An introduction to separation logic

James Brotherston

Programming Principles, Logic and Verification Group
Dept. of Computer Science
University College London, UK
J.Brotherston@ucl.ac.uk

Logic Summer School, ANU, 7 December 2015

Introduction

Verification of imperative programs is classically based on Hoare triples:

$$\{P\}\, C\, \{Q\}$$

where C is a program and P, Q are assertions in some logical language.

Introduction

Verification of imperative programs is classically based on Hoare triples:

$$\{P\}\,C\,\{Q\}$$

where C is a program and P, Q are assertions in some logical language.

These are read, roughly speaking, as

> *for any state σ satisfying P, if C transforms state σ to σ', then σ' satisfies Q.*

Introduction

Verification of imperative programs is classically based on Hoare triples:

$$\{P\}\,C\,\{Q\}$$

where C is a program and P, Q are assertions in some logical language.

These are read, roughly speaking, as

> *for any state σ satisfying P, if C transforms state σ to σ', then σ' satisfies Q.*

(with some wriggle room allowing us to deal with faulting or non-termination in various ways.)

Hoare-style verification

A Hoare-style program logic therefore relies on three main components:

Hoare-style verification

A Hoare-style program logic therefore relies on three main components:

1. a language of programs, and an operational semantics explaining how they transform states;

Hoare-style verification

A Hoare-style program logic therefore relies on three main components:

1. a language of programs, and an operational semantics explaining how they transform states;
2. a language of logical assertions, and a semantics explaining how to read them as true or false in a particular state;

Hoare-style verification

A Hoare-style program logic therefore relies on three main components:

1. a language of programs, and an operational semantics explaining how they transform states;
2. a language of logical assertions, and a semantics explaining how to read them as true or false in a particular state;
3. a formal interpretation of Hoare triples, together with (sound) proof rules for manipulating them.

Hoare-style verification

A Hoare-style program logic therefore relies on three main components:

1. a language of programs, and an operational semantics explaining how they transform states;
2. a language of logical assertions, and a semantics explaining how to read them as true or false in a particular state;
3. a formal interpretation of Hoare triples, together with (sound) proof rules for manipulating them.

We'll look at these informally first, then introduce a little more formal detail.

Programs, informally

We consider a standard `while` language with pointers, memory (de)allocation and recursive procedures.

Programs, informally

We consider a standard `while` language with pointers, memory (de)allocation and recursive procedures.E.g.:

```
deltree(*x) {
    if x=nil then return;
    else {
        l,r := x.left,x.right;
        deltree(l);
        deltree(r);
        free(x);
    }
}
```

Assertions, informally

Our assertion language lets us describe heap data structures such as linked lists and trees.

Assertions, informally

Our assertion language lets us describe heap data structures such as linked lists and trees.

E.g., binary trees with root pointer x can be defined by:

$$\begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & \mathsf{tree}(x) \\ x \neq \mathsf{nil} : x \mapsto (y, z) * \mathsf{tree}(y) * \mathsf{tree}(z) & \Rightarrow & \mathsf{tree}(x) \end{array}$$

Assertions, informally

Our assertion language lets us describe heap data structures such as linked lists and trees.

E.g., binary trees with root pointer x can be defined by:

$$\begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & \mathsf{tree}(x) \\ x \neq \mathsf{nil} : x \mapsto (y, z) * \mathsf{tree}(y) * \mathsf{tree}(z) & \Rightarrow & \mathsf{tree}(x) \end{array}$$

where

- emp denotes the empty heap;

Assertions, informally

Our assertion language lets us describe heap data structures such as linked lists and trees.

E.g., binary trees with root pointer x can be defined by:

$$\begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & \mathsf{tree}(x) \\ x \neq \mathsf{nil} : x \mapsto (y, z) * \mathsf{tree}(y) * \mathsf{tree}(z) & \Rightarrow & \mathsf{tree}(x) \end{array}$$

where

- emp denotes the empty heap;
- $x \mapsto (y, z)$ denotes a single pointer to a pair of data cells;

Assertions, informally

Our assertion language lets us describe heap data structures such as linked lists and trees.

