Lecture 3: Converting Imperative Programs to Formulas

Viktor Kuncak

Verification-Condition Generation for Imperative Non-Deterministic Programs

Program can be represented by a formula relating initial and final state. Consider program with variables x, y, z

$$\begin{array}{ll} \text{program:} & x = x + 2; y = x + 10 \\ \text{relation:} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \text{formula:} & x' = x + 2 \wedge y' = x + 12 \wedge z' = z \end{array}$$

Specification: $z = old(z) \wedge (old(x) > 0 \rightarrow (x > 0 \wedge y > 0))$
Adhering to specification is relation subset:

$$\begin{array}{ll} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \subseteq & \{(x, y, z, x', y', z') \mid z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0))\} \end{array}$$

or validity of the following implication:

$$\begin{array}{ll} & x' = x + 2 \wedge y' = x + 12 \wedge z' = z \\ \rightarrow & z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0)) \end{array}$$

Imperative Presburger Arithmetic Programs

F - formulas, t - terms - as in functional programs so far

Fixed number of mutable integer variables $V = \{x_1, \ldots, x_n\}$

Imperative statements:

- $\mathbf{x = t}$: change $x \in V$ to have value given by t; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if(F) c_1\ else\ c_2}$: if F holds, execute c_1 else execute c_2
- $\mathbf{c_1; c_2}$: first execute c_1, then execute c_2

Statements for introducing and restricting non-determinism:

- $\mathbf{havoc(x)}$: non-deterministically change $x \in V$ to have an arbitrary value; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if}(*)\ \mathbf{c_1\ else\ c_2}$: arbitrarily choose to run c_1 or c_2
- $\mathbf{assume(F)}$: block all executions where F does not hold

Given such loop-free program c with conditionals, compute a polynomial-sized formula $R(c)$ of form: $\exists \bar{z}.F(\bar{x}, \bar{z}, \bar{x}')$ describing relation between initial values of variables $x_1, \ldots, x_n$ and final values of variables $x'_1, \ldots, x'_n$

Construction Formula that Describe Relations

c - imperative command

$R(c)$ - formula describing relation between initial and final states of execution of c

If $\rho(c)$ describes the relation, then $R(c)$ is formula such that

$$\rho(c) = \{(\bar{v}, \bar{v}') \mid R(c)\}$$

$R(c)$ is a formula between unprimed variables $\bar{v}$ and primed variables $\bar{v}'$

Formula for Assignment

$$x = t$$

Formula for Assignment

$$x = t$$

$R(x = t)$:

$$x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Note that the formula must explicitly state which variables remain the same (here: all except x). Otherwise, those variables would not be constrained by the relation, so they could take arbitrary value in the state after the command.

Formula for if-else

After flattening,

$$\mathit{if}\,(b)\ c_1\ \mathit{else}\ c_2$$

Formula for if-else

After flattening,

$$\mathit{if}\,(b)\ c_1\ \mathit{else}\ c_2$$

$R(\mathit{if}\,(b)\ c_1\ \mathit{else}\ c_2)$:

$$(b \wedge R(c_1)) \vee (\neg b \wedge R(c_2))$$

Command semicolon

$$c_1 ; c_2$$

Command semicolon

$$c_1; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \wedge (b, c) \in r_2\}$$

Command semicolon

$$c_1; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \wedge (b, c) \in r_2\}$$

What are $R(c_1)$ and $R(c_2)$ and in terms of which variables they are expressed?

Command semicolon

$$c_1; c_2$$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b.(a, b) \in r_1 \wedge (b, c) \in r_2\}$$

What are $R(c_1)$ and $R(c_2)$ and in terms of which variables they are expressed?

$R(c_1; c_2) \equiv$

$$\exists \bar{z}. \quad R(c_1)[\bar{x}' := \bar{z}] \wedge R(c_2)[\bar{x} := \bar{z}]$$

where $\bar{z}$ are freshly picked names of intermediate states.

- a useful convention: $\bar{z}$ refer to position in program source code

havoc

Definition of HAVOC

1. wide and general destruction: devastation
2. great confusion and disorder

Example of use:

y = 12; havoc(x); assume(x + x = y)

Translation, $R(havoc(x))$:

havoc

Definition of HAVOC

1. wide and general destruction: devastation
2. great confusion and disorder

Example of use:

y = 12; havoc(x); assume(x + x = y)

Translation, $R(havoc(x))$:

$$\bigwedge_{v \in V \setminus \{x\}} v' = v$$

This again illustrates "politically correct" approach to describing the destruction of values of variables: just do not mention them.

