Deductive verification

Software Analysis
Topic 4

Carlo A. Furia

USI – Università della Svizzera Italiana

Today's menu

Hoare logic

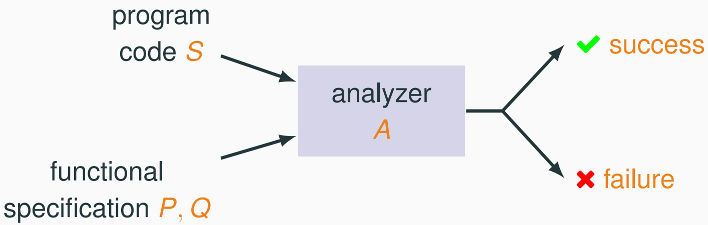
Predicate transformers and verification conditions

Supporting realistic program features

Tools and case studies

Separation logic

Deductive verification: the very idea



Deductive verification:

- analyzes real program code
- verifies arbitrarily complex properties
- properties are mainly functional (input/output)
- is normally sound but incomplete

Hoare logic

Hoare logic

The ultimate goal of deductive verification is expressing correctness as the validity of a logic formula.

To this end, we need a formal semantics that is declarative in style (as opposed to operational semantics).

Hoare logic

The ultimate goal of deductive verification is expressing correctness as the validity of a logic formula.

To this end, we need a formal semantics that is declarative in style (as opposed to operational semantics).

Hoare logic formalizes program statements by means of Hoare triples.

$$\{P\}\ S\ \{Q\}$$

Hoare logic

The ultimate goal of deductive verification is expressing correctness as the validity of a logic formula.

To this end, we need a formal semantics that is declarative in style (as opposed to operational semantics).

Hoare logic formalizes program statements by means of Hoare triples.

$$\text{precondition} \longrightarrow \{P\}\ \overset{\text{statement}}{\overset{\downarrow}{S}}\ \{Q\} \longleftarrow \text{postcondition}$$

Hoare logic

The ultimate goal of deductive verification is expressing correctness as the validity of a logic formula.

To this end, we need a formal semantics that is declarative in style (as opposed to operational semantics).

Hoare logic formalizes program statements by means of Hoare triples.

$$\text{precondition} \longrightarrow \{P\} \overset{\substack{\text{statement}\\ \downarrow}}{S} \{Q\} \longleftarrow \text{postcondition}$$

> $\{P\}\ S\ \{Q\}$ is valid if <u>executing</u> S in a state that satisfies P leads to a state that satisfies Q.

Hoare logic

The ultimate goal of deductive verification is expressing correctness as the validity of a logic formula.

To this end, we need a formal semantics that is declarative in style (as opposed to operational semantics).

Hoare logic formalizes program statements by means of Hoare triples.

statement

precondition $\{P\}\ S\ \{Q\}$ postcondition

> $\{P\}\ S\ \{Q\}$ is valid if <u>executing</u> S in a state that satisfies P leads to a state that satisfies Q.

the post-state the pre-state

Valid Hoare triples define code that is correct with respect to a functional pre/post specification.

Hoare-Floyd logic

Hoare logic is also called Hoare-Floyd logic, because it is the combination of fundamental contributions by Tony Hoare and Bob Floyd.

Robert W. Floyd

C. A. R. Hoare

Specification predicates

P and Q in a Hoare triple $\{P\}\ S\ \{Q\}$ are predicates that constrain program states.

Specification predicates

P and Q in a Hoare triple $\{P\}\ S\ \{Q\}$ are predicates that constrain program states.

Normally, P and Q are first-order formulas that may include:

- program variables (cannot be quantified)
- logic variables
- interpreted theory symbols (typically, arithmetic and other useful theories)

Specification predicates

P and Q in a Hoare triple $\{P\}\ S\ \{Q\}$ are predicates that constrain program states.

Normally, P and Q are first-order formulas that may include:

- program variables (cannot be quantified)
- logic variables
- interpreted theory symbols (typically, arithmetic and other useful theories)

An interpretation $\mathcal{M}$ of a specification predicate P is
an interpretation that includes a program state s:
an assignment of value to every program variable in P.

Therefore, we can equivalently view each predicate P as a set of program states – those corresponding to interpretations that satisfy P.

Some valid Hoare triples

Some valid Hoare triples

```
{ true } y := x { true }
```

Some valid Hoare triples

```
{ true } y := x { true }

{ false } y := x { y > 0 }
```

Some valid Hoare triples

```
{ true } y := x { true }

{ false } y := x { y > 0 }

{ x > 0 } y := x { y > 0 }
```

Some valid Hoare triples

```
{ true } y := x { true }

{ false } y := x { y > 0 }

{ x > 0 } y := x { y > 0 }

{ z = 3 } y := x { z = 3 }
```

Some valid Hoare triples

$\{ \texttt{true} \}\ y := x\ \{ \texttt{true} \}$

$\{ \texttt{false} \}\ y := x\ \{ y > 0 \}$

$\{ x > 0 \}\ y := x\ \{ y > 0 \}$

$\{ z = 3 \}\ y := x\ \{ z = 3 \}$

$\{ x > 0 \}\ x := x + 3\ \{ x > 3 \}$

Some valid Hoare triples

```
{ true } y := x { true }

{ false } y := x { y > 0 }

{ x > 0 } y := x { y > 0 }

{ z = 3 } y := x { z = 3 }

{ x > 0 } x := x + 3 { x > 3 }

{ x > 0 } y := x ; x := 4  { y > 0 ∧ x > 0}
```

Hoare logic

Axiomatic semantics

Axiomatic semantics

An axiomatic semantics is a declarative formal semantics of programs.

It consists of a series of inference rules for Hoare logic, which define the semantics of programs in terms of pre/post specifications.

Since the inference rules are axioms of the programming language theory, this style of semantics is called axiomatic.

Axiomatic semantics

An axiomatic semantics is a declarative formal semantics of programs.

It consists of a series of inference rules for Hoare logic, which define the semantics of programs in terms of pre/post specifications.

Since the inference rules are axioms of the programming language theory, this style of semantics is called axiomatic.

The program state:

$$s : \quad \mathit{Variables} \to \mathit{Values}$$

of the operational semantics is the same on which specification predicates are interpreted in the axiomatic semantics.

$s \models P$	predicate P holds in s
$\{s \mid s \models P\}$	set of states on which P holds

Axiomatic semantics of Helium

To define Helium's semantics axiomatically, we give sound inference rules that define valid Hoare triples for each program statement.

Axiomatic semantics of Helium

To define Helium's semantics axiomatically, we give sound inference rules that define valid Hoare triples for each program statement.

Executing **`skip`** does not affect the state:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

Axiomatic semantics of Helium

To define Helium's semantics axiomatically, we give sound inference rules that define valid Hoare triples for each program statement.

Executing **skip** does not affect the state:

$$\frac{}{\{P\}\ \mathbf{skip}\ \{P\}}$$

Hoare triples can naturally be composed:

$$\frac{\{P\}\ S_1\ \{P'\} \quad \{P'\}\ S_2\ \{Q\}}{\{P\}\ S_1\,;S_2\ \{Q\}}$$

Axiomatic semantics of Helium: assignments

$$\frac{\mathrm{v}_1, \ldots, \mathrm{v}_n \text{ all different}}{\{Q[\mathrm{v}_1 \mapsto E_1, \ldots, \mathrm{v}_n \mapsto E_n]\}\ \mathrm{v}_1, \ldots, \mathrm{v}_n := E_1, \ldots, E_n\ \{Q\}}$$

$Q[\mathrm{v}_1 \mapsto E_1, \ldots, \mathrm{v}_n \mapsto E_n]$ is called the backward substitution of predicate Q through the assignment $\mathrm{v}_1, \ldots, \mathrm{v}_n := E_1, \ldots, E_n$

The backward substitution is a syntactic rewrite, which does not evaluate expressions in any way.

Axiomatic semantics of Helium: assignments

$$\frac{\mathtt{v}_1, \ldots, \mathtt{v}_n \text{ all different}}{\{Q[\mathtt{v}_1 \mapsto E_1, \ldots, \mathtt{v}_n \mapsto E_n]\}\ \mathtt{v}_1, \ldots, \mathtt{v}_n := E_1, \ldots, E_n\ \{Q\}}$$

$Q[\mathtt{v}_1 \mapsto E_1, \ldots, \mathtt{v}_n \mapsto E_n]$ is called the backward substitution of predicate Q through the assignment $\mathtt{v}_1, \ldots, \mathtt{v}_n := E_1, \ldots, E_n$

The backward substitution is a syntactic rewrite, which does not evaluate expressions in any way.

The soundness of the backward substitution rule is somewhat unintuitive because it goes backward: to establish a certain property about x (postcondition), establish a certain property about E (precondition) and then assign E to x.

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
	x := y	{true}
	x := y	{false}
	x := y	$\{z = w + 1\}$
	x := y	$\{y = 3\}$
	x := y	$\{x = 4\}$
	x := x + 2	$\{x = 3\}$
	x, y := y, x	$\{x = 3 \land y = 4\}$
	x, y := x, x	$\{x > 3 \land y > 4\}$

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
{true}	x := y	{true}
	x := y	{false}
	x := y	$\{z = w + 1\}$
	x := y	$\{y = 3\}$
	x := y	$\{x = 4\}$
	x := x + 2	$\{x = 3\}$
	x, y := y, x	$\{x = 3 \wedge y = 4\}$
	x, y := x, x	$\{x > 3 \wedge y > 4\}$

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
{true}	x := y	{true}
{false}	x := y	{false}
	x := y	$\{z = w + 1\}$
	x := y	$\{y = 3\}$
	x := y	$\{x = 4\}$
	x := x + 2	$\{x = 3\}$
	x, y := y, x	$\{x = 3 \wedge y = 4\}$
	x, y := x, x	$\{x > 3 \wedge y > 4\}$

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
{true}	x := y	{true}
{false}	x := y	{false}
{z = w + 1}	x := y	{z = w + 1}
	x := y	{y = 3}
	x := y	{x = 4}
	x := x + 2	{x = 3}
	x, y := y, x	$\{x = 3 \land y = 4\}$
	x, y := x, x	$\{x > 3 \land y > 4\}$

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
$\{\texttt{true}\}$	`x := y`	$\{\texttt{true}\}$
$\{\texttt{false}\}$	`x := y`	$\{\texttt{false}\}$
$\{z = w + 1\}$	`x := y`	$\{z = w + 1\}$
$\{y = 3\}$	`x := y`	$\{y = 3\}$
	`x := y`	$\{x = 4\}$
	`x := x + 2`	$\{x = 3\}$
	`x, y := y, x`	$\{x = 3 \wedge y = 4\}$
	`x, y := x, x`	$\{x > 3 \wedge y > 4\}$

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
$\{\text{true}\}$	x := y	$\{\text{true}\}$
$\{\text{false}\}$	x := y	$\{\text{false}\}$
$\{z = w + 1\}$	x := y	$\{z = w + 1\}$
$\{y = 3\}$	x := y	$\{y = 3\}$
$\{y = 4\}$	x := y	$\{x = 4\}$
	x := x + 2	$\{x = 3\}$
	x, y := y, x	$\{x = 3 \wedge y = 4\}$
	x, y := x, x	$\{x > 3 \wedge y > 4\}$

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
$\{\texttt{true}\}$	`x := y`	$\{\texttt{true}\}$
$\{\texttt{false}\}$	`x := y`	$\{\texttt{false}\}$
$\{z = w + 1\}$	`x := y`	$\{z = w + 1\}$
$\{y = 3\}$	`x := y`	$\{y = 3\}$
$\{y = 4\}$	`x := y`	$\{x = 4\}$
$\{x + 2 = 3\}$	`x := x + 2`	$\{x = 3\}$
	`x, y := y, x`	$\{x = 3 \wedge y = 4\}$
	`x, y := x, x`	$\{x > 3 \wedge y > 4\}$

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
{true}	x := y	{true}
{false}	x := y	{false}
{z = w + 1}	x := y	{z = w + 1}
{y = 3}	x := y	{y = 3}
{y = 4}	x := y	{x = 4}
{x + 2 = 3}	x := x + 2	{x = 3}
{y = 3 ∧ x = 4}	x, y := y, x	{x = 3 ∧ y = 4}
	x, y := x, x	{x > 3 ∧ y > 4}

Backward substitution: examples

BACKWARD SUBSTUTION	ASSIGNMENT	POSTCONDITION
{true}	x := y	{true}
{false}	x := y	{false}
$\{z = w + 1\}$	x := y	$\{z = w + 1\}$
$\{y = 3\}$	x := y	$\{y = 3\}$
$\{y = 4\}$	x := y	$\{x = 4\}$
$\{x + 2 = 3\}$	x := x + 2	$\{x = 3\}$
$\{y = 3 \wedge x = 4\}$	x, y := y, x	$\{x = 3 \wedge y = 4\}$
$\{x > 3 \wedge x > 4\}$	x, y := x, x	$\{x > 3 \wedge y > 4\}$

Rules of consequence

In order to combine different Hoare triples we also need rules to reason about related predicates. These are logic rules that do not depend on the specific programming language we are dealing with.

Rules of consequence

In order to combine different Hoare triples we also need rules to reason about related predicates. These are logic rules that do not depend on the specific programming language we are dealing with.

Strengthening the precondition and weakening the postcondition does not affect validity.

$$\frac{\{P'\}\ S\ \{Q\} \quad P \Longrightarrow P'}{\{P\}\ S\ \{Q\}} \qquad \frac{\{P\}\ S\ \{Q'\} \quad Q' \Longrightarrow Q}{\{P\}\ S\ \{Q\}}$$

Some valid Hoare triples: validity proofs

We can use the rules seen so far to prove the validity of Hoare triples.

Some valid Hoare triples: validity proofs

We can use the rules seen so far to prove the validity of Hoare triples.

$$\frac{}{\{\top\}\ \mathtt{y\ :=\ x}\ \{\top\}}$$

Some valid Hoare triples: validity proofs

We can use the rules seen so far to prove the validity of Hoare triples.

$$\frac{}{\{\top\}\ \texttt{y := x}\ \{\top\}}$$

$$\frac{\dfrac{}{\{x > 0\}\ \texttt{y := x}\ \{y > 0\}} \quad \bot \Longrightarrow x > 0}{\{\bot\}\ \texttt{y := x}\ \{y > 0\}}$$

Some valid Hoare triples: validity proofs

We can use the rules seen so far to prove the validity of Hoare triples.

$$\frac{}{\{\top\}\ \mathtt{y\ :=\ x}\ \{\top\}}$$

$$\frac{\dfrac{}{\{\mathsf{x} > 0\}\ \mathtt{y\ :=\ x}\ \{\mathsf{y} > 0\}} \quad \bot \Longrightarrow \mathsf{x} > 0}{\{\bot\}\ \mathtt{y\ :=\ x}\ \{\mathsf{y} > 0\}}$$

$$\frac{}{\{\mathsf{x} > 0\}\ \mathtt{y\ :=\ x}\ \{\mathsf{y} > 0\}}$$

Some valid Hoare triples: validity proofs

We can use the rules seen so far to prove the validity of Hoare triples.

$$\frac{}{\{\top\}\ \mathtt{y\ :=\ x}\ \{\top\}}$$

$$\frac{\dfrac{}{\{x > 0\}\ \mathtt{y\ :=\ x}\ \{y > 0\}} \quad \bot \Longrightarrow x > 0}{\{\bot\}\ \mathtt{y\ :=\ x}\ \{y > 0\}}$$

$$\frac{}{\{x > 0\}\ \mathtt{y\ :=\ x}\ \{y > 0\}}$$

$$\frac{}{\{z = 3\}\ \mathtt{y\ :=\ x}\ \{z = 3\}}$$

Some valid Hoare triples: validity proofs

$$\dfrac{\dfrac{}{\{x + 3 > 3\}\ \texttt{x := x + 3}\ \{x > 3\}} \quad x > 0 \Longrightarrow x + 3 > 3}{\{x > 0\}\ \texttt{x := x + 3}\ \{x > 3\}}$$

Some valid Hoare triples: validity proofs

$$\frac{\overline{\{x+3>3\}\ \texttt{x := x + 3}\ \{x>3\}} \qquad x>0 \Longrightarrow x+3>3}{\{x>0\}\ \texttt{x := x + 3}\ \{x>3\}}$$

$$\frac{\overline{\{x>0\}\ \texttt{y := x}\ \{y>0\}} \qquad \dfrac{\overline{\{y>0 \land 4>0\}\ \texttt{x := 4}\ \{y>0 \land x>0\}} \qquad y>0 \Longrightarrow y>0 \land 4>0}{\{y>0\}\ \texttt{x := 4}\ \{y>0 \land x>0\}}}{\{x>0\}\ \texttt{y := x ; x := 4}\ \{y>0 \land x>0\}}$$

Axiomatic semantics of Helium: conditionals

A conditional requires complementary conditions to hold for the then and else branches:

$$\frac{\{P \land C\}\ T\ \{Q\} \quad \{P \land \neg C\}\ E\ \{Q\}}{\{P\}\ \texttt{if}\ C\ T\ \texttt{else}\ E\ \{Q\}}$$

Axiomatic semantics of Helium: conditionals

A conditional requires complementary conditions to hold for the then and else branches:

$$\frac{\{P \land C\}\ T\ \{Q\} \quad \{P \land \neg C\}\ E\ \{Q\}}{\{P\}\ \texttt{if}\ C\ T\ \texttt{else}\ E\ \{Q\}}$$

Conditionals without **else** reduce to conditional with empty **else**:

$$\frac{\{P\}\ \texttt{if}\ C\ T\ \texttt{else skip}\ \{Q\}}{\{P\}\ \texttt{if}\ C\ T\ \{Q\}}$$

Correctness proofs of maximum

Using the rule for conditionals, we can prove the correctness of our Helium program computing the maximum of two variables.

Correctness proofs of maximum

Using the rule for conditionals, we can prove the correctness of our Helium program computing the maximum of two variables.

First of all let's write a specification:

$$\{\text{precondition}\}$$

```
if (x > y) max := x else max := y
```

$$\{\text{postcondition}\}$$

Correctness proofs of maximum

Using the rule for conditionals, we can prove the correctness of our Helium program computing the maximum of two variables.

First of all let's write a specification:

$$\{\texttt{true}\}$$
$$\textbf{if}\ (\texttt{x} > \texttt{y})\ \texttt{max := x}\ \textbf{else}\ \texttt{max := y}$$
$$\{\text{postcondition}\}$$

Correctness proofs of maximum

Using the rule for conditionals, we can prove the correctness of our Helium program computing the maximum of two variables.

First of all let's write a specification:

$$\{\texttt{true}\}$$

$$\texttt{if (x > y) max := x else max := y}$$

$$\left\{ \begin{array}{c} (\texttt{x} \geq \texttt{y} \Longrightarrow \texttt{max} = \texttt{x}) \\ \wedge\, (\texttt{x} \leq \texttt{y} \Longrightarrow \texttt{max} = \texttt{y}) \end{array} \right\}$$

Correctness proofs of maximum

Applying the inference rule for conditionals splits the proof into two branches.

$$\frac{\{x > y\}\ \texttt{max := x} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\} \quad \{x \leq y\}\ \texttt{max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}{\{\texttt{true}\}\ \texttt{if (x > y) max := x else max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}$$

Correctness proofs of maximum

Applying the inference rule for conditionals splits the proof into two branches.

$$\frac{\{x > y\}\ \texttt{max := x} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\} \quad \{x \leq y\}\ \texttt{max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}{\{\texttt{true}\}\ \textbf{if}\ (x > y)\ \texttt{max := x}\ \textbf{else}\ \texttt{max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}$$

The left branch is proved by backward substitution followed by a series of simplifications.

$$\frac{\dfrac{}{\left\{ \begin{array}{c} (x \geq y \Longrightarrow x = x) \\ \wedge\, (x \leq y \Longrightarrow x = y) \end{array} \right\} \texttt{max := x} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}{\{x \geq y\}\ \texttt{max := x} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}} \qquad x > y \Longrightarrow x \geq y}{\{x > y\}\ \texttt{max := x} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\, (x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}$$

Correctness proofs of maximum

Applying the inference rule for conditionals splits the proof into two branches.

$$\frac{\{x > y\}\ \texttt{max := x} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\,(x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\} \quad \{x \leq y\}\ \texttt{max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\,(x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}{\{\texttt{true}\}\ \textbf{if}\ (x > y)\ \texttt{max := x}\ \textbf{else}\ \texttt{max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\,(x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}$$

The right branch is similar but slightly simpler.

$$\frac{\overline{\left\{ \begin{array}{c} (x \geq y \Longrightarrow y = x) \\ \wedge\,(x \leq y \Longrightarrow y = y) \end{array} \right\} \texttt{max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\,(x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}}{\{x \leq y\}\ \texttt{max := y} \left\{ \begin{array}{c} (x \geq y \Longrightarrow \texttt{max} = x) \\ \wedge\,(x \leq y \Longrightarrow \texttt{max} = y) \end{array} \right\}}$$

Axiomatic semantics of Helium: loops

A loop repeats until its condition becomes false:

$$\frac{\{J \wedge C\}\ B\ \{J\}}{\{J\}\ \texttt{while}\ C\ B\ \{J \wedge \neg C\}}$$

Predicate J is the loop invariant. It is an inductive predicate that is maintained by iterations of the loop.

A simple program with loops

Let's prove the correctness of this program:

```
var x, n: Integer
x := 0
while x < n
  x := x + 1
```

A simple program with loops

Let's prove the correctness of this program:

```
var x, n: Integer
x := 0
while x < n
  x := x + 1
```

First of all let's write a specification (as usual we omit variable declarations for simplicity):

{precondition}

`x := 0 ; while (x < n) x := x + 1`

{postcondition}

A simple program with loops

Let's prove the correctness of this program:

```
var x, n: Integer
x := 0
while x < n
  x := x + 1
```

First of all let's write a specification (as usual we omit variable declarations for simplicity):

$$\{n \geq 0\}$$

`x := 0 ; while (x < n) x := x + 1`

$$\{\text{postcondition}\}$$

A simple program with loops

Let's prove the correctness of this program:

```
var x, n: Integer
x := 0
while x < n
  x := x + 1
```

First of all let's write a specification (as usual we omit variable declarations for simplicity):

$$\{n \geq 0\}$$
$$\texttt{x := 0 ; while (x < n) x := x + 1}$$
$$\{x = n\}$$

A simple program with loops

Let's prove the correctness of this program:

```
var x, n: Integer
x := 0
while x < n
  x := x + 1
```

First of all let's write a specification (as usual we omit variable declarations for simplicity):

$$\{n \geq 0\}$$
$$\texttt{x := 0 ; while (x < n) x := x + 1}$$
$$\{x = n\}$$

As loop invariant we will use:

$$J \quad = \quad 0 \leq x \leq n$$

Proving a simple program with loops

We prove the following lemma using the inference rule for loops:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{0 \leq x + 1 \leq n\}\ \texttt{x := x + 1}\ \{0 \leq x \leq n\}}}{\{-1 \leq x \leq n - 1\}\ \texttt{x := x + 1}\ \{0 \leq x \leq n\}} \quad 0 \leq x < n \Longrightarrow -1 \leq x \leq n - 1}{\{0 \leq x < n\}\ \texttt{x := x + 1}\ \{0 \leq x \leq n\}}}{\{0 \leq x \leq n \land x < n\}\ \texttt{x := x + 1}\ \{0 \leq x \leq n\}}}{\{0 \leq x \leq n\}\ \textbf{while}\ \texttt{(x < n) x := x + 1}\ \{0 \leq x \leq n \land \neg(x < n)\}}
$$

Then we use other rules to complete the proof:

$$
\dfrac{\dfrac{\dfrac{}{\{n \geq 0 = 0\}\ \texttt{x := 0}\ \{n \geq 0 = x\}}}{\{n \geq 0\}\ \texttt{x := 0}\ \{n \geq 0 = x\}} \quad \dfrac{\dfrac{\dfrac{}{\{0 \leq x \leq n\}\ \textbf{while}\ \ldots\ \{0 \leq x \leq n \land \neg(x < n)\}}}{\{0 \leq x \leq n\}\ \textbf{while}\ \ldots\ \{x = n\}} \quad n \geq 0 = x \Longrightarrow 0 \leq x \leq n}{\{n \geq 0 = x\}\ \textbf{while}\ \ldots\ \{x = n\}}}{\{n \geq 0\}\ \texttt{x := 0 ;}\ \textbf{while}\ \ldots\ \{x = n\}}
$$

Finding loop invariants

Finding suitable loop invariants is one of the hardest part of correctness proofs – and it cannot be completely automated.

Here are some heuristics useful to discover useful loop invariants:

Finding loop invariants

Finding suitable loop invariants is one of the hardest part of correctness proofs – and it cannot be completely automated.

Here are some heuristics useful to discover useful loop invariants:

- A loop invariant J is an inductive predicate:

Finding loop invariants

Finding suitable loop invariants is one of the hardest part of correctness proofs – and it cannot be completely automated.

Here are some heuristics useful to discover useful loop invariants:

- A loop invariant J is an inductive predicate:

  initiation: J holds initially (just before the loop)

Finding loop invariants

Finding suitable loop invariants is one of the hardest part of correctness proofs – and it cannot be completely automated.

Here are some heuristics useful to discover useful loop invariants:

- A loop invariant J is an inductive predicate:

  initiation: J holds initially (just before the loop)

  consecution: the loop body must preserve J

Finding loop invariants

Finding suitable loop invariants is one of the hardest part of correctness proofs – and it cannot be completely automated.

Here are some heuristics useful to discover useful loop invariants:

- A loop invariant J is an inductive predicate:

  initiation: J holds initially (just before the loop)

  consecution: the loop body must preserve J

- A useful loop invariant J is related to the postcondition: $J \land \neg C$ should implies the specification predicate just after the loop

Finding loop invariants

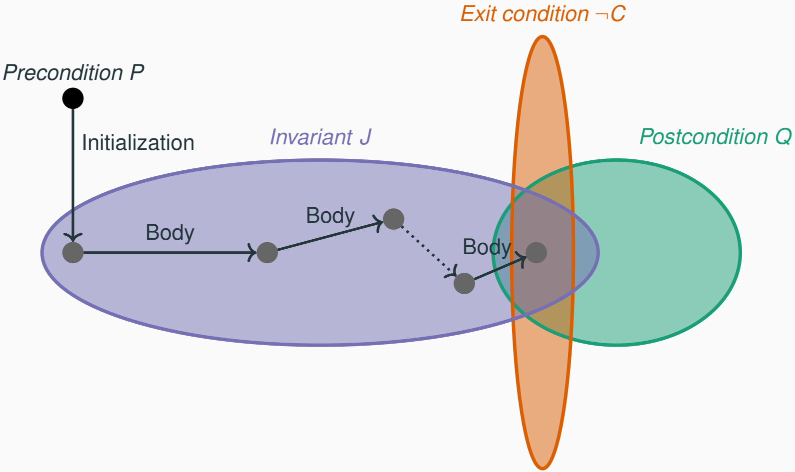
Finding suitable loop invariants is one of the hardest part of correctness proofs – and it cannot be completely automated.

Here are some heuristics useful to discover useful loop invariants:

- A loop invariant J is an inductive predicate:
  initiation: J holds initially (just before the loop)
  consecution: the loop body must preserve J
- A useful loop invariant J is related to the postcondition: $J \wedge \neg C$ should implies the specification predicate just after the loop
- A loop invariant J is an abstract specification of what the loop does: J describes what the loop has done and what remains to be done

Loops as successive approximations

$\{P\}$ Initialization $\{J\}$ **while** C Body $\{Q\}$



Loop invariant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

Loop invariant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

What the loop does:

- increments `n` from the initial value 0 until it is `b`
- multiplies `a` by itself in `pow`

Loop invariant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

What the loop does:

- increments `n` from the initial value 0 until it is `b`
- multiplies `a` by itself in `pow`

How the loop establishes the postcondition:

- after the *k*th iteration, `pow` is a^k
- when the loop terminates, the loop body has executed `b` times

Loop invariant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

What the loop does:

- increments `n` from the initial value 0 until it is `b`
- multiplies `a` by itself in `pow`

How the loop establishes the postcondition:

- after the kth iteration, `pow` is a^k
- when the loop terminates, the loop body has executed `b` times

How variables are related:

- after the kth iteration, $n = k$

Loop invariant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

$J = (0 \leq n \leq b) \land (pow = a^n)$

What the loop does:

- increments n from the initial value 0 until it is b
- multiplies a by itself in pow

How the loop establishes the postcondition:

- after the kth iteration, pow is a^k
- when the loop terminates, the loop body has executed b times

How variables are related:

- after the kth iteration, $n = k$

Loop invariant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

$$J = (0 \le \texttt{n} \le \texttt{b}) \land (\texttt{pow} = \texttt{a}^{\texttt{n}})$$

- J holds initially
- J is invariant (preserved by a loop iteration)
- $J \land \neg(\texttt{n} < \texttt{b})$ implies the postcondition

What the loop does:

- increments `n` from the initial value 0 until it is `b`
- multiplies `a` by itself in `pow`

How the loop establishes the postcondition:

- after the kth iteration, `pow` is $\texttt{a}^k$
- when the loop terminates, the loop body has executed `b` times

How variables are related:

- after the kth iteration, $\texttt{n} = k$

Loop invariant of power (second version)

```
{ n = 1 ∧ pow = 1 }
while n ≤ b
  pow := pow ∗ a
  n := n + 1
{ pow = a^b }
```

What the loop does:

- increments n from the initial value 1 until it is $b + 1$
- multiplies a by itself in pow

How the loop establishes the postcondition:

- after the kth iteration, pow is a^k
- when the loop terminates, the loop body has executed b times

How variables are related:

- after the kth iteration, $n = k + 1$; hence $k = n - 1$

Loop invariant of power (second version)

$\{ \texttt{n} = 1 \wedge \texttt{pow} = 1 \}$

while $\texttt{n} \leq \texttt{b}$

  $\texttt{pow} := \texttt{pow} * \texttt{a}$

  $\texttt{n} := \texttt{n} + 1$

$\{ \texttt{pow} = \texttt{a}^{\texttt{b}} \}$

$$J = (1 \leq \mathtt{n} \leq \mathtt{b} + 1) \wedge (\mathtt{pow} = \mathtt{a}^{\mathtt{n}-1})$$

What the loop does:

- increments n from the initial value 1 until it is $\mathtt{b} + 1$
- multiplies a by itself in pow

How the loop establishes the postcondition:

- after the kth iteration, pow is $\mathtt{a}^k$
- when the loop terminates, the loop body has executed b times

How variables are related:

- after the kth iteration, $\mathtt{n} = k + 1$; hence $k = \mathtt{n} - 1$

Loop invariant of power (third version)

```
{ n = b ∧ pow = 1 }
while n > 0
  pow := pow * a
  n := n - 1
{ pow = a^b }
```

What the loop does:

- decrements `n` from the initial value `b` until it is 0
- multiplies `a` by itself in `pow`

How the loop establishes the postcondition:

- after the kth iteration, `pow` is a^k
- when the loop terminates, the loop body has executed `b` times

How variables are related:

- after the kth iteration, $n = b - k$; hence $k = b - n$

Loop invariant of power (third version)

```
{ n = b ∧ pow = 1 }
while n > 0
  pow := pow * a
  n := n - 1
{ pow = a^b }
```

$J = (0 \le \texttt{n} \le \texttt{b}) \land (\texttt{pow} = \texttt{a}^{\texttt{b}-\texttt{n}})$

What the loop does:

- decrements n from the initial value b until it is 0
- multiplies a by itself in pow

How the loop establishes the postcondition:

- after the kth iteration, pow is $\texttt{a}^k$
- when the loop terminates, the loop body has executed b times

How variables are related:

- after the kth iteration, $\texttt{n} = \texttt{b} - k$; hence $k = \texttt{b} - \texttt{n}$

Specification splitting

Specification conjunction and disjunction rules are useful to split proofs according to the propositional structure of specification.

$$\frac{\{P_1\}\ S\ \{Q_1\} \quad \{P_2\}\ S\ \{Q_2\}}{\{P_1 \wedge P_2\}\ S\ \{Q_1 \wedge Q_2\}} \qquad \frac{\{P_1\}\ S\ \{Q_1\} \quad \{P_2\}\ S\ \{Q_2\}}{\{P_1 \vee P_2\}\ S\ \{Q_1 \vee Q_2\}}$$

Specification splitting

Specification conjunction and disjunction rules are useful to split proofs according to the propositional structure of specification.

$$\frac{\{P_1\}\ S\ \{Q_1\} \quad \{P_2\}\ S\ \{Q_2\}}{\{P_1 \wedge P_2\}\ S\ \{Q_1 \wedge Q_2\}} \qquad \frac{\{P_1\}\ S\ \{Q_1\} \quad \{P_2\}\ S\ \{Q_2\}}{\{P_1 \vee P_2\}\ S\ \{Q_1 \vee Q_2\}}$$

Specification splitting rules are not needed, but it's useful to apply them directly.

Rule of constancy

The rule of constancy is useful to compose specifications involving different program variables:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \wedge R\}\ S\ \{Q \wedge R\}}$$

Rule of constancy

The rule of constancy is useful to compose specifications involving different program variables:

R doesn't mention any (program/free) variable in $\mathcal{F}(S)$

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \land R\}\ S\ \{Q \land R\}}$$

$\mathcal{F}(S)$ is the frame of S: the set of all variables that S may modify. In Helium, this is just the set of all variables appearing in the left-hand side of an assignment.

Rule of constancy

The rule of constancy is useful to compose specifications involving different program variables:

R doesn't mention any (program/free) variable in $\mathcal{F}(S)$

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \wedge R\}\ S\ \{Q \wedge R\}}$$

$\mathcal{F}(S)$ is the frame of S: the set of all variables that S may modify. In Helium, this is just the set of all variables appearing in the left-hand side of an assignment.

The rule of constancy is derivable from the other rules, but it's useful to apply it directly – and will have an important role when we switch to languages that are more expressive than Helium.

Hoare logic

Soundness and completeness

Soundness and completeness of axiomatic semantics

The axiomatic semantics is a first-order theory using the notation of Hoare logic.

We can check whether the inference rules we have introduced are:

sound: $\vdash \{P\}S\{Q\}$ implies $\models \{P\}S\{Q\}$

complete: $\models \{P\}S\{Q\}$ implies $\vdash \{P\}S\{Q\}$

$\vdash \{P\}S\{Q\}$**:** we can prove $\{P\}S\{Q\}$ using the inference rules of <u>axiomatic semantics</u>

$\models \{P\}S\{Q\}$**:** $\{P\}S\{Q\}$ is valid (for example using the operational semantics to express satisfiability)

Soundness of axiomatic semantics

sound: $\vdash \{P\}S\{Q\}$ implies $\models \{P\}S\{Q\}$

We could prove soundness by showing the soundness of each inference rule.