E.g., binary trees with root pointer x can be defined by:

$$\begin{array}{rcl} x = \mathsf{nil} : \mathsf{emp} & \Rightarrow & \mathsf{tree}(x) \\ x \neq \mathsf{nil} : x \mapsto (y, z) * \mathsf{tree}(y) * \mathsf{tree}(z) & \Rightarrow & \mathsf{tree}(x) \end{array}$$

where

- emp denotes the empty heap;
- $x \mapsto (y, z)$ denotes a single pointer to a pair of data cells;
- $*$ means "and separately in memory".

An example proof

```
deltree(*x) {
    if x=nil then return;
    else {
        l,r := x.left,x.right;

        deltree(l);

        deltree(r);

        free(x);

    }
}
```

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;
    else {
        l,r := x.left,x.right;

        deltree(l);

        deltree(r);

        free(x);

    }
}
```

$\{\mathsf{emp}\}$

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;  {emp}
    else {
        l,r := x.left,x.right;

        deltree(l);

        deltree(r);

        free(x);

    }
}  {emp}
```

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;  {emp}
    else {    {x ↦ (y, z) * tree(y) * tree(z)}
        l,r := x.left,x.right;

        deltree(l);

        deltree(r);

        free(x);

    }
}  {emp}
```

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;
```
$\{\mathsf{emp}\}$

```
    else {
```
$\{x \mapsto (y, z) * \mathsf{tree}(y) * \mathsf{tree}(z)\}$

```
        l,r := x.left,x.right;
```
$\{x \mapsto (l, r) * \mathsf{tree}(l) * \mathsf{tree}(r)\}$

```
        deltree(l);

        deltree(r);

        free(x);

    }
}
```
$\{\mathsf{emp}\}$

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;
```
$\{\mathsf{emp}\}$
```
    else {
```
$\{x \mapsto (y, z) * \mathsf{tree}(y) * \mathsf{tree}(z)\}$
```
        l,r := x.left,x.right;
```
$\{x \mapsto (l, r) * \mathsf{tree}(l) * \mathsf{tree}(r)\}$
```
        deltree(l);
```
$\{x \mapsto (l, r) * \mathsf{emp} * \mathsf{tree}(r)\}$
```
        deltree(r);

        free(x);

    }
}
```
$\{\mathsf{emp}\}$

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;  {emp}
    else {   {x ↦ (y, z) ∗ tree(y) ∗ tree(z)}
        l,r := x.left,x.right;
        {x ↦ (l, r) ∗ tree(l) ∗ tree(r)}
        deltree(l);
        {x ↦ (l, r) ∗ emp ∗ tree(r)}
        deltree(r);
        {x ↦ (l, r) ∗ emp ∗ emp}
        free(x);

    }
}  {emp}
```

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;  {emp}
    else {   {x ↦ (y,z) * tree(y) * tree(z)}
        l,r := x.left,x.right;
        {x ↦ (l,r) * tree(l) * tree(r)}
        deltree(l);
        {x ↦ (l,r) * emp * tree(r)}
        deltree(r);
        {x ↦ (l,r) * emp * emp}
        free(x);
        {emp * emp * emp}
    }
}  {emp}
```

An example proof

$\{\mathsf{tree}(x)\}$

```
deltree(*x) {
    if x=nil then return;  {emp}
    else {    {x ↦ (y,z) * tree(y) * tree(z)}
        l,r := x.left,x.right;
        {x ↦ (l,r) * tree(l) * tree(r)}
        deltree(l);
        {x ↦ (l,r) * emp * tree(r)}
        deltree(r);
        {x ↦ (l,r) * emp * emp}
        free(x);
        {emp * emp * emp}
    }  {emp}
}  {emp}
```

Frame property

Consider the following step in the previous example:

$$\{x \mapsto (l, r) * \mathsf{tree}(l) * \mathsf{tree}(r)\}$$