Non-deterministic choice

$$\mathit{if}\,(*)\ c_1\ \mathit{else}\ c_2$$

Non-deterministic choice

$$if\,(*)\ c_1\ else\ c_2$$

$R(if\,(*)\ c_1\ else\ c_2)$:

$$R(c_1) \vee R(c_2)$$

- translation is simply a disjunction – this is why construct is interesting
- corresponds to branching in control-flow graphs

assume

$$assume(F)$$

assume

$$assume(F)$$

$R(assume(F))$:

$$F \wedge \bigwedge_{v \in V} v' = v$$

assume

$$assume(F)$$

$R(assume(F))$:

$$F \wedge \bigwedge_{v \in V} v' = v$$

- This command does not change any state.

assume

$$assume(F)$$

$R(assume(F))$:

$$F \wedge \bigwedge_{v \in V} v' = v$$

- This command does not change any state.
- If F does not hold, it stops with "instantaneous success".

Example of Translation

$$
\begin{array}{l}
0 \\
(\mathit{if}\ (b)\ x = x + 1\ \mathit{else}\ y = x + 2); \\
1 \\
x = x + 5; \\
2 \\
(\mathit{if}\ (*)\ y = y + 1\ \mathit{else}\ x = y) \\
3
\end{array}
$$

becomes

$$
\begin{aligned}
\exists x_1, y_1, x_2, y_2.\ & ((b \wedge \mathbf{x_1} = \mathbf{x} + \mathbf{1} \wedge y_1 = y) \vee (\neg b \wedge x_1 = x \wedge \mathbf{y_1} = \mathbf{x} + \mathbf{2})) \\
& \wedge (\mathbf{x_2} = \mathbf{x_1} + \mathbf{5} \wedge y_2 = y_1) \\
& \wedge ((x' = x_2 \wedge \mathbf{y'} = \mathbf{y_2} + \mathbf{1}) \vee (\mathbf{x'} = \mathbf{y_2} \wedge y' = y_2))
\end{aligned}
$$

Think of execution trace $(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)$ where

- (x_0, y_0) is denoted by (x, y)
- (x_3, y_3) is denoted by (x', y')

Imperative Presburger Arithmetic Programs

F - formulas, t - terms - as in functional programs so far

Fixed number of mutable integer variables $V = \{x_1, \ldots, x_n\}$

Imperative statements:

- $\mathbf{x = t}$: change $x \in V$ to have value given by t; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if(F) c_1\ else\ c_2}$: if F holds, execute c_1 else execute c_2
- $\mathbf{c_1; c_2}$: first execute c_1, then execute c_2

Statements for introducing and restricting non-determinism:

- $\mathbf{havoc(x)}$: non-deterministically change $x \in V$ to have an arbitrary value; leave vars in $V \setminus \{x\}$ unchanged
- $\mathbf{if}(*)\ \mathbf{c_1\ else\ c_2}$: arbitrarily choose to run c_1 or c_2
- $\mathbf{assume(F)}$: block all executions where F does not hold

Given such loop-free program c with conditionals, compute a polynomial-sized formula $R(c)$ of form: $\exists \bar{z}.F(\bar{x}, \bar{z}, \bar{x}')$ describing relation between initial values of variables $x_1, \ldots, x_n$ and final values of variables $x'_1, \ldots, x'_n$

Justifying the name for assume(F)

Compute and simplify as much as possible each of the following expressions:

1. $R(assume(F); c)$

Justifying the name for assume(F)

Compute and simplify as much as possible each of the following expressions:

1. $R(\mathit{assume}(F); c) \quad = \quad F \wedge R(c)$
2. $R(c; \mathit{assume}(F))$

Justifying the name for assume(F)

Compute and simplify as much as possible each of the following expressions:

1. $R(\mathit{assume}(F); c) \quad = \quad F \wedge R(c)$
2. $R(c; \mathit{assume}(F)) \quad = \quad R(c) \wedge F[\bar{x} := \bar{x}']$
   where $F[\bar{x} := \bar{x}']$ denotes F with all variables replaced with primed versions

Expressing **if** through non-deterministic choice and assume

Expressing **if** through non-deterministic choice and assume

if (b) c1 **else** c2

$\Big\|\Big|$

if $(*)$ {
  assume(b);
  c1
} **else** {
  assume(!b);
  c2
}

Indeed, apply translation to both sides and observe that generated formulas are equivalent.

