- Welcome back to Software Testing and Verification!

TODO

Gabriele Keller
g.k.keller@uu.nl

Predicate Logic

- Formulas
- Quantification
- Inference rules
- Proofs
- Proofs involving quantification
- Some basic proof techniques:
  - Contradiction
  - Equational
  - Case split
  - Induction

Equational Reasoning

- The main claim to prove was

  ∀s :: `rev s = rv [ ] s`

- Along the way, we proved a stronger lemma:

  (∀s :: (∀t :: `rev s ++ t = rv t s`))

- This lemma **directly** implies the original claim!
  - without having to prove the latter through induction, namely by instantiating `t` to `[ ]`.

Part I (2.5pt)

1. **Refreshing Predicate Logic** [1pt]. Give a formal proof of the formula in the Lecture Notes Section 3.9 no. 7. Use the proof style as in the Lecture Notes.

   Tip: prove the formula via contradiction.

2. **Induction and Equational Proof** [0.8pt] Consider the following two functions to calculate the sum of the elements in a list/sequence. In the notation below, [ ] denotes an empty list, and x:s denotes a list with x as its first element and s as the remaining of the list.

   From programming perspective, these functions can be seen as *programs* that get a list as input and return an integer as output. In fact, you can write them as programs in a functional programming language like Haskell. In programming, the above mentioned [ ] and : act as list constructors.

   $$\begin{array}{lcl} \texttt{sum1}\ [\,] & = & 0 \\ \texttt{sum1}\ (\texttt{x} : \texttt{s}) & = & \texttt{x} + \texttt{sum1}\ \texttt{s} \end{array}$$

   $$\begin{array}{lcl} \texttt{sum2}\ \texttt{n}\ [\,] & = & \texttt{n} \\ \texttt{sum2}\ \texttt{n}\ (\texttt{x} : \texttt{s}) & = & \texttt{sum2}\ (\texttt{x} + \texttt{n})\ \texttt{s} \end{array}$$

   Prove that for all lists of integers **s**, we have:

   $$\texttt{sum1 s} \;=\; \texttt{sum2 0 s}$$

   To prove a general property about lists, such as the one above, you typically need to use the following list induction rule (quite analogous to the natural number induction rule that you already know).

   $$\frac{P\ [\,] \quad , \quad (\forall x :: (\forall s :: P\ s \Rightarrow P\ (x{:}s)))}{(\forall s :: P\ s)}$$

   Using induction over lists and equational reasoning, to the above claimed equality between **sum1** and **sum2**.

   As a side note, **sum2** has the overhead of having to maintain one extra parameter, but it can be optimized to a more efficient program because it does not actually need to build a call stack.

Hoare Logic

LN chapter 5, 6, 9.3

Hoare Logic is used to reason about the correctness of programs. In the end, it reduces a program and its specification to a set of verifications conditions.

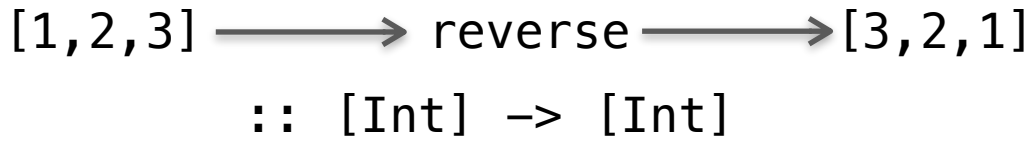
Overview

- Part I
  - Hoare triple
  - Rules for basic statements // SEQ, IF, ASG
  - Weakest pre-condition
  - Example
- Part II : loops
- Handling few more basic constructs
  - array
  - specification at the program level

Part 1 : reasoning about basic statements

Hoare Logic

- We've looked into proofs about ***purely functional programs***:
  - result of a function *only* depends on its arguments
    - a function applied to the same argument will always give the same result
  - enables equational reasoning

