

Separation Logic 2:

project: Annotation Assistant for Partially Specified Programs
plus some high-level observations



Hengle Jiang & John-Paul Ore

Recap : Separating Conjunction

$$P * Q$$

$$\left\{ \begin{array}{c} P \\ \text{heaplet} \end{array} \right\} \cap \left\{ \begin{array}{c} Q \\ \text{heaplet} \end{array} \right\} = \emptyset$$

Recap : Frame Rule

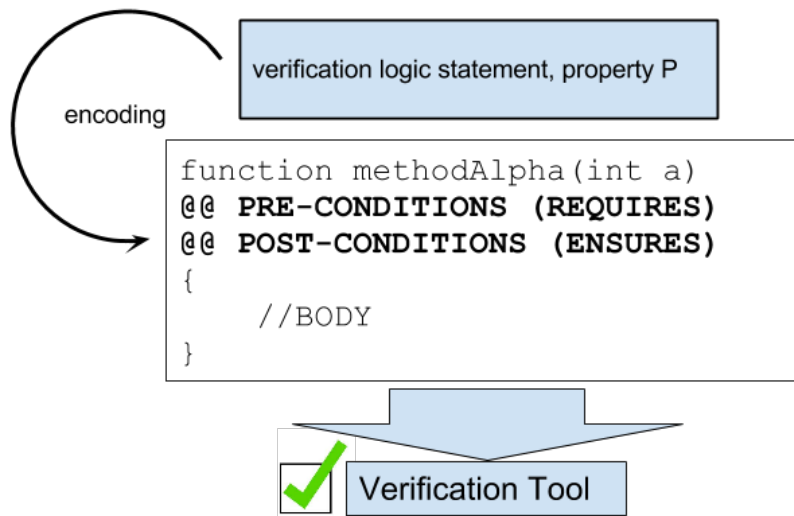
$$\frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}}$$

$$\textit{Modifies}(C) \cap \textit{Free}(R) = \emptyset$$

Le Menu

- Annotations And Verification tools
- Functions of an Annotation Assistant in Nearly Annotated Source
- Some high-level observations about Separation Logic

Annotations



Your Source Code

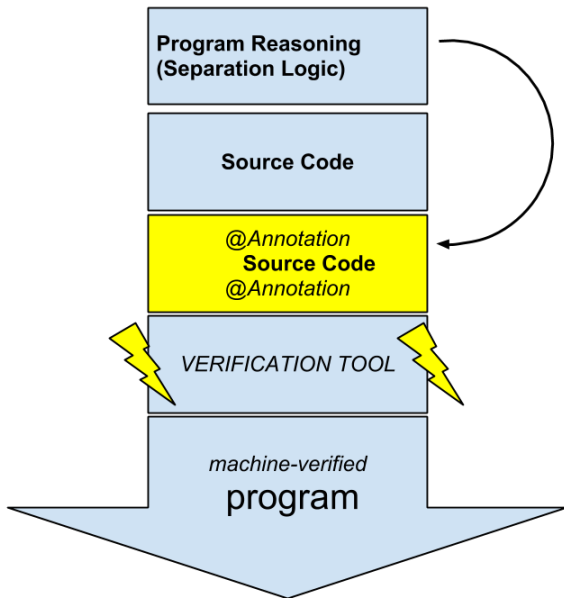
1. `methodAlpha(a);`

2. `function methodAlpha(int a)`
 `@@ PRE-CONDITIONS (REQUIRES)` } *encoding of*
 `@@ POST-CONDITIONS (ENSURES)` } *properties from*
 `{` *Separation Logic*
 `//BODY`
 `}`

3.