For example, a proof sketch for the (single) assignment rule:

$$\frac{}{\{Q[\mathtt{v} \mapsto E]\}\ \mathtt{v}\ \mathtt{:=}\ E\ \{Q\}} \qquad \frac{[\![E]\!]_s = e}{\langle \mathtt{v}\ \mathtt{:=}\ E\ ,\ s\rangle \rightsquigarrow s[\mathtt{v} \mapsto e]}$$

1. Assume that $Q[v \mapsto E]$ holds in the pre-state s: $[\![Q[v \mapsto E]]\!]_s = \top$
2. Evaluation is idempotent: $[\![Q[v \mapsto E]]\!]_s \Longleftrightarrow [\![Q[v \mapsto [\![E]\!]_s]]\!]_s$
3. According to the operational semantics, the post-state s' is $s[v \mapsto [\![E]\!]_s]$
4. $[\![Q]\!]_{s'} \Longleftrightarrow [\![Q[v \mapsto [\![E]\!]_s]]\!]_s$ because the assignment doesn't change any state component other than v
5. Thus, $[\![Q]\!]_{s'} = \top$: Q holds in the post-state

Completeness of axiomatic semantics

complete: $\models \{P\}S\{Q\}$ implies $\vdash \{P\}S\{Q\}$

What we really want is syntactic completeness:

complete: $\mathcal{A} \models \{P\}S\{Q\}$ implies $\mathcal{A} \vdash \{P\}S\{Q\}$

where $\mathcal{A}$ are the axioms of axiomatic semantics.

Completeness of axiomatic semantics

complete: $\models \{P\}S\{Q\}$ implies $\vdash \{P\}S\{Q\}$

What we really want is syntactic completeness:

complete: $\mathcal{A} \models \{P\}S\{Q\}$ implies $\mathcal{A} \vdash \{P\}S\{Q\}$

where $\mathcal{A}$ are the axioms of axiomatic semantics.

Since axiomatic semantics includes arithmetic, it cannot be syntactically complete because of Gödel's incompleteness theorem.

Relative completeness of axiomatic semantics

Relative completeness: a theory with axioms T is complete relative to arithmetic if $T \models F$ implies $\widetilde{A} \cup T \vdash F$

where $\widetilde{A}$ is the set of all valid sentences of arithmetic taken as axioms.

Relative completeness of axiomatic semantics

Relative completeness: a theory with axioms T is complete relative to arithmetic if $T \models F$ implies $\widetilde{A} \cup T \vdash F$

where $\widetilde{A}$ is the set of all valid sentences of arithmetic taken as axioms.

Paul “Cookie” Cook

In 1978, Cook proved that Hoare logic is relatively complete.

$$\mathcal{A} \models \{P\}S\{Q\}$$
implies
$$\widetilde{A} \cup \mathcal{A} \vdash \{P\}S\{Q\}$$

Relative completeness of axiomatic semantics

Relative completeness: a theory with axioms T is complete relative to arithmetic if $T \models F$ implies $\widetilde{A} \cup T \vdash F$

where $\widetilde{A}$ is the set of all valid sentences of arithmetic taken as axioms.

Stephen A. Cook

In 1978, Cook proved that Hoare logic is relatively complete.

$$\mathcal{A} \models \{P\}S\{Q\}$$

implies

$$\widetilde{A} \cup \mathcal{A} \vdash \{P\}S\{Q\}$$

Hoare logic

Termination

Partial vs. total correctness

Is a program that does not terminate correct?

$$\{P\}\ S\ \{Q\}$$

partial correctness: if the execution of S in a state that satisfies P terminates, it terminates in a state that satisfies Q

total correctness: the execution of S in a state that satisfies P terminates in a state that satisfies Q

TOTAL CORRECTNESS $=$ PARTIAL CORRECTNESS $+$ TERMINATION

Partial vs. total correctness

Is a program that does not terminate correct?

$$\{P\}\ S\ \{Q\}$$

partial correctness: if the execution of S in a state that satisfies P terminates, it terminates in a state that satisfies Q

total correctness: the execution of S in a state that satisfies P terminates in a state that satisfies Q

TOTAL CORRECTNESS $=$ PARTIAL CORRECTNESS $+$ TERMINATION

The axiomatic semantics rules we have seen so far are sound for partial and total correctness, with the exception of the rule for loops.

Partial correctness of loops

Loops are the only statement that may not terminate.

The inference rule we used to prove partial correctness is unsound for total correctness:

$$\dfrac{\dfrac{}{\{\top \wedge \top\}\ \textbf{skip}\ \{\top\}}}{\dfrac{\{\top\}\ \textbf{while}\ \texttt{true}\ \textbf{skip}\ \{\top \wedge \neg\top\}}{\{\top\}\ \textbf{while}\ \texttt{true}\ \textbf{skip}\ \{\bot\}}}$$

`while true skip` is not totally correct because it doesn't terminate!

Proving termination

A sound inference rule uses a variant V (also: ranking function) to show progress:

$$\frac{\{J \land C \land V = v\}\; B \;\{J \land V < v\} \qquad J \land C \Longrightarrow V \geq 0}{\{J\}\; \texttt{while}\; C\; B\; \{J \land \neg C\}}$$

where v is a fresh variable that denotes the value of the variant V before each iteration.

Proving termination

A sound inference rule uses a variant V (also: ranking function) to show progress:

$$\frac{\{J \land C \land V = v\}\; B\; \{J \land V < v\} \quad J \land C \Longrightarrow V \geq 0}{\{J\}\; \mathtt{while}\; C\; B\; \{J \land \neg C\}}$$

where v is a fresh variable that denotes the value of the variant V before each iteration.

The variant V is:

- an integer expression
- always nonnegative while the loop iterates
- decreased in every loop iteration

Proving termination

A sound inference rule uses a variant V (also: ranking function) to show progress:

$$\frac{\{J \land C \land V = v\}\ B\ \{J \land V < v\} \quad J \land C \Longrightarrow V \geq 0}{\{J\}\ \texttt{while}\ C\ B\ \{J \land \neg C\}}$$

where v is a fresh variable that denotes the value of the variant V before each iteration.

The variant V is:

- an integer expression
- always nonnegative while the loop iterates
- decreased in every loop iteration

Since the loop decreases V without making it negative, the loop must perform only finitely many iterations after which it terminates (upon V reaching its minimum value).

Proving termination of a simple program with loops

To prove termination of the simple loop that increments x we can use the variant:

$$V \quad = \quad \mathtt{n} - \mathtt{x}$$

$$\dfrac{\begin{array}{c}\{0 \leq \mathtt{x} < \mathtt{n} \wedge \mathtt{n} - \mathtt{x} = v\} \\ \mathtt{x\ :=\ x\ +\ 1} \\ \{0 \leq \mathtt{x} \leq \mathtt{n} \wedge \mathtt{n} - \mathtt{x} < v\}\end{array} \qquad 0 \leq \mathtt{x} < \mathtt{n} \Longrightarrow \mathtt{n} - \mathtt{x} \geq 0}{\{0 \leq \mathtt{x} \leq \mathtt{n}\}\ \mathtt{while\ (x\ <\ n)\ x\ :=\ x\ +\ 1}\ \{0 \leq \mathtt{x} \leq \mathtt{n} \wedge \neg(\mathtt{x} < \mathtt{n})\}}$$

Proving termination of a simple program with loops

To prove termination of the simple loop that increments $\mathtt{x}$ we can use the variant:

$$V \quad = \quad \mathtt{n} - \mathtt{x}$$

$$\dfrac{\begin{array}{c}\{0 \leq \mathtt{x} < \mathtt{n} \wedge \mathtt{n} - \mathtt{x} = v\} \\ \mathtt{x\ :=\ x\ +\ 1} \\ \{0 \leq \mathtt{x} \leq \mathtt{n} \wedge \mathtt{n} - \mathtt{x} < v\}\end{array} \qquad 0 \leq \mathtt{x} < \mathtt{n} \Longrightarrow \mathtt{n} - \mathtt{x} \geq 0}{\{0 \leq \mathtt{x} \leq \mathtt{n}\}\ \mathbf{while}\ \mathtt{(x\ <\ n)\ x\ :=\ x\ +\ 1}\ \{0 \leq \mathtt{x} \leq \mathtt{n} \wedge \neg(\mathtt{x} < \mathtt{n})\}}$$

nonnegative: $0 \leq \mathtt{x} \leq \mathtt{n}$ implies $\mathtt{x} \leq \mathtt{n}$, which is equivalent to $\mathtt{n} - \mathtt{x} \geq 0$

decreasing: the backward substitution of $\mathtt{n} - \mathtt{x} < v$ through $\mathtt{x\ :=\ x\ +\ 1}$ is $\mathtt{n} - \mathtt{x} - 1 < v$, which follows from $\mathtt{n} - \mathtt{x} = v$

The rest of the proof is as for partial correctness.

Finding loop variants

Finding suitable loop variants is another part of deductive verification that cannot be completely automated.

In many cases, however, we can discover a suitable loop variant by looking for an integer expression V that:

nonnegative: is always nonnegative while the loop body executes

decreasing: is decreased by each iteration of the loop

Since property **nonnegative** should follow from the invariant, the invariant itself may suggest the variant.

Finding loop variants

Finding suitable loop variants is another part of deductive verification that cannot be completely automated.

In many cases, however, we can discover a suitable loop variant by looking for an integer expression V that:

nonnegative: is always nonnegative while the loop body executes

decreasing: is decreased by each iteration of the loop

Since property **nonnegative** should follow from the invariant, the invariant itself may suggest the variant.

More generally, the loop variant need not be an integer expression: it can be an expression over any well-founded ordered domain (of which the nonnegative integers are a special case).

Loop variant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

$$J = (0 \leq \mathtt{n} \leq \mathtt{b}) \wedge (\mathtt{pow} = \mathtt{a}^{\mathtt{n}})$$

Variable `n`:

- increases in each iteration
- is always less than or equal to `b`

Loop variant of power

```
{ n = 0 ∧ pow = 1 }
while n < b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

$$J = (0 \le \mathtt{n} \le \mathtt{b}) \wedge (\mathtt{pow} = \mathtt{a}^{\mathtt{n}})$$

Variable `n`:

- increases in each iteration
- is always less than or equal to `b`

$$V = \mathtt{b} - \mathtt{n}$$

- V remains nonnegative (from J)
- V is decreased in each iteration (because `n` increases and `b` is constant)

Loop variant of power (second version)

```
{ n = 1 ∧ pow = 1 }
while n ≤ b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

$$J = (1 \leq \mathtt{n} \leq \mathtt{b} + 1) \land (\mathtt{pow} = \mathtt{a}^{\mathtt{n}-1})$$

Variable `n`:

- increases in each iteration
- is always less than or equal to `b + 1`

Loop variant of power (second version)

```
{ n = 1 ∧ pow = 1 }
while n ≤ b
  pow := pow * a
  n := n + 1
{ pow = a^b }
```

$$J = (1 \leq \mathtt{n} \leq \mathtt{b} + 1) \wedge (\mathtt{pow} = \mathtt{a}^{\mathtt{n}-1})$$

Variable `n`:

- increases in each iteration
- is always less than or equal to `b + 1`

$$V = \mathtt{b} - \mathtt{n} + 1$$

- V remains nonnegative (from J)
- V is decreased in each iteration (because `n` increases and `b` is constant)

Loop variant of power (third version)

```
{ n = b ∧ pow = 1 }
while n > 0
  pow := pow * a
  n := n - 1
{ pow = a^b }
```

$$J = (0 \le \mathtt{n} \le \mathtt{b}) \wedge (\mathtt{pow} = \mathtt{a}^{\mathtt{b}-\mathtt{n}})$$

Variable `n`:

- decreases in each iteration
- is always greater than or equal to zero

Loop variant of power (third version)

```
{ n = b ∧ pow = 1 }
while n > 0
  pow := pow * a
  n := n - 1
{ pow = a^b }
```

$$J \;=\; (0 \leq \mathtt{n} \leq \mathtt{b}) \wedge (\mathtt{pow} = \mathtt{a}^{\mathtt{b}-\mathtt{n}})$$

Variable `n`:

- decreases in each iteration
- is always greater than or equal to zero

$$V \quad = \quad \mathtt{n}$$

- V remains nonnegative (from J)
- V is decreased in each iteration (because `n` decreases)

Embedding invariants and variants in Helium

Since reasoning about loops requires invariants and variants, and these cannot be easily inferred from the executable code, we provide keywords to embed such annotations directly in the code of loops.

$$
\begin{aligned}
\textit{Loop} \;::=\; & \texttt{while}\ \textit{Expression} \\
& [\texttt{invariant}\ \textit{Expression}]^{+}\ [\texttt{variant}\ \textit{Expression}] \\
& \textit{Statement}^{+}
\end{aligned}
$$

- We can declare multiple `invariant` clauses; the invariant is their conjunction.
- When no `invariant` clause is declared, it is the same as declaring `true` as invariant.

Embedding invariants and variants in Helium

Since reasoning about loops requires invariants and variants, and these cannot be easily inferred from the executable code, we provide keywords to embed such annotations directly in the code of loops.

$$
\begin{aligned}
\textit{Loop} \ ::= \ & \mathtt{while}\ \textit{Expression} \\
& [\mathtt{invariant}\ \textit{Expression}]^{+}\ [\mathtt{variant}\ \textit{Expression}] \\
& \textit{Statement}^{+}
\end{aligned}
$$

- We can declare multiple `invariant` clauses; the invariant is their conjunction.
- When no `invariant` clause is declared, it is the same as declaring `true` as invariant.

A fully annotated loop has all the ingredients to apply the inference rule of axiomatic semantics:

$$
\frac{\{J \land C \land V = v\}\ B\ \{J \land V < v\} \quad J \land C \Longrightarrow V \geq 0}{\{J\}\ \mathtt{while}\ C\ \mathtt{invariant}\ J\ \mathtt{variant}\ V\ B\ \{J \land \neg C\}}
$$

Predicate transformers and verification conditions

Predicate transformers and verification conditions

Weakest precondition calculus

Calculating proofs in Hoare logic

Applying the inference rules of axiomatic semantics is an impractical technique to prove programs correct.

We present the weakest precondition calculus: a more calculational approach to applying the rules of axiomatic semantics.

“Calculational” means that we can apply the steps mechanically – based on the program text and its specification.

Calculating proofs in Hoare logic

Applying the inference rules of axiomatic semantics is an impractical technique to prove programs correct.

We present the weakest precondition calculus: a more calculational approach to applying the rules of axiomatic semantics.

"Calculational" means that we can apply the steps mechanically – based on the program text and its specification.

Edsger W. Dijkstra

Dijkstra invented the weakest precondition method as a way to incrementally develop programs starting from their specification.

We will use a different version of Dijkstra's calculus that is geared towards a posteriori correctness proofs.

Weakest preconditions

Given a program S and a specification predicate Q,
the weakest precondition $\mathbf{wp}(S, Q)$ is the weakest predicate P
such that $\{P\}\ S\ \{Q\}$ is valid.

Weakest means that, for every other predicate P' such that
$\{P'\}\ S\ \{Q\}$ is valid, $P' \Longrightarrow P$.

Weakest preconditions

Given a program S and a specification predicate Q,
the weakest precondition $\mathbf{wp}(S, Q)$ is the weakest predicate P
such that $\{P\}\ S\ \{Q\}$ is valid.

Weakest means that, for every other predicate P' such that $\{P'\}\ S\ \{Q\}$ is valid, $P' \Longrightarrow P$.

$\mathbf{wp}(S, Q)$ is called predicate transformer because it transforms predicate Q into another predicate.

Weakest preconditions

Given a program S and a specification predicate Q,
the weakest precondition $\mathbf{wp}(S, Q)$ is the weakest predicate P
such that $\{P\}\ S\ \{Q\}$ is valid.

Weakest means that, for every other predicate P' such that $\{P'\}\ S\ \{Q\}$ is valid, $P' \implies P$.

$\mathbf{wp}(S, Q)$ is called predicate transformer because it transforms predicate Q into another predicate.

Sometimes it is convenient to assume that $\mathbf{wp}(S, Q)$ is a set of predicates that are implicitly conjoined.

Weakest preconditions

Given a program S and a specification predicate Q,
the weakest precondition $\mathbf{wp}(S, Q)$ is the weakest predicate P
such that $\{P\}\ S\ \{Q\}$ is valid.

Weakest means that, for every other predicate P' such that $\{P'\}\ S\ \{Q\}$ is valid, $P' \Longrightarrow P$.

$\mathbf{wp}(S, Q)$ is called predicate transformer because it transforms predicate Q into another predicate.

Sometimes it is convenient to assume that $\mathbf{wp}(S, Q)$ is a set of predicates that are implicitly conjoined.

We focus on a variant $\mathbf{wlp}(S, Q)$ of weakest precondition called weakest liberal precondition which does not check for termination:

$$\begin{array}{rcl} \{P\}\ S\ \{Q\} \text{ is totally correct} & \text{iff} & P \Longrightarrow \mathbf{wp}(S, Q) \\ \{P\}\ S\ \{Q\} \text{ is partially correct} & \text{iff} & P \Longrightarrow \mathbf{wlp}(S, Q) \end{array}$$

Weakest precondition of Helium

The weakest precondition of statements is derived from the corresponding inference rules of axiomatic semantics:

$$
\begin{aligned}
\mathbf{wlp}(\texttt{skip}, Q) &= Q \\
\mathbf{wlp}(S_1\,;S_2, Q) &= \mathbf{wlp}(S_1, \mathbf{wlp}(S_2, Q)) \\
\mathbf{wlp}(\mathrm{v}_1\,, \ldots\,, \mathrm{v}_n \;\texttt{:=}\; E_1\,, \ldots\,, E_n, Q) &= Q[\mathrm{v}_1 \mapsto E_1, \ldots, \mathrm{v}_n \mapsto E_n] \\
\mathbf{wlp}(\texttt{if}\ C\ T\ \texttt{else}\ E, Q) &= \begin{array}{ll} & (C \implies \mathbf{wlp}(T, Q)) \\ \wedge & (\neg C \implies \mathbf{wlp}(E, Q)) \end{array}
\end{aligned}
$$

Weakest precondition of Helium

The weakest precondition of statements is derived from the corresponding inference rules of axiomatic semantics:

$$\begin{aligned}
\mathbf{wlp}(\texttt{skip}, Q) &= Q \\
\mathbf{wlp}(S_1\,;S_2, Q) &= \mathbf{wlp}(S_1, \mathbf{wlp}(S_2, Q)) \\
\mathbf{wlp}(\mathtt{v}_1, \ldots, \mathtt{v}_n \mathrel{:=} E_1, \ldots, E_n, Q) &= Q[\mathtt{v}_1 \mapsto E_1, \ldots, \mathtt{v}_n \mapsto E_n] \\
\mathbf{wlp}(\texttt{if}\ C\ T\ \texttt{else}\ E, Q) &= \begin{array}{ll} & (C \implies \mathbf{wlp}(T, Q)) \\ \land & (\neg C \implies \mathbf{wlp}(E, Q)) \end{array}
\end{aligned}$$

For example, to prove

$$\{\top\}\ \texttt{if (x > y) max := x else max := y}\ \left\{ \begin{array}{ll} & (\mathtt{x} \geq \mathtt{y} \implies \mathtt{max} = \mathtt{x}) \\ \land & (\mathtt{x} \leq \mathtt{y} \implies \mathtt{max} = \mathtt{y}) \end{array} \right\}$$

we check that the following formula is valid:

$$\top \implies \left(\begin{array}{ll} & (\mathtt{x} > \mathtt{y} \implies (\mathtt{x} \geq \mathtt{y} \implies \mathtt{x} = \mathtt{x}) \land (\mathtt{x} \leq \mathtt{y} \implies \mathtt{x} = \mathtt{y})) \\ \land & (\mathtt{x} \leq \mathtt{y} \implies (\mathtt{x} \geq \mathtt{y} \implies \mathtt{y} = \mathtt{x}) \land (\mathtt{x} \leq \mathtt{y} \implies \mathtt{y} = \mathtt{y})) \end{array} \right)$$

Weakest precondition of Helium: loops

The weakest precondition of loops is a bit more involved:

$$\mathbf{wlp}(\texttt{while}\ C\ \texttt{invariant}\ J\ B, Q) = \begin{array}{ll} & \overset{\text{initiation}}{J} \\ \wedge & \underbrace{(\forall \mathbf{v} \bullet (J \wedge C \Longrightarrow \mathbf{wlp}(B, J))[\mathcal{V}(J, C, B) \mapsto \mathbf{v}])}_{\text{consecution}} \\ \wedge & \underbrace{(\forall \mathbf{v} \bullet (J \wedge \neg C \Longrightarrow Q)[\mathcal{V}(J, C, Q) \mapsto \mathbf{v}])}_{\text{closing}} \end{array}$$

v is a vector of universally quantified variables, one for every program variable mentioned in J, C, and B or Q

$\mathcal{V}(X)$ is the set of program (free) variables mentioned in X

initiation is evaluated in the loop's pre-state

consecution must hold in every initial state where the invariant and the loop condition hold

closing must hold in every final state where the invariant and the exit condition hold

Universally quantified state conditions

Parts of the weakest precondition of loops require to reason about universally quantified states:

$$\forall \mathbf{v} \bullet (J \wedge C \Longrightarrow \mathbf{wlp}(B, J))[\mathcal{V}(J, C, B) \mapsto \mathbf{v}]$$

Without universal quantification, we could use other predicates – obtained by propagating the weakest precondition before the loop – to prove consecution – which instead should follow solely from J and C.

Universally quantified state conditions

Parts of the weakest precondition of loops require to reason about universally quantified states:

$$\forall \mathbf{v} \bullet (J \wedge C \Longrightarrow \mathbf{wlp}(B, J))[\mathcal{V}(J, C, B) \mapsto \mathbf{v}]$$

Without universal quantification, we could use other predicates – obtained by propagating the weakest precondition before the loop – to prove consecution – which instead should follow solely from J and C.

Without universal quantification:

```
x := 0
while x < 2
invariant x < 2
  x := x + 1
{ false }
```

$$\mathbf{wlp}(\mathtt{while}\ldots, \bot) = \begin{array}{ll} & \mathtt{x} < 2 \\ \wedge & (\mathtt{x} < 2 \wedge \mathtt{x} < 2 \Longrightarrow \mathtt{x} + 1 < 2) \\ \wedge & (\mathtt{x} < 2 \wedge \mathtt{x} \geq 2 \Longrightarrow \mathtt{false}) \end{array}$$

$$\mathbf{wlp}(\text{all program}, \bot) = \begin{array}{ll} & 0 < 2 \\ \wedge & (0 < 2 \wedge 0 < 2 \Longrightarrow 0 + 1 < 2) \\ \wedge & (0 < 2 \wedge 0 \geq 2 \Longrightarrow \mathtt{false}) \end{array}$$

Expressing universally quantified state conditions

A trick to express such universally quantified states implicitly is to replace the quantified variables with fresh (program) variables that are not used in the existing program: this way all that we know about these fresh variables comes from the only predicates where they appear.

With fresh variable a replacing x:

```
x := 0
while x < 2
invariant x < 2
  x := x + 1
{ false }
```

$$\mathbf{wlp}(\texttt{while}\ldots, \bot) = \begin{array}{ll} & \mathtt{x} < 2 \\ \wedge & (\mathtt{a} < 2 \wedge \mathtt{a} < 2 \Longrightarrow \mathtt{a} + 1 < 2) \\ \wedge & (\mathtt{a} < 2 \wedge \mathtt{a} \geq 2 \Longrightarrow \mathtt{false}) \end{array}$$

$$\mathbf{wlp}(\text{all program}, \bot) = \begin{array}{ll} & 0 < 2 \\ \wedge & (\mathtt{a} < 2 \wedge \mathtt{a} < 2 \Longrightarrow \mathtt{a} + 1 < 2) \\ \wedge & (\mathtt{a} < 2 \wedge \mathtt{a} \geq 2 \Longrightarrow \mathtt{false}) \end{array}$$

Syntactic loop invariants

The value v of any variable v that is not modified by a loop's body is an obvious invariant of the loop. However, we still have to specify $\mathtt{v} = v$ explicitly in the loop invariant if we want to be able to propagate this fact after the loop.

To automatically account for such syntactic invariants we can universally quantify only variables $\mathcal{F}(B)$ that may be changed by the loop's body B – that is, that appear to the left of an assignment in B.

With fresh variable a replacing x:

```
x, y := 0, 0
while x < 2
invariant x ≤ 2
  x := x + 1
{ y = 0 }
```

$$\mathbf{wlp}(\mathtt{while}\ldots, \bot) = \begin{array}{ll} & \mathtt{x} \le 2 \\ \land & (\mathtt{a} \le 2 \land \mathtt{a} < 2 \Longrightarrow \mathtt{a} + 1 \le 2) \\ \land & (\mathtt{a} \le 2 \land \mathtt{a} \ge 2 \Longrightarrow \mathtt{y} = 0) \end{array}$$

$$\mathbf{wlp}(\text{all program}, \bot) = \begin{array}{ll} & 0 \le 2 \\ \land & (\mathtt{a} \le 2 \land \mathtt{a} < 2 \Longrightarrow \mathtt{a} + 1 \le 2) \\ \land & (\mathtt{a} \le 2 \land \mathtt{a} \ge 2 \Longrightarrow 0 = 0) \end{array}$$

Weakest precondition for total correctness

Weakest liberal preconditions are the same as weakest precondition except for loops, where proving total correctness requires to reason about the variant as well.

$$\mathbf{wp}(\texttt{while}\ C\ \texttt{invariant}\ J\ \texttt{variant}\ V\ B, Q) =$$

$$\begin{array}{ll} & J \\ \wedge & \left(\forall\, \mathbf{v}, v \bullet \left(\begin{array}{c} J \wedge C \wedge V = v \implies \\ \mathbf{wp}(B, J \wedge V < v) \end{array} \right)[\mathcal{V}(J, C, B, V) \mapsto \mathbf{v}]\right) \\ \wedge & \left(\forall\, \mathbf{v} \bullet (J \wedge \neg C \implies Q \wedge V \geq 0)[\mathcal{V}(J, C, Q, V) \mapsto \mathbf{v}]\right) \end{array}$$

Inductive weakest precondition

The weakest precondition of loops approximates the loop's body behavior with the loop's invariant. Therefore it is a weakest precondition only up to this approximation.

Inductive weakest precondition

The weakest precondition of loops approximates the loop's body behavior with the loop's invariant. Therefore it is a weakest precondition only up to this approximation.

If we want a weakest precondition based on the exact semantics of the loop body we end up with an inductive (that is, recursive) formula:

$$\mathbf{wlp}(\texttt{while}\ C\ B, Q) = \begin{cases} \mathbf{wlp}(B, \mathbf{wlp}(\texttt{while}\ C\ B, Q)) & \text{if } C \\ Q & \text{otherwise} \end{cases}$$

In deductive verification, we normally use the approximate semantics based on loop invariants.

Some people prefer to use weakest precondition only to denote the inductive semantics, referring to the invariant-based semantics as "verification condition calculation" (see later).

Proof outlines embedded in code

We can annotate programs with intermediate state predicates, corresponding to the weakest preconditions that are checked locally. Whenever two predicates come one after the other, we should check that the first implies the second.

```
var a, b, max: Integer
{ true }
if a > b
  { a > b }
  { (a ≥ b ⟹ a = a) ∧ (a ≤ b ⟹ a = b) }
  max := a
else
  { a ≤ b }
  { (a ≥ b ⟹ b = a) ∧ (a ≤ b ⟹ b = b) }
  max := b
{ (a ≥ b ⟹ max = a) ∧ (a ≤ b ⟹ max = b) }
```

Proof outline of increment loop

```
var x, n: Integer
{ n ≥ 0 }
{ 0 ≤ 0 ≤ n }
x := 0
{ 0 ≤ x ≤ n }
while x < n
invariant 0 ≤ x ≤ n
  { 0 ≤ x ≤ n ∧ x < n }
  { -1 ≤ x < n }
  { 0 ≤ x + 1 ≤ n }
  x := x + 1
  { 0 ≤ x ≤ n }
{ 0 ≤ x ≤ n ∧ x ≥ n }
{ x = n }
```

Proof outline of integer power

```
var a, b, pow, n: Integer
{ b ≥ 0 }
{ 0 ≤ 0 ≤ b ∧ 1 = a^0 }
n, pow := 0, 1
{ 0 ≤ n ≤ b ∧ pow = a^n }
while n < b
invariant 0 ≤ n ≤ b ∧ pow = a^n
  { 0 ≤ n ≤ b ∧ pow = a^n ∧ n < b }
  { 0 ≤ n + 1 ≤ b ∧ pow * a = a^(n + 1) }
  pow := pow * a
  { 0 ≤ n + 1 ≤ b ∧ pow = a^(n + 1) }
  n := n + 1
  { 0 ≤ n ≤ b ∧ pow = a^n }
{ 0 ≤ n ≤ b ∧ pow = a^n ∧ n ≥ b }
{ pow = a^b }
```

Proof outline of factorial

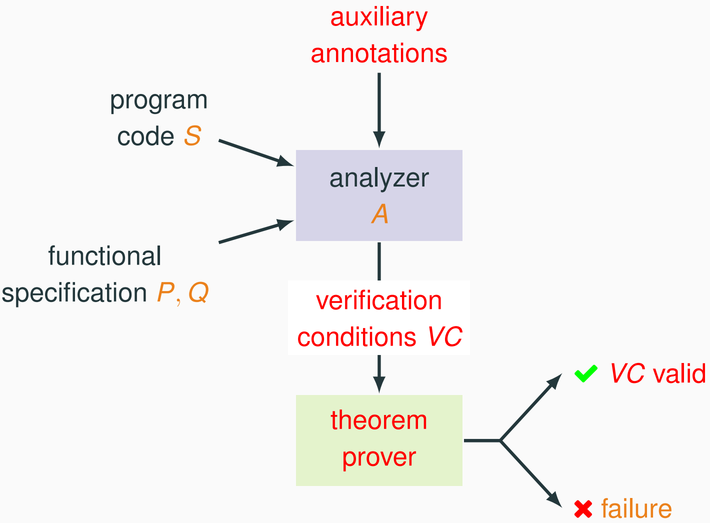
```
var m, n, fac: Integer
{ n ≥ 0 }
{ 0 ≤ 0 ≤ n ∧ 1 = 0! }
m, fac := 0, 1
{ 0 ≤ m ≤ n ∧ fac = m! }
while m < n
invariant 0 ≤ m ≤ n ∧ fac = m!
  { 0 ≤ m ≤ n ∧ fac = m! ∧ m < n }
  { 0 ≤ m + 1 ≤ n ∧ fac ∗ (m + 1) = (m + 1)! }
  m := m + 1
  { 0 ≤ m ≤ n ∧ fac ∗ m = m! }
  fac := fac ∗ m
  { 0 ≤ m ≤ n ∧ fac = m! }
{ 0 ≤ m ≤ n ∧ fac = m! ∧ m ≥ n }
{ fac = n! }
```

Predicate transformers and verification conditions

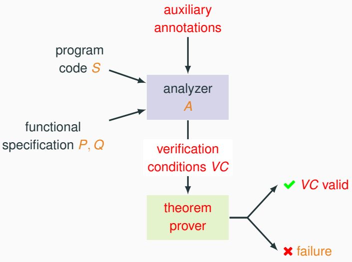
Verification conditions

Automating deductive verification

The ultimate goal of deductive verification is expressing correctness as the validity of a logic formula.



Automating deductive verification

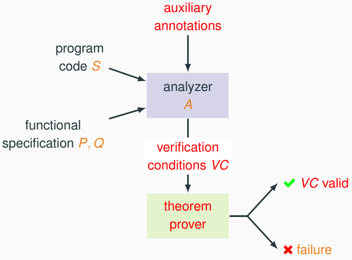


Auxiliary annotations include loop invariants and variants, and other intermediate assertions.

The verification conditions are logic formulas whose conjunction is valid iff the program P is correct with respect to P, Q.

A theorem prover can determine whether an arbitrary logic formula is valid or not.

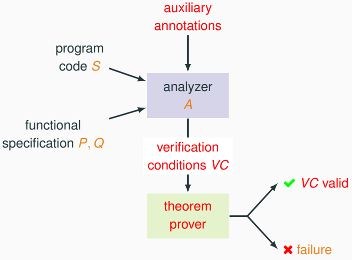
Verification condition calculation



For example, using the weakest precondition calculus:

$$VC(\{P\}\ S\ \{Q\}) \quad = \quad P \Longrightarrow \mathbf{wp}(S, Q)$$

Verification condition calculation



For example, using the weakest precondition calculus:

$$VC(\{P\}\ S\ \{Q\}) \quad = \quad P \Longrightarrow \mathbf{wp}(S, Q)$$

- Different encodings of *VC* have different characteristics
- Sometimes **wp** is used to only denote the actual transformed predicates:

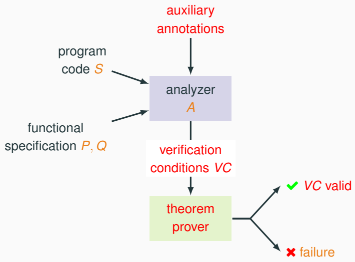
$$VC = \mathbf{wp} + \text{obligations}$$

  where the additional proof obligations are the check that loop invariants are indeed invariants
- Sometimes **wp** is used to only denote the loop's exact inductive semantics:

$$VC = \mathbf{wp} - \text{inductive loops} + \text{loop invariants}$$

Deductive verification: soundness and completeness

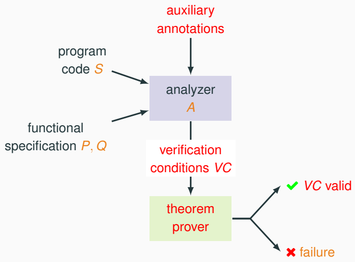
The decidability of the logic used to encode verification conditions, and the soundness of the corresponding theorem prover, determine the overall properties of deductive verifiers.



When it targets first-order logic specifications with arithmetic, deductive verification is:

Deductive verification: soundness and completeness

The decidability of the logic used to encode verification conditions, and the soundness of the corresponding theorem prover, determine the overall properties of deductive verifiers.



When it targets first-order logic specifications with arithmetic, deductive verification is:

- sound
- undecidable
  (also: $\{\top\}\ S\ \{\bot\}$ is partially correct iff S doesn't halt)
- incomplete

Deductive verification: complexity and automation

Deductive verification's level of automation depends on:

Since we can help the theorem prover by providing more auxiliary annotations, deductive verification is an auto-active analysis technique.

Deductive verification: complexity and automation

Deductive verification's level of automation depends on:

- the complexity of the specification:
  the more detailed the specification, the bigger the verification effort

Since we can help the theorem prover by providing more auxiliary annotations, deductive verification is an auto-active analysis technique.

Deductive verification: complexity and automation

Deductive verification's level of automation depends on:

- the complexity of the specification:
  the more detailed the specification, the bigger the verification effort
- the size of the program:
  programs with more complex control structure lead to bigger *VC*

Since we can help the theorem prover by providing more auxiliary annotations, deductive verification is an auto-active analysis technique.

Deductive verification: complexity and automation

Deductive verification's level of automation depends on:

- the complexity of the specification:
  the more detailed the specification, the bigger the verification effort
- the size of the program:
  programs with more complex control structure lead to bigger *VC*
- the size of the *VC*, relative to the size of the program:
  the naive encoding of **wp** is exponential in the number of conditionals, but there exist more efficient encodings that are quadratic or even linear in the size of the program

Since we can help the theorem prover by providing more auxiliary annotations, deductive verification is an auto-active analysis technique.