```
deltree(l)
```

$$\{x \mapsto (l, r) * \mathsf{emp} * \mathsf{tree}(r)\}$$

Frame property

Consider the following step in the previous example:

$$\{x \mapsto (l, r) * \mathsf{tree}(l) * \mathsf{tree}(r)\}$$
`deltree(l)`
$$\{x \mapsto (l, r) * \mathsf{emp} * \mathsf{tree}(r)\}$$

Implicitly, this relies on a framing property, namely:

$$\frac{\{\mathsf{tree}(l)\}\ \texttt{deltree(l)}\ \{\mathsf{emp}\}}{\{x \mapsto (l, r) * \mathsf{tree}(l) * \mathsf{tree}(r)\}\ \texttt{deltree(l)}\ \{x \mapsto (l, r) * \mathsf{emp} * \mathsf{tree}(r)\}}$$

Classical failure of frame rule

The so-called frame rule,

$$\frac{\{P\}\,C\,\{Q\}}{\{F \wedge P\}\,C\,\{F \wedge Q\}}$$

is well known to fail in standard Hoare logic.

Classical failure of frame rule

The so-called frame rule,

$$\frac{\{P\}\,C\,\{Q\}}{\{F \wedge P\}\,C\,\{F \wedge Q\}}$$

is well known to fail in standard Hoare logic.E.g.,

$$\frac{\{x = 0\}\,x := 2\,\{x = 2\}}{\{y = 0 \wedge x = 0\}\,x := 2\,\{y = 0 \wedge x = 2\}}$$

is not valid (because y could alias x).

Classical failure of frame rule

The so-called frame rule,

$$\frac{\{P\}\,C\,\{Q\}}{\{F \wedge P\}\,C\,\{F \wedge Q\}}$$

is well known to fail in standard Hoare logic.E.g.,

$$\frac{\{x = 0\}\,x := 2\,\{x = 2\}}{\{y = 0 \wedge x = 0\}\,x := 2\,\{y = 0 \wedge x = 2\}}$$

is not valid (because y could alias x).

As we'll see, using the "separating conjunction" $*$ instead of $\wedge$ will however give us a valid frame rule.

Heap memory model

- We assume an infinite set Val of values of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are allocable locations; nil is a non-allocable value.

Heap memory model

- We assume an infinite set Val of values of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are allocable locations; nil is a non-allocable value.
- Stacks map variables to values, $s : \mathsf{Var} \to \mathsf{Val}$.

Heap memory model

- We assume an infinite set Val of values of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are allocable locations; nil is a non-allocable value.
- Stacks map variables to values, $s : \mathsf{Var} \to \mathsf{Val}$.
- Heaps map finitely many locations to values, $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. We write e for the empty heap (undefined on all locations).

Heap memory model

- We assume an infinite set Val of values of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are allocable locations; nil is a non-allocable value.
- Stacks map variables to values, $s : \mathsf{Var} \to \mathsf{Val}$.
- Heaps map finitely many locations to values, $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. We write e for the empty heap (undefined on all locations).
- Heap composition $h_1 \circ h_2$ is defined to be $h_1 \cup h_2$ if their domains are non-overlapping, and undefined otherwise.

Heap memory model

- We assume an infinite set Val of values of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are allocable locations; nil is a non-allocable value.
- Stacks map variables to values, $s : \mathsf{Var} \to \mathsf{Val}$.
- Heaps map finitely many locations to values, $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$. We write e for the empty heap (undefined on all locations).
- Heap composition $h_1 \circ h_2$ is defined to be $h_1 \cup h_2$ if their domains are non-overlapping, and undefined otherwise.
- A state is simply a stack paired with a heap, (s, h).

Program semantics

- A configuration is given by (C, s, h), where C is a program, and (s, h) a (stack-heap) state.

Program semantics