Expressing assignment through havoc and assume

Expressing assignment through havoc and assume

x = e

|||

havoc(x);
assume(x == e)

Under what conditions this holds?

Expressing assignment through havoc and assume

x = e

|||

havoc(x);
assume(x == e)

Under what conditions this holds?
$x \notin FV(e)$

Illustration of the problem: $havoc(x);\ assume(x == x + 1)$

Expressing assignment through havoc and assume

x = e

|||

havoc(x);
assume(x == e)

Under what conditions this holds?
$x \notin FV(e)$

Illustration of the problem: $havoc(x);\ assume(x == x + 1)$

Luckily, we can rewrite it into $x_{fresh} = x + 1; x = x_{fresh}$

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope.
Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) =$

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope.
Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) =$
$y' = x + 3 \wedge z' = 2x + 3 + z \wedge x' = x$

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope.
Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) =$
$y' = x + 3 \wedge z' = 2x + 3 + z \wedge x' = x$

$R(\{\textbf{var } y; y = x + 3; z = x + y + z\}) =$

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope.
Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) =$
$y' = x + 3 \wedge z' = 2x + 3 + z \wedge x' = x$

$R(\{\textbf{var } y; y = x + 3; z = x + y + z\}) = \; z' = 2x + 3 + z \wedge x' = x$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V
For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define
$R_V(\{var\ y; P\}) \;=$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V
For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define
$R_V(\{var\ y; P\}) \;=\; \exists y, y'.R_{V \cup \{y\}}(P)$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V
For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define
$R_V(\{var\ y; P\}) \;=\; \exists y, y'.R_{V \cup \{y\}}(P)$

Exercise: express havoc(x) using var.

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V
For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define
$R_V(\{var\ y; P\}) \;=\; \exists y, y'.R_{V \cup \{y\}}(P)$

Exercise: express havoc(x) using var.

$$R_V(havoc(x)) \iff R_V(\{var\ y;\ x = y\})$$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V
For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define
$R_V(\{var\ y; P\}) \;=\; \exists y, y'.R_{V \cup \{y\}}(P)$

Exercise: express havoc(x) using var.

$$R_V(havoc(x)) \iff R_V(\{var\ y;\ x = y\})$$

Exercise: give transformation that lifts all variables to be global

Expressing Specifications as Commands

Shorthand: Havoc Multiple Variables at Once

Variables $V = \{x_1, \ldots, x_n\}$
Translation of $R(havoc(y_1, \ldots, y_m))$:

Shorthand: Havoc Multiple Variables at Once

Variables $V = \{x_1, \ldots, x_n\}$
Translation of $R(havoc(y_1, \ldots, y_m))$:

$$\bigwedge_{v \in V \setminus \{y_1, \ldots, y_m\}} v' = v$$

Exercise: the resulting formula is the same as for:

$$havoc(y_1); \ldots; havoc(y_m)$$

Thus, the order of distinct havoc-s does not matter.

Programs and Specs are Relations

$$\begin{array}{ll} \text{program:} & x = x + 2; y = x + 10 \\ \text{relation:} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \text{formula:} & x' = x + 2 \wedge y' = x + 12 \wedge z' = z \end{array}$$

Specification:

$$z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0)$$

Adhering to specification is relation subset:

$$\begin{array}{ll} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \subseteq & \{(x, y, z, x', y', z') \mid z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0))\} \end{array}$$

Non-deterministic programs are a way of writing specifications

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Use local variables to store initial values.