Deductive verification: complexity and automation

Deductive verification's level of automation depends on:

- the complexity of the specification:
  the more detailed the specification, the bigger the verification effort
- the size of the program:
  programs with more complex control structure lead to bigger *VC*
- the size of the *VC*, relative to the size of the program:
  the naive encoding of **wp** is exponential in the number of conditionals, but there exist more efficient encodings that are quadratic or even linear in the size of the program
- the amount, detail, and style of auxiliary annotations:
  theorem prover are quite sensitive to the form in which formulas are expressed

Since we can help the theorem prover by providing more auxiliary annotations, deductive verification is an auto-active analysis technique.

SAT and SMT solvers

The theorem provers used to check the validity of *VC* are often SMT solvers.

A Satisfiability Modulo Theory solver (SMT solver) is
a theorem prover for propositional logic (SAT)
combined with fragments of logic theories.

Example of decidable theories:

- linear integer arithmetic: decidable
- quantifier-free linear integer arithmetic: NP-complete (reduces to integer linear programming)
- quantifier-free equality (with uninterpreted functions): in P

SAT and SMT solvers

The theorem provers used to check the validity of *VC* are often SMT solvers.

> A Satisfiability Modulo Theory solver (SMT solver) is
> a theorem prover for propositional logic (SAT)
> combined with fragments of logic theories.

Example of decidable theories:

- linear integer arithmetic: decidable
- quantifier-free linear integer arithmetic: NP-complete (reduces to integer linear programming)
- quantifier-free equality (with uninterpreted functions): in P

If a pre/post specification is quantifier free, so are the *VC* computed using **wp**.

SAT and SMT solvers

The theorem provers used to check the validity of *VC* are often SMT solvers.

> A Satisfiability Modulo Theory solver (SMT solver) is
> a theorem prover for propositional logic (SAT)
> combined with fragments of logic theories.

Example of decidable theories:

- linear integer arithmetic: decidable
- quantifier-free linear integer arithmetic: NP-complete (reduces to integer linear programming)
- quantifier-free equality (with uninterpreted functions): in P

If a pre/post specification is quantifier free, so are the *VC* computed using **wp**.

An SMT solver typically accepts input with unrestricted quantification, but in that case it relies on incomplete heuristics to try to build a proof.

Auxiliary annotations in Helium

Since specification annotations are central to deductive verification, it would be convenient to support them directly in the programming language.

We introduce three kinds of annotations in Helium:

pre- and postconditions to conveniently specify the input/output behavior of a piece of code

assert statements to embed in code intermediate lemmas in the style of proof outlines

assume statements to encode assumptions about the input

Procedures

A procedure declaration is just a convenient way to define and specify a self-contained piece of code with clearly defined input/output behavior.

procedure's name

output arguments

input arguments

$$
\begin{aligned}
He &::= (\mathit{Statement} \mid \mathit{Procedure})^* \\
\mathit{Procedure} &::= \texttt{procedure}\ \mathit{Identifier} \\
&\quad [(\mathit{VarDeclaration}^+)] : [(\mathit{VarDeclaration}^+)] \\
&\quad \mathit{Precondition}^*\ \mathit{Frame}^*\ \mathit{Postcondition}^*\ \mathit{Statement}^+ \\
\mathit{Precondition} &::= \texttt{require}\ \texttt{BooleanExpression} \\
\mathit{Frame} &::= \texttt{modify}\ \texttt{v}_1, \ldots, \texttt{v}_n \\
\mathit{Postcondition} &::= \texttt{ensure}\ \texttt{BooleanExpression}
\end{aligned}
$$

Procedures: semantics

The correctness of a procedure `proc` is equivalent to the correctness of a Hoare triple:

$$\frac{\{P\}\ B\ \{Q\} \qquad \mathcal{F}(B) \cap \mathcal{G} \subseteq F}{\texttt{procedure proc (in: T): (out: T) require } P \texttt{ modify } F \texttt{ ensure } Q\ B}$$

where `proc`'s input `in` and output `out` arguments behave like variables local to `proc`'s body B.

P is a predicate over `in` and any global variables

Q is a predicate over `in`, `out`, and any global variables

F is a set of global variables

Frame conditions

The frame condition F (also: modify or write clause) is the specification of what global variables `proc` modifies.

$$\frac{\{P\}\ B\ \{Q\} \qquad \mathcal{F}(B) \cap \mathcal{G} \subseteq F}{\texttt{procedure proc (in: T): (out: T) require } P \texttt{ modify } F \texttt{ ensure } Q\ B}$$

Procedure `proc` is correct if the set $\mathcal{F}(B) \cap \mathcal{G}$ of all variables that are assigned to in B (in $\mathcal{F}(B)$) and are global (in $\mathcal{G}$) is included in the frame specification F.

Frame conditions

The frame condition F (also: modify or write clause) is the specification of what global variables `proc` modifies.

$$\frac{\{P\}\ B\ \{Q\} \qquad \mathcal{F}(B) \cap \mathcal{G} \subseteq F}{\texttt{procedure proc (in: T): (out: T) require } P \texttt{ modify } F \texttt{ ensure } Q\ B}$$

Procedure `proc` is correct if the set $\mathcal{F}(B) \cap \mathcal{G}$ of all variables that are assigned to in B (in $\mathcal{F}(B)$) and are global (in $\mathcal{G}$) is included in the frame specification F.

Frame conditions express the memory footprint of a procedure. We will use them to reason about procedure calls in an extension of Helium.

Multiple clauses and old expressions

As usual, multiple specification clauses of the same kind are implicitly conjoined:

```
procedure proc (in: T): (out: T)
  require P₁ require P₂ ··· require Pᵣ
  modify F₁ modify F₂ ··· modify Fₘ
  ensure Q₁ ensure Q₂ ··· ensure Qₑ
```

`proc`'s precondition: $\bigwedge_k P_k$ (if $r = 0$, the precondition is $\top$)

`proc`'s frame condition: $\bigcup_k F_k$ (if $m = 0$, the frame condition is $\emptyset$)

`proc`'s postcondition: $\bigwedge_k Q_k$ (if $e = 0$, the postcondition is $\top$)

Multiple clauses and old expressions

As usual, multiple specification clauses of the same kind are implicitly conjoined:

```
procedure proc (in: T): (out: T)
  require P_1 require P_2 ··· require P_r
  modify F_1 modify F_2 ··· modify F_m
  ensure Q_1 ensure Q_2 ··· ensure Q_e
```

`proc`'s precondition: $\bigwedge_k P_k$ (if $r = 0$, the precondition is $\top$)

`proc`'s frame condition: $\bigcup_k F_k$ (if $m = 0$, the frame condition is $\emptyset$)

`proc`'s postcondition: $\bigwedge_k Q_k$ (if $e = 0$, the postcondition is $\top$)

Anywhere within `proc`'s annotations, the expression **`old`**`(`e`)` denotes the value of e in `proc`'s pre-state – where P holds and the execution of `proc` begins.

Programs as procedures

We split the two conjuncts in `max`'s postcondition into two **ensure** clauses.

```
procedure max (a, b: Integer): (max: Integer)
ensure a ≥ b ⟹ max = a
ensure a ≤ b ⟹ max = b
  if a > b
    max := a
  else
    max := b
```

Programs as procedures

The **modify** clause is empty because `power` only needs input and output arguments.

```
procedure power (a, b: Integer): (pow: Integer)
require b ≥ 0
ensure pow = a^b
  var n: Integer
  n, pow := 0, 1
  while n < b
  invariant 0 ≤ n ≤ b ∧ pow = a^n
    pow := pow * a
    n := n + 1
```

Programs as procedures

In this variant of `factorial` we directly modify the input argument; hence we need to refer to `old(n)` in the specification.

```
procedure factorial (n: Integer): (fac: Integer)
require n ≥ 0
ensure fac = old(n)!
  fac := 1
  while n > 0
  invariant 0 ≤ n ≤ old(n) ∧ fac * n! = old(n)!
  variant n
    fac := fac * n
    n := n - 1
```

Assert and assume

We add two passive (specification) statements to Helium:

$$\begin{aligned}
\mathit{Statement} &::= \mathit{Declaration} \mid \mathit{Active} \mid \mathit{Passive} \\
\mathit{Passive} &::= \mathit{Assert} \mid \mathit{Assume} \\
\mathit{Assert} &::= \texttt{assert}\ \mathit{BooleanExpression} \\
\mathit{Assume} &::= \texttt{assume}\ \mathit{BooleanExpression}
\end{aligned}$$

The operational semantics of `assert` P is equivalent to **`skip`** if P holds; otherwise it leads to an **error** state that forces termination.

$$\frac{[\![P]\!]_s = \top}{\langle \texttt{assert}\ P, s\rangle \rightsquigarrow s} \qquad \frac{[\![P]\!]_s = \bot}{\langle \texttt{assert}\ P, s\rangle \rightsquigarrow \textbf{error}}$$

The semantics of `assume` P is equivalent to `assert` P if P holds; otherwise, execution is simply undefined. This means we only consider executions where P holds.

$$\frac{[\![P]\!]_s = \top}{\langle \texttt{assume}\ P, s\rangle \rightsquigarrow s}$$

Assert and assume: axiomatic semantics

The axiomatic semantics of `assert` *A* requires that *A* hold, independent of what postcondition is to be established.

$$\frac{P \implies A \land Q}{\{P\}\ \texttt{assert}\ A\ \{Q\}} \qquad \textbf{wp}(\texttt{assert}\ P, Q) \ = \ P \land Q$$

Assert and assume: axiomatic semantics

The axiomatic semantics of `assert` A requires that A hold, independent of what postcondition is to be established.

$$\frac{P \implies A \land Q}{\{P\}\ \texttt{assert}\ A\ \{Q\}} \qquad \textbf{wp}(\texttt{assert}\ P, Q) \ = \ P \land Q$$

The axiomatic semantics of `assume` A only considers the states such that A hold; if A is false, any Q is established trivially.

$$\frac{P \land A \implies Q}{\{P\}\ \texttt{assume}\ A\ \{Q\}} \qquad \textbf{wp}(\texttt{assume}\ P, Q) \ = \ P \implies Q$$

Semantic statements for specification

Using passive statements we can equivalently express the semantics of a procedure as a generic program fragment:

```
procedure proc (in: T): (out: T)
require P
ensure Q
  B
```

```
var in, out: T
assume P  // assume pre
B
assert Q  // assert post
```

Semantic statements for specification

Using passive statements we can equivalently express the semantics of a procedure as a generic program fragment:

```
procedure proc (in: T): (out: T)
require P
ensure Q
  B
```

```
var in, out: T
assume P  // assume pre
B
assert Q  // assert post
```

We can also add proof assertions into the program code as `assert`s:

```
procedure max (a, b: Integer): (max: Integer)
ensure a ≥ b ⟹ max = a
ensure a ≤ b ⟹ max = b
  if a > b
    { assert a ≥ b ⟹ a = a; max := a }
  else
    { assert a ≤ b; max := b }
```

Loop invariants: operational semantics

The axiomatic semantics of Helium relies on loop invariants. To be consistent, we could give loop invariants (and variants) an operational semantics that consists in checking that the invariant and variants satisfy their fundamental properties.

$$\frac{[\![C]\!]_s \quad \langle B, s\rangle \leadsto s' \quad [\![J]\!]_s \quad [\![J]\!]_{s'} \quad [\![V]\!]_s > [\![V]\!]_{s'} \geq 0 \quad \langle \texttt{while}\, C\ B, s'\rangle \leadsto s''}{\langle \texttt{while}\, C\ \texttt{invariant}\, J\ \texttt{variant}\ V\ B, s\rangle \leadsto s''}$$

$$\frac{\neg[\![C]\!]_s \quad [\![J]\!]_s}{\langle \texttt{while}\, C\ \texttt{invariant}\, J\ \texttt{variant}\ V\ B, s\rangle \leadsto s}$$

Loop invariants: operational semantics

The axiomatic semantics of Helium relies on loop invariants. To be consistent, we could give loop invariants (and variants) an operational semantics that consists in checking that the invariant and variants satisfy their fundamental properties.

$$\frac{[\![C]\!]_s \quad \langle B, s\rangle \leadsto s' \quad [\![J]\!]_s \quad [\![J]\!]_{s'} \quad [\![V]\!]_s > [\![V]\!]_{s'} \geq 0 \quad \langle \texttt{while}\ C\ B, s'\rangle \leadsto s''}{\langle \texttt{while}\ C\ \texttt{invariant}\ J\ \texttt{variant}\ V\ B, s\rangle \leadsto s''}$$

$$\frac{\neg[\![C]\!]_s \quad [\![J]\!]_s}{\langle \texttt{while}\ C\ \texttt{invariant}\ J\ \texttt{variant}\ V\ B, s\rangle \leadsto s}$$

An alternative would be ignoring loop invariants and variants in the operational semantics, considering them only proof aids. Which alternative is more appropriate depends on context, but normally we prefer to keep the runtime (operational) and proof (axiomatic) semantics <u>as aligned as possible</u>.

Predicate transformers and verification conditions

Forward reasoning and strongest postcondition

Forward assignment axiom

The assignment axiom in Hoare logic works backward:

$$\left\{Q[\mathsf{v} \mapsto E]\right\} \quad \mathsf{v} := E \quad \left\{Q\right\}$$

It is possible to write an equivalent one that works forward:

$$\left\{P\right\} \quad \mathsf{v} := E \quad \left\{\exists \bar{v} \bullet (\mathsf{v} = E[\mathsf{v} \mapsto \bar{v}] \land P[\mathsf{v} \mapsto \bar{v}])\right\}$$

Intuitively: $\bar{v}$ is the value of v before the assignment – or `old(v)`.

Forward assignment axiom

The assignment axiom in Hoare logic works backward:

$$\Big\{ Q[\mathsf{v} \mapsto E] \Big\} \quad \mathtt{v\ :=\ } E \quad \Big\{ Q \Big\}$$

It is possible to write an equivalent one that works forward:

$$\Big\{ P \Big\} \quad \mathtt{v\ :=\ } E \quad \Big\{ \exists \bar{v} \bullet (\mathsf{v} = E[\mathsf{v} \mapsto \bar{v}] \land P[\mathsf{v} \mapsto \bar{v}]) \Big\}$$

Intuitively: $\bar{v}$ is the value of v before the assignment – or `old(v)`.

Example of application:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathsf{x} = 1\}\ \mathtt{x\ :=\ x\ +\ 1}\ \{\exists \bar{x}(\mathsf{x} = (\mathtt{x\ +\ 1})[\mathsf{x} \mapsto \bar{x}] \land (\mathsf{x} = 1)[\mathsf{x} \mapsto \bar{x}])\}}{\{\mathsf{x} = 1\}\ \mathtt{x\ :=\ x\ +\ 1}\ \{\exists \bar{x}(\mathsf{x} = \bar{x} + 1 \land \bar{x} = 1)\}}}{\{\mathsf{x} = 1\}\ \mathtt{x\ :=\ x\ +\ 1}\ \{\exists \bar{x}(\mathsf{x} = 1 + 1 \land \bar{x} = 1)\}}}{\{\mathsf{x} = 1\}\ \mathtt{x\ :=\ x\ +\ 1}\ \{\mathsf{x} = 2 \land \exists \bar{x}(\bar{x} = 1)\}}}{\{\mathsf{x} = 1\}\ \mathtt{x\ :=\ x\ +\ 1}\ \{\mathsf{x} = 2\}}$$

Strongest postconditions

Using the forward assignment axiom, we can define a strongest postcondition predicate transformer:

Given a program S and a specification predicate P,
the strongest postcondition $\mathbf{sp}(S, P)$ is the strongest predicate Q
such that $\{P\}\ S\ \{Q\}$ is valid.

Strongest means that, for every other predicate Q' such that $\{P\}\ S\ \{Q'\}$ is valid, $Q \Longrightarrow Q'$.

Strongest postconditions

Using the forward assignment axiom, we can define a strongest postcondition predicate transformer:

Given a program S and a specification predicate P,
the strongest postcondition $\mathbf{sp}(S, P)$ is the strongest predicate Q
such that $\{P\}\ S\ \{Q\}$ is valid.

Strongest means that, for every other predicate Q' such that $\{P\}\ S\ \{Q'\}$ is valid, $Q \Longrightarrow Q'$.

$\{P\}\ S\ \{Q\}$ is (partially) correct iff $\mathbf{sp}(S, P) \implies Q$

Strongest postcondition of Helium

$$\begin{aligned}
\mathbf{sp}(\texttt{skip}, P) &= P \\
\mathbf{sp}(S_1 \,;\, S_2, P) &= \mathbf{sp}(S_2, \mathbf{sp}(S_1, P)) \\
\mathbf{sp}(\mathsf{v} := E, P) &= \exists \bar{v}\big(\mathsf{v} = E[\mathsf{v} \mapsto \bar{v}] \;\wedge\; P[\mathsf{v} \mapsto \bar{v}]\big) \\
\mathbf{sp}(\texttt{if}\ C\ T\ \texttt{else}\ E, P) &= (C \Longrightarrow \mathbf{sp}(T, P)) \;\wedge\; (\neg C \Longrightarrow \mathbf{sp}(E, P)) \\
&= \mathbf{sp}(T, C \wedge P) \;\vee\; \mathbf{sp}(E, \neg C \wedge P)
\end{aligned}$$

alternative forms

$$\begin{aligned}
\mathbf{sp}(\texttt{while}\ C\ \texttt{invariant}\ J\ B, P) = \;& \forall \mathbf{v} \bullet (P \Longrightarrow J)[\mathcal{V}(P, J) \mapsto \mathbf{v}] \\
& \wedge\; \forall \mathbf{v} \bullet (\mathbf{sp}(B, J \wedge C) \Longrightarrow J)\begin{bmatrix} \mathcal{V}(J, C, B) \\ \mapsto \\ \mathbf{v} \end{bmatrix} \\
& \wedge\; (J \wedge \neg C)
\end{aligned}$$

$$\begin{aligned}
\mathbf{sp}(\texttt{assert}\ A, P) &= (P \wedge A) \;\wedge\; \forall \mathbf{v} \bullet (P \Longrightarrow A)[\mathcal{V}(P, A) \mapsto \mathbf{v}] \\
\mathbf{sp}(\texttt{assume}\ A, P) &= P \wedge A
\end{aligned}$$

Strongest postcondition of Helium

$$
\begin{aligned}
\mathbf{sp}(\texttt{skip}, P) &= P \\
\mathbf{sp}(S_1\,;S_2, P) &= \mathbf{sp}(S_2, \mathbf{sp}(S_1, P)) \\
\mathbf{sp}(\mathsf{v} \;\texttt{:=}\; E, P) &= \exists \bar{v}\big(\mathsf{v} = E[\mathsf{v} \mapsto \bar{v}] \;\wedge\; P[\mathsf{v} \mapsto \bar{v}]\big) \\
\mathbf{sp}(\texttt{if}\; C\; T\; \texttt{else}\; E, P) &= (C \Longrightarrow \mathbf{sp}(T, P)) \;\wedge\; (\neg C \Longrightarrow \mathbf{sp}(E, P)) \\
&= \mathbf{sp}(T, C \wedge P) \;\vee\; \mathbf{sp}(E, \neg C \wedge P)
\end{aligned}
$$

alternative forms

$$
\begin{aligned}
\mathbf{sp}(\texttt{while}\; C\; \texttt{invariant}\; J\; B, P) = \;& \forall \mathbf{v} \bullet (P \Longrightarrow J)[\mathcal{V}(P, J) \mapsto \mathbf{v}] \\
& \wedge\; \forall \mathbf{v} \bullet (\mathbf{sp}(B, J \wedge C) \Longrightarrow J)\begin{bmatrix} \mathcal{V}(J, C, B) \\ \mapsto \\ \mathbf{v} \end{bmatrix} \\
& \wedge\; (J \wedge \neg C)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{sp}(\texttt{assert}\; A, P) &= (P \wedge A) \;\wedge\; \forall \mathbf{v} \bullet (P \Longrightarrow A)[\mathcal{V}(P, A) \mapsto \mathbf{v}] \\
\mathbf{sp}(\texttt{assume}\; A, P) &= P \wedge A
\end{aligned}
$$

As usual, the universal quantifications are needed so that the only facts about the loop that are propagated forward are the loop invariant and the exit condition.

Inductive strongest postcondition

The strongest postcondition of loops approximates the loop's body behavior with the loop's invariant. Therefore it is a strongest postcondition only up to this approximation.

Inductive strongest postcondition

The strongest postcondition of loops approximates the loop's body behavior with the loop's invariant. Therefore it is a strongest postcondition only up to this approximation.

If we want a strongest postcondition based on the exact semantics of the loop body we end up with an inductive (that is, recursive) formula:

$$\begin{aligned}\mathbf{sp}(\texttt{while}\ C\ B, P) &= \begin{cases}\mathbf{sp}(\texttt{while}\ C\ B, \mathbf{sp}(B, P \land C)) & \text{if } C \\ P & \text{otherwise}\end{cases} \\ &= \mathbf{sp}(\texttt{while}\ C\ B, \mathbf{sp}(B, P \land C)) \lor (P \land \neg C)\end{aligned}$$

Inductive strongest postcondition

The strongest postcondition of loops approximates the loop's body behavior with the loop's invariant. Therefore it is a strongest postcondition only up to this approximation.

If we want a strongest postcondition based on the exact semantics of the loop body we end up with an inductive (that is, recursive) formula:

$$
\begin{aligned}
\mathbf{sp}(\texttt{while}\ C\ B, P) &= \begin{cases} \mathbf{sp}(\texttt{while}\ C\ B, \mathbf{sp}(B, P \land C)) & \text{if } C \\ P & \text{otherwise} \end{cases} \\
&= \mathbf{sp}(\texttt{while}\ C\ B, \mathbf{sp}(B, P \land C)) \lor (P \land \neg C)
\end{aligned}
$$

Some people prefer to use strongest postcondition to denote only the inductive semantics, referring to the invariant-based semantics as "*VC* calculation".

Weakest precondition vs. strongest postcondition

The main advantage of backward reasoning (weakest precondition) is that the formulas are simpler – in particular, the assignment rule is purely syntactic and does not introduce quantifiers.

This makes the weakest precondition calculus preferable for deductive proofs based on the *VC* approach: reducing correctness to logic validity.

Weakest precondition vs. strongest postcondition

The main advantage of backward reasoning (weakest precondition) is that the formulas are simpler – in particular, the assignment rule is purely syntactic and does not introduce quantifiers.

This makes the weakest precondition calculus preferable for deductive proofs based on the *VC* approach: reducing correctness to logic validity.

The main advantage of forward reasoning (strongest postcondition) is that it is a form of symbolic execution. Thus, there is a notion of symbolic current state, which we can simplify dynamically – thus pruning the execution in specific case.

This makes the strongest postcondition calculus preferable for symbolic proofs that are often interactive.

Weakest-precondition vs. strongest-postcondition proof

$$\{x \neq 0\}\ \texttt{if (x = 0) x := 1 else x := 0}\ \{x = 0\}$$

In a weakest precondition proof we cannot simplify until the last step:

1. $x = 0 \Longrightarrow \mathbf{wp}(\texttt{x := 1}, x = 0)$
2. $x \neq 0 \Longrightarrow \mathbf{wp}(\texttt{x := 0}, x = 0)$
3. From 1: $x = 0 \Longrightarrow 1 = 0$
4. From 2: $x \neq 0 \Longrightarrow 0 = 0$
5. *VC*: $x \neq 0 \Longrightarrow (3) \wedge (4)$
6. *VC* is valid

In a strongest postcondition proof we can prune unreachable branches:

1. $\mathbf{sp}(\texttt{x := 1}, x = 0 \wedge x \neq 0)$
2. From 1: $\mathbf{sp}(\texttt{x := 1}, \bot) = \bot$
3. $\mathbf{sp}(\texttt{x := 0}, x \neq 0 \wedge x \neq 0)$
4. From 3: $\exists \bar{x}(\bar{x} \neq 0 \wedge x = 0)$
5. Overall $\mathbf{sp}$: $\bot \vee x = 0 \equiv x = 0$
6. *VC*: $x = 0 \Longrightarrow x = 0$
7. *VC* is valid

Note that: $\mathbf{sp}(C, \bot) = \bot$, since $\bot$ is a state that is never reached.

Predicate transformers and verification conditions

Deductive verification in practice

Dafny: deductive verification in action

We now present a brief tutorial of the Dafny deductive verifier.

Dafny's main developer is Rustan Leino, who has greatly contributed to making deductive verification more practical. Leino's work, in turn, has been greatly influenced by Greg Nelson's.

Rustan Leino

Greg Nelson

From Helium to Dafny: factorial

```
procedure factorial
   (n: Integer): (fac: Integer)
require n ≥ 0
ensure fac = old(n)!
  fac := 1
  while n > 0
  invariant 0 ≤ n ≤ old(n)
  invariant fac ∗ n! = old(n)!
  variant n
    fac := fac ∗ n
    n := n - 1
```

Dafny can sometimes infer simple invariants and variants ($0 \le m \le n$ and the variant in the example).

```
method factorial (n: int) returns (fac: int)
  requires n >= 0;
  ensures fac == bang(n);
{
  var m := n;  // arguments are constant
  fac := 1;    // statements terminated by ;
  while m > 0
    invariant 0 <= m <= n;
    invariant fac ∗ bang(m) == bang(n);
    decreases m;  // variant
  {
    fac := fac ∗ m;
    m := m - 1;
  }
} // C/Java-style braces

// recursive definition of math function
function bang (n: int): int
{ // conditional expression
  if (n <= 1) then 1 else n ∗ bang(n - 1)
}
```

Dafny: maximum

```
function fmax(x: int, y: int): int
{ if x >= y then x else y }

method max(x: int, y: int) returns(max: int)
  ensures x >= y ==> max == x;
  ensures x <= y ==> max == y;
  ensures max == fmax(x, y);  // alternative spec
{
  if x > y
  { max := x; }  // braces always required
  else
  { max := y;}
}
```

Dafny: power

```
      // cannot abbreviate as (a, b: int)
method power (a: int, b: int) returns (pow: int)
  requires b >= 0;
  ensures pow == to(a, b);
{
  var n: int;
  n, pow := 0, 1;
  while n < b
    invariant 0 <= n <= b && pow == to(a, n);
  {
    pow := pow * a;
    n := n + 1;
  }
}

// a^b
function to(a: int, b: int): int
  requires b >= 0;
{
  if b == 0 then 1 else a * to(a, b - 1)
}
```

Is Helium a realistic programming language?

It is easy to translate Helium into Java or any other real-world programming language. But are the proof techniques we present on Helium applicable to realistic programs too?

We will see how to reason about important language features:

- arrays
- procedure calls (including recursion)
- references/pointers of objects allocated on the heap

Is Helium a realistic programming language?

It is easy to translate Helium into Java or any other real-world programming language. But are the proof techniques we present on Helium applicable to realistic programs too?

We will see how to reason about important language features:

- arrays
- procedure calls (including recursion)
- references/pointers of objects allocated on the heap

Even with these features there are a number of important details that we should consider to perform correctness proofs of realistic programs:

- machine (bounded) numbers
- exceptions and jumps
- side effects
- errors and undefined behavior

Machine numbers

Type **`Integer`** represents mathematical integers, that can take any of the infinitely many integer values.

A realistic programming language normally uses machine integers, which have a finite/bounded range.

GENERIC NAME	EXAMPLE TYPE	RANGE
32-bit signed	C: **`int32_t`** Java: **`int`**	from -2^{32} to $2^{32} - 1$
32-bit unsigned	C: **`uint32_t`** C#: **`uint`**	from 0 to $2^{32} - 1$
64-bit signed	C: **`int64_t`** Java: **`long`**	from -2^{64} to $2^{64} - 1$

Machine numbers

Type **`Integer`** represents mathematical integers, that can take any of the infinitely many integer values.

A realistic programming language normally uses machine integers, which have a finite/bounded range.

GENERIC NAME	EXAMPLE TYPE	RANGE
32-bit signed	C: **`int32_t`** Java: **`int`**	from -2^{32} to $2^{32} - 1$
32-bit unsigned	C: **`uint32_t`** C#: **`uint`**	from 0 to $2^{32} - 1$
64-bit signed	C: **`int64_t`** Java: **`long`**	from -2^{64} to $2^{64} - 1$

Machine numbers (integers and floating-point) have behavior that cannot happen with mathematical numbers (integers and rationals/reals):

`n + m`	may overflow
`x == y`	rounding error
`int n = (long) m;`	narrowing conversion

Exceptions and jumps

Helium is a highly structured programming language:

- single entry and exit point in every code block
- statements executed in textual order

Realistic programming languages often include features that go beyond pure structured programming:

control-flow breaking statements such as `return`, `break`, and `continue`

exceptions and exception handling

jumps such as `goto` statements

Exceptions and jumps

Helium is a highly structured programming language:

- single entry and exit point in every code block
- statements executed in textual order

Realistic programming languages often include features that go beyond pure structured programming:

control-flow breaking statements such as `return`, `break`, and `continue`

exceptions and exception handling

jumps such as `goto` statements

```java
public int exceptions() {
  try {
    throw new Error();
  } finally {
    return 42;
  }
}
```

In this piece of Java code:

- What does `exceptions()` return?
- Does `exceptions()` return normally or with an exception?

(For details see Martin Nordio's PhD thesis)

Side effects

Several rules of axiomatic semantics are sound only if evaluating an expression does not have any side effects – that is, it does not change the state in any way.

expression with side effects

$$\{x = 3\}\ \texttt{x := x++}\ \{x = 3\}$$ but x is 4

Side effects

Several rules of axiomatic semantics are sound only if evaluating an expression does not have any side effects – that is, it does not change the state in any way.

expression with side effects

$\{x = 3\}$ `x := x++` $\{x = 3\}$ but x is 4

Other side-effect behavior that may happen in a realistic program but is abstracted away in basic axiomatic semantics includes:

- memory allocation problems (for example, out of memory)
- input/ouput problems (for example, file not found)
- concurrency problems (for example, race conditions)

Errors and undefined behavior

Even if expression evaluation has no side effects, there are still operations that may lead to an error (or to undefined behavior):

division by zero is an error – unless ∞ is a valid numeric value

overflows and other bounded-number problems are undefined behavior in C/C++

Errors and undefined behavior

Even if expression evaluation has no side effects, there are still operations that may lead to an error (or to undefined behavior):

division by zero is an error – unless ∞ is a valid numeric value

overflows and other bounded-number problems are undefined behavior in C/C++

An error is a behavior where execution cannot continue.

Undefined behavior is a behavior that is not defined by the language standard; hence different language implementations may do different things, all valid.

Errors and undefined behavior

Even if expression evaluation has no side effects, there are still operations that may lead to an error (or to undefined behavior):

division by zero is an error – unless ∞ is a valid numeric value

overflows and other bounded-number problems are undefined behavior in C/C++

An error is a behavior where execution cannot continue.

Undefined behavior is a behavior that is not defined by the language standard; hence different language implementations may do different things, all valid.

```
int main (void) {
  printf ("%d\n",
          (INT_MAX+1) < 0);
  return 0;
}
```

overflow

Valid behavior of this C program:

- Printing `1`
- Printing `0` or any other integer
- Formatting the disk
- ...

(For details see John Regehr's blog posts)

Deductive verification of realistic programs

To reason about verification unfriendly features of realistic programming languages – which introduce "special" behavior – we can do a combination of:

- modeling special behavior using simpler program features
- assuming that special behavior does not occur, restricting the scope of the verification results

Deductive verification of realistic programs

To reason about verification unfriendly features of realistic programming languages – which introduce "special" behavior – we can do a combination of:

- modeling special behavior using simpler program features
- assuming that special behavior does not occur, restricting the scope of the verification results

For example for machine integers:

Model special behavior:

```
procedure plusone(x, y: Integer): (res: Int32)
ensure x + y ≤ Int32.MAX ⟹ res = x + y
ensure x + y > Int32.MAX ⟹ res = error
  res := x + y
```

Assume normal behavior:

```
procedure plusone(x, y: Integer): (res: Integer)
require x + y ≤ Int32.MAX
ensure res = x + y ≤ Int32.MAX
  res := x + y
```

Deductive verification of realistic programs

To reason about verification unfriendly features of realistic programming languages – which introduce "special" behavior – we can do a combination of:

- modeling special behavior using simpler program features
- assuming that special behavior does not occur, restricting the scope of the verification results

For example for machine integers:

Model special behavior:

```
procedure plusone(x, y: Integer): (res: Int32)
ensure x + y ≤ Int32.MAX ⟹ res = x + y
ensure x + y > Int32.MAX ⟹ res = error
  res := x + y
```

Assume normal behavior:

```
procedure plusone(x, y: Integer): (res: Integer)
require x + y ≤ Int32.MAX
ensure res = x + y ≤ Int32.MAX
  res := x + y
```

Formalizing the semantics of realistic programming languages is a very useful exercise to understand rigorously their behavior.

Supporting realistic program features

Supporting realistic program features

Arrays

Arrays

Let's add arrays to Helium:

$$
\begin{aligned}
\textit{Type} &::= \texttt{Array}\langle \textit{Type} \rangle \mid \ldots \\
\textit{Expression} &::= \texttt{a[}\textit{ArithmeticExpression}\texttt{]} \mid \ldots \\
\textit{ArithmeticExpression} &::= \texttt{a.size} \\
\textit{Assignment} &::= \texttt{a[}\textit{ArithmeticExpression}\texttt{] :=} \ \textit{Expression} \\
&\quad \mid \texttt{a.size :=} \ \textit{ArithmeticExpression} \mid \ldots
\end{aligned}
$$

In the operational semantics, the evaluation $[\![\texttt{a}]\!]_s$ of an array variable `a` of type $\texttt{Array}\langle\texttt{T}\rangle$ is a total function $\texttt{Integer} \to \texttt{T}$ – even though we normally use array elements at index `0` (included) to `a.size` (excluded).

Note that arrays cannot be used in parallel assignments, whereas assignments to `size` attributes can.

Arrays: the aliasing problem

Array variable declarations extend the state with mappings to undefined functions of the integers:

$$\frac{\texttt{a} \notin \mathit{domain}(s)}{\langle \texttt{var a: Array}{<}\texttt{T}{>}, s\rangle \rightsquigarrow s \cup \{\texttt{a} \to \bigcup_{x \in \texttt{Integer}} \{x \to ?\}\} \cup \{\texttt{a.size} \to ?\}}$$

Evaluating an array expression evaluates the corresponding state component:

$$\frac{}{[\![\texttt{a[}k\texttt{]}]\!]_s = [\![\texttt{a}]\!]_s([\![k]\!]_s)}$$

Array assignment sets a component of the array function:

$$\frac{}{\langle \texttt{a[}k\texttt{] := }e, s\rangle \rightsquigarrow s[\texttt{a} \mapsto \texttt{a}[[\![k]\!]_s \mapsto [\![e]\!]_s]]}$$

where $f[x \mapsto y]$ denotes a function that is identical to f except possibly at x where $f(x) = y$.

Arrays: axiomatic semantics

The backward substitution rule we used for scalar variables is unsound for arrays.

Arrays: axiomatic semantics

The backward substitution rule we used for scalar variables is unsound for arrays.