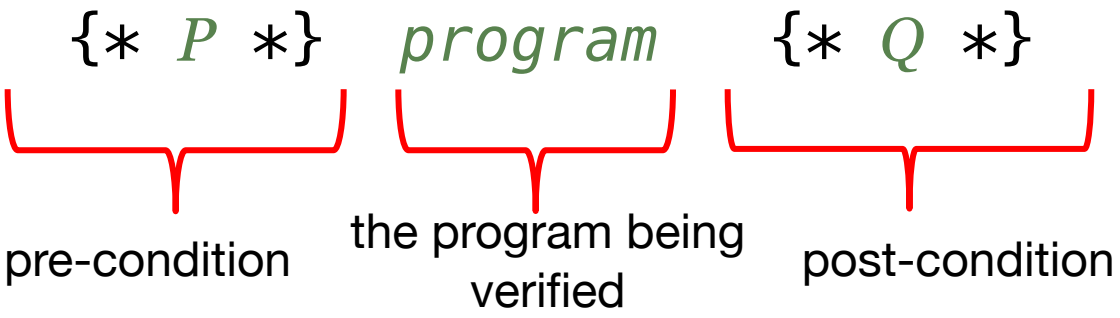


Hoare Logic

- *Proofs about **imperative/object oriented programs:***
  - argue about how the execution of the program affects the state of the
    - memory
    - devices
    - world

Hoare Logic

- Proposed by Tony Hoare, Robert W. Floyd in 1969
  - specifies the ***effect*** a program has on some state
- A ***Hoare triple*** is a simple way to specify the relation between a program's initial state and end state:



properties of the state which have to hold for the program to execute as intended

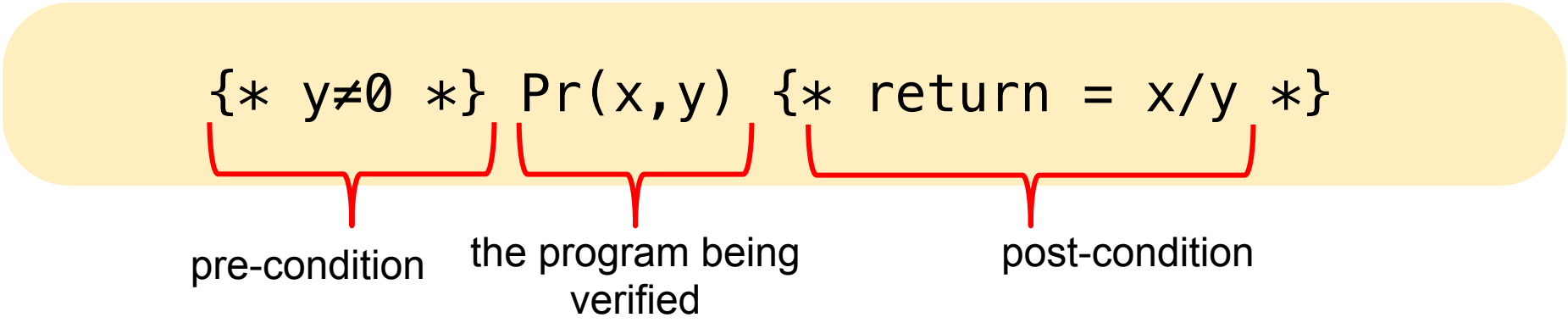
properties we want to hold after program execution

Hoare Logic

- We'll use a very simple imperative language
  - arithmetic expressions
  - skip
  - assignments
  - sequence of statements
  - if-statements
  - while-loops
  - procedure calls
  - (infinite) arrays
- The state we're looking at is just the value of certain variables
  - value of a variable in a stateful language: content of the memory location denoted by that variables, can change during execution of a program
  - value of a variable in a functional language is like a mathematical variable: bound to a value once, does not change

Hoare triple

- Example:



Partial and total correctness interpretation

$$\{* \; P \; *\} \; program \; \{* \; Q \; *\}$$

- ***Total correctness interpretation*** of Hoare triple:
  - if the program is executed on a state satisfying P, it will **terminate** in a state satisfying Q.

- ***Partial correctness interpretation:***
  - if the program is executed on a state satisfying P, and if it terminates, it will terminate in a state satisfying Q.

- Partial correctness is of course weaker, but on the other hand easier to prove. Useful in situations where we can afford to postpone concerns about termination.

Difficult to capture with standard Hoare triples

- **Limitations**: not every type of behavioural properties can easily captured with Hoare triples.
  - Certain actions within a program must occur *in a* ***certain order***. An alternative formalism to express this: CSP (communicating sequential processes)
  - A certain state in the program must be visited ***infinitely*** often. Alternative formalism: temporal logic.
  - A certain goal must be reached despite the ***presence of adversaries*** in the program's environment that may try to fake information. Alternative formalism: logic of belief.

Hoare Logic

Hoare Logic often used to prove the correctness of a program in terms of the validity of Hoare triples.

