.5}{\mapsto} v_1 * g_2 \overset{.5}{\mapsto} v_2$

What Can Ghost State Be?

- Ghost state can be anything with the right algebraic structure!
- Any $(M,\oplus)$ such that

$$a \oplus b = b \oplus a \text{ and } (a \oplus b) \oplus c = a \oplus (b \oplus c)$$

(partial commutative monoid, resource algebra, separation algebra)

- Essentially, anything that can be inserted into separation logic:

$$\mathrm{ghost}(a) * \mathrm{ghost}(b) \Leftrightarrow \mathrm{ghost}(a \oplus b)$$

- Examples: variables, shares, graphs, state machines, histories, …

$$(g, \pi_1, v) \oplus (g, \pi_2, v) = (g, \pi_1 \oplus \pi_2, v) \quad g \overset{\pi}{\mapsto} v ::= \mathrm{ghost}\big((g, \pi, v)\big)$$

Updating Ghost State

- We can change the value of the ghost variable when we have all of it.
- More generally, we can make *frame-preserving updates*:

$$a \rightsquigarrow b ::= \forall c, a \oplus c \text{ defined} \Rightarrow b \oplus c \text{ defined}$$

- I.e., we can't invalidate anyone else's ghost state
- Shared variables: $\pi_1 \oplus \pi_2 ::= \pi_1 + \pi_2$ when $\pi_1, \pi_2 > 0, \pi_1 + \pi_2 \leq 1$
- In CSL:

$$\frac{a \rightsquigarrow b}{\text{ghost}(a) \Rrightarrow \text{ghost}(b)}$$

What Can Ghost State Do?

- Ghost state is useful for tracking facts we could have recorded in the original program.
- With a non-trivial join operation, it also places restrictions on the possible states of *other parts* of the program. I.e., it enforces protocols between threads.
- Ghost state is essentially a generalization of the SL heap to an arbitrary mathematical object.
- We can use it to do global reasoning locally!

Ghost State and Global Invariants

$\{\mathrm{inv}(\exists a\ b.\mathrm{x} \mapsto a * \mathrm{y} \mapsto b * a \mid b)\}$

```
*y = *x + *y;
```

$\{\mathrm{inv}(\exists a\ b.\mathrm{x} \mapsto a * \mathrm{y} \mapsto b * a \mid b)\}$

```
*x = 1;
```

$\{\mathrm{inv}(\exists a\ b.\mathrm{x} \mapsto a * \mathrm{y} \mapsto b * a \mid b)\}$

$(I * \mathrm{ghost}(\mathbf{off})) \vee \mathrm{ghost}(\mathbf{on})$

- We can define $\mathrm{lock}(\mathrm{l}, R) ::= \mathrm{inv}((\mathrm{l} \mapsto 0 * R) \vee \mathrm{l} \mapsto 1)$

Based on “Iris from the Ground Up”, Jung et al., to appear in JFP ‘18

Histories and Linearizability

- A concurrent data structure is *linearizable* if its behavior cannot be distinguished from that of a sequential implementation

$$R ::= \exists H.\,\mathrm{hashtable}(H, \mathrm{p}) * \exists h.\,\mathrm{consistent}(H, h) \wedge \mathrm{ghost}(h)$$

$$\{\mathrm{inv}(R) * \mathrm{ghost}(h)\}\ \mathrm{x} = \mathrm{get}(\mathrm{p}, \mathrm{k});\ \{\mathrm{inv}(\mathrm{R}) * \mathrm{ghost}(h; \mathrm{Get\ k\ x})\}$$

$$\{\mathrm{inv}(R) * \mathrm{ghost}(h)\}\ \mathrm{set}(\mathrm{p}, \mathrm{k}, \mathrm{x});\ \{\mathrm{inv}(\mathrm{R}) * \mathrm{ghost}(h; \mathrm{Set\ k\ x})\}$$

Ghost State in Verifiable C

Verified Software Toolchain

```
ghost_alloc (ghost_var Tsh 0).
```

…

```
viewshift_SEP 2 (ghost_var Tsh 1 g).
```

{ *prove* $g_1 \mapsto 0 \rightsquigarrow g_1 \mapsto 1$ }

```
*x = 0;
```

$\{x \mapsto 0 * \mathrm{lock}(l, R)\}$

$\{x \mapsto 0 * \mathrm{lock}(l, R) * g_1 \mapsto 0\}$

…

$\{\mathrm{lock}(l, R) * g_1 \mapsto 0 * \cdots\}$

$\{\mathrm{lock}(l, R) * g_1 \mapsto 1 * \cdots\}$

- Double-increment example shipped with VST, brief manual available online

More Verified Concurrent Programs

- “Mailbox” communication system – atomic exchange, ghost variables
- Hashtable and key-value node – atomic load/store/CAS, histories and linearizability
- Future: high-performance database implementation (Masstree) – relaxed load/store, hand-over-hand locking, etc.

Conclusion

- Concurrent separation logic without ghost state lets us prove race freedom, and verify steady-state programs with lock invariants.
- But ghost state unlocks the true power of CSL, letting us use the same local reasoning to prove global properties like linearizability.
- Ghost state fits neatly into VST, so we can now prove linearizability of concurrent C programs.
- Looking forward to applying it to weak memory, fine-grained data structures, and any concurrent applications you might have!

Note: On Sequential Consistency

- All of this only works under SC!
- Global invariants are unsound under more relaxed models (*)
- The earlier approach falls apart when we try to connect the ghost state to the values of data in memory (what does “data in memory” even mean?)
- RSL and GPS provide more limited, per-location invariant reasoning