For example the backward substitution $\texttt{a[y]}[\texttt{a[x]} \mapsto 0] = \texttt{a[y]}$ since $\texttt{a[x]}$ does not occur in $\texttt{a[y]}$. Thus we could deduce the following Hoare triple:

$$\{\texttt{x} = \texttt{y} \ \wedge \ \texttt{a[y] = 1}\}\ \texttt{a[x] := 0}\ \{\texttt{x} = \texttt{y} \ \wedge \ \texttt{a[y] = 1}\}$$

which is however invalid since $\texttt{a[x] = a[y] = 0}$ after the assignment since $\texttt{x = y}$.

Arrays: axiomatic semantics

The backward substitution rule we used for scalar variables is unsound for arrays.

For example the backward substitution $\texttt{a[y]}[\texttt{a[x]} \mapsto 0] = \texttt{a[y]}$ since $\texttt{a[x]}$ does not occur in $\texttt{a[y]}$. Thus we could deduce the following Hoare triple:

$$\{\texttt{x} = \texttt{y} \ \wedge \ \texttt{a[y] = 1}\}\ \texttt{a[x] := 0}\ \{\texttt{x} = \texttt{y} \ \wedge \ \texttt{a[y] = 1}\}$$

which is however invalid since $\texttt{a[x] = a[y] = 0}$ after the assignment since $\texttt{x = y}$.

Syntactically different array expressions $\texttt{a[x]}$ and $\texttt{a[y]}$ are semantically equivalent when their index expressions $\texttt{x}$ and $\texttt{y}$ have the same value: $\texttt{a[x]}$ and $\texttt{a[y]}$ are aliases.

The aliasing problem happens whenever there may be different syntactic synonyms – for example with references/pointers.

Arrays: axiomatic semantics

A sound backward substitution rule for arrays:

$$\frac{}{\{Q[\mathsf{a} \mapsto \mathsf{a}[k \mapsto E]]\}\ \texttt{a[}k\texttt{] :=}\ E\ \{Q\}}$$

where $\mathsf{a}[x \mapsto y]$ is an array identical to a except possibly at x where it stores value y.

In other words: we have expressed assignment to a single array element as an assignment between whole <u>array variables</u> – to which the usual assignment axiom applies.

This rule looks as simple as the scalar assignment rule; however, it may introduce complexity in the proofs because it introduces different cases according to whether array indexes are aliased or not.

Maximum of array

```
procedure max(a: Array<Integer>): (max: Integer)
```

Maximum of array

```
procedure max(a: Array<Integer>): (max: Integer)

require a.size > 0
ensure ∃ m: Integer (0 ≤ m < a.size ∧ a[m] = max)
ensure ∀ k: Integer (0 ≤ k < a.size ⟹ a[k] ≤ max)
```

Maximum of array

```
procedure max(a: Array<Integer>): (max: Integer)

require a.size > 0
ensure ∃ m: Integer (0 ≤ m < a.size ∧ a[m] = max)
ensure ∀ k: Integer (0 ≤ k < a.size ⟹ a[k] ≤ max)

  var n: Integer
  n, max := 1, a[0]
  while n < a.size
    if a[n] > max
      max := a[n]
    n := n + 1
```

Maximum of array: proof outline

```
procedure max(a: Array<Integer>): (max: Integer)
require a.size > 0
ensure ∃ m: Integer (0 ≤ m < a.size ∧ a[m] = max)
ensure ∀ k: Integer (0 ≤ k < a.size ⟹ a[k] ≤ max)
  var n: Integer
  { 0 ≤ 1 ≤ a.size ∧ a[0] = a[0] ∧ a[0] ≤ a[0] }
  n, max := 1, a[0]
  { 0 ≤ n ≤ a.size ∧ ∃ m: Integer (0 ≤ m < n ∧ a[m] = max)
                   ∧ ∀ k: Integer (0 ≤ k < n ⟹ a[k] ≤ max) }
  while n < a.size
  invariant 0 ≤ n ≤ a.size
  invariant ∃ m: Integer (0 ≤ m < n ∧ a[m] = max)
  invariant ∀ k: Integer (0 ≤ k < n ⟹ a[k] ≤ max)
    { 0 ≤ n < a.size ∧ ∃ m: Integer (0 ≤ m < n ∧ a[m] = max)
                     ∧ ∀ k: Integer (0 ≤ k < n ⟹ a[k] ≤ max) }
    { -1 ≤ n < a.size ∧ (a[n] > max ⟹ ∃ m: Integer (0 ≤ m ≤ n ∧ a[m] = a[n])
                                      ∧ ∀ k: Integer (0 ≤ k ≤ n ⟹ a[k] ≤ a[n]))
                      ∧ (a[n] ≤ max ⟹ ∃ m: Integer (0 ≤ m ≤ n ∧ a[m] = max)
                                      ∧ ∀ k: Integer (0 ≤ k ≤ n ⟹ a[k] ≤ max)) }
    if (a[n] > max) max := a[n]
    { -1 ≤ n < a.size ∧ ∃ m: Integer (0 ≤ m ≤ n ∧ a[m] = max)
                      ∧ ∀ k: Integer (0 ≤ k ≤ n ⟹ a[k] ≤ max) }
    n := n + 1
    { 0 ≤ n ≤ a.size ∧ ∃ m: Integer (0 ≤ m < n ∧ a[m] = max)
                     ∧ ∀ k: Integer (0 ≤ k < n ⟹ a[k] ≤ max) }
```

Array initialization

```
procedure new_array(n, v: Integer): (a: Array<Integer>)
require n ≥ 0
ensure a.size = n ∧ ∀ k: Integer (0 ≤ k < a.size ⟹ a[k] = v)
  var x: Integer
  x, a.size := 0, n
  while x < a.size
  invariant 0 ≤ x ≤ a.size ∧ ∀ k: Integer (0 ≤ k < x ⟹ a[k] = v)
    { 0 ≤ x < a.size ∧ ∀ k: Integer (0 ≤ k < x ⟹ a[k] = v) }
    { -1 ≤ x < a.size ∧ ∀ k: Integer (0 ≤ k ≤ x ⟹ a[x ↦ v][k] = v) }
    a[x] := v
    { -1 ≤ x < a.size ∧ ∀ k: Integer (0 ≤ k ≤ x ⟹ a[k] = v) }
    x := x + 1
    { 0 ≤ x ≤ a.size ∧ ∀ k: Integer (0 ≤ k < x ⟹ a[k] = v) }
```

To prove the following implication we consider two cases:

$$\overbrace{\forall\ \mathtt{k}\colon \mathtt{Integer}\ (0 \le \mathtt{k} < \mathtt{x} \Longrightarrow \mathtt{a[k]} = \mathtt{v})}^{A} \Longrightarrow \overbrace{\forall\ \mathtt{k}\colon \mathtt{Integer}\ (0 \le \mathtt{k} \le \mathtt{x} \Longrightarrow \mathtt{a[x} \mapsto \mathtt{v][k]} = \mathtt{v})}^{B}$$

1. $\mathtt{k} \neq \mathtt{x}$: $\mathtt{a} = \mathtt{a[x} \mapsto \mathtt{v]}$, and hence A is the same as B over the range $0 \le \mathtt{k} < x$
2. $\mathtt{k} = \mathtt{x}$: $\mathtt{a[x} \mapsto \mathtt{v][k]} = \mathtt{a[x} \mapsto \mathtt{v][x]} = \mathtt{v}$, and the implication in B reduces to $\cdots \Longrightarrow \top$

Dafny: maximum of array

```
method max(a: array<int>) returns(max: int)
  requires a != null;     // array is a reference type
  requires a.Length > 0; // a.size in Helium
     // :: is required and its scope extends until ;
  ensures forall k: int :: 0 <= k < a.Length ==> a[k] <= max;
  ensures exists k: int :: 0 <= k < a.Length && a[k] == max;
{
  ghost var m: int;
  var j: int;
  j, max := 0, a[0];
  m := 0;
  while j < a.Length
    invariant 0 <= j <= a.Length;
    invariant forall k: int :: 0 <= k < j ==> a[k] <= max;
    invariant 0 <= m < a.Length && a[m] == max;
    decreases a.Length - j;
  {
    if a[j] > max
    { max := a[j]; m := j; }
    j := j + 1;
  }
}
```

More complex examples

More complex examples of fully annotated and verified programs:

- the description of the first assignment includes pointers to online examples and documentation; in particular, the Docker image with Dafny includes the verified Dafny examples we presented in the slides, as well as a few more
- Loop Invariants: Analysis, Classification, and Examples surveys a variety of algorithms and their loop invariants for functional correctness
- Rotation of Sequences: Algorithms and Proofs describes implementation, specification, and correctness proofs of four algorithms, of increasing complexity, to rotate an array
- VerifyThis is a yearly program verification competition; complex algorithms verified using different tools are available in its archive of challenge problems

Supporting realistic program features

Procedure calls

Procedure calls in Helium

The next language feature we add to Helium is procedure call.

$$\begin{aligned} \textit{Statement} &::= \textit{ProcedureCall} \mid \ldots \\ \textit{ProcedureCall} &::= \mathtt{u}_1\,\mathtt{,}\,\ldots\,\mathtt{,}\ \mathtt{u}_n\ \mathtt{:=}\ \mathtt{p(}\textit{Expression}_1\,\mathtt{,}\,\ldots\,\mathtt{,}\textit{Expression}_m\mathtt{)} \end{aligned}$$

output written to — callee procedure (callee) — actual input arguments

Where `p` is the name of a procedure declared as:

```
procedure p (in₁: T₁, ..., inₘ: Tₘ): (out₁: U₁, ..., outₙ: Uₘ)
require P
modify F
ensure Q
  B
```

Procedure call: operational semantics

The operational semantics of procedure call reduces to executing the callee's body and storing the results in the variables local to the caller.

$$\frac{\overline{s} = s[\mathtt{in}_1, \ldots, \mathtt{in}_m \mapsto [\![E_1]\!]_s, \ldots, [\![E_m]\!]_s] \quad [\![P]\!]_{\overline{s}} \quad \langle B, \overline{s}\rangle \leadsto s'}{\langle \mathtt{u}_1, \ldots, \mathtt{u}_n := \mathtt{p}(E_1, \ldots, E_m), s\rangle \leadsto s'[\mathtt{u}_1, \ldots, \mathtt{u}_n \mapsto \mathtt{out}_1, \ldots, \mathtt{out}_n]}$$

In the operational semantics:

- $\overline{s}$ is the state s augmented with fresh input variables for `p`'s execution, initialized to the actual arguments $E_1, \ldots, E_m$
- `p`'s precondition P must hold for the call to be conforming to `p`'s specification
- after execution, the output values are stored in $\mathtt{u}_1, \ldots, \mathtt{u}_m$

We slightly simplify the notation – for example, we assume `p`'s arguments have names different from any other variables at the call context, and $\mathtt{out}_k$ denotes the value stored in $\mathtt{out}_k$ after `p` executes – but the overall meaning is the usual one.

Procedure call: operational semantics

The operational semantics of procedure call reduces to executing the callee's body and storing the results in the variables local to the caller.

$$\frac{\overline{s} = s[\mathtt{in}_1, \ldots, \mathtt{in}_m \mapsto [\![E_1]\!]_s, \ldots, [\![E_m]\!]_s] \quad \neg[\![P]\!]_{\overline{s}}}{\langle \mathtt{u}_1\mathtt{,\ \ldots,\ u}_n\ \mathtt{:=\ p(}E_1\mathtt{,\ \ldots,}E_m\mathtt{)}, s\rangle \rightsquigarrow \textbf{error}}$$

In the operational semantics:

- $\overline{s}$ is the state s augmented with fresh input variables for `p`'s execution, initialized to the actual arguments $E_1, \ldots, E_m$
- `p`'s precondition P must hold for the call to be conforming to `p`'s specification
- after execution, the output values are stored in $\mathtt{u}_1, \ldots, \mathtt{u}_m$

To complete the operational semantics, execution halts with an error when a call's actual arguments violate the callee's precondition.

Procedure call: axiomatic semantics

There are two main options to define the axiomatic semantics of procedure call:

Inlining semantics | Modular semantics

The inlining semantics replicates the operational semantics in a declarative way, by inlining the callee's body into the call context – using fresh local variables to avoid name clashes with other variables at the call context.

The modular semantics ignores the callee's body and uses only its specification to define the effects of the call at the call context. It is called "modular" because it abstracts implementations by using their specifications.

Most deductive verification tools use the modular semantics.

Yay! 👍: the modular semantics permits scalability

Nay! 👎: it requires writing detailed specifications of procedures

Procedure call inlining semantics

Under the inlining call semantics, proving a procedure call boils down to proving:

1. The caller's pre-state P' satisfies the callee's precondition P
2. The callee's body leads to a post-state Q' when it is executed in state P'

$$
\frac{\{P'\}\ \texttt{var in}_1\texttt{, ..., in}_m \texttt{ := } E_1\texttt{, ...,}E_m\ \{P\} \qquad \{P'\}\ \begin{array}{l}\texttt{var in}_1\texttt{, ..., in}_m \texttt{ := } E_1\texttt{, ...,}E_m\\ \texttt{var out}_1\texttt{, ..., out}_n\\ B\\ \texttt{u}_1\texttt{, ..., u}_n \texttt{ := out}_1\texttt{, ..., out}_n\end{array}\ \{Q'\}}{\{P'\}\ \texttt{u}_1\texttt{, ..., u}_n \texttt{ := p(}E_1\texttt{, ...,}E_m\texttt{)}\ \{Q'\}}
$$

There are a number of details that we gloss over:

- `var x := e` is a shorthand for `var x: T; x := e`
- We assume name clashes are taken care of

The callee's postcondition Q does not play any role in the inlining semantics.

Procedure call modular semantics

Under the modular call semantics, proving a procedure call boils down to proving:

1. The caller's pre-state P' satisfies the callee's precondition P
2. The callee's post-state Q satisfies the caller's postcondition Q'

$$\frac{\{P'\}\ \texttt{var in}_1\texttt{, ..., in}_m\texttt{ := }E_1\texttt{, ...,}E_m\ \{P\} \quad \{Q\}\ \texttt{u}_1\texttt{, ..., u}_n\texttt{ := out}_1\texttt{, ..., out}_n\ \{Q'\}}{\{P'\}\ \texttt{u}_1\texttt{, ..., u}_n\texttt{ := p(}E_1\texttt{, ...,}E_m\texttt{)}\ \{Q'\}}$$

There are a number of details that we gloss over:

- Q may refer to input arguments $\texttt{in}_1, \ldots, \texttt{in}_m$; may use expressions with **`old`**; and may modify global variables in F. These result in a complete rule that is more complex than the one shown, since it has to keep track of these other references by means of auxiliary variables.

The callee's body B does not play any role in the modular semantics.

Inlining vs. modular semantics

All three variants of `abs` are correct:

```
procedure abs_1
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_2
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure x > 0 ⟹ abs = x
ensure x < 0 ⟹ abs = -x
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_3
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
ensure x > 0 ⟹ abs = x
  if x > 0
    abs := x
  else abs := 10
```

	INLINE			MODULAR		
	_1	_2	_3	_1	_2	_3
`{x = 3} y := abs(x) {y = 3}`						

Inlining vs. modular semantics

All three variants of `abs` are correct:

```
procedure abs_1
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_2
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure x > 0 ⟹ abs = x
ensure x < 0 ⟹ abs = -x
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_3
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
ensure x > 0 ⟹ abs = x
  if x > 0
    abs := x
  else abs := 10
```

	INLINE			MODULAR		
	_1	_2	_3	_1	_2	_3
`{x = 3} y := abs(x) {y = 3}`	✓	✓	✓	✗	✓	✓
`{x = -3} y := abs(x) {y = 3}`						

Inlining vs. modular semantics

All three variants of `abs` are correct:

```
procedure abs_1
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_2
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure x > 0 ⟹ abs = x
ensure x < 0 ⟹ abs = -x
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_3
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
ensure x > 0 ⟹ abs = x
  if x > 0
    abs := x
  else abs := 10
```

	INLINE _1	INLINE _2	INLINE _3	MODULAR _1	MODULAR _2	MODULAR _3
{x = 3} y := abs(x) {y = 3}	✔	✔	✔	✘	✔	✔
{x = -3} y := abs(x) {y = 3}	✔	✔	✘	✘	✔	✘
{ true } y := abs(x) {y ≥ x}						

Inlining vs. modular semantics

All three variants of `abs` are correct:

```
procedure abs_1
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_2
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure x > 0 ⟹ abs = x
ensure x < 0 ⟹ abs = -x
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_3
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
ensure x > 0 ⟹ abs = x
  if x > 0
    abs := x
  else abs := 10
```

	INLINE			MODULAR		
	_1	_2	_3	_1	_2	_3
{x = 3} y := abs(x) {y = 3}	✔	✔	✔	✘	✔	✔
{x = -3} y := abs(x) {y = 3}	✔	✔	✘	✘	✔	✘
{ true } y := abs(x) {y ≥ x}	✔	✔	✔	✘	✔	✔
{ x > 0 } y := abs(x) {y = x}						

Inlining vs. modular semantics

All three variants of `abs` are correct:

```
procedure abs_1
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_2
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure x > 0 ⟹ abs = x
ensure x < 0 ⟹ abs = -x
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_3
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
ensure x > 0 ⟹ abs = x
  if x > 0
    abs := x
  else abs := 10
```

	INLINE			MODULAR		
	_1	_2	_3	_1	_2	_3
{x = 3} y := abs(x) {y = 3}	✔	✔	✔	✘	✔	✔
{x = -3} y := abs(x) {y = 3}	✔	✔	✘	✘	✔	✘
{ true } y := abs(x) {y ≥ x}	✔	✔	✔	✘	✔	✔
{ x > 0 } y := abs(x) {y = x}	✔	✔	✔	✘	✔	✔
{ x < 0 } y := abs(x) {y = -x}						

Inlining vs. modular semantics

All three variants of `abs` are correct:

```
procedure abs_1
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_2
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure x > 0 ⟹ abs = x
ensure x < 0 ⟹ abs = -x
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_3
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
ensure x > 0 ⟹ abs = x
  if x > 0
    abs := x
  else abs := 10
```

	INLINE			MODULAR		
	_1	_2	_3	_1	_2	_3
{x = 3} y := abs(x) {y = 3}	✔	✔	✔	✘	✔	✔
{x = -3} y := abs(x) {y = 3}	✔	✔	✘	✘	✔	✘
{ true } y := abs(x) {y ≥ x}	✔	✔	✔	✘	✔	✔
{ x > 0 } y := abs(x) {y = x}	✔	✔	✔	✘	✔	✔
{ x < 0 } y := abs(x) {y = -x}	✔	✔	✘	✘	✔	✘
{ x = 0 } y := abs(x) {y ≥ x}						

Inlining vs. modular semantics

All three variants of `abs` are correct:

```
procedure abs_1
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_2
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure x > 0 ⟹ abs = x
ensure x < 0 ⟹ abs = -x
  if x > 0
    abs := x
  else abs := -x
```

```
procedure abs_3
  (x: Integer):
    (abs: Integer)
require x ≠ 0
ensure abs > 0
ensure x > 0 ⟹ abs = x
  if x > 0
    abs := x
  else abs := 10
```

	INLINE _1	INLINE _2	INLINE _3	MODULAR _1	MODULAR _2	MODULAR _3
{x = 3} y := abs(x) {y = 3}	✔	✔	✔	✘	✔	✔
{x = -3} y := abs(x) {y = 3}	✔	✔	✘	✘	✔	✘
{ true } y := abs(x) {y ≥ x}	✔	✔	✔	✘	✔	✔
{ x > 0 } y := abs(x) {y = x}	✔	✔	✔	✘	✔	✔
{ x < 0 } y := abs(x) {y = -x}	✔	✔	✘	✘	✔	✘
{ x = 0 } y := abs(x) {y ≥ x}	✘	✘	✘	✘	✘	✘

Advantages of the modular semantics

The main reason that deductive verifiers use the modular semantics is that it scales:

- under the inline semantics, we prove the same procedure again and again in each call context, leading to a monolithic *VC*
- under the modular semantics, we prove the procedure once when it is declared, leading to a *VC* that is split into independent chunks

Another important reason is that only the modular semantics supports proofs of recursive procedures:

- under the inline semantics, a recursive calls may require an unbounded number of inlinings
- under the modular semantics, a recursive call is just like any other call, where we summarize the effects of the call using the callee's specification

Loop invariants vs. loop unrolling

Using the modular semantics for procedure calls is also consistent with using loop invariants to summarize loop iterations (instead of unrolling the loop body an indefinite number of times).

Unrolling is what the operational semantics does, which is why it is hard to reason about loops using operational semantics.

Later in the course we'll see symbolic execution, a static analysis technique which is basically based on the inlining and unrolling semantics – even though it is symbolic.

Procedures: semantics with calls

The axiom that defines the correctness of a procedure (definition) remains valid when the procedure's body may itself call procedures.

$$\frac{\{P\}\ B\ \{Q\} \qquad \mathcal{F}(B) \cap \mathcal{G} \subseteq F}{\texttt{procedure proc (in: T): (out: T) } \textbf{require}\ P\ \textbf{modify}\ F\ \textbf{ensure}\ Q\ B}$$

However, $\mathcal{F}(B)$ now includes not only all global variables that appear to the left-hand side of assignments, but also those that belong to frame specifications F' of any procedures called within B.

Weakest precondition of procedure call

Just like the inference rule from which it is derived, the weakest precondition of procedure calls consists of two clauses.

$$
\begin{aligned}
&\mathbf{wp}(\texttt{u}_1\texttt{, ..., u}_n \texttt{ := p(}E_1\texttt{, ..., }E_m\texttt{)}, Q') = \\
&\qquad\qquad \mathbf{wp}(\texttt{var in}_1\texttt{, ..., in}_m \texttt{ := } E_1\texttt{, ..., }E_m, P) \\
&\qquad \wedge \quad \forall\, \mathbf{f} \bullet \mathbf{wp}\left(\begin{array}{l}\texttt{var out}_1\texttt{, ..., out}_n \\ \texttt{assume } Q \\ \texttt{u}_1\texttt{, ..., u}_n \texttt{ := out}_1\texttt{, ..., out}_n\end{array}, Q'\right)[F \mapsto \mathbf{f}]
\end{aligned}
$$

The second clause is within the scope of a universal quantification on a set **f** of variables that replace each program variable mentioned in the callee `p`'s frame F. This effectively "forgets" all that is known about the frame variables at the call context except the postcondition – similarly to how we forget all about previous loop iterations except the loop invariant.

Nondeterministic assignment

Since it is useful to be able to model the modular semantics directly in the code, we introduce a special nondeterministic value ?:

```
v := ?
```

assigns a nondeterministic value to `v`, effectively forgetting any previous value and not assuming anything specific about the new value.

This nondeterministic assignment is sometimes called `havoc` or `shuffle` in programming languages explicitly designed for verification (such as Boogie).

Nondeterministic assignment: semantics

In the operational semantics, $[\![?]\!]_s$ evaluates to the special value `?` used for uninitialized variable.

In the axiomatic semantics:

$$\frac{\{P\}\ \texttt{var new\_v: T; v := new\_v}\ \{Q\}}{\{P\}\ \texttt{v := ?}\ \{Q\}}$$

where `new_v` is a fresh variable of the same type `T` as `v`.

Correspondingly, we can express the weakest precondition of nondeterministic assignment:

$$\begin{aligned}\mathbf{wp}(\texttt{v := ?}, Q) &= \mathbf{wp}(\texttt{var new\_v: T; v := new\_v}, Q)\\ &= \forall\, \mathit{new\_v} \bullet Q[\texttt{v} \mapsto \mathit{new\_v}]\end{aligned}$$

Axiomatic semantics of calls using nondeterminism

```
procedure p (in₁: T₁, ..., inₘ: Tₘ): (out₁: U₁, ..., outₙ: Uₘ)
require P
modify f₁, ..., f_f
ensure Q
  B
```

Under the modular semantics, the usual generic call to `p`:

```
u₁, ..., uₙ := p(E₁, ..., Eₘ)
```

is equivalent to:

```
var in₁, ..., inₘ := E₁, ..., Eₘ   // initialize actual arguments
assert P                           // check precondition
var old₁, ..., oldₒ := O₁, ..., Oₒ // save old expressions
var out₁, ..., outₙ                // make room for procedure output
var f₁, ..., f_f := ?, ..., ?      // forget modified variables in F
assume Q_old                // assume postcondition with old(Oₖ) ↦ oldₖ
u₁, ..., uₙ := out₁, ..., outₙ     // store final result
```

Lithium

We call Lithium the language Helium extended with:

1. arrays
2. procedures
3. procedure calls

Supporting realistic program features

References/pointers

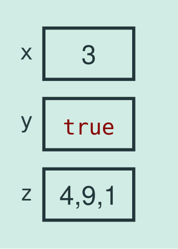
Value types

All types in Lithium are value types: a variable of type `T` corresponds to a memory location where an instance of `T` is stored.

The part of the memory where variables of value types are stored is usually called store or stack.

```
var x: Integer
var y: Boolean
var z: Array<Integer>
x, y, z := 3, true, (4,9,1)
```

Store

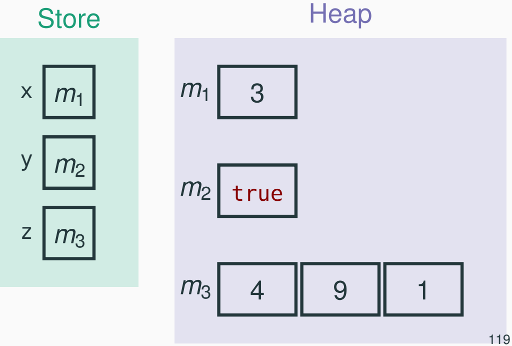


Reference types

We extend Lithium with reference types: a variable of type `ref T` corresponds to a memory location that stores the memory address of another memory location where an instance of `T` is stored.

The part of the memory where the content referenced by variables of reference types are stored is usually called heap.

```
var x: ref Integer
var y: ref Boolean
var z: ref Array<Integer>
  // store with indirection
[x] := new 3
[y] := new true
[z] := new (4,9,1)
```



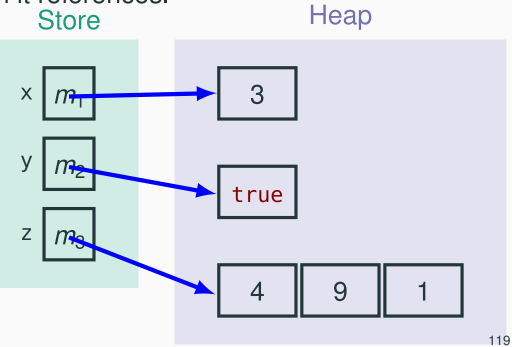
Reference types

We extend Lithium with reference types: a variable of type `ref T` corresponds to a memory location that stores the memory address of another memory location where an instance of `T` is stored.

The part of the memory where the content referenced by variables of reference types are stored is usually called heap.

Since the absolute values of the memory addresses of references does not matter, we represent a reference variable with an arrow that points to the memory location it references.

```
var x: ref Integer
var y: ref Boolean
var z: ref Array<Integer>
  // store with indirection
[x] := new 3
[y] := new true
[z] := new (4,9,1)
```



Adding references/pointers to Lithium

We actually add a form of pointers to Lithium; references are just syntactic sugar for pointer indirection.

$$
\begin{aligned}
\textit{Type} &::= \texttt{ref}\ \textit{Type} \mid \ldots \\
\textit{ReferenceExpression} &::= \texttt{r} \in \textit{ReferenceVariables} \mid \texttt{new}\ \textit{Type} \mid \texttt{new}\ \textit{Expression} \\
&\quad\mid \textit{ReferenceExpression}\texttt{.field} \\
&\quad\mid \textit{ReferenceExpression} + \textit{ReferenceExpression} \\
&\quad\mid \textit{ReferenceExpression} - \textit{ReferenceExpression} \\
&\quad\mid \ldots \\
\textit{Expression} &::= \texttt{[}\textit{ReferenceExpression}\texttt{]} \mid \ldots \\
\textit{Assignment} &::= \texttt{[}\textit{ReferenceExpression}\texttt{]}\ \texttt{:=}\ \textit{Expression} \\
\textit{ActiveStatement} &::= \texttt{dispose}\ \textit{ReferenceExpression} \mid \ldots
\end{aligned}
$$

Adding references/pointers to Lithium

We actually add a form of pointers to Lithium; references are just syntactic sugar for pointer indirection.

allocation expression (uninitialized)

allocation expression (w/ initialization)

variable of `ref` type

$$
\begin{aligned}
\textit{Type} &::= \texttt{ref}\ \textit{Type} \mid \ldots \\
\textit{ReferenceExpression} &::= \texttt{r} \in \textit{ReferenceVariables} \mid \textbf{new}\ \textit{Type} \mid \textbf{new}\ \textit{Expression} \\
&\mid \textit{ReferenceExpression}\texttt{.field} \\
&\mid \textit{ReferenceExpression} + \textit{ReferenceExpression} \\
&\mid \textit{ReferenceExpression} - \textit{ReferenceExpression} \\
&\mid \ldots \\
\textit{Expression} &::= \texttt{[}\textit{ReferenceExpression}\texttt{]} \mid \ldots \\
\textit{Assignment} &::= \texttt{[}\textit{ReferenceExpression}\texttt{]}\ \texttt{:=}\ \textit{Expression} \\
\textit{ActiveStatement} &::= \textbf{dispose}\ \textit{ReferenceExpression} \mid \ldots
\end{aligned}
$$

field offset

pointer arithmetic

dereference/indirection

deallocation

heap assignment

State with references

The program state should capture the whole memory content. So far that was just the store; now it must also include a model of the heap.

The store maps variables to values:

$$s: \quad \mathit{Variables} \to \mathit{Values}$$

The heap maps addresses to values:

$$h: \quad \mathit{Addresses} \to \mathit{Values}$$

We assume that:

- h is a partial function, defined only for addresses that correspond to allocated memory
- an address is just a particular type of values
- the special address value `nil` (`null` in other languages) is such that it is never used for allocated memory: `nil` $\notin \mathit{domain}(h)$

Finally the state is just a pair (s, h).

Reference expressions: operational semantics

The evaluation of a reference expression gives an address.

$$
\begin{aligned}
[\![\mathtt{r}]\!]_{(s,h)} &= s(\mathtt{r}) & \mathtt{r} &\in \mathit{ReferenceVariables} \\
[\![r_1 + r_2]\!]_{(s,h)} &= [\![r_1]\!]_{(s,h)} + [\![r_2]\!]_{(s,h)} & r_1, r_2 &\in \mathit{ReferenceExpression} \\
[\![r.\mathtt{field}]\!]_{(s,h)} &= [\![r]\!]_{(s,h)} + \mathit{offset}(\mathtt{field}) & r &\in \mathit{ReferenceExpression}
\end{aligned}
$$

Reference expressions: operational semantics

The evaluation of a reference expression gives an address.

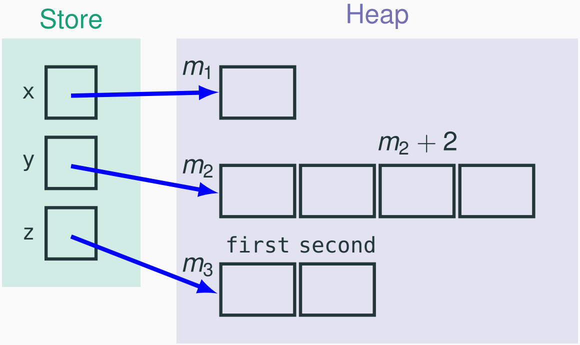
$$\llbracket \mathtt{r} \rrbracket_{(s,h)} = s(\mathtt{r}) \qquad \mathtt{r} \in \mathit{ReferenceVariables}$$

$$\llbracket r_1 + r_2 \rrbracket_{(s,h)} = \llbracket r_1 \rrbracket_{(s,h)} + \llbracket r_2 \rrbracket_{(s,h)} \qquad r_1, r_2 \in \mathit{ReferenceExpression}$$

$$\llbracket r.\mathtt{field} \rrbracket_{(s,h)} = \llbracket r \rrbracket_{(s,h)} + \mathit{offset}(\mathtt{field}) \qquad r \in \mathit{ReferenceExpression}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

r	$\llbracket r \rrbracket_{(s,h)}$
`x`	
`x + 1`	
`y + 2`	
`z.second`	



Reference expressions: operational semantics

The evaluation of a reference expression gives an address.

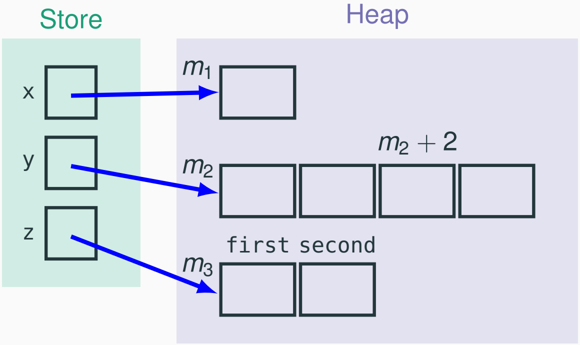
$$[\![\mathtt{r}]\!]_{(s,h)} = s(\mathtt{r}) \qquad \mathtt{r} \in \mathit{ReferenceVariables}$$

$$[\![r_1 + r_2]\!]_{(s,h)} = [\![r_1]\!]_{(s,h)} + [\![r_2]\!]_{(s,h)} \qquad r_1, r_2 \in \mathit{ReferenceExpression}$$

$$[\![r.\mathtt{field}]\!]_{(s,h)} = [\![r]\!]_{(s,h)} + \mathit{offset}(\mathtt{field}) \qquad r \in \mathit{ReferenceExpression}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

r	$[\![r]\!]_{(s,h)}$
x	m_1
x + 1	
y + 2	
z.second	



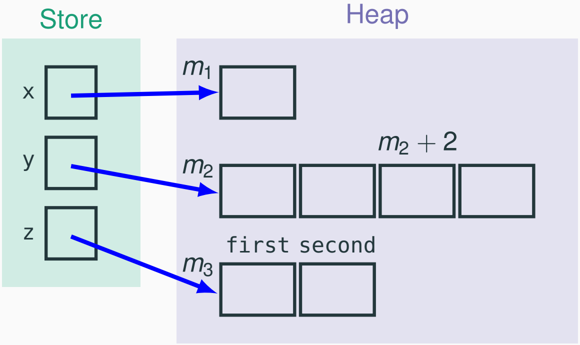
Reference expressions: operational semantics

The evaluation of a reference expression gives an address.

$$
\begin{array}{rclr}
[\![\mathtt{r}]\!]_{(s,h)} & = & s(\mathtt{r}) & \mathtt{r} \in \mathit{ReferenceVariables} \\
[\![r_1 + r_2]\!]_{(s,h)} & = & [\![r_1]\!]_{(s,h)} + [\![r_2]\!]_{(s,h)} & r_1, r_2 \in \mathit{ReferenceExpression} \\
[\![r.\mathtt{field}]\!]_{(s,h)} & = & [\![r]\!]_{(s,h)} + \mathit{offset}(\mathtt{field}) & r \in \mathit{ReferenceExpression}
\end{array}
$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

r	$[\![r]\!]_{(s,h)}$
`x`	m_1
`x + 1`	$m_1 + 1$
`y + 2`	
`z.second`	



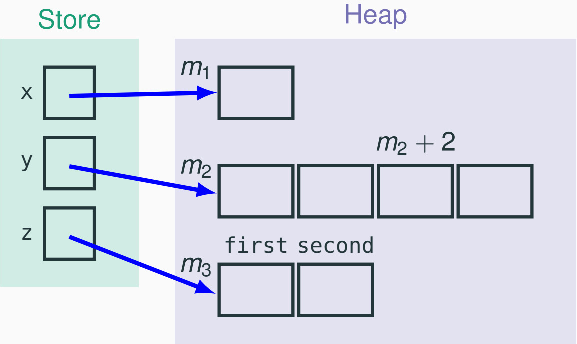
Reference expressions: operational semantics

The evaluation of a reference expression gives an address.

$$
\begin{aligned}
[\![\mathtt{r}]\!]_{(s,h)} &= s(\mathtt{r}) && \mathtt{r} \in \mathit{ReferenceVariables} \\
[\![r_1 + r_2]\!]_{(s,h)} &= [\![r_1]\!]_{(s,h)} + [\![r_2]\!]_{(s,h)} && r_1, r_2 \in \mathit{ReferenceExpression} \\
[\![r.\mathtt{field}]\!]_{(s,h)} &= [\![r]\!]_{(s,h)} + \mathit{offset}(\mathtt{field}) && r \in \mathit{ReferenceExpression}
\end{aligned}
$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

r	$[\![r]\!]_{(s,h)}$
`x`	m_1
`x + 1`	$m_1 + 1$
`y + 2`	$m_2 + 2$
`z.second`	



Reference expressions: operational semantics

The evaluation of a reference expression gives an address.