For each language construct, we need an inference rule to specify how the instruction affects the state

Let's start with `skip`:

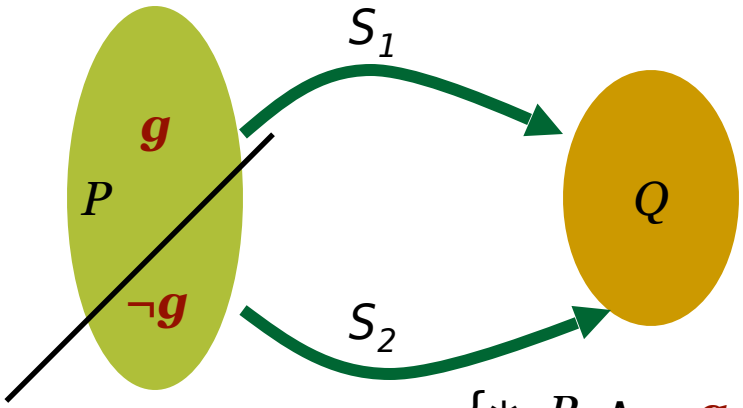
$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

That is: for any precondition P, if P holds, then P still holds after executing the program `skip`

Inference rule for IF

$$\frac{}{\{*\ P\ *\}\quad \texttt{if } g \texttt{ then } S_1 \texttt{ else } S_2\quad \{*\ Q\ *\}}$$

$\{*\ P \wedge g\ *\}\quad S_1\ \{*\ Q\ *\}$



$\{*\ P \wedge \neg g\ *\} S_2 \{*\ Q\ *\}$

Inference rule for assignment, attempt-1

$$\frac{??}{\{*\ P\ *\}\ x := e\ \{*\ Q\ *\}}$$

- **Idea:** find a sufficient pre-condition W, that holds before the assignment, if and only if Q holds after the assignment.
- Then we prove $P \Rightarrow W$

Examples

- {* *W*? *} x:=10 {* x = y *}

  W: 10 = y

- {* *W*? *} x:=x+a {* x = y *}

  W: x+a = y

- {* *W*? *} x:=x+1 {* x > 10 *}

  W: x+1 > 10

- More generally, the weakest precondition W such that
  - {* W *} x := e {* Q *} holds

    can be obtained by replacing/substituting all free occurrences of x in Q by the expression e
- We write the substitution of x in Q by the expression e

  $Q[e/x]$ (note: many different notations used in literature)

Assignment

- Theorem:

  Q holds after x:=e

  iff

  Q[e/x] holds before the assignment.

- Which leads to the following *proof rule* :

$$\{^* P ^*\}\ \mathrm{x{:=}e}\ \{^* Q ^*\} \quad \equiv \quad P \Rightarrow Q[e/x]$$

$$\frac{P \Rightarrow Q[e/x]}{\{*\ P\ *\}\ x\ :=\ e\ \{*\ Q\ *\}}$$

Sequence of statements

- Now that we looked at rules for individual statements, lets look at rules about the combination of statements:

$$\frac{}{\{*\ P\ *\}\ S_1\ ;\ S_2\ \{*\ R\ *\}}$$

- Note: this rule does not tell us how to find Q

Few other basic rules

- **Pre-condition strengthening:**

$$\frac{P \Rightarrow Q \quad , \quad \{* Q *\} \quad S \quad \{* R *\}}{\{* P *\} \quad S \quad \{* R *\}}$$

Few other basic rules

- **Pre-condition strengthening:**

$$\frac{P \Rightarrow Q \quad , \quad \{* Q *\} \quad S \quad \{* R *\}}{\{* P *\} \quad S \quad \{* R *\}}$$

- **Post-condition weakening:**

$$\frac{\{* P *\} \quad S \quad \{* Q *\} \quad , \quad Q \Rightarrow R}{\{* P *\} \quad S \quad \{* R *\}}$$

Few other basic rules

- **Conjunction of Hoare triples:**

$$\frac{\{*\ P_1\ *\}\ \ S\ \ \{*\ Q_1\ *\}\ \ ,\ \{*\ P_2\ *\}\ \ S\ \ \{*\ Q_2\ *\}}{\{*\ P_1 \wedge P_2\ *\}\ \ S\ \ \{*\ Q_1 \wedge Q_2\ *\}}$$

Few other basic rules

- **Conjunction of Hoare triples:**

$$\frac{\{*\ P_1\ *\}\ \ S\ \ \{*\ Q_1\ *\}\ \ ,\ \{*\ P_2\ *\}\ \ S\ \ \{*\ Q_2\ *\}}{\{*\ P_1 \wedge P_2\ *\}\ \ S\ \ \{*\ Q_1 \wedge Q_2\ *\}}$$

- **Disjunction of Hoare triples**

$$\frac{\{*\ P_1\ *\}\ \ S\ \ \{*\ Q_1\ *\}\ \ ,\ \{*\ P_2\ *\}\ \ S\ \ \{*\ Q_2\ *\}}{\{*\ P_1 \vee P_2\ *\}\ \ S\ \ \{*\ Q_1 \vee Q_2\ *\}}$$

How does a proof proceed now ?