- A configuration is given by (C, s, h), where C is a program, and (s, h) a (stack-heap) state.
- C could be empty, in which case we call (C, s, h) final (and usually just write $\langle s, h \rangle$).

Program semantics

- A configuration is given by (C, s, h), where C is a program, and (s, h) a (stack-heap) state.
- C could be empty, in which case we call (C, s, h) final (and usually just write $\langle s, h \rangle$).
- *fault* is a special configuration used to catch memory errors.

Program semantics

- A configuration is given by (C, s, h), where C is a program, and (s, h) a (stack-heap) state.
- C could be empty, in which case we call (C, s, h) final (and usually just write $\langle s, h \rangle$).
- *fault* is a special configuration used to catch memory errors.
- The small-step semantics of programs is then given by a relation $\rightsquigarrow$ between configurations:

$$(C, s, h) \rightsquigarrow (C', s', h')$$

Semantics of assignment and (de)allocation

$$\frac{}{(x := E, s, h) \rightsquigarrow (s[x \mapsto [\![E]\!]s], h)}$$

$$\frac{[\![E]\!]s \in \mathsf{dom}(h)}{(x := E.f, s, h) \rightsquigarrow (s[x \mapsto h([\![E]\!]s).f], h)}$$

$$\frac{[\![E]\!]s \in \mathsf{dom}(h)}{(E.f := E', s, h) \rightsquigarrow (s, h[[\![E]\!]s.f \mapsto [\![E']\!]s])}$$

$$\frac{\ell \in \mathsf{Loc} \setminus \mathsf{dom}(h) \quad v \in \mathsf{Val}}{(E := \mathtt{new}(), s, h) \rightsquigarrow (s[x \mapsto \ell], h[\ell \mapsto v])}$$

$$\frac{[\![E]\!]s = \ell \in \mathsf{dom}(h)}{(\mathtt{free}(E), s, h) \rightsquigarrow (s, (h \upharpoonright (\mathsf{dom}(h) \setminus \{\ell\}))}$$

$$\frac{C \equiv x := E.f \mid E.f := E' \mid \mathtt{free}(E) \quad [\![E]\!]s \notin \mathsf{dom}(h)}{(C, s, h) \rightsquigarrow \mathit{fault}}$$

Symbolic-heap assertions

- Terms t are either variables $x, y, z \ldots$ or the constant nil.

Symbolic-heap assertions

- Terms t are either variables $x, y, z \ldots$ or the constant nil.
- Pure formulas π and spatial formulas F are given by:

$$
\begin{array}{lcl}
\pi & ::= & t = t \mid t \neq t \\
F & ::= & \mathsf{emp} \mid x \mapsto \mathbf{t} \mid P\mathbf{t} \mid F * F
\end{array}
$$

(where P a predicate symbol, $\mathbf{t}$ a tuple of terms).

Symbolic-heap assertions

- Terms t are either variables $x, y, z \ldots$ or the constant nil.
- Pure formulas π and spatial formulas F are given by:

$$\begin{array}{rcl} \pi & ::= & t = t \mid t \neq t \\ F & ::= & \mathsf{emp} \mid x \mapsto \mathbf{t} \mid P\mathbf{t} \mid F * F \end{array}$$

  (where P a predicate symbol, $\mathbf{t}$ a tuple of terms).
- A symbolic heap is $\exists \mathbf{x}.\ \Pi : F$, for Π a set of pure formulas.

Symbolic-heap assertions

- Terms t are either variables $x, y, z \ldots$ or the constant nil.
- Pure formulas π and spatial formulas F are given by:

$$\begin{array}{rcl} \pi & ::= & t = t \mid t \neq t \\ F & ::= & \mathsf{emp} \mid x \mapsto \mathbf{t} \mid P\mathbf{t} \mid F * F \end{array}$$

(where P a predicate symbol, $\mathbf{t}$ a tuple of terms).

- A symbolic heap is $\exists \mathbf{x}.\ \Pi : F$, for Π a set of pure formulas.
- The predicate symbols might come from a hard-coded set, or might be user-defined.

Semantics of assertions

We define the forcing relation $s, h \models A$:

Semantics of assertions

We define the forcing relation $s, h \models A$:

$$
\begin{array}{lcl}
s, h \models_\Phi t_1 = (\neq) t_2 & \Leftrightarrow & s(t_1) = (\neq) s(t_2) \\
s, h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models_\Phi x \mapsto \mathbf{t} & \Leftrightarrow & \mathsf{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(\mathbf{t}) \\