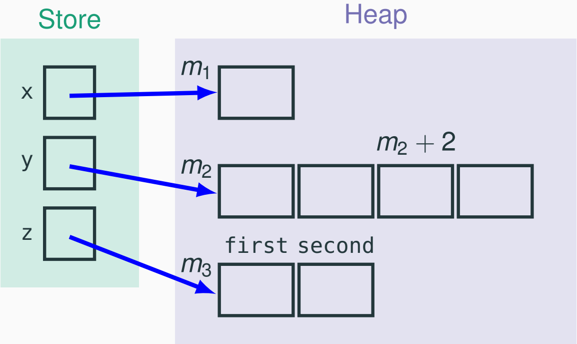
$$[\![\mathrm{r}]\!]_{(s,h)} = s(\mathrm{r}) \qquad \mathrm{r} \in \mathit{ReferenceVariables}$$

$$[\![r_1 + r_2]\!]_{(s,h)} = [\![r_1]\!]_{(s,h)} + [\![r_2]\!]_{(s,h)} \qquad r_1, r_2 \in \mathit{ReferenceExpression}$$

$$[\![r.\mathtt{field}]\!]_{(s,h)} = [\![r]\!]_{(s,h)} + \mathit{offset}(\mathtt{field}) \qquad r \in \mathit{ReferenceExpression}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

r	$[\![r]\!]_{(s,h)}$
`x`	m_1
`x + 1`	$m_1 + 1$
`y + 2`	$m_2 + 2$
`z.second`	$m_3 + 1$



Dereferencing: operational semantics

Dereferencing a reference expression gives a value – but fails if the reference expression is not a valid memory address.

$$[\![[\, r\,]]\!]_{(s,h)} = \begin{cases} h([\![r]\!]_{(s,h)}) & [\![r]\!]_{(s,h)} \in \mathit{domain}(h) \\ \textbf{error} & \text{otherwise} \end{cases}$$

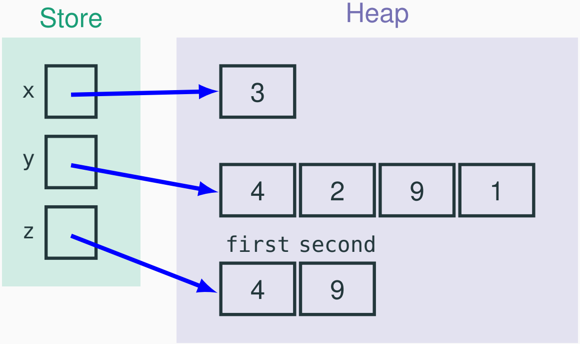
Dereferencing: operational semantics

Dereferencing a reference expression gives a value – but fails if the reference expression is not a valid memory address.

$$[\![\, [\, r \,] \,]\!]_{(s,h)} = \begin{cases} h([\![r]\!]_{(s,h)}) & [\![r]\!]_{(s,h)} \in \mathit{domain}(h) \\ \textbf{error} & \text{otherwise} \end{cases}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

d	$[\![d]\!]_{(s,h)}$
[x]	
[x + 1]	
[y + 2]	
[z.second]	



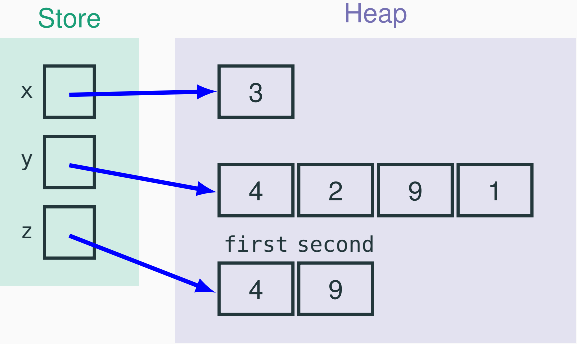
Dereferencing: operational semantics

Dereferencing a reference expression gives a value – but fails if the reference expression is not a valid memory address.

$$[\![\,[r]\,]\!]_{(s,h)} = \begin{cases} h([\![r]\!]_{(s,h)}) & [\![r]\!]_{(s,h)} \in \mathit{domain}(h) \\ \textbf{error} & \text{otherwise} \end{cases}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

d	$[\![d]\!]_{(s,h)}$
`[x]`	3
`[x + 1]`	
`[y + 2]`	
`[z.second]`	



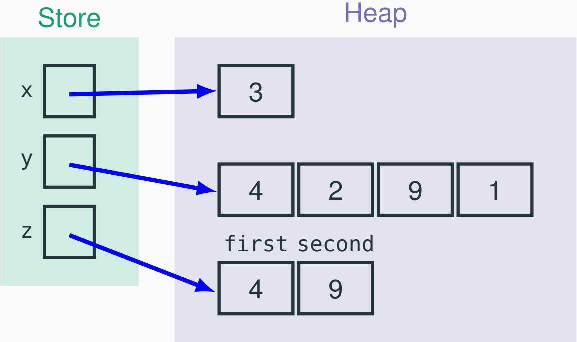
Dereferencing: operational semantics

Dereferencing a reference expression gives a value – but fails if the reference expression is not a valid memory address.

$$[\![[r]]\!]_{(s,h)} = \begin{cases} h([\![r]\!]_{(s,h)}) & [\![r]\!]_{(s,h)} \in \mathit{domain}(h) \\ \textbf{error} & \text{otherwise} \end{cases}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

d	$[\![d]\!]_{(s,h)}$
`[x]`	3
`[x + 1]`	**error**
`[y + 2]`	
`[z.second]`	



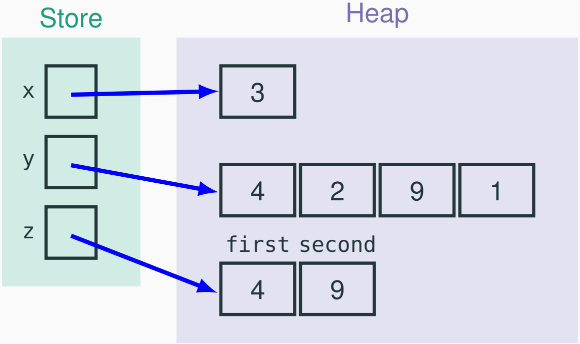
Dereferencing: operational semantics

Dereferencing a reference expression gives a value – but fails if the reference expression is not a valid memory address.

$$[\![\, \texttt{[}r\texttt{]} \,]\!]_{(s,h)} = \begin{cases} h([\![r]\!]_{(s,h)}) & [\![r]\!]_{(s,h)} \in \mathit{domain}(h) \\ \textbf{error} & \text{otherwise} \end{cases}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

d	$[\![d]\!]_{(s,h)}$
`[x]`	3
`[x + 1]`	**error**
`[y + 2]`	9
`[z.second]`	



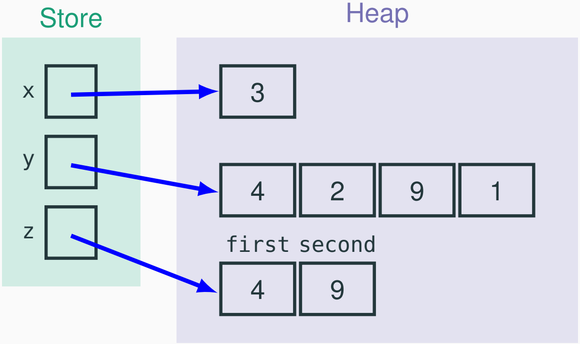
Dereferencing: operational semantics

Dereferencing a reference expression gives a value – but fails if the reference expression is not a valid memory address.

$$[\![\, [r] \,]\!]_{(s,h)} = \begin{cases} h([\![r]\!]_{(s,h)}) & [\![r]\!]_{(s,h)} \in \mathit{domain}(h) \\ \textbf{error} & \text{otherwise} \end{cases}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

d	$[\![d]\!]_{(s,h)}$
`[x]`	3
`[x + 1]`	**error**
`[y + 2]`	9
`[z.second]`	9



Fetch assignments: operational semantics

Assigning a dereferenced expression to a variable is called fetch assignment. This behaves like a regular assignment except that the whole computation may fail if dereferencing fails.

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \in \mathit{domain}(h)}{\langle \mathtt{v} \ \mathtt{:=} \ \mathtt{[}E\mathtt{]}, (s,h)\rangle \rightsquigarrow (s[\mathtt{v} \mapsto h(e)], h)} \qquad \frac{[\![E]\!]_{(s,h)} = e \quad e \notin \mathit{domain}(h)}{\langle \mathtt{v} \ \mathtt{:=} \ \mathtt{[}E\mathtt{]}, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

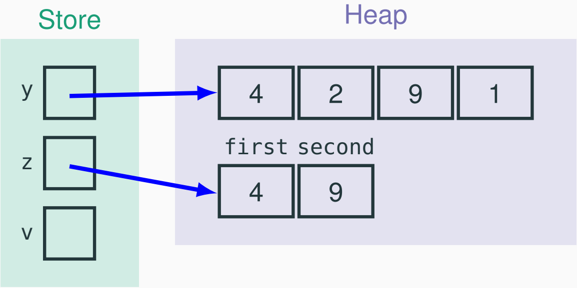
Fetch assignments: operational semantics

Assigning a dereferenced expression to a variable is called fetch assignment. This behaves like a regular assignment except that the whole computation may fail if dereferencing fails.

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \in domain(h)}{\langle \texttt{v := [}E\texttt{]}, (s,h)\rangle \rightsquigarrow (s[\texttt{v} \mapsto h(e)], h)} \qquad \frac{[\![E]\!]_{(s,h)} = e \quad e \notin domain(h)}{\langle \texttt{v := [}E\texttt{]}, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var y: ref Array<Integer>
var a: ref Pair
var v: ref Integer
```

r	`v := [`r`]`
`y + 2`	
`z.first`	
`y + 5`	
`z.second + 1`	



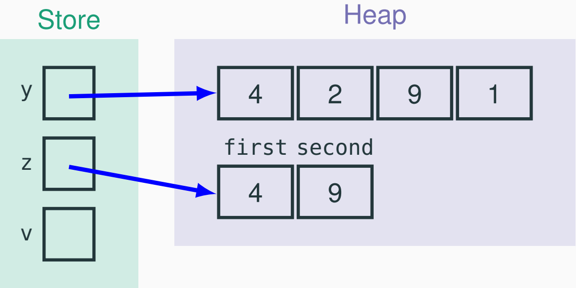
Fetch assignments: operational semantics

Assigning a dereferenced expression to a variable is called fetch assignment. This behaves like a regular assignment except that the whole computation may fail if dereferencing fails.

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \in domain(h)}{\langle \mathtt{v} \ \mathtt{:=}\ \mathtt{[}E\mathtt{]}, (s,h)\rangle \rightsquigarrow (s[\mathtt{v} \mapsto h(e)], h)} \qquad \frac{[\![E]\!]_{(s,h)} = e \quad e \notin domain(h)}{\langle \mathtt{v} \ \mathtt{:=}\ \mathtt{[}E\mathtt{]}, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var y: ref Array<Integer>
var a: ref Pair
var v: ref Integer
```

r	`v := [`r`]`
`y + 2`	✔ `v = 9`
`z.first`	
`y + 5`	
`z.second + 1`	



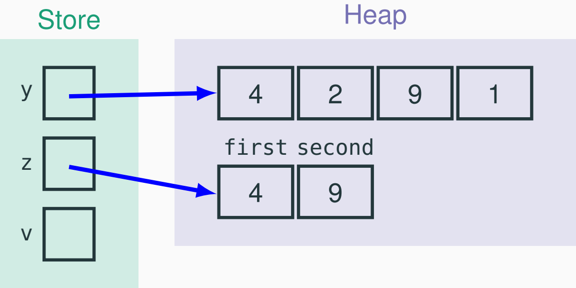
Fetch assignments: operational semantics

Assigning a dereferenced expression to a variable is called fetch assignment. This behaves like a regular assignment except that the whole computation may fail if dereferencing fails.

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \in domain(h)}{\langle \texttt{v := [}E\texttt{]}, (s,h)\rangle \rightsquigarrow (s[\texttt{v} \mapsto h(e)], h)} \qquad \frac{[\![E]\!]_{(s,h)} = e \quad e \notin domain(h)}{\langle \texttt{v := [}E\texttt{]}, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var y: ref Array<Integer>
var a: ref Pair
var v: ref Integer
```

r	`v := [`r`]`
`y + 2`	✔ `v = 9`
`z.first`	✔ `v = 4`
`y + 5`	
`z.second + 1`	



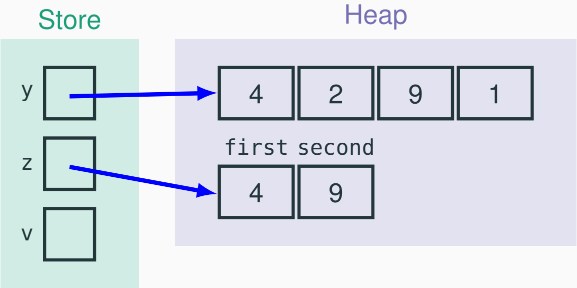
Fetch assignments: operational semantics

Assigning a dereferenced expression to a variable is called fetch assignment. This behaves like a regular assignment except that the whole computation may fail if dereferencing fails.

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \in domain(h)}{\langle \texttt{v := [}E\texttt{]}, (s,h)\rangle \rightsquigarrow (s[\texttt{v} \mapsto h(e)], h)} \qquad \frac{[\![E]\!]_{(s,h)} = e \quad e \notin domain(h)}{\langle \texttt{v := [}E\texttt{]}, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var y: ref Array<Integer>
var a: ref Pair
var v: ref Integer
```

r	`v := [`r`]`
`y + 2`	✓ `v = 9`
`z.first`	✓ `v = 4`
`y + 5`	✗
`z.second + 1`	



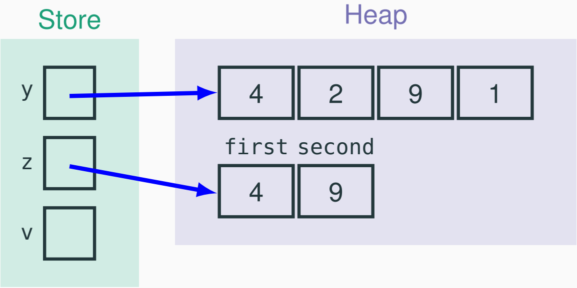
Fetch assignments: operational semantics

Assigning a dereferenced expression to a variable is called fetch assignment. This behaves like a regular assignment except that the whole computation may fail if dereferencing fails.

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \in \mathit{domain}(h)}{\langle \mathtt{v\ :=\ [}E\mathtt{]}, (s,h)\rangle \rightsquigarrow (s[\mathtt{v} \mapsto h(e)], h)} \qquad \frac{[\![E]\!]_{(s,h)} = e \quad e \notin \mathit{domain}(h)}{\langle \mathtt{v\ :=\ [}E\mathtt{]}, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var y: ref Array<Integer>
var a: ref Pair
var v: ref Integer
```

r	`v := [`r`]`
`y + 2`	✔ `v = 9`
`z.first`	✔ `v = 4`
`y + 5`	✖
`z.second + 1`	✖



Allocation expressions: operational semantics

The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is not problematic because it still does not affect the current state – it only adds new state.

$$\frac{m \notin \mathit{domain}(h) \quad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r} \ \mathtt{:=}\ \mathbf{new}\ \mathtt{T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

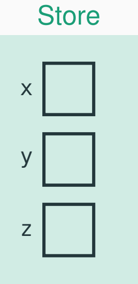
Allocation expressions: operational semantics

The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is not problematic because it still does not affect the current state – it only adds new state.

$$\frac{m \notin \mathit{domain}(h) \qquad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r} \ := \ \mathbf{new} \ \mathtt{T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

```
var x: Integer
var y: ref Array<Integer>
var z: ref Pair
```



Allocation expressions: operational semantics

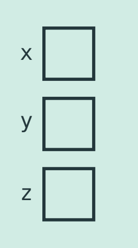
The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is not problematic because it still does not affect the current state – it only adds new state.

$$\frac{m \notin \mathit{domain}(h) \qquad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r\ :=\ new\ T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

```
var x: Integer
var y: ref Array<Integer>
var z: ref Pair

x := new Integer
```

Store



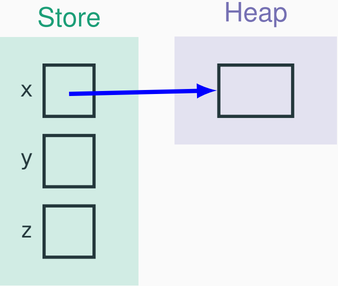
Allocation expressions: operational semantics

The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is <u>not problematic</u> because it still does not affect the current state – it only adds new state.

$$\frac{m \notin domain(h) \quad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r} := \mathbf{new}\ \mathtt{T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

```
var x: Integer
var y: ref Array<Integer>
var z: ref Pair

x := new Integer
```



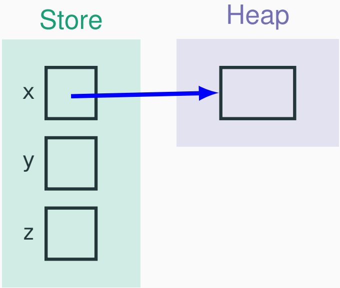
Allocation expressions: operational semantics

The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is <u>not problematic</u> because it still does not affect the current state – it only adds new state.

$$\frac{m \notin domain(h) \quad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r} := \mathbf{new}\ \mathtt{T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

```
var x: Integer
var y: ref Array<Integer>
var z: ref Pair

x := new Integer
y := new Array(3)
```



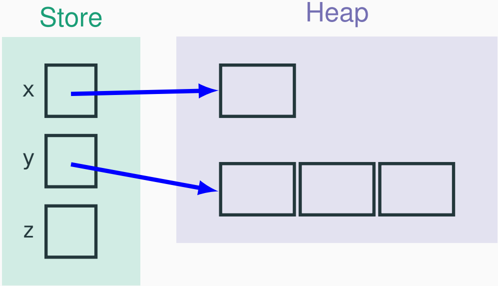
Allocation expressions: operational semantics

The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is <u>not problematic</u> because it still does not affect the current state – it only adds new state.

$$\frac{m \notin domain(h) \quad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r} \ := \ \mathbf{new} \ \mathtt{T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

```
var x: Integer
var y: ref Array<Integer>
var z: ref Pair

x := new Integer
y := new Array(3)
```



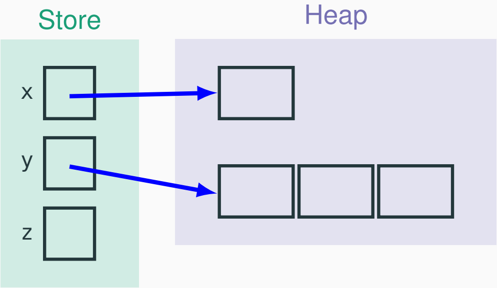
Allocation expressions: operational semantics

The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is not problematic because it still does not affect the current state – it only adds new state.

$$\frac{m \notin \mathit{domain}(h) \qquad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r} \ \mathtt{:=} \ \mathtt{new} \ \mathtt{T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

```
var x: Integer
var y: ref Array<Integer>
var z: ref Pair
x := new Integer
y := new Array(3)
z := new Pair
```



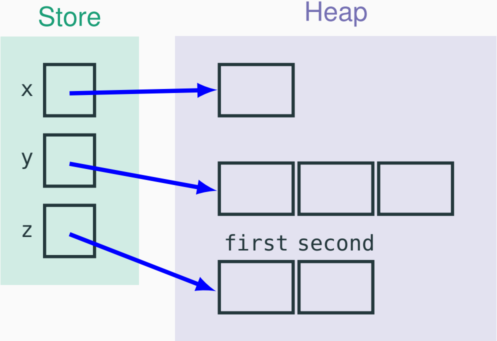
Allocation expressions: operational semantics

The evaluation of an allocation expression gives an address but also has the side effect of extending the heap's domain.

Since we will only use allocation expressions to initialize variables, this side effect is not problematic because it still does not affect the current state – it only adds new state.

$$\frac{m \notin \mathit{domain}(h) \quad m \text{ is an available memory address for type } \mathtt{T}}{\langle \mathtt{r} \ \mathtt{:=} \ \mathbf{new} \ \mathtt{T}, (s, h)\rangle \rightsquigarrow (s[\mathtt{r} \mapsto m], h[m \mapsto \mathtt{?}])}$$

```
var x: Integer
var y: ref Array<Integer>
var z: ref Pair

x := new Integer
y := new Array(3)
z := new Pair
```



Heap assignments: operational semantics

A heap assignments writes allocated heap memory whose address is given using a reference expression. This fails if the reference expression is not a valid memory address.

$$\frac{[\![r]\!]_{(s,h)} = m \quad m \in domain(h) \quad [\![E]\!]_{(s,h)} = e}{\langle \texttt{[}r\texttt{]} \ \texttt{:=} \ E, (s,h)\rangle \rightsquigarrow (s, h[m \mapsto e])} \qquad \frac{[\![r]\!]_{(s,h)} = m \quad m \notin domain(h)}{\langle \texttt{[}r\texttt{]} \ \texttt{:=} \ E, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

Heap assignments: operational semantics

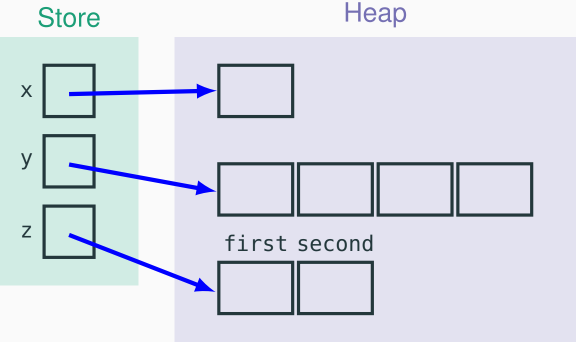
A heap assignments writes allocated heap memory whose address is given using a reference expression. This fails if the reference expression is not a valid memory address.

$$\frac{[\![r]\!]_{(s,h)} = m \quad m \in domain(h) \quad [\![E]\!]_{(s,h)} = e}{\langle \texttt{[}r\texttt{]}\ \texttt{:=}\ E, (s,h)\rangle \rightsquigarrow (s, h[m \mapsto e])} \qquad \frac{[\![r]\!]_{(s,h)} = m \quad m \notin domain(h)}{\langle \texttt{[}r\texttt{]}\ \texttt{:=}\ E, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

OK?

```
[x] := 3
[x + 3] := 3
[y] := (1,2,3,4)
[y + 1] := 3
[z.first] := 0
```



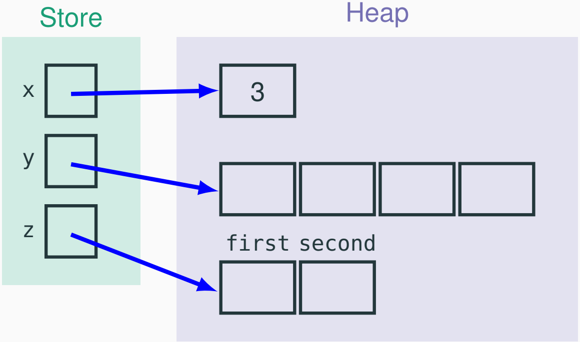
Heap assignments: operational semantics

A heap assignments writes allocated heap memory whose address is given using a reference expression. This fails if the reference expression is not a valid memory address.

$$\frac{[\![r]\!]_{(s,h)} = m \quad m \in domain(h) \quad [\![E]\!]_{(s,h)} = e}{\langle \texttt{[}r\texttt{]}\ \texttt{:=}\ E, (s,h)\rangle \rightsquigarrow (s, h[m \mapsto e])} \qquad \frac{[\![r]\!]_{(s,h)} = m \quad m \notin domain(h)}{\langle \texttt{[}r\texttt{]}\ \texttt{:=}\ E, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

	OK?
`[x] := 3`	✔
`[x + 3] := 3`	
`[y] := (1,2,3,4)`	
`[y + 1] := 3`	
`[z.first] := 0`	



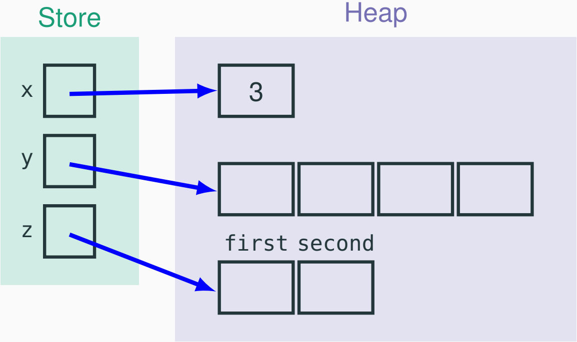
Heap assignments: operational semantics

A heap assignments writes allocated heap memory whose address is given using a reference expression. This fails if the reference expression is not a valid memory address.

$$\frac{[\![r]\!]_{(s,h)} = m \quad m \in domain(h) \quad [\![E]\!]_{(s,h)} = e}{\langle \texttt{[}r\texttt{]} \texttt{ := } E, (s,h)\rangle \rightsquigarrow (s, h[m \mapsto e])} \qquad \frac{[\![r]\!]_{(s,h)} = m \quad m \notin domain(h)}{\langle \texttt{[}r\texttt{]} \texttt{ := } E, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

	OK?
`[x] := 3`	✓
`[x + 3] := 3`	✗
`[y] := (1,2,3,4)`	
`[y + 1] := 3`	
`[z.first] := 0`	



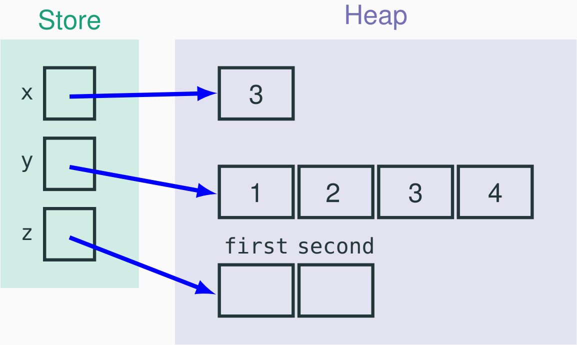
Heap assignments: operational semantics

A heap assignments writes allocated heap memory whose address is given using a reference expression. This fails if the reference expression is not a valid memory address.

$$\frac{[\![r]\!]_{(s,h)} = m \quad m \in domain(h) \quad [\![E]\!]_{(s,h)} = e}{\langle \texttt{[}r\texttt{] := } E, (s,h)\rangle \rightsquigarrow (s, h[m \mapsto e])} \qquad \frac{[\![r]\!]_{(s,h)} = m \quad m \notin domain(h)}{\langle \texttt{[}r\texttt{] := } E, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

	OK?
`[x] := 3`	✓
`[x + 3] := 3`	✗
`[y] := (1,2,3,4)`	✓
`[y + 1] := 3`	
`[z.first] := 0`	



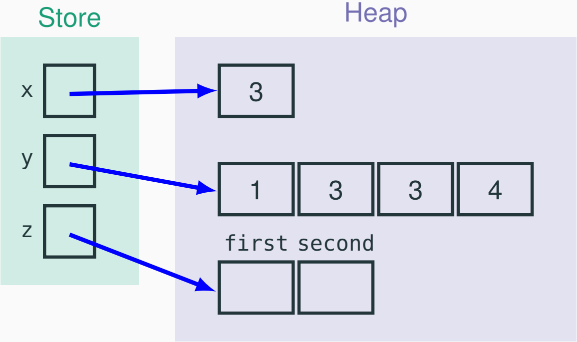
Heap assignments: operational semantics

A heap assignments writes allocated heap memory whose address is given using a reference expression. This fails if the reference expression is not a valid memory address.

$$\frac{[\![r]\!]_{(s,h)} = m \quad m \in domain(h) \quad [\![E]\!]_{(s,h)} = e}{\langle \texttt{[}r\texttt{] := } E, (s,h)\rangle \rightsquigarrow (s, h[m \mapsto e])} \qquad \frac{[\![r]\!]_{(s,h)} = m \quad m \notin domain(h)}{\langle \texttt{[}r\texttt{] := } E, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

	OK?
`[x] := 3`	✓
`[x + 3] := 3`	✗
`[y] := (1,2,3,4)`	✓
`[y + 1] := 3`	✓
`[z.first] := 0`	



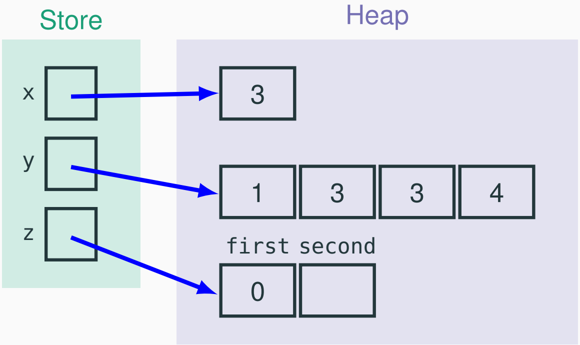
Heap assignments: operational semantics

A heap assignments writes allocated heap memory whose address is given using a reference expression. This fails if the reference expression is not a valid memory address.

$$\frac{[\![r]\!]_{(s,h)} = m \quad m \in domain(h) \quad [\![E]\!]_{(s,h)} = e}{\langle \texttt{[}r\texttt{]} \ \texttt{:=} \ E, (s,h)\rangle \rightsquigarrow (s, h[m \mapsto e])} \qquad \frac{[\![r]\!]_{(s,h)} = m \quad m \notin domain(h)}{\langle \texttt{[}r\texttt{]} \ \texttt{:=} \ E, (s,h)\rangle \rightsquigarrow \textbf{error}}$$

```
var x: ref Integer
var y: ref Array<Integer>
var z: ref Pair
```

	OK?
`[x] := 3`	✓
`[x + 3] := 3`	✗
`[y] := (1,2,3,4)`	✓
`[y + 1] := 3`	✓
`[z.first] := 0`	✓



Deallocation statement: operational semantics

A deallocation statement shrinks the heap's domain – it invalidates existing state. As usual it may fail if we try to deallocate an invalid heap address.

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \in domain(h) \quad h' = h \setminus \{e \to \_\}}{\langle \texttt{dispose}\ E, (s,h)\rangle \leadsto (s,h')}$$

$$\frac{[\![E]\!]_{(s,h)} = e \quad e \notin domain(h)}{\langle \texttt{dispose}\ E, (s,h)\rangle \leadsto \textbf{error}}$$

$h' = h$ except that
h' is undefined at e

Berillium

We call Berillium the language Lithium extended with:

1. reference types
2. allocation and deallocation
3. heap reading and writing

Supporting realistic program features

The aliasing problem

Challenges of reasoning about reference types

Reference types are commonly used in programming languages – mainly for efficiency reasons.

The reasoning rules we have used so far have to be significantly refined to remain sound with reference types.

nullness: a reference variable can take a memory location that is not allocated or `nil` (invalid); operations on an invalid reference variable fail

(de)allocation: memory in the heap must be allocated when needed and deallocated (released) when done using it (Deallocation is less of a problem in languages, such as Java, that use automatic memory management (garbage collection)

aliasing: different reference variables may share the same heap location; this makes reasoning non-local

The aliasing problem

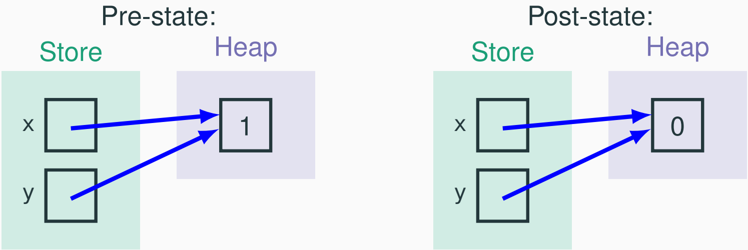
Let's get an idea of the problems introduced by aliasing.

We have already seen the aliasing problem with array assignment.

Similarly, the usual backward substitution rule is unsound if reference variables are involved. For example, we could deduce:

$$\{x = y \land [y] = 1\}\ \texttt{[x] := 0}\ \{x = y \land [y] = 1\}$$

for two variables `x, y: ref Integer`, even though `[y] = 0` after the assignment because `y` aliases `x`.



Assignment with references

A workaround for the unsoundness of the assignment rule is adding a semantic condition that there is no aliasing:

$$\{x \neq y \land [y] = 1\}\ \texttt{[x] := 0}\ \{x \neq y \land [y] = 1\}$$

is valid for any variables `x, y: ref Integer`.

The limitation of this approach is that it turns a simple syntactic rule into a more complex one – as it requires to reason semantically about equality of references.

Assignment with references

A workaround for the unsoundness of the assignment rule is adding a semantic condition that there is no aliasing:

$$\{x \neq y \wedge [y] = 1\}\ \texttt{[x] := 0}\ \{x \neq y \wedge [y] = 1\}$$

is valid for any variables `x, y: ref Integer`.

The limitation of this approach is that it turns a simple syntactic rule into a more complex one – as it requires to reason semantically about equality of references.

Another issue with reasoning explicitly about aliasing is that it quickly introduces a lot of annotation overhead:

$$\left\{ C(z_1, \ldots, z_n) \ \wedge \bigwedge_{1 \leq k \leq n} \bigwedge_{1 \leq j \leq m} z_k \neq x_j \right\}$$

$$\texttt{[}x_1\texttt{],...,[}x_m\texttt{] := 0,...,0}$$

$$\{C(z_1, \ldots, z_n)\}$$

Rule of constancy

R doesn't mention any variable in *S*'s frame

Another victim of aliasing is the rule of constancy:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \wedge R\}\ S\ \{Q \wedge R\}}$$

Even if S doesn't assign to any variable mentioned in R, some variables in R may be aliased to some variables modified by S.

Rule of constancy

R doesn't mention any variable in S's frame

Another victim of aliasing is the rule of constancy:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \land R\}\ S\ \{Q \land R\}}$$

Even if S doesn't assign to any variable mentioned in R, some variables in R may be aliased to some variables modified by S.

This problem generalizes the problem with assignments since it applies to any piece of code that modifies shared memory – for example a procedure.