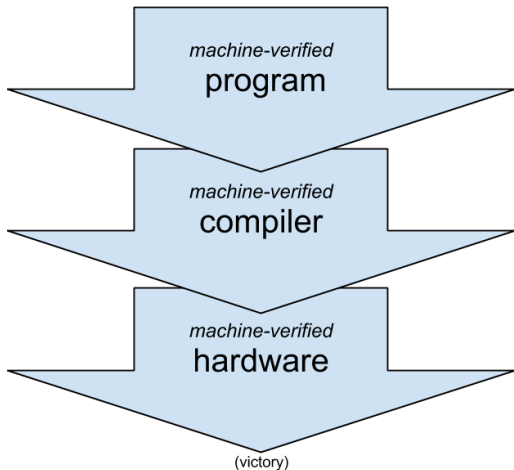
```
void Program() {
    ...
    methodAlpha(a);
    methodBeta(b);
    ...
}
```

} **ENTIRE**
*program has
desired property*



”The future of program verification is to connect machine-verified source programs to machine-verified compilers, and run the object code on machine-verified hardware.” - Andrew Appel & Sandrine

Blazy, 2007



Source Code

```
void Program() {
```

```
    PRECONDITION P
```

```
    ...  
    methodAlpha(a);  
    ...  
    methodBeta(b);  
    ...
```

```
    POSTCONDITION Q
```

```
}
```

Source Code

```
void Program() {
```

PRECONDITION P



PRECONDITION P'

```
...  
methodAlpha(a);
```

```
...
```

```
methodBeta(b);
```

```
...
```

POSTCONDITION Q'



POSTCONDITION Q

```
}
```

}
Frame rule
transformations

}
Frame rule
transformations

Source Code

```
void Program() {
```

PRECONDITION P1*P2

...

PRECONDITION P1'

C1;

POSTCONDITION Q1'

PRECONDITION P2'

C2;

POSTCONDITION Q2'

...

POSTCONDITION Q1*Q2

```
}
```

Use Separation Logic to infer multiple disjoint local specifications.

$\{P1'\}C1\{Q1'\} \quad \{P2'\}C2\{Q2'\}$

$\{P1*P2\} \text{Program} \{P2*Q2\}$

If the footprints of the code chunks are disjoint, the tool still can work by passing local specifications to corresponding code chunks.

Annotation Assistance tool

Example 1: want to verify setBalance method, but do not know the specification of deposit method.

```
typedef struct account{
    int balance;
} Account

void setBalance(Account *a, int newBalance)
    requires a->balance == _;
    requires newBalance > 0;
    ensures a->balance == newBalance;
{
    clearBalance(a); //clear balance to 0
    deposit(a, newBalance);
}
```

The precondition of deposit call:
{a->balance == 0 &&& newBalance > 0}

The postcondition of deposit call:
{a->balance == newBalance}

The specification of deposit method:
void deposit(Account* a, int amount)
 requires a->balance == 0;
 requires amount > 0;
 ensures a->balance == amount;
{
}

Annotation Assistance tool

Example 2: Two methods calls on disjoint heaplets.

```
void setBalance(Account *a1,
  Account *a2, int newBalance)
  requires a1->balance == _
  && a2->balance == _;
  requires newBalance > 0;
  ensures a1->balance == newBalance;
  && a2->balance == newBalance;
{
  clearBalance(a1);
  deposit(a1, newBalance);
  clearBalance(a2);
  deposit(a2, newBalance);
}
```

The VCs of two deposit calls:

```
{a1->balance == 0 && newBalance > 0}
deposit(a1, newBalance);
{a1->balance == newBalance}
```

```
{a2->balance == 0 && newBalance > 0}
deposit(a2, newBalance);
{a2->balance == newBalance}
```

The specification of deposit method:

```
void deposit(Account* a, int amount)
  requires a->balance == 0;
  requires amount > 0;
  ensures a->balance == amount;
{
}
```

Annotation Assistance tool

Example 3: Two methods calls on two symbolic execution paths.
The specifications inferred are dependent on the contexts.

```
void setBalance(Account *a, int nb)
  requires a->balance == _;
  ensures a->balance == nb;
{
  int cb = getBalance(a);
  if(nb > cb){
    deposit(a, nb - cb);
  }else{
    withdraw(a, cb - nb);
  }
}
```

The VCs of the deposit call:
{a->balance == cb && nb > cb}
deposit(a, nb - cb);
{a->balance == nb}

The specification of deposit method:
void deposit(Account* a, int amount)
 requires a->balance == ?cb;
 requires amount > 0;
 ensures a->balance == cb + amount;