- {* x≠y *} `tmp:= x ; x:=y ; y:=tmp` {* x≠y *}

- Rule for **SEQ** requires us to come up with intermediate assertions:

{* x≠y *} `tmp:= x` {* ? *} `x:=y` {* ? *} `y:=tmp` {* x≠y *}

- If we manage to come up with the intermediate assertions, Hoare logic's rules tell us how to prove the claim. However, the rules **do not** tell us how to come up with these intermediate assertions in the first place,

$$\frac{\{*\ P\ *\}S_1\{*\ Q\ *\},\ \{*\ Q\ *\}S_2\{*\ R\ *\}}{\{*\ P\ *\}\ S_1\ ;\ S_2\ \{*\ R\ *\}}$$

Weakest pre-condition

- We can *characterise* ***wp***, as follows (Def. 6.2.3):

$$\{^* \; P \; ^*\} \;\; S \;\; \{^* \; Q \;\; ^*\} \qquad \equiv \qquad P \Rightarrow \boldsymbol{wp} \; S \; Q$$

- ***wp*** always produces a valid pre-cond. You can prove:

$$\{^* \;\; \boldsymbol{wp} \; S \; Q \; ^*\} \quad S \quad \{^* \; Q \; ^*\}$$

- ***wp*** reduces program verification problems to proving implications, for which we already have a tool for (Predicate Logic).

Weakest pre-condition

- Again, the characterisation of ***wp***:

$$\{^* P ^*\}\ S\ \{^* Q\ ^*\} \quad \equiv \quad P \Rightarrow \mathit{wp}\ S\ Q$$

- The reduction is complete. That is, if the implication above is not valid, neither is the specification.

  Note: a counter example demonstrating the invalidity of the implication essentially describes an input for the program/ statement S, that leads to a final state that is not in Q. Such an input is useful for debugging S.

But...

- But this characterisation is **not constructive**:

$$\{* P *\}\ S\ \{* Q\ *\} \quad \equiv \quad P \Rightarrow \boldsymbol{wp}\, S\, Q$$

- That is, it does not tell us how to actually calculate this weakest pre-condition that ***wp*** is supposed to produce.
- To be actually usable, we need to come up with a constructive definition for ***wp***.

Some notes about the weakest precondition

- This is the meta property we want to have:

$$P \Rightarrow \boldsymbol{wp}\, S\, Q \quad \equiv \quad \{*\, P\, *\}\ S\ \{*\, Q\ *\}$$

- A definition/implementation of ***wp*** should be at least **sound**; it should produce a safe pre-condition:

$$P \Rightarrow \boldsymbol{wp}\, S\, Q \quad \Rightarrow \quad \{*\, P\, *\}\ S\ \{*\, Q\ *\}$$

- A ***wp*** definition that satisfies the reverse property is called complete. I formulate it in contraposition to emphasize that it means, if the implication is invalid, the Hoare triple is also not valid:

$$P \not\Rightarrow \boldsymbol{wp}\, S\, Q \quad \Rightarrow \quad \cancel{\{*\, P\, *\}\ S\ \{*\, Q\ *\}}$$