```
procedure p ():
ensure [x] = 0
  [x] := 0
```

Procedure `p` has an empty frame since it does not assign to any global variable.
However, it modifies indirectly the global heap state by writing to the memory location referenced by `x`; every variable pointing to the same location will also reference `0`.

Framing with references

A way to specify meaningful frame conditions for global variables of reference types is interpreting a modifies clause as a set of objects (that is, memory locations) that may modified.

$$\frac{\{P\}\ B\ \{Q\} \qquad \mathcal{F}(B) \cap \mathcal{G} \subseteq F}{\texttt{procedure proc (in: T): (out: T) } \textbf{require}\ P\ \textbf{modify}\ F\ \textbf{ensure}\ Q\ B}$$

Now $\mathcal{F}(B)$ is the set of:

- all value variables that are the target of any assignments in B
- all reference variables that are the target of any heap assignments in B
- all variables mentioned in the **modify** clause of any procedure called in B

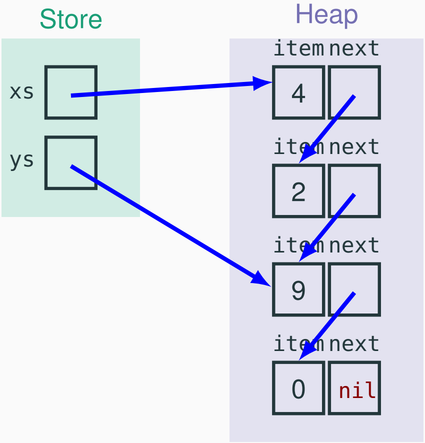
Supporting realistic program features

Reasoning about objects

Partial sharing

The problem introduced by aliasing is even more acute when it involves data structures allocated in the heap, such as the familiar linked list.

In this case there is a partial sharing of the lists pointed to by `xs` and `ys`.
We would like to be able to reason about when a procedure operating on `xs` interferes with any other procedure that partially shares the list.



How do we specify framing of procedures operating on heap-allocated data structures?

Framing methodologies

A number of framing methodologies (also called protocols) have been developed to specify framing precisely, and to be able to reason about code in the presence of sharing.

ownership methodologies express the collection of objects that a data structure owns – usually through a class invariant. An object may generally read any other objects, but may modify only objects it owns.

semantic collaboration expresses the collections of objects that a data structure depends on and other objects that depend on the structure. A fine-grained control of read/write dependencies between objects supports the specification and verification of idiomatic object structures such as the object-oriented design patterns.

dynamic frames methodologies rely on explicit representation of the frame conditions and on an explicit reasoning about interference.

Reasoning about the observer pattern

Here's an example of object-oriented design pattern, implemented in Java, that makes reasoning challenging.

```java
class Subject<T> {
  T value;
  List<Observer> subscribers;

  void update(T value) {
    this.value = value;
    for (Observer o: subscribers)
      o.notify();
  }

  void register(Observer o) {
    subscribers.add(o);
  }
}
```

```java
class Observer<T> {
  T cache;
  Subject<T> subject;

  void notify() {
    cache = subject.value;
  }

  boolean invariant() {
    return (subject.subscribers.has(this)
            && cache == subject.value);
  }
}
```

Reasoning about the observer pattern

Here's an example of object-oriented design pattern, implemented in Java, that makes reasoning challenging.

```java
class Subject<T> {
  T value;
  List<Observer> subscribers;

  void update(T value) {
    this.value = value;
    for (Observer o: subscribers)
      o.notify();
  }

  void register(Observer o) {
    subscribers.add(o);
  }
}
```

```java
class Observer<T> {
  T cache;
  Subject<T> subject;

  void notify() {
    cache = subject.value;
  }

  boolean invariant() {
    return (subject.subscribers.has(this)
             && cache == subject.value);
  }
}
```

The observer's invariant depends on the subject, but it's inappropriate to assume that the observer owns the subject.

Reasoning about the observer pattern

Here's an example of object-oriented design pattern, implemented in Java, that makes reasoning challenging.

```java
class Subject<T> {
  T value;
  List<Observer> subscribers;

  void update(T value) {
    this.value = value;
    for (Observer o: subscribers)
      o.notify();
  }

  void register(Observer o) {
    subscribers.add(o);
  }
}
```

```java
class Observer<T> {
  T cache;
  Subject<T> subject;

  void notify() {
    cache = subject.value;
  }

  boolean invariant() {
    return (subject.subscribers.has(this)
            && cache == subject.value);
  }
}
```

Conversely, the subject also cannot own the observers since it should be decoupled from them.

Reasoning about the observer pattern

Here's an example of object-oriented design pattern, implemented in Java, that makes reasoning challenging.

```
class Subject<T> {
  T value;
  List<Observer> subscribers;

  void update(T value) {
    this.value = value;
    for (Observer o: subscribers)
      o.notify();
  }

  void register(Observer o) {
    subscribers.add(o);
  }
}
```

```
class Observer<T> {
  T cache;
  Subject<T> subject;

  void notify() {
    cache = subject.value;
  }

  boolean invariant() {
    return (subject.subscribers.has(this)
            && cache == subject.value);
  }
}
```

Neither `notify` nor `update` may assume the invariant as precondition: they execute when the invariant's false.

Reasoning about the observer pattern

Here's an example of object-oriented design pattern, implemented in Java, that makes reasoning challenging.

```
class Subject<T> {
  T value;
  List<Observer> subscribers;

  void update(T value) {
    this.value = value;
    for (Observer o: subscribers)
      o.notify();
  }

  void register(Observer o) {
    subscribers.add(o);
  }
}
```

```
class Observer<T> {
  T cache;
  Subject<T> subject;

  void notify() {
    cache = subject.value;
  }

  boolean invariant() {
    return (subject.subscribers.has(this)
            && cache == subject.value);
  }
}
```

How to specify that the `update` method modifies any number of registered observers?

Dynamic frames with Dafny

We now explore the challenges of reasoning about programs with shared mutable state using a series of examples of implementing a linked list in Dafny.

As we have seen already, Dafny is object-based – in particular, all variables except those of primitive/mathematical types are references that are used without explicit dereferencing.

In this sense it is similar to languages like Java, and hence it serves well our purpose of illustrating the challenges of reasoning about realistic (object-oriented) programs.

Linked list with basic specification

```
class Node<T>
{
  var item: T;        // stored value
  var next: Node<T>;  // ref to next node

  // create a node storing `item`
  constructor (item: T)
    modifies this;
    ensures this.item = item;
    ensures this.next = null;
  {
    this.item := item;
    this.next := null;
  }
}
```

```
class List<T>
{
  // ref to first node
  var head: Node<T>;
  // number of elements in list
  var size: int;

  // add to front a node storing `item`
  method extend(item: T)
    modifies this;
    ensures size = old(size) + 1;
    ensures head ≠ null;
    ensures head.item = item;
  {
    var first := new Node(item);
    first.next := head;
    head := first;
    size := size + 1;
  }
}
```

Weak specification

The specification of `extend` is quite weak as it doesn't guarantee that the method doesn't change the existing nodes in the list.

```
// add to front a node storing `item`
method extend(item: T)
  modifies this;
  ensures size = old(size) + 1;
  ensures head ≠ null;
  ensures head.item = item;
{
  var first := new Node(item);
  first.next := head;
  head := first;
  size := size + 1;
}
```

```
method fake_extend(item: T)
  modifies this;
  ensures size = old(size) + 1;
  ensures head ≠ null;
  ensures head.item = item;
{
  var first := new Node(item);
  // doesn't connect existing list
  head := first;
  size := size + 1;
}
```

Weak specification

The specification of `extend` is quite weak as it doesn't guarantee that the method doesn't change the existing nodes in the list.

```
// add to front a node storing `item`
method extend(item: T)
  modifies this;
  ensures size == old(size) + 1;
  ensures head != null;
  ensures head.item == item;
{
  var first := new Node(item);
  first.next := head;
  head := first;
  size := size + 1;
}
```

```
method fake_extend(item: T)
  modifies this;
  ensures size == old(size) + 1;
  ensures head != null;
  ensures head.item == item;
{
  var first := new Node(item);
  // doesn't connect existing list
  head := first;
  size := size + 1;
}
```

We need to express how the sequence of elements stored in the chain of nodes changes after each operation.

Ghost variables

We add a ghost (also: auxiliary) variable to `List`, which keeps track of the sequence of elements using a mathematical sequence as model.

```
// sequence of `item`s
// in chain of nodes
ghost var sequence: seq<T>;
```

```
// add to front a node storing `item`
method extend(item: T)
  modifies this;
  ensures size == old(size) + 1;
  ensures head ≠ null;
  ensures head.item == item;
  ensures sequence == [item] + old(sequence);
{
  var first := new Node(item);
  first.next := head;
  head := first;
  size := size + 1;
  sequence := [item] + sequence;
}
```

Ghost variables are only used in proofs but can be discarded when executing the program after we have proved correctness.

Model consistency

Model variables need to be connected to the actual implementation, otherwise we can still trivially satisfy a specification using `sequence`.

```
// sequence of `item`s
// in chain of nodes
ghost var sequence: seq<T>;
```

```
method fake_extend(item: T)
  modifies this;
  ensures size == old(size) + 1;
  ensures head != null;
  ensures head.item == item;
  ensures sequence == [item] + old(sequence);
{
  var first := new Node(item);
  // doesn't connect existing list
  head := first;
  size := size + 1;
  sequence := [item] + sequence;
  // now model and list are inconsistent
}
```

Inductive predicates

One way to that is to introduce inductive function that recursively enumerates the elements reachable in the list.

```
function elements<T>(head: Node<T>): seq<T>
  reads head;
{
  if (head == null) then []
  else ([head.item]
        + elements(head.next))
}
```

```
// add to front a node storing `item`
method extend(item: T)
    // require consistency
  requires elements(head) == sequence;
  modifies this;
  ensures size == old(size) + 1;
  ensures head != null;
  ensures head.item == item;
  ensures sequence == [item] + old(sequence);
    // preserve consistency
  ensures elements(head) == sequence;
```

A reads clause is the (mathematical) functions' counterpart to procedures' modify clause: by specifying on what a function's value depends, we can reason precisely about when a function evaluation may change.

Strong specifications

Now `fake_extend` cannot verify the same strong specification that connects model and representation.

```
function elements<T>(head: Node<T>): seq<T>
  reads head;
{
  if (head == null) then []
  else ([head.item]
        + elements(head.next))
}
```

```
method fake_extend(item: T)
  requires elements(head) == sequence;
  modifies this;
  ensures size == old(size) + 1;
  ensures head != null;
  ensures head.item == item;
  ensures sequence == [item] + old(sequence);
    // may not hold
  ensures elements(head) == sequence;
```

`elements(head)` depends on `head` (reads clause), which has been changed; hence `elements(head)` has changed as well.

Well-formed inductive predicates

Dafny cannot prove the termination of function `elements` because it is not a well-formed definition of finite sequence:

```
function elements<T>(head: Node<T>): seq<T>
  reads head;
{
  if (head == null) then []
  else ([head.item]
        + elements(head.next))
}
```

- Its value is undefined when we follow a chain of nodes not terminated by **`null`**
- Its value is undefined when we follow a chain of nodes with a loop

These are well-formedness conditions that the list itself should have.

Invariants

Let's switch to a different style of specification – using a predicate `valid` that should hold before and after operations that leave the list in a consistent state (like an invariant).

Now we keep track directly of the sequence of nodes.

```
// chain of nodes from `head`
ghost var nodes: seq<Node<T>>;

function valid(): bool
  reads this, nodes; {
  (∀ k: int • 0 ≤ k < |nodes| ⟹ nodes[k] ≠ null) // no null nodes
    ∧ size = |nodes| /* size is consistent */ ∧ (
      (head = null ∧ |nodes| = 0)  // empty list
      ∨ (head ≠ null ∧ |nodes| > 0 ∧ head = nodes[0]
              // connected nodes
          ∧ (∀ k: int • 0 ≤ k < |nodes| - 1 ⟹ nodes[k].next = nodes[k+1])
              // nonrepeating nodes
          ∧ (∀ k: int • 0 ≤ k < |nodes| ⟹ nodes[k] ∉ nodes[0..k] + nodes[k+1..]))
    )
}
```

Abstraction in specification

If we add `valid()` as precondition and postcondition of `extend` and `fake_extend`, Dafny proves the first correct but not the second – since it does not leave the list in a valid state.

However, a specification that relies on a low-level implementation detail (namely the sequence of nodes) may be not abstract enough for public clients.

We add a more abstract specification on top of the fundamental one – for example by reintroducing the ghost variable `sequence` and linking it to `nodes`

```
// sequence of `item`s
// in chain of nodes
ghost var sequence: seq<T>;
```

New clause in `valid()`:

```
∧ ( |nodes| = |sequence| ∧  // sequence is node's projection on `item`
     (∀ k: int • 0 ≤ k < |nodes| ⟹ nodes[k].item = sequence[k]) )
```

Tools and case studies

Tools

Some notable tools for deductive verification:

- **Boogie** is an intermediate verification language – similar to an intermediate representation for auto-active verifiers
- **Dafny** is an auto-active verifier with support for objects and dynamic frames
- **Why3** is a deductive verification for a functional language (a dialect of ML)
- **VeriFast** is an interactive prover for separation logic specifications and C/Java programs
- **AutoProof** is an auto-active prover for the Eiffel programming language, supporting powerful methodologies for class invariants
- **KeY** is an interactive proof system for Java programs annotated with JML
- **SPARK** is a language and system to incrementally develop high-integrity software

Case studies

Some notable case studies carried out using deductive verification:

EiffelBase2: a fully-verified realistic container library

Schorr-Waite: a complex graph-marking algorithm that is used for garbage collection – verified using VeriFast, Why3, and Dafny

TimSort: a complex general-purpose sorting algorithm used in Java's and Python's standard libraries – verified using KeY

See the VerifyThis annual verification competition for other examples of the state-of-the-art in deductive verification.

Separation logic

Separation logic

We have seen a few examples of how Hoare logic can be equipped with methodologies to cope with the issue brought by shared mutable state.

We now describe a more fundamental approach to the same problem: separation logic.

Separation logic is an extension of Hoare logic geared towards preserving some of the nice syntactic features of axiomatic reasoning in the presence of references/pointers.

With separation logic we can write assertions that describe the shape of memory cells in the heap. These assertions are amenable to local (modular) reasoning in a similar way as traditional Hoare logic on programs without pointers.

Separation logic

We have seen a few examples of how Hoare logic can be equipped with methodologies to cope with the issue brought by shared mutable state.

We now describe a more fundamental approach to the same problem: separation logic.

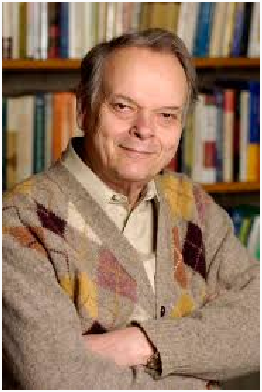
Separation logic is an extension of Hoare logic geared towards preserving some of the nice syntactic features of axiomatic reasoning in the presence of references/pointers.

With separation logic we can write assertions that describe the shape of memory cells in the heap. These assertions are amenable to local (modular) reasoning in a similar way as traditional Hoare logic on programs without pointers.

This presentation is based on various materials by O'Hearn, Calcagno, Parkinson, van Staden, and Poskitt.

Separation logic: the inventors

Separation logic was first developed by O'Hearn, Yang, and the late John Reynolds around 2000 – based on foundational work done by Rod Burstall in the 1970s.



John C. Reynolds

Peter O'Hearn

Separation logic in industrial practice

Startup Monoidics – founded by O'Hearn, Calcagno, Distefano, and others – turned some of the theoretical work on separation logic into usable verification technology. It was acquired by Facebook in 2013.

theguardian

Facebook buys code-checking Silicon Roundabout startup Monoidics

MONOIDICS

facebook

Separation logic

Separation logic assertions

Predicates in separation logic

Separation logic extends the vocabulary to write predicates about program states – to support describing the heap's content with ease.

Separation logic introduces these new constructs:

separating conjunction:	$\mathit{Formula} \ast \mathit{Formula}$
separating implication:	$\mathit{Formula} \mathrel{-\!\!\ast} \mathit{Formula}$
points to relation:	$\mathit{Expression} \mapsto \mathit{Expression}$
empty heap:	emp

Predicates in separation logic

Separation logic extends the vocabulary to write predicates about program states – to support describing the heap's content with ease.

Separation logic introduces these new constructs:

“star”

separating conjunction:	*Formula* $*$ *Formula*
separating implication:	*Formula* $-\!\!*$ *Formula*
points to relation:	*Expression* $\mapsto$ *Expression*
empty heap:	*emp*

“magic wand”

Semantics of empty heap

The empty heap predicate holds only in heaps that do not contain any memory:

$$s, h \models \mathit{emp} \qquad\qquad \text{iff} \qquad\qquad \mathit{domain}(h) = \emptyset$$

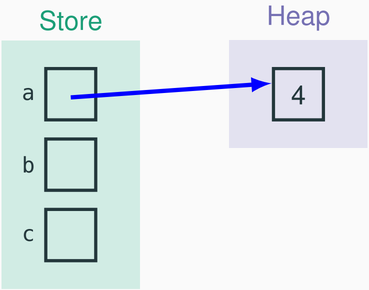
emp means that the heap is empty

Semantics of points to

The points to relations holds only in heaps with exactly one memory location:

$$s, h \models X \mapsto Y \quad \text{iff} \quad \mathit{domain}(h) = \{[\![X]\!]_{(s,h)}\} \text{ and } h([\![X]\!]_{(s,h)}) = [\![Y]\!]_{(s,h)}$$

$X \mapsto Y$ means that the heap has exactly one location whose address is X's value and that stores Y's value



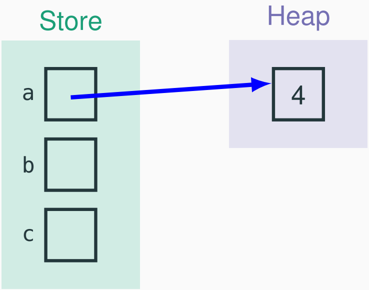
Semantics of points to

The points to relations holds only in heaps with exactly one memory location:

$$s, h \models X \mapsto Y \quad \text{iff} \quad \mathit{domain}(h) = \{[\![X]\!]_{(s,h)}\} \text{ and } h([\![X]\!]_{(s,h)}) = [\![Y]\!]_{(s,h)}$$

$X \mapsto Y$ means that the heap has exactly one location whose address is X's value and that stores Y's value

$a \mapsto 4$



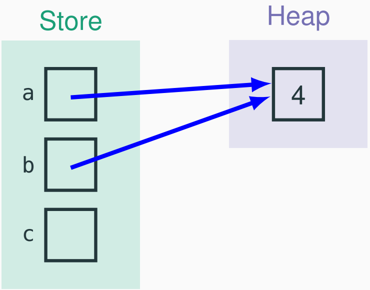
Semantics of points to

The points to relations holds only in heaps with exactly one memory location:

$$s, h \models X \mapsto Y \quad \text{iff} \quad domain(h) = \{[\![X]\!]_{(s,h)}\} \text{ and } h([\![X]\!]_{(s,h)}) = [\![Y]\!]_{(s,h)}$$

$X \mapsto Y$ means that the heap has exactly one location whose address is X's value and that stores Y's value

$\mathsf{a} \mapsto 4$



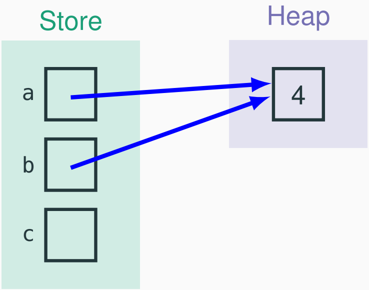
Semantics of points to

The points to relations holds only in heaps with exactly one memory location:

$$s, h \models X \mapsto Y \quad \text{iff} \quad \mathit{domain}(h) = \{[\![X]\!]_{(s,h)}\} \text{ and } h([\![X]\!]_{(s,h)}) = [\![Y]\!]_{(s,h)}$$

$X \mapsto Y$ means that the heap has exactly one location whose address is X's value and that stores Y's value

$\mathsf{a} \mapsto 4 \ \wedge \ \mathsf{b} \mapsto 4$

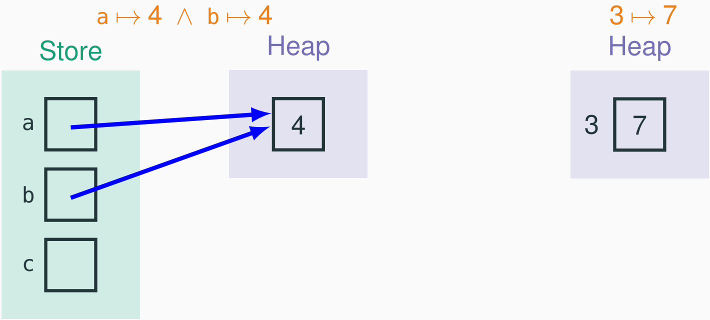


Semantics of points to

The points to relations holds only in heaps with exactly one memory location:

$$s, h \models X \mapsto Y \quad \text{iff} \quad domain(h) = \{[\![X]\!]_{(s,h)}\} \text{ and } h([\![X]\!]_{(s,h)}) = [\![Y]\!]_{(s,h)}$$

$X \mapsto Y$ means that the heap has exactly one location whose address is X's value and that stores Y's value



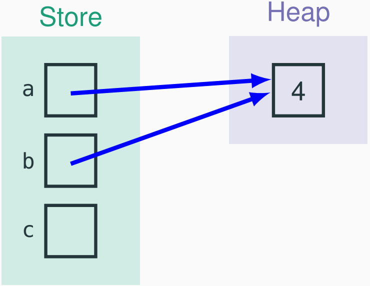
Semantics of points to

The points to relations holds only in heaps with exactly one memory location:

$$s, h \models X \mapsto Y \quad \text{iff} \quad domain(h) = \{[\![X]\!]_{(s,h)}\} \text{ and } h([\![X]\!]_{(s,h)}) = [\![Y]\!]_{(s,h)}$$

$X \mapsto Y$ means that the heap has exactly one location whose address is X's value and that stores Y's value

$a \mapsto 4 \ \wedge \ b \mapsto 4$



$3 \mapsto 7$

What about larger heaps?

Semantics of separating conjunction

We use the separating conjunction (star operator) to compose elementary assertions made using the points to relation on disjoint heaps:

disjoint domains

$$s, h \models P * Q \quad \text{iff} \quad \text{there exist } h_1, h_2 \text{ such that:}$$

$$domain(h_1) \cap domain(h_2) = \emptyset$$

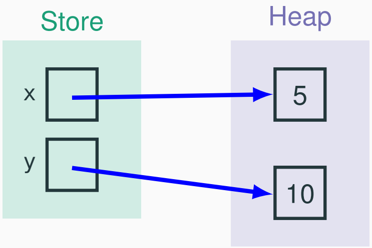
$$h_1 \cup h_2 = h$$

union of functions

$$s, h_1 \models P \quad \text{and} \quad s, h_2 \models Q$$

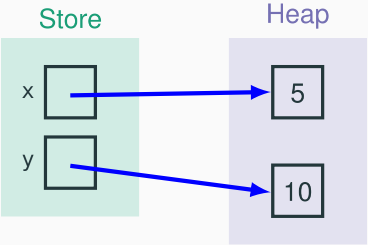
$P * Q$ means that the heap can be split into two so that
P holds in one part and Q holds in the other

Examples of separating conjunction



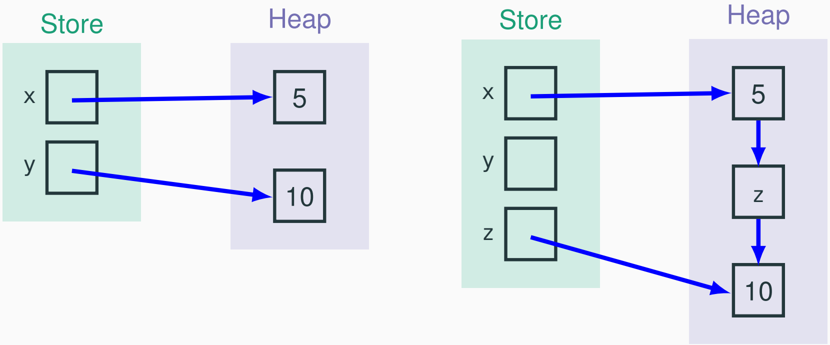
Examples of separating conjunction

$$x \mapsto 5 \quad * \quad y \mapsto 10$$



Examples of separating conjunction

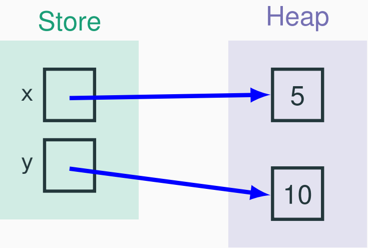
$x \mapsto 5 \quad * \quad y \mapsto 10$



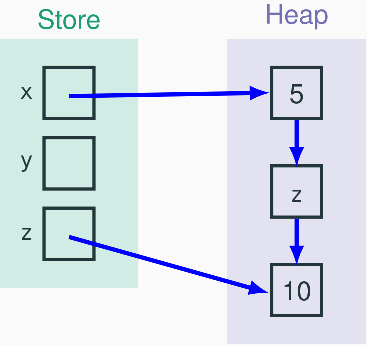
We can always partition the heap into disjoint sets of cells.

Examples of separating conjunction

$x \mapsto 5 \; * \; y \mapsto 10$

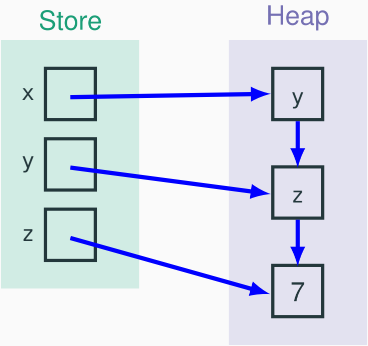


$x \mapsto 5 \; * \; 5 \mapsto z \; * \; z \mapsto 10$



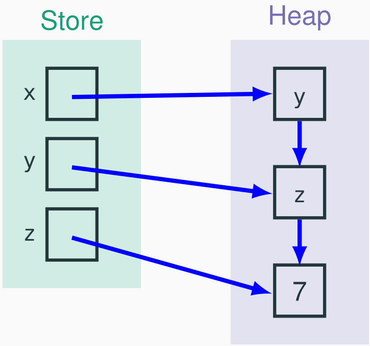
We can always partition the heap into disjoint sets of cells.

Examples of separating conjunction



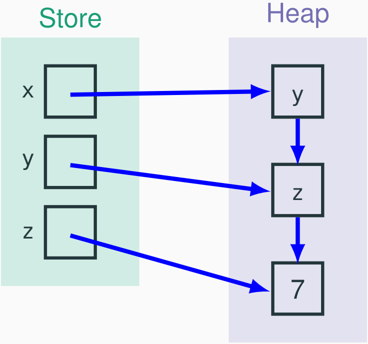
Examples of separating conjunction

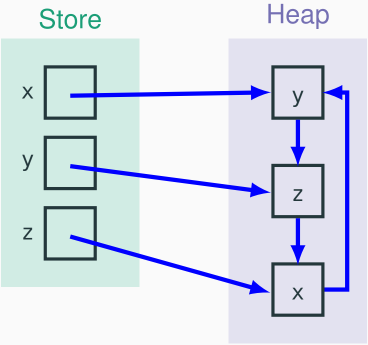
$$x \mapsto y \quad * \quad y \mapsto z \quad * \quad z \mapsto 7$$



Examples of separating conjunction

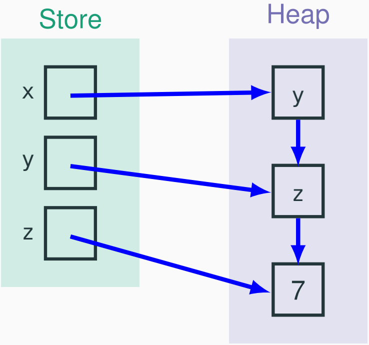
$$x \mapsto y \;\; * \;\; y \mapsto z \;\; * \;\; z \mapsto 7$$



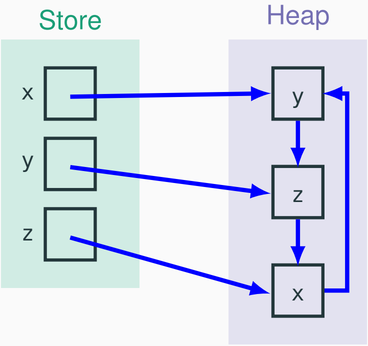


Examples of separating conjunction

$x \mapsto y \; * \; y \mapsto z \; * \; z \mapsto 7$



$\mathit{emp} \; * \; x \mapsto y \; * \; y \mapsto z \; * \; z \mapsto x$



Shorthand for adjacent locations

$$X \mapsto Y_0, \ldots, Y_n$$

is a shorthand for

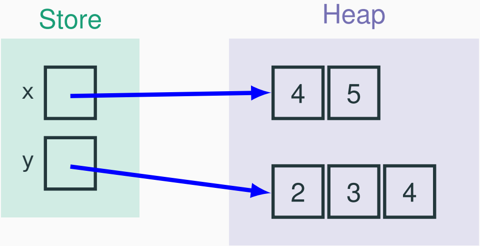
$$X \mapsto Y_0 \; * \; X+1 \mapsto Y_1 \; * \; \cdots \; * \; X+n \mapsto Y_n$$

Shorthand for adjacent locations

$$X \mapsto Y_0, \ldots, Y_n$$

is a shorthand for

$$X \mapsto Y_0 \;\ast\; X+1 \mapsto Y_1 \;\ast\; \cdots \;\ast\; X+n \mapsto Y_n$$

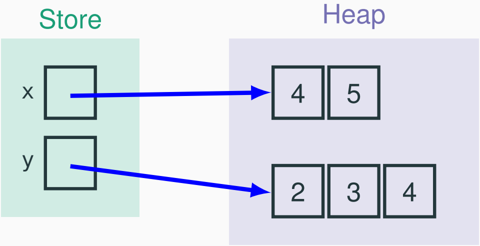


Shorthand for adjacent locations

$$X \mapsto Y_0, \ldots, Y_n$$

is a shorthand for

$$X \mapsto Y_0 \ * \ X + 1 \mapsto Y_1 \ * \ \cdots \ * \ X + n \mapsto Y_n$$



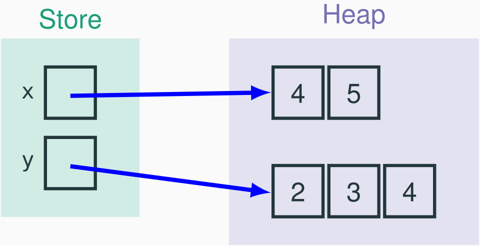
$$\mathrm{x} \mapsto 4, 5 \ * \ \mathrm{y} \mapsto 2, 3, 4$$

Shorthand for adjacent locations

$$X \mapsto Y_0, \ldots, Y_n$$

is a shorthand for

$$X \mapsto Y_0 \;\; * \;\; X + 1 \mapsto Y_1 \;\; * \;\; \cdots \;\; * \;\; X + n \mapsto Y_n$$



$$\mathsf{x} \mapsto 4, 5 \;\; * \;\; \mathsf{y} \mapsto 2, 3, 4$$

$$\mathsf{x} \mapsto 4, 5 \;\; * \;\; \top$$

Shorthand for “allocated”

$$X \mapsto -$$

is a shorthand for

$$\exists x \bullet (X \mapsto x)$$

that is X is a valid (allocated) memory address.

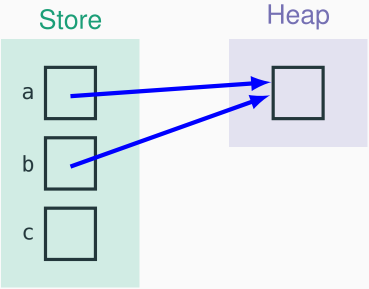
Shorthand for "allocated"

$$X \mapsto -$$

is a shorthand for

$$\exists x \bullet (X \mapsto x)$$

that is X is a valid (allocated) memory address.



$a \mapsto -$

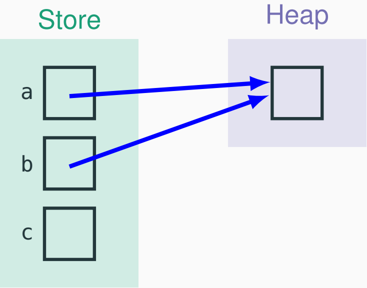
Shorthand for "allocated"

$$X \mapsto -$$

is a shorthand for

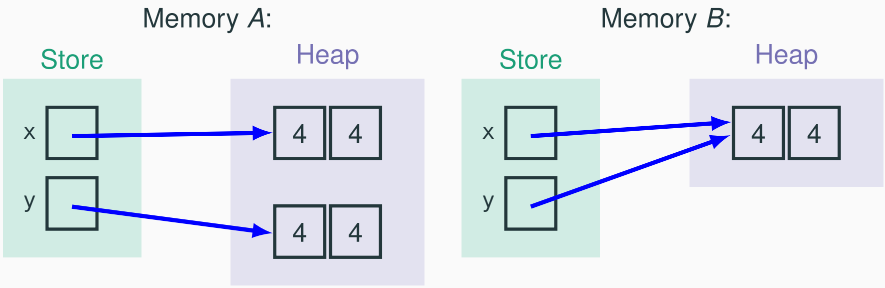
$$\exists x \bullet (X \mapsto x)$$

that is X is a valid (allocated) memory address.

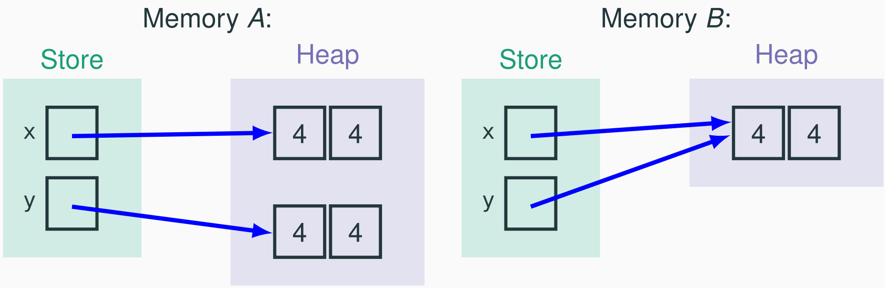


$$\mathsf{a} \mapsto - \;\wedge\; \mathsf{b} \mapsto -$$

More examples of separation logic assertions

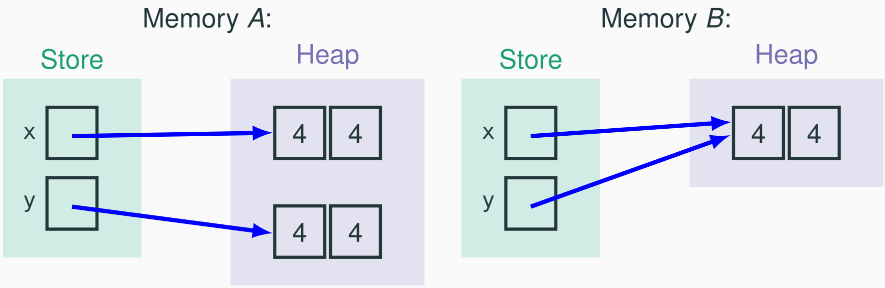


More examples of separation logic assertions



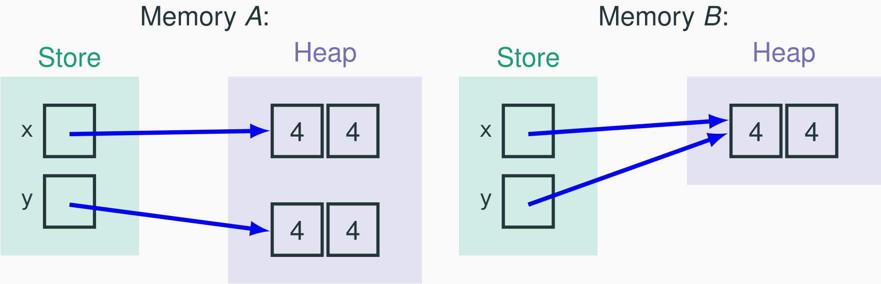
F	$A \models F$	$B \models F$
$x \mapsto 4, 4$		

More examples of separation logic assertions



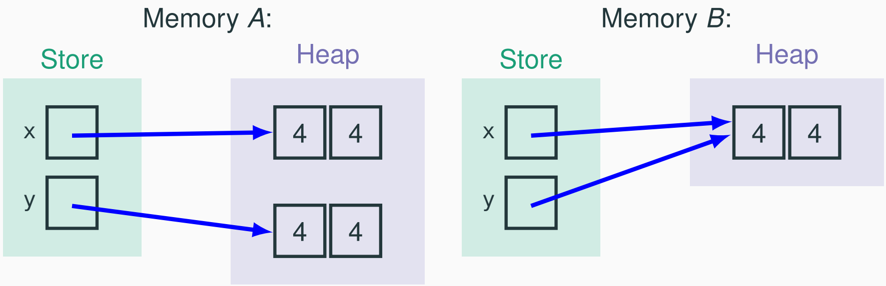
F	$A \models F$	$B \models F$
$x \mapsto 4, 4$	✗	✓
$x \mapsto 4, 4 \ * \ \top$		

More examples of separation logic assertions



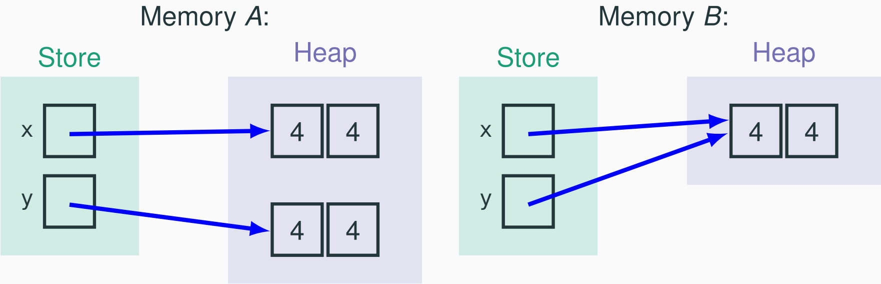
F	$A \models F$	$B \models F$
$\mathsf{x} \mapsto 4, 4$	✗	✓
$\mathsf{x} \mapsto 4, 4 \ * \ \top$	✓	✓
$\mathsf{x} \mapsto 4, 4 \ * \ \mathsf{y} \mapsto 4, 4$		

More examples of separation logic assertions



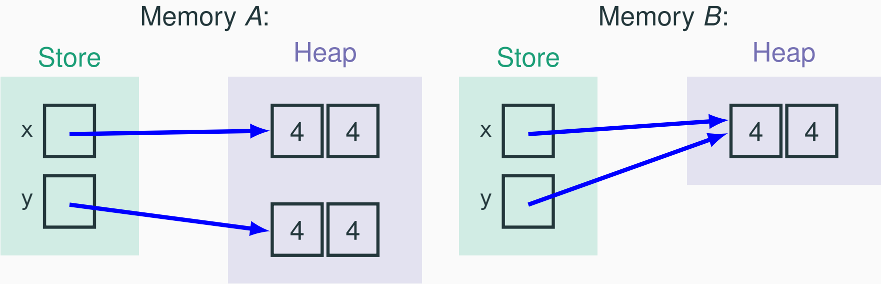
F	$A \models F$	$B \models F$
$\mathsf{x} \mapsto 4, 4$	✗	✓
$\mathsf{x} \mapsto 4, 4 \ * \ \top$	✓	✓
$\mathsf{x} \mapsto 4, 4 \ * \ \mathsf{y} \mapsto 4, 4$	✓	✗
$\mathsf{x} \mapsto 4, 4 \wedge \mathsf{y} \mapsto 4, 4$		

More examples of separation logic assertions



F	$A \models F$	$B \models F$
$x \mapsto 4, 4$	✗	✓
$x \mapsto 4, 4 \ * \ \top$	✓	✓
$x \mapsto 4, 4 \ * \ y \mapsto 4, 4$	✓	✗
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$	✗	✓
$(x \mapsto 4, 4 \ * \ \top) \wedge (y \mapsto 4, 4 \ * \ \top)$		

More examples of separation logic assertions



F	$A \models F$	$B \models F$
$x \mapsto 4, 4$	✗	✓
$x \mapsto 4, 4 \ * \ \top$	✓	✓
$x \mapsto 4, 4 \ * \ y \mapsto 4, 4$	✓	✗
$x \mapsto 4, 4 \wedge y \mapsto 4, 4$	✗	✓
$(x \mapsto 4, 4 \ * \ \top) \wedge (y \mapsto 4, 4 \ * \ \top)$	✓	✓

Semantics of separating implication

We use the separating implication (magic wand operator) to compose elementary assertions made using the points to relation on disjoint heaps:

$$s, h \models P \mathrel{-\!\!*} Q \quad \text{iff} \quad \text{for all heaps } h' :$$
$$\text{if } \mathit{domain}(h') \cap \mathit{domain}(h) = \emptyset \text{ and } s, h' \models P$$
$$\text{then } s, h \cup h' \models Q$$

$P \mathrel{-\!\!*} Q$ means that if the heap is extended in a way that
P holds in the disjoint extension,
then Q holds in the whole extended heap

Semantics of separating implication

We use the separating implication (magic wand operator) to compose elementary assertions made using the points to relation on disjoint heaps:

$$s, h \models P \mathrel{-\!\!*} Q \quad \text{iff} \quad \text{for all heaps } h' : \\ \text{if } \mathit{domain}(h') \cap \mathit{domain}(h) = \emptyset \text{ and } s, h' \models P \\ \text{then } s, h \cup h' \models Q$$

$P \mathrel{-\!\!*} Q$ means that if the heap is extended in a way that
P holds in the disjoint extension,
then Q holds in the whole extended heap

The separating implication is mainly used to prove theoretical results such as the completeness of certain separation logic fragments.

Regular vs. separating conjunction

Similarities between $\wedge$ and $*$:

$\wedge$			$*$		
$P \wedge Q$	iff	$Q \wedge P$	$P * Q$	iff	$Q * P$
$P \wedge \top$	iff	P	$P * emp$	iff	P
$P \wedge (P \Longrightarrow Q)$	implies	Q	$P * (P \mathrel{-\!\!*} Q)$	implies	Q

Differences between $\wedge$ and $*$:

$\wedge$			$*$		
P	implies	$P \wedge P$	P	does <u>not</u> imply	$P * P$
$P \wedge P$	implies	P	$P * P$	does <u>not</u> imply	P
$P \wedge \neg P$	iff	$\bot$	$P * \neg P$	is	satisfiable

Regular vs. separating conjunction

Similarities between $\wedge$ and $*$:

$\wedge$			$*$		
$P \wedge Q$	iff	$Q \wedge P$	$P * Q$	iff	$Q * P$
$P \wedge \top$	iff	P	$P * \mathit{emp}$	iff	P
$P \wedge (P \Longrightarrow Q)$	implies	Q	$P * (P \mathrel{-\!\!*} Q)$	implies	Q

Differences between $\wedge$ and $*$:

$\wedge$			$*$		
P	implies	$P \wedge P$	P	does <u>not</u> imply	$P * P$
$P \wedge P$	implies	P	$P * P$	does <u>not</u> imply	P
$P \wedge \neg P$	iff	$\bot$	$P * \neg P$	is	satisfiable

For example, if $\mathit{one} \triangleq \exists x, y \bullet (x \mapsto y)$:

$\mathit{one} \wedge \neg(\mathit{one} * \mathit{one})$	is satisfiable
$\neg\mathit{one} \wedge (\mathit{one} * \mathit{one})$	is satisfiable
$\mathit{one} * \neg\mathit{one}$	is satisfiable

Locality of separation logic

$$one * \neg one$$

Locality of separation logic

in some portion of the heap

$$\mathit{one} * \neg \mathit{one}$$

in some other portion of the heap

Locality of separation logic

in some portion of the heap

$$\mathit{one} * \neg \mathit{one}$$

in some other portion of the heap

To understand separation logic assertions you should always think locally.

Peter O'Hearn 

Separation logic

Axiomatic semantics

Axiomatic semantics with separation logic

The proof rules for store assignments, procedure (calls), sequencing, conditionals, and loops are the same as in standard Hoare logic.

We need new axioms to reason about statements that manipulate the heap. These are called the small axioms of separation logic, since they axiomatize commands with local predicates.

$$\{E \mapsto -\}\ \texttt{[}E\texttt{]} \mathrel{\texttt{:=}} F\ \{E \mapsto F\}$$

$$\{E \mapsto -\}\ \textbf{dispose}\ E\ \{emp\}$$

E must not mention v

$$\{emp\}\ \mathsf{v} \mathrel{\texttt{:=}} \textbf{new}\ E\ \{\mathsf{v} \mapsto E\}$$

$$\{\mathsf{v} = \bar{v} \wedge E \mapsto e\}\ \mathsf{v} \mathrel{\texttt{:=}} \texttt{[}E\texttt{]}\ \{v = e\ \wedge\ E[\mathsf{v} \mapsto \bar{v}] \mapsto e\}$$

We also need a new rule of constancy – called the frame rule – which uses the separating conjunction and enables composition of local proofs:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P * R\}\ S\ \{Q * R\}}$$

Small axiom: writing to heap

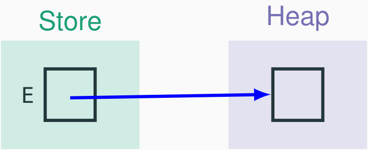
$$\frac{}{\{E \mapsto -\} \quad [E] := F \quad \{E \mapsto F\}}$$

If E is allocated (it "points to something"), then it points to F after writing F to E.

Small axiom: writing to heap

$$\frac{}{\{E \mapsto -\} \quad [E] := F \quad \{E \mapsto F\}}$$

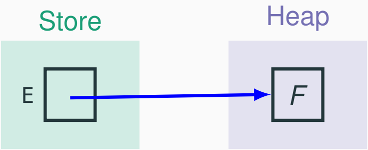
If E is allocated (it "points to something"), then it points to F after writing F to E.



Small axiom: writing to heap

$$\overline{\{E \mapsto -\} \quad [E] := F \quad \{E \mapsto F\}}$$

If E is allocated (it "points to something"), then it points to F after writing F to E.



Small axiom: deallocation

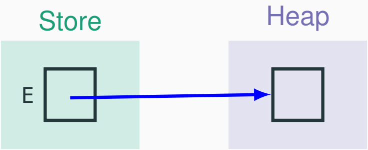
$$\overline{\{E \mapsto -\} \quad \texttt{dispose}\ E \quad \{emp\}}$$

If only E is allocated, then
the heap is empty after deallocating E.

Small axiom: deallocation

$$\frac{}{\{E \mapsto -\} \quad \texttt{dispose}\ E \quad \{emp\}}$$

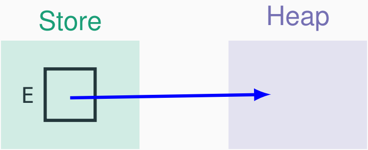
If only E is allocated, then
the heap is empty after deallocating E.



Small axiom: deallocation

$$\overline{\{E \mapsto -\} \quad \texttt{dispose}\ E \quad \{emp\}}$$

If only E is allocated, then
the heap is empty after deallocating E.



Small axiom: allocation

E must not mention v

$$\frac{}{\{emp\} \quad \mathsf{v} := \mathbf{new}\ E \quad \{\mathsf{v} \mapsto E\}}$$

If the heap is empty, then v points to
a newly allocated cell storing E after allocation.

The axiom for when E references v is a bit more complex.

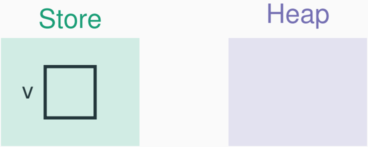
Small axiom: allocation

E must not mention v

$$\frac{}{\{emp\} \quad \mathsf{v} := \mathbf{new}\ E \quad \{\mathsf{v} \mapsto E\}}$$

If the heap is empty, then v points to a newly allocated cell storing *E* after allocation.

The axiom for when *E* references v is a bit more complex.



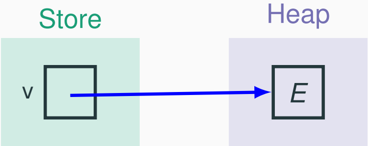
Small axiom: allocation

E must not mention $\mathtt{v}$

$$\frac{}{\{\mathit{emp}\} \quad \mathtt{v} := \mathtt{\textbf{new}}\ E \quad \{\mathtt{v} \mapsto E\}}$$

If the heap is empty, then $\mathtt{v}$ points to
a newly allocated cell storing E after allocation.

The axiom for when E references $\mathtt{v}$ is a bit more complex.



Small axiom: reading from heap

$$\overline{\{\mathsf{v} = \bar{v} \wedge E \mapsto e\} \quad \mathtt{v := [E]} \quad \{v = e \ \wedge \ E[\mathsf{v} \mapsto \bar{v}] \mapsto e\}}$$

If E points to e, then v stores e after dereferencing E (reading from the heap at location E) and assigning the result to v.

This is just a variant of Hoare logic's forward assignment axiom ($\bar{v}$ is `old(v)` in the post-state).

The substitution $E[\mathsf{v} \mapsto \bar{v}]$ is needed when E is an expression that involves v – it must be re-expressed to refer to $\bar{v}$ in the post-state.

Small axiom: reading from heap

$$\frac{}{\{\mathsf{v} = \bar{v} \wedge E \mapsto e\} \quad \mathtt{v := [}E\mathtt{]} \quad \{v = e \ \wedge \ E[\mathsf{v} \mapsto \bar{v}] \mapsto e\}}$$

If E points to e, then v stores e after dereferencing E (reading from the heap at location E) and assigning the result to v.

This is just a variant of Hoare logic's forward assignment axiom ($\bar{v}$ is `old(v)` in the post-state).

The substitution $E[\mathsf{v} \mapsto \bar{v}]$ is needed when E is an expression that involves v – it must be re-expressed to refer to $\bar{v}$ in the post-state.

Small axiom: reading from heap

$$\frac{}{\{\mathsf{v} = \bar{v} \wedge E \mapsto e\} \quad \texttt{v := [}E\texttt{]} \quad \{v = e \ \wedge \ E[\mathsf{v} \mapsto \bar{v}] \mapsto e\}}$$

If E points to e, then v stores e after dereferencing E (reading from the heap at location E) and assigning the result to v.

This is just a variant of Hoare logic's forward assignment axiom ($\bar{v}$ is `old(v)` in the post-state).

The substitution $E[\mathsf{v} \mapsto \bar{v}]$ is needed when E is an expression that involves v – it must be re-expressed to refer to $\bar{v}$ in the post-state.

Rule of constancy

Remember that the rule of constancy is unsound in the presence of reference variables: even if R does not refer to any reference variable that S may modify, it may still refer to variables that are aliased by some variable that S may modify.

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \wedge R\}\ S\ \{Q \wedge R\}}$$

Using the separating conjunction of separation logic, we can define a variant of the rule of constancy – called the frame rule – that is sound:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P * R\}\ S\ \{Q * R\}}$$

Just like the rule of constancy, the frame rule is syntactic: the frame $\mathcal{F}(S)$ of S is the set of all store variables written to in an assignment or in the frame of a called procedure. Since P, Q and R refer to disjoint parts of the heap, there can be no implicit aliasing of R's variables!

Fault-avoiding semantics of Hoare triples

With heap-manipulating programs, we have to specify what happens when a command tries to access an unallocated address. In separation logic, it is customary to use the fault-avoiding interpretation.

$\{P\}\ S\ \{Q\}$ is valid under the fault-avoiding interpretation if
executing S in a state that satisfies P
does not fault and leads to a state that satisfies Q.

Without this interpretation, every program that faults would trivially be correct.

Separation logic

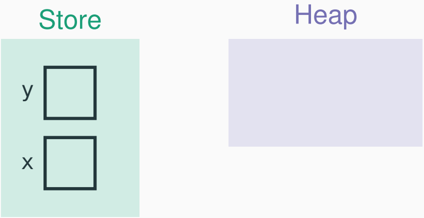
Proofs

Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 * true }
```