Similarly, the specification of withdraw method:
void withdraw(Account* a, int amount)
 requires a->balance == ?cb;
 requires amount >= 0;
 ensures a->balance == cb - amount;

Annotation Assistance tool

Example 4: Same methods calls on two symbolic execution paths.
The specifications inferred are dependent on the contexts.

```
void setBalance(Account *a1,
  Account *a2, int nb)
  requires a1->balance == _;
  &&& a2->balance == _;
  requires nb > 0;
  ensures a1->balance == nb;
  &&& a2->balance == nb;
{
  clearBalance(a1);
  deposit(a1, nb);

  int cb = getBalance(a2);
  if(nb > cb){
    deposit(a2, nb - cb);
  }else{
    withdraw(a2, cb - nb);
  }
}
```

The VCs of first deposit calls:
{a1->balance == 0 &&& nb > 0}
deposit(a1, nb);
{a1->balance == nb}

The inferred spec from first deposit call:
void deposit(Account* a, int amount)
 requires a->balance == 0;
 requires amount > 0;
 ensures a->balance == amount;

The inferred spec from second deposit call:
void deposit(Account* a, int amount)
 requires a->balance == ?cb;
 requires amount > 0;
 ensures a->balance == cb + amount;

We may find the second spec and also satisfy
the first context, but not on the other
direction.

So the tool needs to output the second spec.

Annotation Assistance tool

Example 5: Multiple calls on the same symbolic path.

```
void setBalance(Account *a, int nb)
  requires a->balance == _;
  requires nb > 0;
  ensures a->balance == nb;
{
  clearBalance(a);
  deposit(a, nb);
}
```

- ▶ The tool cannot work in this scenario. We may use a SMT solver to guess the two specifications, but it would not be very helpful to programmers.

Annotation Assistance tool

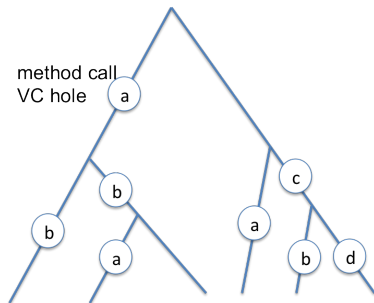
Example 6: A trivial bug

```
void setBalance(Account *a, int nb)
  requires a->balance == _;
  requires nb > 0;
  ensures a->balance == nb;
{
  clearBalance(a);
}
```

- ▶ In the other direction, if we find footprints mismatch between global specification and local specifications, we know we cannot never verify it no matter what local specification we provide. The tool should detect such trivial error.

Annotation Assistance tool

Verification and Inferring through Symbolic Execution.



Use loop invariant to verify loop, so symbolic execution here does not have unroll bound problem.

1. Generate symbolic execution paths;
2. On each path verify and shrink the specifications by separation logic;
3. Solve as many as possible local specifications;
4. Refine multiple specifications of the same method.

Annotation Assistance tool

Houdini, an Annotation Assistant for ESC/Java (2001)



Cormac Flanagan and K. Rustan M. Leino

- ▶ Generates a large number of candidate annotations heuristically from program contexts;
- ▶ Uses ESC/Java to verify or refute each of these annotations.
- ▶ Different from our proposed approach.
- ▶ have not found many related works on this direction.

Annotation Assistance tool

Example 7: Global to local reasoning

```
QuickSort(a:array<int>, p:int, r:int)
  requires a != null;
  ensures sorted(a,p,r);
{
  if(p < r){
    var q := Partition(a,p,r);
    QuickSort(a,p,q-1);
    QuickSort(a,q+1,r);
  }
}
```

- ▶ If we know the specification of QuickSort, we can infer the specification of Partition method by separation logic.
- ▶ It is natural for programmer to construct verification this way.