Weakest pre-condition of skip and assignment

$$wp\ \texttt{skip}\ Q \quad = \quad Q$$

$$wp\ (x\texttt{:=}e)\ Q \quad = \quad Q[e/x]$$

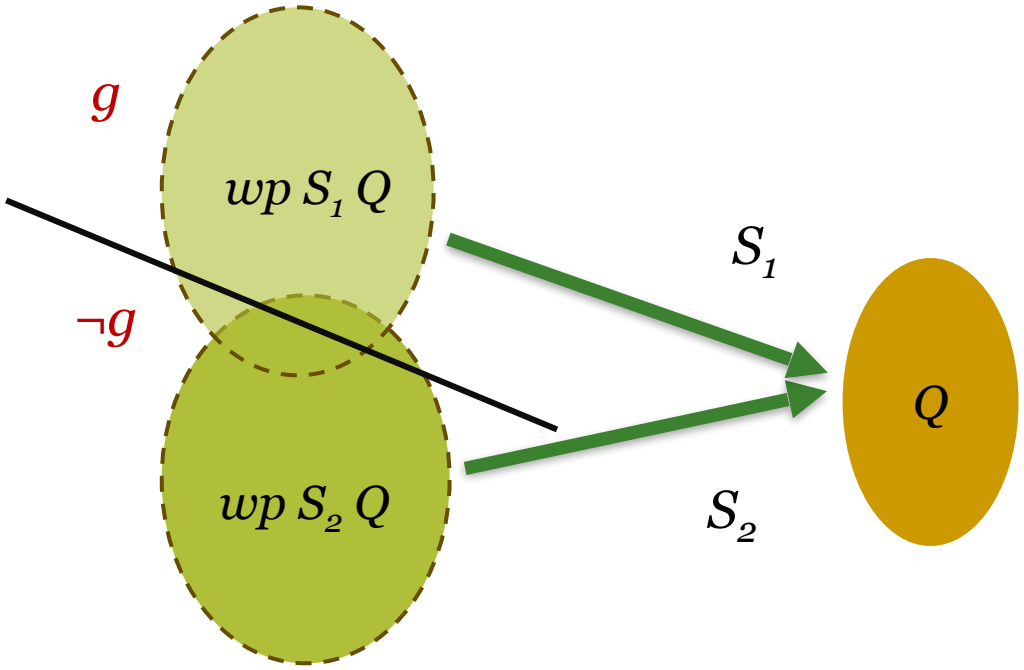
wp of SEQ

$$wp\ (S_1 ; S_2)\ Q\ =\ wp\ S_1\ (wp\ S_2\ Q)$$

- first, calculate $R\ =\ wp\ S_2\ Q$
- then, calculate $P\ =\ wp\ S_1\ R$

wp of IF

wp (if g then S_1 else S_2) Q =



Alternative formulation of ***wp*** IF

$$\textit{wp}\ (\textbf{if}\ g\ \textbf{then}\ S_1\ \textbf{else}\ S_2)\ Q \;=\; (g \wedge \textit{wp}\ S_1\ Q) \vee (\neg g \wedge \textit{wp}\ S_2\ Q)$$

$$=$$

$$(g \Rightarrow \textit{wp}\ S_1\ Q) \wedge (\neg g \Rightarrow \textit{wp}\ S_2\ Q)$$

This formulation for the ***wp*** of if-then-else is more convenient for working out proofs. E.g. in a deductive proof its conjunctive form would translate to proving two goals.

How does a proof proceed now ?

- {* x≠y *} `tmp:= x; x:=y; y:=tmp` {* x≠y *}

Example (informal)

How can we prove that

$\{* \; *\}$ **if** $(x \geq 0)$ **then** $x := x+1$ **else** $x := -x$ $\{* \; x>0 \; *\}$

holds?

$$\frac{\{* \; P \wedge g \; *\} S_1 \{* \; Q \; *\}, \quad \{* \; P \wedge \neg g \; *\} S_2 \{* \; Q \; *\}}{\{* \; P \; *\} \text{ if } g \text{ then } S_1 \text{ else } S_2 \{* \; Q \; *\}}$$

we need to show that

[G1] $\{* \quad x \geq 0 \; *\}$ $x := x+1$ $\{* \; x>0 \; *\}$

and

[G2] $\{* \; \neg(x \geq 0) \; *\}$ $x := -x$ $\{* \; x>0 \; *\}$

Some notes about the verification approach

- In the exercises, we prove $P \Rightarrow W$ by hand.
- In practice, to some degree this can be automatically checked using e.g. a SAT solver. With a SAT solver we instead check if:

  $\neg(P \Rightarrow W)$ is satisfiable

  If it is, then $P \Rightarrow W$ is **not** valid. Otherwise $P \Rightarrow W$ is valid.

- If the definition of ***wp*** is also complete (in addition to being sound), the witness of the satisfiability of $\neg(P \Rightarrow W)$ is essentially an input that will expose a bug in the program → useful for debugging.