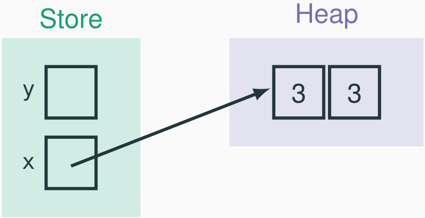


Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 ∗ true }
```

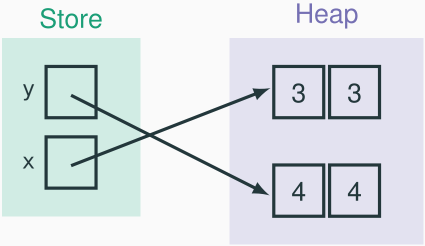


Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 ∗ true }
```

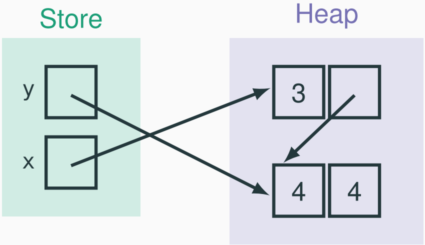


Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 ∗ true }
```

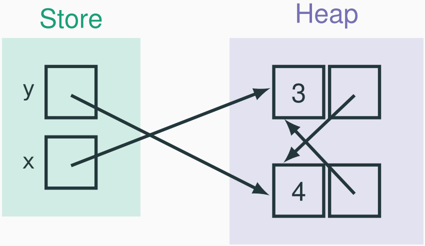


Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 ∗ true }
```

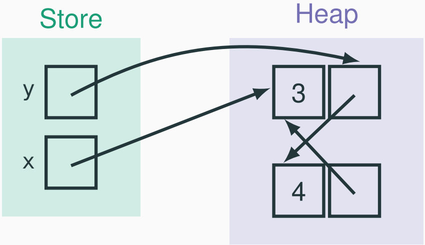


Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 ∗ true }
```

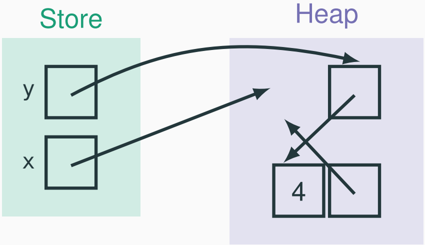


Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 ∗ true }
```

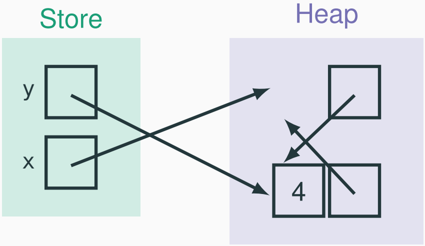


Proof of a simple program

Let's write a proof outline of this pointer-manipulating program.

- Reason forward
- Perform local proofs using the small axioms
- Combine local proofs into the overall proof using the frame rule

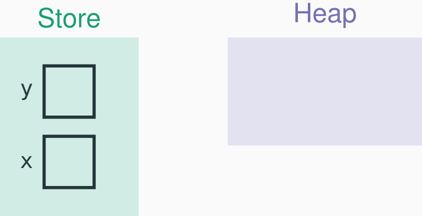
```
{ emp }
x := new 3, 3
y := new 4, 4
[x + 1] := y
[y + 1] := x
y := x + 1
dispose x
y := [y]
{ y ↦ 4 ∗ true }
```



Proof of a simple program

$\{emp\}$

```
x := new 3, 3

y := new 4, 4

[x + 1] := y

[y + 1] := x

y := x + 1

dispose x

y := [y]
```



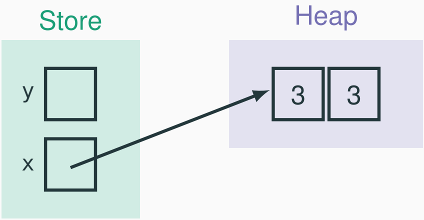
Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4

[x + 1] := y

[y + 1] := x

y := x + 1

dispose x

y := [y]
```



Small axiom for allocation:

$\{emp\}$ `v := new` E $\{v \mapsto E\}$

Proof of a simple program

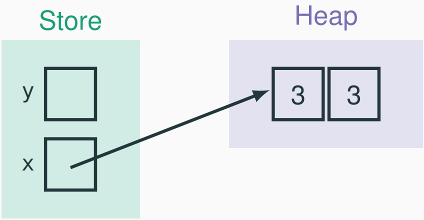
$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

$\{emp * x \mapsto 3, 3\}$

```
y := new 4, 4

[x + 1] := y

[y + 1] := x

y := x + 1

dispose x

y := [y]
```



Identity of $*$ *emp*:

$$P \text{ iff } emp * P$$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

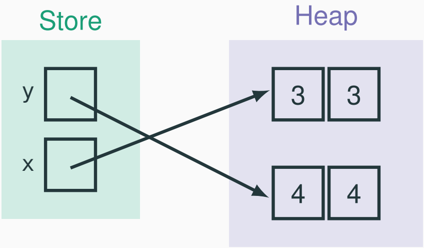
$\{x \mapsto 3, 3\}$

$\{emp * x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 * x \mapsto 3, 3\}$

```
[x + 1] := y

[y + 1] := x

y := x + 1

dispose x

y := [y]
```



Local proof using allocation axiom:

$\{emp\}$ `y := new 4, 4` $\{y \mapsto 4, 4\}$

Combination using the frame rule:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P * R\}\ S\ \{Q * R\}}$$

Proof of a simple program

$\{emp\}$

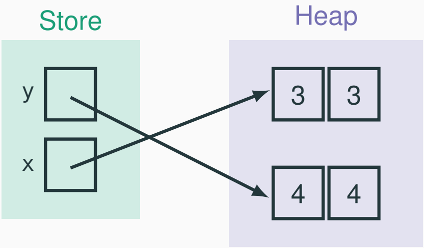
```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \, * \, x \mapsto 3, 3\}$

```
[x + 1] := y


[y + 1] := x


y := x + 1


dispose x


y := [y]
```



Local proof using allocation axiom:

$$\{emp\}\ \texttt{y := new 4, 4}\ \{y \mapsto 4, 4\}$$

Combination using the frame rule:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \, * \, R\}\ S\ \{Q \, * \, R\}}$$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, 3\}$

$\{y \mapsto 4, 4 \ast x \mapsto 3 \ast x + 1 \mapsto 3\}$

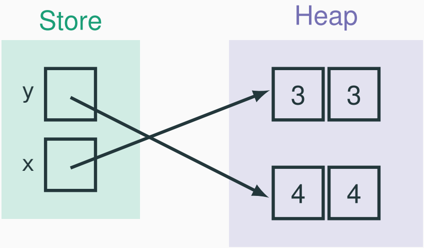
```
[x + 1] := y
```

```
[y + 1] := x
```

```
y := x + 1
```

```
dispose x
```

```
y := [y]
```



Definition of $E \mapsto F_1, F_2$:

$E \mapsto F_1, F_2$ iff $E \mapsto F_1 \ast E + 1 \mapsto F_2$

Proof of a simple program

$\{emp\}$

`x := new 3, 3`

$\{x \mapsto 3, 3\}$

`y := new 4, 4`

$\{y \mapsto 4, 4 * x \mapsto 3, 3\}$

$\{y \mapsto 4, 4 * x \mapsto 3 * x + 1 \mapsto 3\}$

`[x + 1] := y`

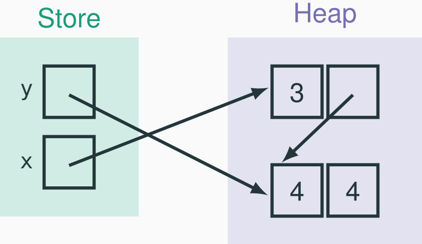
$\{y \mapsto 4, 4 * x \mapsto 3 * x + 1 \mapsto y\}$

`[y + 1] := x`

`y := x + 1`

`dispose x`

`y := [y]`



Local proof using
heap writing axiom:

$\{x+1 \mapsto -\}$ `[x + 1] := y` $\{x+1 \mapsto y\}$

Combination using the frame rule:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P * R\}\ S\ \{Q * R\}}$$

Note $\mathcal{F}(S) = \emptyset$ because no store variables are written to.

Proof of a simple program

$\{emp\}$

```
 x := new 3, 3
```

$\{x \mapsto 3, 3\}$

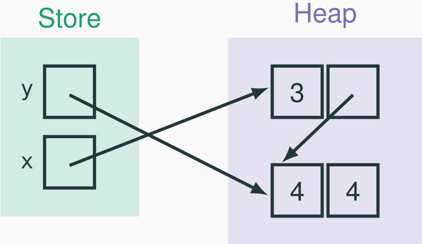
```
 y := new 4, 4
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, 3\}$

```
 [x + 1] := y
```

$\{y \mapsto 4, 4 \ast x \mapsto 3 \ast x + 1 \mapsto y\}$

```
 [y + 1] := x

 y := x + 1

 dispose x

 y := [y]
```



Local proof using
<u>heap writing axiom</u>:

$\{x{+}1 \mapsto -\}$ `[x + 1] := y` $\{x{+}1 \mapsto y\}$

Combination using the frame rule:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \ast R\}\ S\ \{Q \ast R\}}$$

Note $\mathcal{F}(S) = \emptyset$ because no store variables are written to.

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{\mathtt{x} \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{\mathtt{y} \mapsto 4, 4 \ * \ \mathtt{x} \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{\mathtt{y} \mapsto 4, 4 \ * \ \mathtt{x} \mapsto 3 \ * \ \mathtt{x} + 1 \mapsto \mathtt{y}\}$

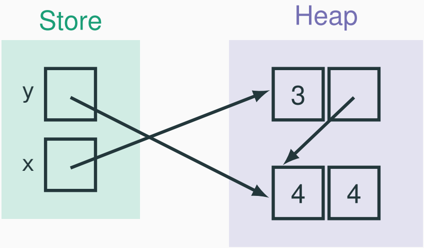
$\{\mathtt{y} \mapsto 4 \ * \ \mathtt{y} + 1 \mapsto 4 \ * \ \mathtt{x} \mapsto 3 \ * \ \mathtt{x} + 1 \mapsto \mathtt{y}\}$

```
[y + 1] := x
```

```
y := x + 1
```

```
dispose x
```

```
y := [y]
```



Definition of $E \mapsto F_1, F_2$:

$$E \mapsto F_1, F_2 \text{ iff } E \mapsto F_1 \ * \ E + 1 \mapsto F_2$$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

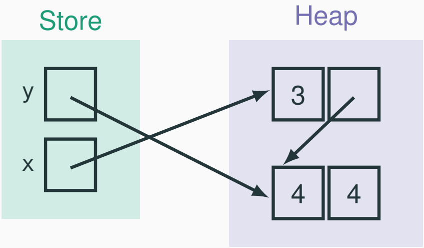
```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4 \ast y + 1 \mapsto 4 \ast x \mapsto 3 \ast x + 1 \mapsto y\}$

```
[y + 1] := x

y := x + 1

dispose x

y := [y]
```



Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4 \ast y + 1 \mapsto 4 \ast x \mapsto 3 \ast x + 1 \mapsto y\}$

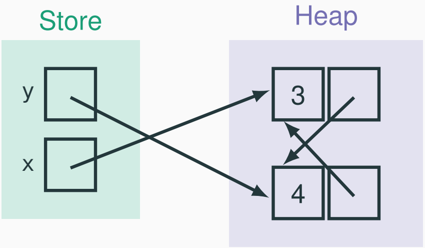
```
[y + 1] := x
```

$\{y \mapsto 4 \ast y + 1 \mapsto x \ast x \mapsto 3 \ast x + 1 \mapsto y\}$

```
y := x + 1
```

```
dispose x
```

```
y := [y]
```



Local proof using heap writing axiom:

$$\{y+1 \mapsto -\}\ \texttt{[y + 1] := x}\ \{y+1 \mapsto x\}$$

Combination using the frame rule:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \ast R\}\ S\ \{Q \ast R\}}$$

Note $\mathcal{F}(S) = \emptyset$ because no store variables are written to.

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4 \ * \ y + 1 \mapsto 4 \ * \ x \mapsto 3 \ * \ x + 1 \mapsto y\}$

```
[y + 1] := x
```

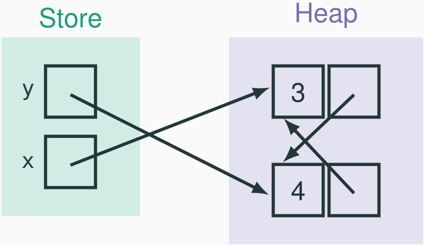
$\{y \mapsto 4 \ * \ y + 1 \mapsto x \ * \ x \mapsto 3 \ * \ x + 1 \mapsto y\}$

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

```
y := x + 1
```

```
dispose x
```

```
y := [y]
```



Definition of $E \mapsto F_1, F_2$:

$E \mapsto F_1, F_2$ iff $E \mapsto F_1 \ * \ E + 1 \mapsto F_2$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

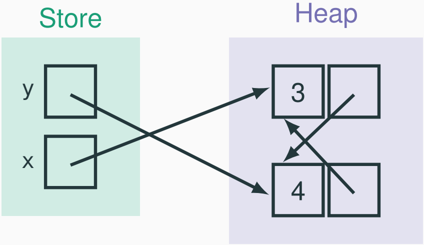
```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

```
y := x + 1

dispose x

y := [y]
```



Definition of $E \mapsto F_1, F_2$:

$$E \mapsto F_1, F_2 \text{ iff } E \mapsto F_1 \ * \ E + 1 \mapsto F_2$$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{\mathrm{x} \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{\mathrm{y} \mapsto 4, 4 \; * \; \mathrm{x} \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{\mathrm{y} \mapsto 4, 4 \; * \; \mathrm{x} \mapsto 3, \mathrm{y}\}$

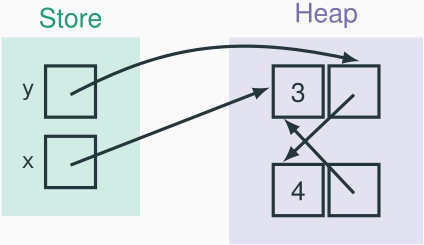
```
[y + 1] := x
```

$\{\mathrm{y} \mapsto 4, \mathrm{x} \; * \; \mathrm{x} \mapsto 3, \mathrm{y}\}$

```
y := x + 1
```

$\{\bar{y} \mapsto 4, \mathrm{x} \; * \; \mathrm{x} \mapsto 3, \bar{y} \wedge \mathrm{y} = \mathrm{x} + 1\}$

```
dispose x

y := [y]
```



Standard Hoare (global) forward assignment axiom:

$$\{P\} \; \texttt{y := x + 1} \; \left\{\exists \bar{y} \begin{pmatrix} P[\mathrm{y} \mapsto \bar{y}] \wedge \\ \mathrm{y} = \mathrm{x} + 1 \end{pmatrix}\right\}$$

old(y)

We leave the quantification implicit using a fresh variable.
There is no local reasoning here.

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

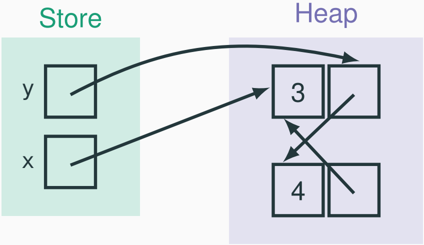
```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ * \ x \mapsto 3, \bar{y} \wedge y = x + 1\}$

$\{\bar{y} \mapsto 4, x \ * \ x \mapsto 3 \ * \ x + 1 \mapsto \bar{y} \wedge y = x + 1\}$

```
dispose x
```

```
y := [y]
```



Definition of $E \mapsto F_1, F_2$:

$$E \mapsto F_1, F_2 \text{ iff } E \mapsto F_1 \ * \ E + 1 \mapsto F_2$$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{\mathrm{x} \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{\mathrm{y} \mapsto 4, 4 \ast \mathrm{x} \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{\mathrm{y} \mapsto 4, 4 \ast \mathrm{x} \mapsto 3, \mathrm{y}\}$

```
[y + 1] := x
```

$\{\mathrm{y} \mapsto 4, \mathrm{x} \ast \mathrm{x} \mapsto 3, \mathrm{y}\}$

```
y := x + 1
```

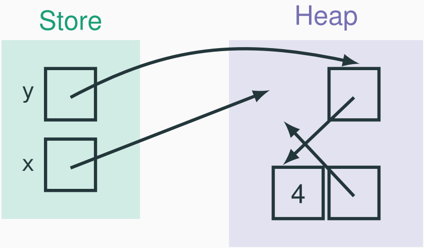
$\{\bar{y} \mapsto 4, \mathrm{x} \ast \mathrm{x} \mapsto 3, \bar{y} \wedge \mathrm{y} = \mathrm{x} + 1\}$

$\{\bar{y} \mapsto 4, \mathrm{x} \ast \mathrm{x} \mapsto 3 \ast \mathrm{x} + 1 \mapsto \bar{y} \wedge \mathrm{y} = \mathrm{x} + 1\}$

```
dispose x
```

$\{\bar{y} \mapsto 4, \mathrm{x} \ast emp \ast \mathrm{x} + 1 \mapsto \bar{y} \wedge \mathrm{y} = \mathrm{x} + 1\}$

```
y := [y]
```



Local proof using deallocation axiom:

$$\{\mathrm{x} \mapsto -\}\ \textbf{dispose}\ \mathrm{x}\ \{emp\}$$

Combination using the frame rule:

$$\frac{\{P\}\ S\ \{Q\} \quad \mathcal{V}(R) \cap \mathcal{F}(S) = \emptyset}{\{P \ast R\}\ S\ \{Q \ast R\}}$$

Note $\mathcal{F}(S) = \emptyset$.