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Use local variables to store initial values.

```
{ var x0; var y0;
  x0 = x; y0 = y;
  havoc(x,y);
  assume(x > x0 && y > y0)
}
```

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \ldots, x_n\}$
Specification

$$F(x_1, \ldots, x_n, x_1', \ldots, x_n')$$

Becomes

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \ldots, x_n\}$
Specification

$$F(x_1, \ldots, x_n, x_1', \ldots, x_n')$$

Becomes

$$\begin{array}{l} \{\ \mathit{var}\ y_1, \ldots, y_n; \\ \quad y_1 = x_1; \ldots; y_n = x_n; \\ \quad \mathit{havoc}(x_1, \ldots, x_n); \\ \quad \mathit{assume}(F(y_1, \ldots, y_n, x_1, \ldots, x_n))\ \} \end{array}$$

Program Refinement and Equivalence

For two programs, define **refinement** $P_1 \sqsubseteq P_2$ iff

$$R(P_1) \rightarrow R(P_2)$$

is a valid formula.
(Some books use the opposite meaning of $\sqsubseteq$.)
As usual, $P_2 \sqsupseteq P_1$ iff $P_1 \sqsubseteq P_2$.

- $P_1 \sqsubseteq P_2$ iff $\rho(P_1) \subseteq \rho(P_2)$

Define **equivalence** $P_1 \equiv P_2$ iff $P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$

- $P_1 \equiv P_2$ iff $\rho(P_1) = \rho(P_2)$

Example for $V = \{x, y\}$

$$\{\mathit{var}\ x0; x0 = x; \mathit{havoc}(x); \mathit{assume}(x > x0)\} \sqsupseteq (x = x + 1)$$

Proof: Use R to compute formulas for both sides and simplify.

Program Refinement and Equivalence

For two programs, define **refinement** $P_1 \sqsubseteq P_2$ iff

$$R(P_1) \rightarrow R(P_2)$$

is a valid formula.
(Some books use the opposite meaning of $\sqsubseteq$.)
As usual, $P_2 \sqsupseteq P_1$ iff $P_1 \sqsubseteq P_2$.

- $P_1 \sqsubseteq P_2$ iff $\rho(P_1) \subseteq \rho(P_2)$

Define **equivalence** $P_1 \equiv P_2$ iff $P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$

- $P_1 \equiv P_2$ iff $\rho(P_1) = \rho(P_2)$

Example for $V = \{x, y\}$

$$\{var\ x0; x0 = x; havoc(x); assume(x > x0)\} \sqsupseteq (x = x + 1)$$

Proof: Use R to compute formulas for both sides and simplify.

$$x' = x + 1 \wedge y' = y \ \rightarrow \ x' > x \wedge y' = y$$

Stepwise Refinement Methodology

Start form a possibly non-deterministic specification P_0
Refine the program until it becomes deterministic and efficiently executable.

$$P_0 \sqsupseteq P_1 \sqsupseteq \ldots \sqsupseteq P_n$$

Example:

$$\begin{aligned}
& \mathit{havoc}(x); \mathit{assume}(x > 0); \mathit{havoc}(y); \mathit{assume}(x < y) \\
\sqsupseteq\ & \mathit{havoc}(x); \mathit{assume}(x > 0); y = x + 1 \\
\sqsupseteq\ & x = 42; y = x + 1 \\
\sqsupseteq\ & x = 42; y = 43
\end{aligned}$$

In the last step program equivalence holds as well

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Theorem: if $P_1 \sqsubseteq P_2$ then $(P; P_1) \sqsubseteq (P; P_2)$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Theorem: if $P_1 \sqsubseteq P_2$ then $(P; P_1) \sqsubseteq (P; P_2)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p \circ p_1) \subseteq (p \circ p_2)$

Theorem: if $P_1 \sqsubseteq P_2$ and $Q_1 \sqsubseteq Q_2$ then

$$(\mathit{if}\ (*) P_1\ \mathit{else}\ Q_1) \sqsubseteq (\mathit{if}\ (*) P_2\ \mathit{else}\ Q_2)$$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Theorem: if $P_1 \sqsubseteq P_2$ then $(P; P_1) \sqsubseteq (P; P_2)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p \circ p_1) \subseteq (p \circ p_2)$

Theorem: if $P_1 \sqsubseteq P_2$ and $Q_1 \sqsubseteq Q_2$ then

$$(\mathit{if}\ (*) P_1\ \mathit{else}\ Q_1) \sqsubseteq (\mathit{if}\ (*) P_2\ \mathit{else}\ Q_2)$$

Version for relations:
$(p_1 \subseteq p_2) \wedge (q_1 \subseteq q_2) \quad \rightarrow \quad (p_1 \cup q_1) \subseteq (p_2 \cup q_2)$