Weakest pre-condition (wp), LN Sec. 6.2

- Imagine we have this function:

  wp : Stmt → Pred → Pred

  such that

  wp S Q

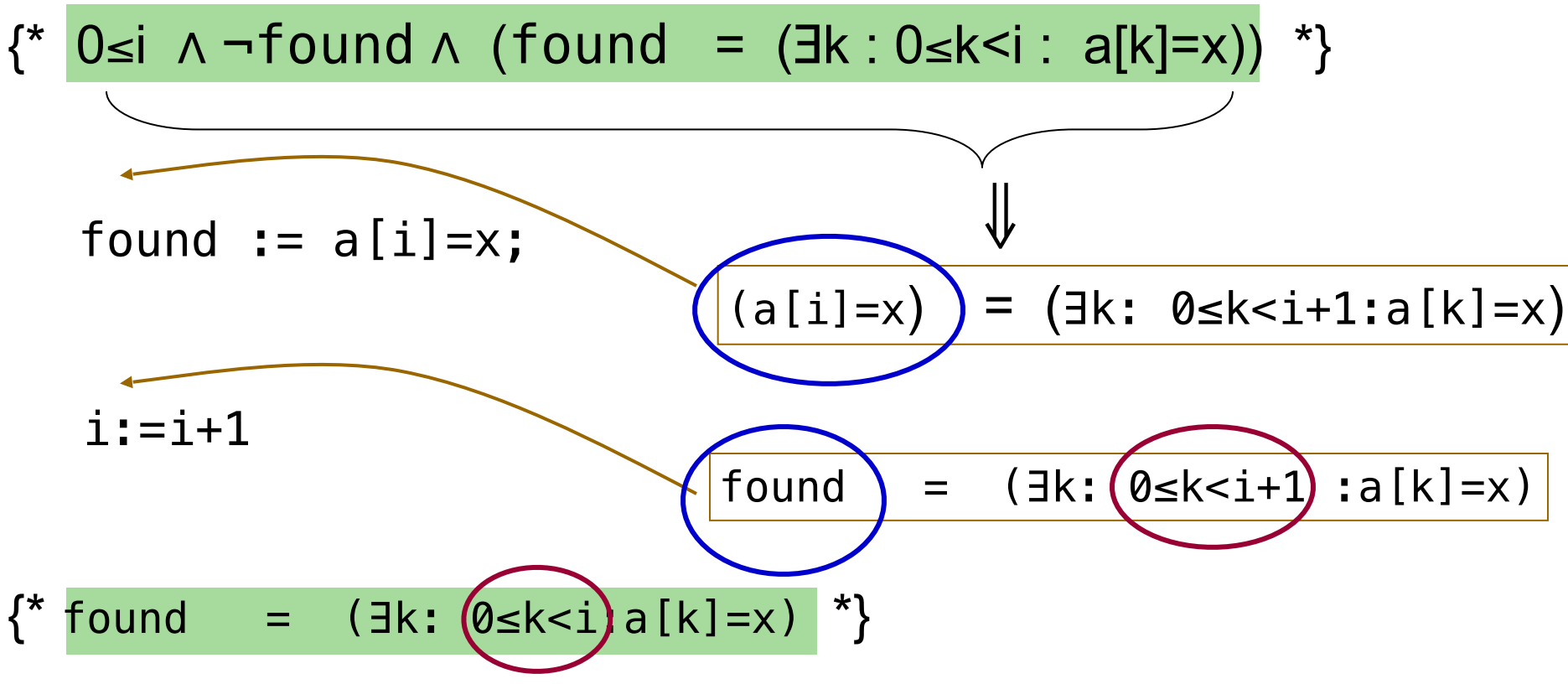
  gives the ***weakest valid*** pre-cond P.

- That is, executing S in any state which satisfies P results in a state that satisfies Q.

Weakest pre-condition (wp), LN Sec. 6.2

- Again, we can have two variations of such a function:
  - Partial correctness based ***wp*** assumes *S* to always terminate.
  - Total correctness ***wp:*** the produced pre-condition guarantees that *S* terminates when executed on that pre-condition.

Example



$wp\ (\texttt{x:=e})\ Q = Q[\texttt{e/x}]$

Questions

(a) $\{* \ \mathbf{wp}\ S\ Q\ *\}\ \ S\ \ \{* \ Q\ *\}$ is always a valid specification.

(b) $\{* \ P \Rightarrow (\mathbf{wp}\ S\ Q)\ *\}\ \ S\ \ \{* \ Q\ *\}$ is always a valid specification.

(c) $\{* \ Q\ *\}\ \ S\ \ \{* \ \mathbf{wp}\ S\ Q\ *\}$ is always a valid specification.

(d) $\{* \ P\ *\}\ \ S\ \ \{* \ Q\ *\}$ is valid if and only if the predicate $\mathbf{wp}\ S\ (P \Rightarrow Q)$ is valid.