Annotation Assistance tool

Example 8: Loop invariant

```
Partition(a:array<int>, p:int, r:int)
returns (q: int)
  requires a != null;
  ensures finalPosition(a,q);
{
  var x: int := a[r];
  var i: int := p - 1;
  var j: int := p;
  var swap: int;
  while(j < r)
  {
    if(a[j] <= x){
      i := i + 1;
      swap := a[i];
      a[i] := a[j];
      a[j] := swap;
    }
    j := j + 1;
  }
  swap := a[i+1];
  a[i+1] := a[r];
  a[r] := swap;
  q := i + 1;
}
```

- ▶ Can we infer loop invariant since we have local specification of the loop?
- ▶ To verify "while(C) LOOP;":
 $\{P \wedge C\} \Rightarrow \{INV\}$
 $\{INV \wedge C\} LOOP \{INV\}$
 $\{INV \wedge \neg C\} \Rightarrow \{Q\}$
- ▶ Can a SMT solver solve *INV* just based on *P* and *Q*?

Annotation Assistance tool

Loop invariants: analysis, classification, and examples(2012)
by Carlo A. Furia, Bertrand Meyer and Sergey Velder

Invariant inference relies on implementing a number of heuristics for mutating postconditions into candidate invariants.

- ▶ **Constant relaxation:** replace a constant n by a variable i , and use $i = n$ as part or all of the exit condition.
- ▶ **Uncoupling:** replace a variable v by two, using their equality as part or all of the exit condition.
- ▶ **Term dropping:** remove a sub formula, which gives a straightforward weakening.
- ▶ **Aging:** replace a variable (more generally, an expression) by an expression that represents the value the variable had at previous iterations of the loop.

Annotation Assistance tool

Other possible approaches on static loop invariant inference

- ▶ **Abstract interpretation:** is a symbolic execution of programs over abstract domains that over-approximates the semantics of loop iteration.

To verify "while(C) LOOP;":

$$\{P \wedge C\} \Rightarrow \{INV\}$$

$$\{INV \wedge C\} LOOP \{INV\}$$

$$\{INV \wedge \neg C\} \Rightarrow \{Q\}$$

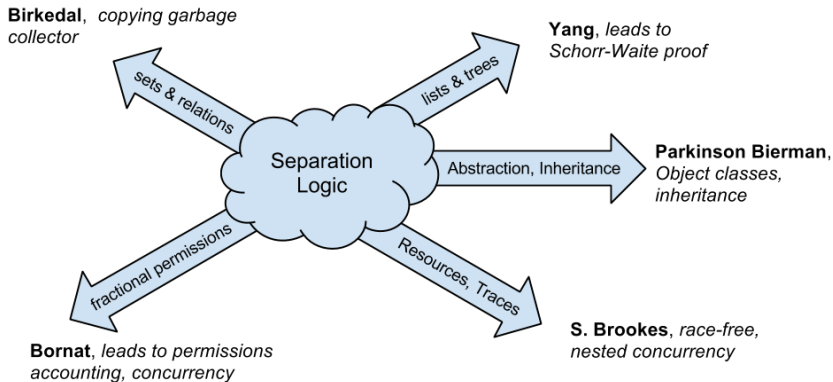
- ▶ **Constraint-based** techniques rely on sophisticated decision procedures over non-trivial mathematical domains to represent concisely the semantics of loops with respect to certain template properties.
- ▶ **Local Reasoning about While-Loops**, by Thomas Tuerk
"When using separation logic, recursive implementations are often much easier to specify and verify than the corresponding imperative ones."

Annotation Assistance tool

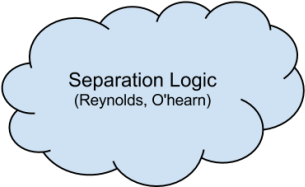
Goals of the tool

- ▶ Use separation logic to infer local specifications of methods on a partially specified program.
- ▶ Infer loop invariant using some advanced techniques.
- ▶ Detect some footprint errors in partially specified program.
- ▶ Could be part of a verification tool.

Part 2: *Some high-level observations*



“Thus it is natural to ask whether one has to make a new extension of separation logic for every proof one wants to make.” – Bodil Biering, Lars Birkedal, and Noah Torp-Smith



Separation Logic
(Reynolds, O'hearn)



**Bunched
Implications**
(O'Hearn & Pym)

Separation Logic

BI-Hyperdoctrines, Higher-Order Separation Logic

(BIERING et al).

Bunched
Implications
(O'Hearn & Pym)

Separation
Logic

The End.