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{\mathsf{x} \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

```
[y + 1] := x
```

$\{\mathsf{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

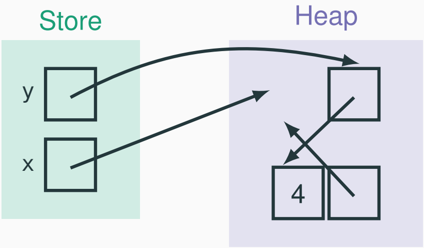
```
y := x + 1
```

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

```
dispose x
```

$\{\bar{y} \mapsto 4, \mathsf{x} \ast emp \ast \mathsf{x} + 1 \mapsto \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

```
y := [y]
```



Since $\mathsf{y} = \mathsf{x} + 1$ only refers to the store, and it applies to all heap assertions that involve $\mathsf{x} + 1$ or y. Thus:

$R = \bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} + 1 \mapsto \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1$

in the previous application of the frame rule.

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{\mathrm{x} \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{\mathrm{y} \mapsto 4, 4 \ast \mathrm{x} \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{\mathrm{y} \mapsto 4, 4 \ast \mathrm{x} \mapsto 3, \mathrm{y}\}$

```
[y + 1] := x
```

$\{\mathrm{y} \mapsto 4, \mathrm{x} \ast \mathrm{x} \mapsto 3, \mathrm{y}\}$

```
y := x + 1
```

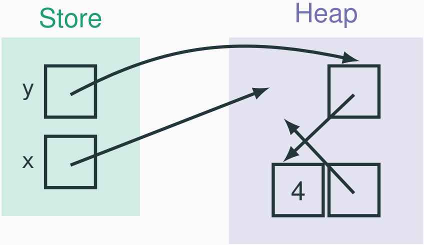
$\{\bar{y} \mapsto 4, \mathrm{x} \ast \mathrm{x} \mapsto 3, \bar{y} \wedge \mathrm{y} = \mathrm{x} + 1\}$

```
dispose x
```

$\{\bar{y} \mapsto 4, \mathrm{x} \ast emp \ast \mathrm{x} + 1 \mapsto \bar{y} \wedge \mathrm{y} = \mathrm{x} + 1\}$

$\{\bar{y} \mapsto 4, \mathrm{x} \ast \mathrm{x} + 1 \mapsto \bar{y} \wedge \mathrm{y} = \mathrm{x} + 1\}$

```
y := [y]
```



Identity of $\ast$ *emp*:

$$P \text{ iff } emp \ast P$$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

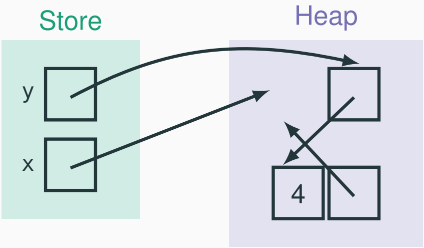
```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ * \ x \mapsto 3, \bar{y} \wedge y = x + 1\}$

```
dispose x
```

$\{\bar{y} \mapsto 4, x \ * \ x + 1 \mapsto \bar{y} \wedge y = x + 1\}$

```
y := [y]
```



Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{\mathsf{x} \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

```
[y + 1] := x
```

$\{\mathsf{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

```
y := x + 1
```

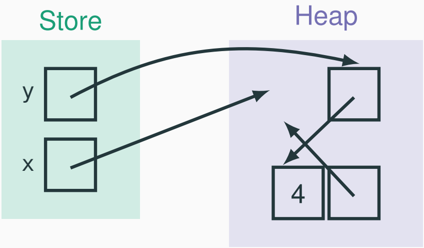
$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

```
dispose x
```

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} + 1 \mapsto \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{y} \mapsto \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

```
y := [y]
```



Congruence (logic equivalence).

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

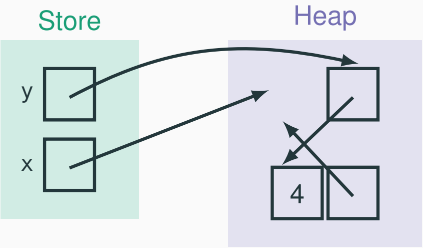
```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ * \ x \mapsto 3, \bar{y} \wedge y = x + 1\}$

```
dispose x
```

$\{\bar{y} \mapsto 4, x \ * \ y \mapsto \bar{y} \wedge y = x + 1\}$

```
y := [y]
```



Proof of a simple program

$\{\mathit{emp}\}$

`x := new 3, 3`

$\{\mathsf{x} \mapsto 3, 3\}$

`y := new 4, 4`

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, 3\}$

`[x + 1] := y`

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

`[y + 1] := x`

$\{\mathsf{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

`y := x + 1`

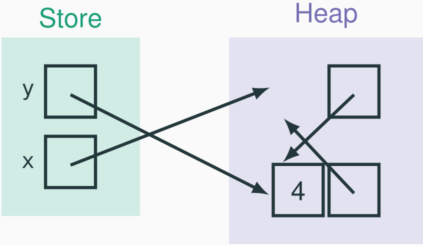
$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

`dispose x`

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{y} \mapsto \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

`y := [y]`

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = \mathsf{x} + 1 \wedge \mathsf{y} = \bar{y}\}$



Local proof using heap reading axiom:

$$\{\mathsf{y} \mapsto \bar{y} \wedge \mathsf{y} = \bar{\bar{y}}\}$$

`y := [y]`

$$\{y = \bar{y} \ \wedge \ \mathsf{y}[\mathsf{y} \mapsto \bar{\bar{y}}] \mapsto \bar{y}\}$$

Combination using the frame rule (and the rule of constancy to handle the conjunct `y = x + 1` in the pre-state).

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ * \ x \mapsto 3, \bar{y} \wedge y = x + 1\}$

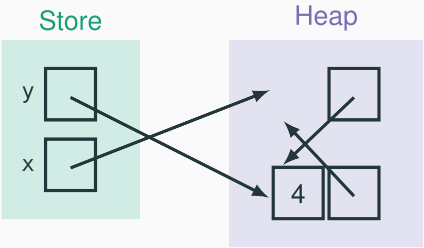
```
dispose x
```

$\{\bar{y} \mapsto 4, x \ * \ y \mapsto \bar{y} \wedge y = x + 1\}$

```
y := [y]
```

$\{\bar{y} \mapsto 4, x \ * \ \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = x + 1 \wedge y = \bar{y}\}$

$\{y \mapsto 4, x \ * \ \bar{\bar{y}} \mapsto y \wedge \bar{\bar{y}} = x + 1 \wedge y = \bar{y}\}$



Congruence (logic equivalence).

Proof of a simple program

$\{\mathit{emp}\}$

`x := new 3, 3`

$\{\mathsf{x} \mapsto 3, 3\}$

`y := new 4, 4`

$\{\mathsf{y} \mapsto 4, 4 \ * \ \mathsf{x} \mapsto 3, 3\}$

`[x + 1] := y`

$\{\mathsf{y} \mapsto 4, 4 \ * \ \mathsf{x} \mapsto 3, \mathsf{y}\}$

`[y + 1] := x`

$\{\mathsf{y} \mapsto 4, \mathsf{x} \ * \ \mathsf{x} \mapsto 3, \mathsf{y}\}$

`y := x + 1`

$\{\bar{y} \mapsto 4, \mathsf{x} \ * \ \mathsf{x} \mapsto 3, \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

`dispose x`

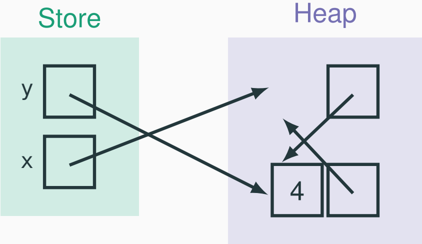
$\{\bar{y} \mapsto 4, \mathsf{x} \ * \ \mathsf{y} \mapsto \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

`y := [y]`

$\{\bar{y} \mapsto 4, \mathsf{x} \ * \ \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = \mathsf{x} + 1 \wedge \mathsf{y} = \bar{y}\}$

$\{\mathsf{y} \mapsto 4, \mathsf{x} \ * \ \bar{\bar{y}} \mapsto \mathsf{y} \wedge \bar{\bar{y}} = \mathsf{x} + 1 \wedge \mathsf{y} = \bar{y}\}$

$\{\mathsf{y} \mapsto 4, \mathsf{x} \ * \ \bar{\bar{y}} \mapsto \mathsf{y}\}$



Rule of consequence (postcondition weakening).

Proof of a simple program

$\{\mathit{emp}\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ast x \mapsto 3, y\}$

```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ast x \mapsto 3, \bar{y} \wedge y = x + 1\}$

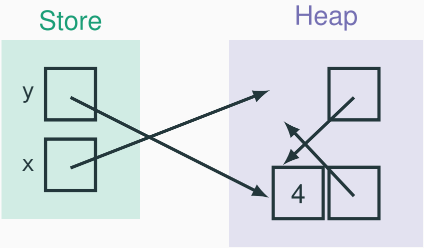
```
dispose x
```

$\{\bar{y} \mapsto 4, x \ast y \mapsto \bar{y} \wedge y = x + 1\}$

```
y := [y]
```

$\{\bar{y} \mapsto 4, x \ast \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = x + 1 \wedge y = \bar{y}\}$

$\{y \mapsto 4, x \ast \bar{\bar{y}} \mapsto y\}$



Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ * \ x \mapsto 3, \bar{y} \wedge y = x + 1\}$

```
dispose x
```

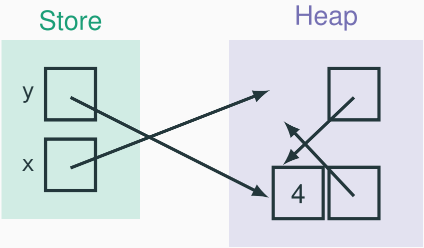
$\{\bar{y} \mapsto 4, x \ * \ y \mapsto \bar{y} \wedge y = x + 1\}$

```
y := [y]
```

$\{\bar{y} \mapsto 4, x \ * \ \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = x + 1 \wedge y = \bar{y}\}$

$\{y \mapsto 4, x \ * \ \bar{\bar{y}} \mapsto y\}$

$\{y \mapsto 4 \ * \ (y + 1 \mapsto x \ * \ \bar{\bar{y}} \mapsto y)\}$



Definition of $E \mapsto F_1, F_2$:

$$E \mapsto F_1, F_2 \text{ iff } E \mapsto F_1 \ * \ E + 1 \mapsto F_2$$

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{\mathsf{x} \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{\mathsf{y} \mapsto 4, 4 \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

```
[y + 1] := x
```

$\{\mathsf{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \mathsf{y}\}$

```
y := x + 1
```

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{x} \mapsto 3, \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

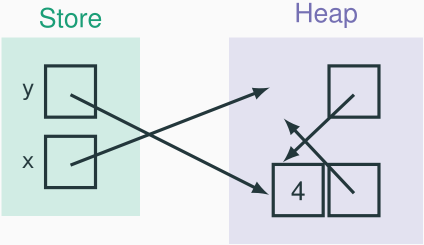
```
dispose x
```

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \mathsf{y} \mapsto \bar{y} \wedge \mathsf{y} = \mathsf{x} + 1\}$

```
y := [y]
```

$\{\bar{y} \mapsto 4, \mathsf{x} \ast \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = \mathsf{x} + 1 \wedge \mathsf{y} = \bar{y}\}$

$\{\mathsf{y} \mapsto 4 \ast (\mathsf{y} + 1 \mapsto \mathsf{x} \ast \bar{\bar{y}} \mapsto \mathsf{y})\}$



Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ * \ x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ * \ x \mapsto 3, y\}$

```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ * \ x \mapsto 3, \bar{y} \wedge y = x + 1\}$

```
dispose x
```

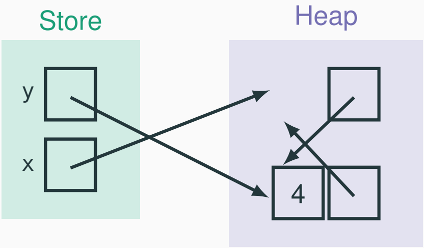
$\{\bar{y} \mapsto 4, x \ * \ y \mapsto \bar{y} \wedge y = x + 1\}$

```
y := [y]
```

$\{\bar{y} \mapsto 4, x \ * \ \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = x + 1 \wedge y = \bar{y}\}$

$\{y \mapsto 4 \ * \ (y + 1 \mapsto x \ * \ \bar{\bar{y}} \mapsto y)\}$

$\{y \mapsto 4 \ * \ \top\}$



Rule of consequence
(postcondition weakening).

Proof of a simple program

$\{emp\}$

```
x := new 3, 3
```

$\{x \mapsto 3, 3\}$

```
y := new 4, 4
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, 3\}$

```
[x + 1] := y
```

$\{y \mapsto 4, 4 \ast x \mapsto 3, y\}$

```
[y + 1] := x
```

$\{y \mapsto 4, x \ast x \mapsto 3, y\}$

```
y := x + 1
```

$\{\bar{y} \mapsto 4, x \ast x \mapsto 3, \bar{y} \wedge y = x + 1\}$

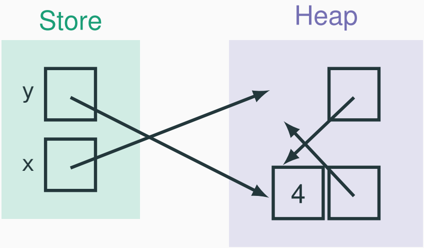
```
dispose x
```

$\{\bar{y} \mapsto 4, x \ast y \mapsto \bar{y} \wedge y = x + 1\}$

```
y := [y]
```

$\{\bar{y} \mapsto 4, x \ast \bar{\bar{y}} \mapsto \bar{y} \wedge \bar{\bar{y}} = x + 1 \wedge y = \bar{y}\}$

$\{y \mapsto 4 \ast \top\}$



Tree disposal

Let us now consider a more challenging example: a tree deallocation procedure.

```
procedure dispose_tree (root: ref Node):
require // root is a tree
ensure  // the tree has been deallocated
  var left, right: ref Node
  if root ≠ nil
    left := [root.left]
    right := [root.right]
    dispose_tree(left)
    dispose_tree(right)
    dispose root
```

First let's express the specification using separation logic.

Tree predicate

A predicate `tree`(p) expresses that p is the root of a well-formed binary tree.

Since the tree can have an arbitrary height, we need an inductive predicate.

Tree predicate

A predicate $\texttt{tree}(p)$ expresses that p is the root of a well-formed binary tree.

Since the tree can have an arbitrary height, we need an inductive predicate.

$\texttt{nil}$ is an empty tree

$$\texttt{tree}(p) \iff \begin{cases} emp & \text{if } p = \texttt{nil} \\ p \mapsto p.\texttt{left}, p.\texttt{right} * \texttt{tree}(p.\texttt{left}) * p.\texttt{right} & \text{otherwise} \end{cases}$$

with typed references, this
would be guaranteed by the type system

Tree predicate

A predicate $\texttt{tree}(p)$ expresses that p is the root of a well-formed binary tree.

Since the tree can have an arbitrary height, we need an inductive predicate.

`nil` is an empty tree

$$\texttt{tree}(p) \Longleftrightarrow \begin{cases} emp & \text{if } p = \texttt{nil} \\ p \mapsto p.\texttt{left}, p.\texttt{right} \;*\; \texttt{tree}(p.\texttt{left}) \;*\; p.\texttt{right} & \text{otherwise} \end{cases}$$

with typed references, this would be guaranteed by the type system

We treat `left` and `right` as if they indicated fixed memory offsets in the heap. With this assumption, how to apply the axioms of separation logic will be straightforward.

Tree disposal: specification

```
procedure dispose_tree (root: ref Node):
require // root is a tree
ensure  // the tree has been deallocated
```

We formalize the pre- and postcondition using predicate `tree`.

Tree disposal: specification

```
procedure dispose_tree (root: ref Node):
require // root is a tree
ensure  // the tree has been deallocated
```

We formalize the pre- and postcondition using predicate `tree`.

```
procedure dispose_tree (root: ref Node):
require tree(root)
ensure  emp
```

Tree disposal: specification

```
procedure dispose_tree (root: ref Node):
require // root is a tree
ensure  // the tree has been deallocated
```

We formalize the pre- and postcondition using predicate `tree`.

```
procedure dispose_tree (root: ref Node):
require tree(root)
ensure  emp
```

The specification is local: it only talks about the heap portion that includes the tree.

Framing does not require a modify clause: pre- and postcondition describe exactly how the local portion of the heap changes.

Tree disposal: Hoare logic specification

Expressing the same specification without separation logic would require auxiliary variables to do framing precisely:

```
procedure dispose_tree (root: ref Node):
require tree(root)
modify reachable(root)
ensure ∀ n (n ∈ old(reachable(root)) ⟹ ¬ allocated(n))
```

`reachable`(n)**:** set of nodes reachable from n
`allocated`(n)**:** is n a valid reference to the heap?

Tree disposal: Hoare logic specification

Expressing the same specification without separation logic would require auxiliary variables to do framing precisely:

```
procedure dispose_tree (root: ref Node):
require tree(root)
modify reachable(root)
ensure ∀ n (n ∈ old(reachable(root)) ⟹ ¬ allocated(n))
```

reachable(n): set of nodes reachable from n
allocated(n): is n a valid reference to the heap?

If we make `dispose_tree` a method of class `Tree`, precondition `tree(root)` could be a class invariant; and the information about allocated nodes could be implicit given garbage collection:

```
procedure dispose_tree ():
require consistent()
modify tree_nodes
ensure tree_nodes = { }
```

Proof of tree disposal: outline

We want to prove:

```
{ tree(root) }
if root ≠ nil
  left := [root.left]
  right := [root.right]
  dispose_tree(left)
  dispose_tree(right)
  dispose root
{ emp }
```

Proof of tree disposal: outline

We want to prove:

```
{ tree(root) }
if root ≠ nil
  left := [root.left]
  right := [root.right]
  dispose_tree(left)
  dispose_tree(right)
  dispose root
{ emp }
```

Using the axiom for conditionals, it reduces to two proofs:

```
{ tree(root) ∧ root ≠ nil }
  left := [root.left]
  right := [root.right]
  dispose_tree(left)
  dispose_tree(right)
  dispose root
{ emp }
```

```
{ tree(root) ∧ root = nil }
  // implies
{ emp }
```

Proof of tree disposal: base case

The (empty) else branch is trivial using the definition of `tree`:

$\{\mathtt{tree}(\mathtt{root}) \wedge \mathtt{root} = \mathtt{nil}\}$
$\{emp \wedge \mathtt{root} = \mathtt{nil}\}$
$\{emp\}$

$$\mathtt{tree}(p) \iff \begin{cases} emp & \text{if } p = \mathtt{nil} \\ p \mapsto p.\mathtt{left}, p.\mathtt{right} \ast \mathtt{tree}(p.\mathtt{left}) \ast p.\mathtt{right} & \text{otherwise} \end{cases}$$

Proof of tree disposal: recursive case

$\{\texttt{tree}(\texttt{root}) \land \texttt{root} \neq \texttt{nil}\}$

```
left := [root.left]
```

```
right := [root.right]
```

```
dispose_tree(left)
```

```
dispose_tree(right)
```

```
dispose root
```

Proof of tree disposal: recursive case

$\{\mathtt{tree}(\mathtt{root}) \land \mathtt{root} \neq \mathtt{nil}\}$

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
left := [root.left]

right := [root.right]

dispose_tree(left)

dispose_tree(right)

dispose root
```

Definition of `tree`.

(Omitting $\mathtt{root} \neq \mathtt{nil}$ for readability.)

Proof of tree disposal: recursive case

$\{\texttt{tree}(\texttt{root}) \wedge \texttt{root} \neq \texttt{nil}\}$

$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{root.left}) * \texttt{tree}(\texttt{root.right})\}$

```
left := [root.left]
```

$\{\texttt{left} = \texttt{root.left} \wedge \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{root.left}) * \texttt{tree}(\texttt{root.right})\}$

```
right := [root.right]
```

```
dispose_tree(left)
```

```
dispose_tree(right)
```

```
dispose root
```

Local proof using heap reading axiom (we ignore the "old" value of `left` since it's immaterial):

$\{\texttt{root}{\mapsto}\texttt{root.left}\}$ `left := [root.left]` $\{\texttt{root}{\mapsto}\texttt{root.left}{\wedge}\texttt{left} = \texttt{root.left}\}$

Combination using the frame rule.

Proof of tree disposal: recursive case

$\{\mathtt{tree}(\mathtt{root}) \land \mathtt{root} \neq \mathtt{nil}\}$

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
left := [root.left]
```

$\{\mathtt{left} = \mathtt{root.left} \land \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
right := [root.right]
```

```
dispose_tree(left)
```

```
dispose_tree(right)
```

```
dispose root
```

Substituting `left` $=$ `root.left` and omitting it for readability.

Proof of tree disposal: recursive case

$\{\texttt{tree}(\texttt{root}) \land \texttt{root} \neq \texttt{nil}\}$

$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{root.left}) * \texttt{tree}(\texttt{root.right})\}$

```
left := [root.left]
```

$\{\texttt{left} = \texttt{root.left} \land \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{root.left}) * \texttt{tree}(\texttt{root.right})\}$

$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{left}) * \texttt{tree}(\texttt{root.right})\}$

```
right := [root.right]
```

$\{\texttt{right} = \texttt{root.right} \land \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{left}) * \texttt{tree}(\texttt{root.right})\}$

```
dispose_tree(left)

dispose_tree(right)

dispose root
```

Local proof using heap reading axiom (we ignore the "old" value of `right` since it's immaterial; N is a node's size):

$\{\texttt{root} + N \mapsto \texttt{root.right}\}$ `right := [root.right]` $\{\texttt{root} + N \mapsto \texttt{root.right} \land \texttt{right} = \texttt{root.right}\}$

Combination using the frame rule.

Proof of tree disposal: recursive case

$\{\mathtt{tree}(\mathtt{root}) \wedge \mathtt{root} \neq \mathtt{nil}\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
left := [root.left]
```

$\{\mathtt{left} = \mathtt{root.left} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
right := [root.right]
```

$\{\mathtt{right} = \mathtt{root.right} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{root.right})\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{right})\}$

```
dispose_tree(left)

dispose_tree(right)

dispose root
```

Substituting $\mathtt{right} = \mathtt{root.right}$ and omitting it for readability.

Proof of tree disposal: recursive case

$\{\texttt{tree}(\texttt{root}) \wedge \texttt{root} \neq \texttt{nil}\}$

$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{root.left}) * \texttt{tree}(\texttt{root.right})\}$

```
left := [root.left]
```

$\{\texttt{left} = \texttt{root.left} \wedge \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{root.left}) * \texttt{tree}(\texttt{root.right})\}$

$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{left}) * \texttt{tree}(\texttt{root.right})\}$

```
right := [root.right]
```

$\{\texttt{right} = \texttt{root.right} \wedge \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{left}) * \texttt{tree}(\texttt{root.right})\}$

$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree}(\texttt{left}) * \texttt{tree}(\texttt{right})\}$

```
dispose_tree(left)
```

$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \mathit{emp} * \texttt{tree}(\texttt{right})\}$

```
dispose_tree(right)
```

```
dispose root
```

Local proof using `dispose_tree`'s specification:

$$\{\texttt{tree}(\texttt{left})\}\ \texttt{dispose\_tree(left)}\ \{\mathit{emp}\}$$

Combination using the frame rule.

Proof of tree disposal: recursive case

$\{\mathtt{tree}(\mathtt{root}) \wedge \mathtt{root} \neq \mathtt{nil}\}$

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
left := [root.left]
```

$\{\mathtt{left} = \mathtt{root.left} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
right := [root.right]
```

$\{\mathtt{right} = \mathtt{root.right} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{root.right})\}$

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{right})\}$

```
dispose_tree(left)
```

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathit{emp} * \mathtt{tree}(\mathtt{right})\}$

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{right})\}$

```
dispose_tree(right)
```

```
dispose root
```

Identity of $*$ *emp*:

$$P \text{ iff } \mathit{emp} * P$$

Proof of tree disposal: recursive case

$\{\mathtt{tree}(\mathtt{root}) \wedge \mathtt{root} \neq \mathtt{nil}\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
left := [root.left]
```

$\{\mathtt{left} = \mathtt{root.left} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{root.left}) * \mathtt{tree}(\mathtt{root.right})\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{root.right})\}$

```
right := [root.right]
```

$\{\mathtt{right} = \mathtt{root.right} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{root.right})\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{left}) * \mathtt{tree}(\mathtt{right})\}$

```
dispose_tree(left)
```

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathit{emp} * \mathtt{tree}(\mathtt{right})\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree}(\mathtt{right})\}$

```
dispose_tree(right)
```

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathit{emp}\}$

```
dispose root
```

Local proof using `dispose_tree`'s specification:

$$\{\mathtt{tree}(\mathtt{right})\}\ \mathtt{dispose\_tree(right)}\ \{\mathit{emp}\}$$

Combination using the frame rule.

Proof of tree disposal: recursive case

$\{\texttt{tree(root)} \land \texttt{root} \neq \texttt{nil}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(root.left)} * \texttt{tree(root.right)}\}$
```
left := [root.left]
```
$\{\texttt{left} = \texttt{root.left} \land \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(root.left)} * \texttt{tree(root.right)}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(left)} * \texttt{tree(root.right)}\}$
```
right := [root.right]
```
$\{\texttt{right} = \texttt{root.right} \land \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(left)} * \texttt{tree(root.right)}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(left)} * \texttt{tree(right)}\}$
```
dispose_tree(left)
```
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \mathit{emp} * \texttt{tree(right)}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(right)}\}$
```
dispose_tree(right)
```
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \mathit{emp}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right}\}$
```
dispose root
```

Identity of $*$ *emp*:

$$P \text{ iff } \mathit{emp} * P$$

Proof of tree disposal: recursive case

$\{\texttt{tree(root)} \wedge \texttt{root} \neq \texttt{nil}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(root.left)} * \texttt{tree(root.right)}\}$
```
left := [root.left]
```
$\{\texttt{left} = \texttt{root.left} \wedge \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(root.left)} * \texttt{tree(root.right)}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(left)} * \texttt{tree(root.right)}\}$
```
right := [root.right]
```
$\{\texttt{right} = \texttt{root.right} \wedge \texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(left)} * \texttt{tree(root.right)}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(left)} * \texttt{tree(right)}\}$
```
dispose_tree(left)
```
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \mathit{emp} * \texttt{tree(right)}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \texttt{tree(right)}\}$
```
dispose_tree(right)
```
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right} * \mathit{emp}\}$
$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right}\}$
```
dispose root
```
$\{\mathit{emp}\}$

Local proof using deallocation axiom (both `root.left` and `root.right`):

$$\{\texttt{root} \mapsto \texttt{root.left}, \texttt{root.right}\}\ \textbf{dispose}\ \texttt{root}\ \{\mathit{emp}\}$$

Proof of tree disposal: recursive case

$\{\mathtt{tree(root)} \wedge \mathtt{root} \neq \mathtt{nil}\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree(root.left)} * \mathtt{tree(root.right)}\}$

```
left := [root.left]
```

$\{\mathtt{left} = \mathtt{root.left} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree(root.left)} * \mathtt{tree(root.right)}\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree(left)} * \mathtt{tree(root.right)}\}$

```
right := [root.right]
```

$\{\mathtt{right} = \mathtt{root.right} \wedge \mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree(left)} * \mathtt{tree(root.right)}\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree(left)} * \mathtt{tree(right)}\}$

```
dispose_tree(left)
```

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathit{emp} * \mathtt{tree(right)}\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathtt{tree(right)}\}$

```
dispose_tree(right)
```

$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right} * \mathit{emp}\}$
$\{\mathtt{root} \mapsto \mathtt{root.left}, \mathtt{root.right}\}$

```
dispose root
```

$\{\mathit{emp}\}$

Separation logic

Predicate transformers

Predicate transformers for separation logic

Weakest precondition and strongest postcondition predicate transformers can be derived from the small axioms of separation logic, with additional complications to accommodate applications of the frame rule in the predicates being transformed.

- The original presentation of separation logic, which includes weakest-precondition rules, is in Reynold: Separation Logic: A Logic for Shared Mutable Data Structures, LICS, 2002
- A strongest postcondition calculus for separation logic is in Sims: Extending separation logic with fixpoints and postponed substitution, TCS, 2006

When is the strongest postcondition defined?

Remember that heap-manipulating programs fault if command tries to access an unallocated address. Therefore, the strongest postcondition $\mathbf{sp}C, P$ of some commands C is only defined if C is always executable in states that satisfy P.

For example, the strongest postcondition of deallocation is not defined when deallocating an address that is not allocated:

$$\{\mathtt{x} = \mathtt{nil} \wedge \mathit{emp}\}\ \mathbf{dispose}\ \mathtt{x}\ \{?\}$$

error because x is not allocated

Consistently with the fault-avoiding interpretation of separation logic, we assume that strongest postconditions are only calculated in states where the command is executable without faults.

Formally, command C is executable without faults in a state that satisfies P if $P \Longrightarrow \mathbf{wp}(C,)$ is valid.

Predicate transformers: deallocation

The small axiom for deallocation requires that the deallocated address be allocated.

$$\{E \mapsto -\} \quad \texttt{dispose}\ E \quad \{\mathit{emp}\}$$

Thus, for example, the strongest postcondition is not defined if E is not allocated:

$$\{\mathtt{x} = \mathtt{nil} \land \mathit{emp}\}\ \texttt{dispose}\ \mathtt{x}\ \{\mathbf{error}\}$$

For such cases we use the separating implication to encode the consistency requirements on the transformed predicate.

Predicate transformers: deallocation

The small axiom for deallocation requires that the deallocated address be allocated.

$$\{E \mapsto -\} \quad \texttt{dispose}\ E \quad \{emp\}$$

Thus, for example, the strongest postcondition is not defined if E is not allocated:

$$\{\mathtt{x} = \mathtt{nil} \wedge emp\}\ \texttt{dispose}\ \mathtt{x}\ \{\mathbf{error}\}$$

For such cases we use the separating implication to encode the consistency requirements on the transformed predicate.

Q must not predicate about location E

$$\mathbf{wp}(\texttt{dispose}\ E, Q) = E \mapsto - * Q$$

$$\mathbf{sp}(\texttt{dispose}\ E, P) = E \mapsto - \mathrel{-\!\!*} P$$

whenever $E \mapsto -$ holds separately from P

Predicate transformers: allocation

The small axiom for allocation ensures that the allocated address is fresh.

E must not mention v

$$\{emp\} \quad \mathtt{v} := \textbf{new}\ E \quad \{\mathtt{v} \mapsto E\}$$

Q holds regardless of the fresh location's address

$$\textbf{wp}(\mathtt{v} := \textbf{new}\ E, Q) = \forall v' \big((v' \mapsto E) \mathrel{-\!\!*} Q[\mathtt{v} \mapsto v']\big)$$

$$\textbf{sp}(\mathtt{v} := \textbf{new}\ E, P) = \exists \bar{v} \big(\mathtt{v} \mapsto E * P[\mathtt{v} \mapsto \bar{v}]\big)$$

`old(v)`

v is fresh memory

Predicate transformers: writing to heap

Writing to heap requires that the written-to location is allocated.

$$\{E \mapsto -\} \quad [E] := F \quad \{E \mapsto F\}$$

E is allocated, and *Q* holds
in every post-state *E* points to *F*

$$\mathbf{wp}([E] := F, Q) = (E \mapsto -) * ((E \mapsto F) \mathrel{-\!\!*} Q)$$

$$\mathbf{sp}([E] := F, P) = E \mapsto F * \big((E \mapsto -) \mathrel{-\!\!*} P\big)$$

E points to *F*, and *P* holds
in every pre-state where *E* is allocated

Predicate transformers: reading from heap

The small axiom for heap reading works forward:

$$\{\mathsf{v} = \bar{v} \wedge E \mapsto e\} \quad \mathtt{v\ :=\ [}E\mathtt{]} \quad \{v = e \ \wedge \ E[\mathsf{v} \mapsto \bar{v}] \mapsto e\}$$

but we can formulate one that works backward:

$$\{E \mapsto e \wedge Q[\mathsf{v} \mapsto e]\} \quad \mathtt{v\ :=\ [}E\mathtt{]} \quad \{Q\}$$

$$\mathbf{wp}(\mathtt{v\ :=\ [}E\mathtt{]}, Q) = \exists e\big((E \mapsto e * \top) \wedge Q[\mathsf{v} \mapsto e]\big)$$

$$\mathbf{sp}(\mathtt{v\ :=\ [}E\mathtt{]}, P) = \exists \bar{v}\big(P[\mathsf{v} \mapsto \bar{v}] \wedge \mathsf{v} = E[v \mapsto \bar{v}]\big)$$

Separation logic

Decidability & complexity

Relative completeness of separation logic

Hongseok Yang

In his 2001 PhD thesis, Yang proved that the inference rules given by the small axioms, the frame rule, and the Hoare logic axioms that remain valid with references are a relatively complete deductive system for separation logic – relative to an oracle for implication.

Decidability of separation logic

Separation logic is quite expressive, as it embeds a notion of heap and pointers within the heap. As a result, only small fragments of it are decidable.

propositional separation logic with $*$, $\mathrel{-\!\!*}$, but no $\mapsto$ is undecidable

first-order separation logic with $\mapsto$, but no $*$ or $\mathrel{-\!\!*}$ is undecidable

quantifier-free separation logic with $\mapsto$, but no $*$ or $\mathrel{-\!\!*}$ (and no data constraints) is PSPACE-complete

Separation logic tools

Practical tools based on separation logic follow different strategies:

interactive provers require users to provide steps in a proof, such as which inference rules to apply, and can thus work with arbitrarily complex specifications and programs.
VeriFast is an example of interactive tool.

lightweight specification checkers target small fragments of separation logic and somewhat limited programming languages.
Smallfoot is an example of automated tool for lightweight specifications.

push-button analyzers check implicit properties, such as absence of null-pointer dereferences, and may produce false positives but work on realistic programming languages.
Infer is an example of fully automated static analyzer.

Summary

Deductive verification: techniques

Deductive verification is a family of techniques for program analysis based on formal proofs of correctness.

Deductive verification techniques are normally based on reducing correctness to validity of a logic formula.

soundness/completeness: sound and relatively complete

complexity: often undecidable or highly complex

automation: from interactive to auto-active

expressiveness: supports arbitrarily complex properties – traded off against automation

Deductive verification: tools and practice

Dafny is a representative tool of the state of the art in auto-active verification – achieving a fairly high degree of automation by leveraging SMT solvers.

Case studies of deductive verification include complex algorithms, data-structure libraries with full specifications, and fully-verified systems developed incrementally by refinement.

Main outstanding challenges:

- reducing required user expertise and effort (annotations)
- modeling realistic programming-language features
- scalability

Credits and further references

Some examples and parts of the presentation of predicate transformers for Hoare logic are based on Mike Gordon's slides on Hoare logic.

The presentation of separation logic follows the tutorial given by Chris Poskitt as part of the Software Verification course given at ETH Zurich in 2013–2015. In turn, Poskitt's material was based on van Staden's for the same course in 2009–2012, which in turn reused material by Calcagno, Parkinson, and O'Hearn.

Peter O'Hearn's tutorial A primer on separation logic, given at Marktoberdorf 2011, includes a detailed introduction to separation logic and automated reasoning with it.

These slides' license