1

Questions

The specification $\{*\ Q\ *\}\ S\ \{*\ \mathbf{true}\ *\}$ is known to be valid under **total correctness**. Which of the following conclusions is correct?

(a) S will terminate when executed in any state.

(b) If the implication $P \Rightarrow Q$ is valid, then S will terminate when executed in any state satisfying P.

(c) The specification $\{*\ Q\ *\}\ S\ \{*\ Q \Rightarrow R\ *\}$ is also valid under partial correctness.

(d) The specification $\{*\ P \Rightarrow Q\ *\}\ S\ \{*\ \mathbf{true}\ *\}$ is also valid under total correctness.

Part II : reasoning about loops

LN Section 6.3 -- 6.7, and 9.3

How to prove this ?

$$\{^* P ^*\} \quad \textbf{while } g \textbf{ do } S \quad \{^* Q ^*\}$$

the while loop corresponds to:

```
if g
then
  S; if g
     then
       S; if g
          then S; if g
                       …
          else skip
     else skip
else skip
```

$$\{^* P \wedge \neg \mathrm{g} ^*\} \quad \mathtt{skip} \quad \{^* Q ^*\}$$

$$\{^* P \wedge g ^*\}\, \underbrace{S;S;S;S;S;\ldots;S}\, \{^* Q ^*\}$$

we can't know how many times

How to prove this ?

$$\{^* P ^*\} \quad \texttt{while } g \texttt{ do } S \quad \{^* Q ^*\}$$

Calculating ***wp*** won't work here!

how many times will the loop iterate?

We might know that it will iterate for example *N* times, where *N* is a parameter of the program, but calculating the ***wp*** of SEQ requires us to know concretely how many statements are being composed in the sequence.

Example

```
{* found = false,   i = 0,   0≤N *}

while (not found) && (i < N) do {
   found := a[i] = x;
   i     := i + 1;
}

{* found = ∃k:0≤k<N:  a[k]=x     *}
```

Idea

Come up with a predicate I , so-called "**invariant**" , that holds at the end of every iteration.

iter_1 : // g // ; S {* I *}

iter_2 : // g // ; S {* I *}

…

iter_n : // g // ; S {* I *} // last iteration

exit : // $\neg g$ //

- Observation:
  - $I \wedge \neg g$ holds when the loop terminates
  - The post-condition Q can be established if $I \wedge \neg g$ implies Q.

Ok, what kind of predicate is I ??

- I needs to hold at the end of every iteration.
  - inductively, we can assume it holds at the start of the iteration (established by the previous iteration).

- So, it is sufficient to have an I satisfying this property:

$$\{^* \; I \wedge g \; ^*\} \quad S \quad \{^* \; I \; ^*\}$$

Example

{* x > 0 *} **while** (x > 0) **do** x := x−1 {* x = 0 *}

- We need to find I such that :
  - $I \wedge \neg(x>0)$ holds when the loop terminates
  - $I \wedge \neg(x>0)$ implies that x = 0.
  - {* $I \wedge (x>0)$ *} x := x−1 {* I *}

Establishing the base case

The inductive argument

$$\{^* \boldsymbol{I} \wedge \boldsymbol{g} \, ^*\} \quad \boldsymbol{S} \quad \{^* \boldsymbol{I} \, ^*\}$$

does not apply for the first iteration

We need to guarantee that the first iteration can assume $\boldsymbol{I}$ as its pre-condition.

We can establish this by requiring that the original precondition $\boldsymbol{P}$ **implies this** $\boldsymbol{I}$.

To Summarize

- Capture this in an inference rule (Rule 6.3.2):

$$\frac{\begin{array}{ll} P \Rightarrow \boldsymbol{I} & \text{// setting up } \boldsymbol{I} \\ \{* \; g \wedge \boldsymbol{I} \; *\} \;\; S \;\; \{* \; \boldsymbol{I} \; *\} & \text{// invariance} \\ \boldsymbol{I} \wedge \neg g \Rightarrow Q & \text{// exit cond} \end{array}}{\{* \; P \; *\} \;\; \textbf{while} \;\; g \;\; \textbf{do} \;\; S \;\; \{* \; Q \; *\}}$$

- This rule is only good for **partial correctness** though.