s, h \models_\Phi P\mathbf{t} & \Leftrightarrow & (s(\mathbf{t}), h) \in [\![P]\!] \\
s, h \models_\Phi F_1 * F_2 & \Leftrightarrow & \exists h_1, h_2.\ h = h_1 \circ h_2 \text{ and } s, h_1 \models_\Phi F_1 \\
 & & \text{and } s, h_2 \models_\Phi F_2 \\
s, h \models_\Phi \exists \mathbf{z}.\ \Pi : F & \Leftrightarrow & \exists \mathbf{v} \in \mathsf{Val}^{|\mathbf{z}|}.\ s[\mathbf{z} \mapsto \mathbf{v}], h \models_\Phi \pi \text{ for all} \\
 & & \pi \in \Pi \text{ and } s[\mathbf{z} \mapsto \mathbf{v}], h \models_\Phi F
\end{array}
$$

The semantics $[\![P]\!]$ of inductive predicate P has a standard construction (but outside the scope of this talk).

Interpretation of Hoare triples

Our interpretation of Hoare triples is almost standard, except we take a fault-avoiding interpretation:

Interpretation of Hoare triples

Our interpretation of Hoare triples is almost standard, except we take a fault-avoiding interpretation:

Definition

$\{P\}\, C\, \{Q\}$ is valid if, whenever $s, h \models P$,

1. $(C, s, h) \not\rightsquigarrow^* \mathit{fault}$ (i.e. is memory-safe), and

Interpretation of Hoare triples

Our interpretation of Hoare triples is almost standard, except we take a fault-avoiding interpretation:

Definition

$\{P\}\,C\,\{Q\}$ is valid if, whenever $s, h \models P$,

1. $(C, s, h) \not\rightsquigarrow^* \mathit{fault}$ (i.e. is memory-safe), and
2. if $(C, s, h) \rightsquigarrow^* (\epsilon, s, h)$, then $s, h \models Q$.

Interpretation of Hoare triples

Our interpretation of Hoare triples is almost standard, except we take a fault-avoiding interpretation:

Definition

$\{P\}\, C\, \{Q\}$ is valid if, whenever $s, h \models P$,

1. $(C, s, h) \not\rightsquigarrow^* \mathit{fault}$ (i.e. is memory-safe), and
2. if $(C, s, h) \rightsquigarrow^* (\epsilon, s, h)$, then $s, h \models Q$.

If we are interested in total correctness, simply replace the memory-safety condition above by (safe) termination: everything still works!

Axioms and proof rules for triples

$$\frac{}{\{\mathsf{emp}\}\, x := E\, \{x = E[x'/x] : \mathsf{emp}\}} \qquad \frac{}{\{E.f \mapsto \_\}\, E.f := E'\, \{E.f \mapsto E'\}}$$

$$\frac{}{\{E.f \mapsto t\}\, x := E.f\, \{x = t[x'/x] : E.f \mapsto t[x'/x]\}}$$

$$\frac{}{\{\mathsf{emp}\}\, x := \mathtt{new}()\, \{x \mapsto x'\}} \qquad \frac{}{\{E \mapsto \_\}\, \mathtt{free}(E)\, \{\mathsf{emp}\}}$$

$$\frac{\{P\}\, C_1\, \{R\} \quad \{R\}\, C_2\, \{Q\}}{\{P\}\, C_1; C_2\, \{Q\}} \qquad \frac{\{B : P\}\, C_1\, \{Q\} \quad \{\neg B : P\}\, C_2\, \{Q\}}{\{P\}\, \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\, \{Q\}}$$

(Note that $E.f \mapsto E'$ is a shorthand for $E \mapsto (\ldots, E', \ldots)$ where E' occurs at the fth position in the tuple.)

The frame rule

The general frame rule of separation logic can be stated as follows:

$$\frac{\{P\}\,C\,\{Q\}}{\{F * P\}\,C\,\{F * Q\}}$$

The frame rule

The general frame rule of separation logic can be stated as follows:

$$\frac{\{P\}\,C\,\{Q\}}{\{F * P\}\,C\,\{F * Q\}}$$

subject to the obvious sanity condition: C does not modify any variable mentioned in the "frame" F.

The frame rule

The general frame rule of separation logic can be stated as follows:

$$\frac{\{P\}\, C\, \{Q\}}{\{F * P\}\, C\, \{F * Q\}}$$

subject to the obvious sanity condition: C does not modify any variable mentioned in the "frame" F.

This rule is exactly what is needed to carry out proofs like the one we saw before for `deltree`.

Soundness of frame rule

Soundness of the frame rule depends on the following two facts about the programming language:

Soundness of frame rule

Soundness of the frame rule depends on the following two facts about the programming language:

Lemma (Safety monotonicity)

If $(C, s, h) \not\rightsquigarrow^* \mathit{fault}$ *then* $(C, s, h \circ h') \not\rightsquigarrow^* \mathit{fault}$ *(for any* h' *such that* $h \circ h'$ *is defined).*

Soundness of frame rule

Soundness of the frame rule depends on the following two facts about the programming language:

Lemma (Safety monotonicity)

If $(C, s, h) \not\rightsquigarrow^* fault$ *then* $(C, s, h \circ h') \not\rightsquigarrow^* fault$ *(for any* h' *such that* $h \circ h'$ *is defined).*

Lemma (Frame property)

Suppose $(C, s, h_1 \circ h_2) \rightsquigarrow^* \langle s, h \rangle$, *and that* $(C, s, h_1) \not\rightsquigarrow^* fault$. *Then there exists* h' *such that* $(C, s, h_1) \rightsquigarrow^* \langle s, h' \rangle$, *and, moreover,* $h = h' \circ h_2$.

Soundness of frame rule

Soundness of the frame rule depends on the following two facts about the programming language:

Lemma (Safety monotonicity)

If $(C, s, h) \not\rightsquigarrow^* \mathit{fault}$ *then* $(C, s, h \circ h') \not\rightsquigarrow^* \mathit{fault}$ *(for any* h' *such that* $h \circ h'$ *is defined).*

Lemma (Frame property)

Suppose $(C, s, h_1 \circ h_2) \rightsquigarrow^* \langle s, h \rangle$, *and that* $(C, s, h_1) \not\rightsquigarrow^* \mathit{fault}$. *Then there exists* h' *such that* $(C, s, h_1) \rightsquigarrow^* \langle s, h' \rangle$, *and, moreover,* $h = h' \circ h_2$.

Together, these lemmas imply the locality of all commands. N.B.: this is an operational fact about the programming language, and nothing at all to do with logic!

Closing remarks

- What we call separation logic is really a combination of
  - programming language,
  - assertion language
  - and rules for Hoare triples.

Closing remarks

- What we call separation logic is really a combination of
  - programming language,
  - assertion language
  - and rules for Hoare triples.

- The power of separation logic comes from compositionality: proofs of sub-programs can be combined into proofs of whole programs.

Closing remarks

- What we call separation logic is really a combination of
  - programming language,
  - assertion language
  - and rules for Hoare triples.
- The power of separation logic comes from compositionality: proofs of sub-programs can be combined into proofs of whole programs.
- Compositionality depends on the frame rule:

$$\frac{\{P\}\,C\,\{Q\}}{\{F * P\}\,C\,\{F * Q\}}$$

Closing remarks

- What we call separation logic is really a combination of
  - programming language,
  - assertion language
  - and rules for Hoare triples.
- The power of separation logic comes from compositionality: proofs of sub-programs can be combined into proofs of whole programs.
- Compositionality depends on the frame rule:

$$\frac{\{P\}\, C\, \{Q\}}{\{F * P\}\, C\, \{F * Q\}}$$

- And the soundness of the frame rule is essentially a reflection of the locality of commands.

Further reading

S. Ishtiaq and P. O'Hearn.
BI as an assertion language for mutable data structures.
In *Proc. POPL-28*. ACM, 2001.
(Winner of *Most Influential POPL Paper 2001* award.)

J.C. Reynolds.
Separation logic: A logic for shared mutable data structures.
In *Proc. LICS-17*. IEEE, 2002.

H. Yang and P. O'Hearn.
A semantic basis for local reasoning.
In *Proc. FoSSaCS-5*. Springer. 2002.

C. Calcagno, D. Distefano, P. O'Hearn and H. Yang.
Compositional shape analysis by means of bi-abduction.
In *Journal of the ACM* 58(6). ACM, 2011.
Original version in *Proc. POPL-36*. ACM, 2009.