Few things to note

- A special instance the previous rule is this (by taking $\boldsymbol{I}$ itself as P):

$$\frac{\{*\ g\ \wedge \boldsymbol{I}\ *\}\quad S\quad \{*\ \boldsymbol{I}\ *\} \qquad \boldsymbol{I}\ \wedge \neg g\ \Rightarrow\ Q}{\{*\ \boldsymbol{I}\ *\}\quad \textbf{while}\ \ g\ \ \textbf{do}\quad S\quad \{*\ Q\ *\}}$$

- If $\boldsymbol{I}$ satisfies the above two conditions, we can extend an implementation of $\boldsymbol{wp}$ to take this $\boldsymbol{I}$ as the $\boldsymbol{wp}$ of the loop over Q as the post-condition.
- This allows you to chain $\boldsymbol{I}$ into the rest of $\boldsymbol{wp}$ calculation.
- Such a modified $\boldsymbol{wp}$ function would still be sound, though it is no longer complete (in other words, the resulting pre-condition may not be the weakest one).

Examples

$$\frac{\{* \; g \wedge I \; *\} \quad S \quad \{* \; I \; *\} \qquad I \wedge \neg g \Rightarrow Q}{\{* \; I \; *\} \;\; \textbf{while} \;\; g \;\; \textbf{do} \;\; S \;\; \{* \; Q \; *\}}$$

- Prove the correctness of the following loops (partial correctness) :

```
{* true *} while  i≠n  do  i++  {* i=n *}
```

```
{*  i=0 ∧ n=10 *} while i<n do i++ {* i=n *}
```

```
{* i=0 ∧ s=0 ∧ n=10 *} while i<n do{s = s+2; i++} {* s=20 *}
```

Proving termination

- Again, consider:

how can we show that g will eventually be false?

$$\{^* \; P \; ^*\} \quad \texttt{while} \; g \; \texttt{do} \quad S \quad \{^* \; Q \; ^*\}$$

- Idea: come up with an **integer** expression m, called ***termination metric***, satisfying :
  1. At the start of every iteration $m > 0$
  2. Each iteration decreases m

- Since m has a lower bound (condition 1), it follows that the loop cannot iterate forever. In other words, it will terminate.

Capturing the termination conditions

- At the start of every iteration $m > 0$:
  - $g \Rightarrow m > 0$
  - If you have an invariant: $I \wedge g \Rightarrow m > 0$
- Each iteration decreases m :

$$\{^* \; I \wedge g \; ^*\} \quad \mathtt{C} := m; S \quad \{^* \; m < \mathtt{C} \; ^*\}$$

old value of termination metric

new value of termination metric after execution of S

To Summarize

- Total correctness (Rule B.1.8):

$$\frac{\begin{array}{ll} P \Rightarrow I & \text{// setting up } I \\ \{* \, g \wedge I \, *\} \;\; S \;\; \{* \, I \, *\} & \text{// invariance} \\ I \wedge \neg g \Rightarrow Q & \text{// exit cond} \\ \{* \, I \wedge g \, *\} \;\; \texttt{C:=}m\texttt{;}\, S \;\; \{* \, m < \texttt{C} \, *\} & \text{// } m \text{ decreasing} \\ I \wedge g \Rightarrow m > o & \text{// } m \text{ bounded below} \end{array}}{\{* \, P \, *\} \;\; \textbf{while } g \textbf{ do } \; S \;\; \{* \, Q \, *\}}$$

Alternatively it can be formulated as follows

- Rule B.1.7

$$\frac{\begin{array}{l}\{* g \wedge I *\}\ \ S\ \ \{* I *\} \qquad \text{// invariance}\\ I \wedge \neg g \Rightarrow Q \qquad \text{// exit cond}\\ \{* I \wedge g *\}\ \ C{:=}m; S\ \ \{* m{<}C *\} \qquad \text{// m decreasing}\\ I \wedge g \Rightarrow m > o \qquad \text{// m bounded below}\end{array}}{\{* I *\}\ \ \textbf{while}\ g\ \textbf{do}\ S\ \{* Q *\}}$$

- This is a special instance of B.1.8. On the other hand, together with the pre-condition strengthening rule it also implies B.1.8.

Examples

$P \Rightarrow I$

$\{* g \wedge I *\} \quad S \quad \{* I *\}$

$TC1 : \{* I \wedge g *\} \quad C:=m; S \quad \{* m<C *\}$

$TC2: \quad I \wedge g \Rightarrow m > o$

- Prove that the following programs terminate:

```
{* i≤n *} while i≠n do i := i+1   {* true *}
```

Examples

- Prove that the following programs terminate:

```
{* i≤n *} while i≠n do i := i+1   {* true *}
```

```
{* candy = 100 ∧ apple = 100 }
while candy > 0 ∨ apple > 0  do {
   if candy > 0
     then { candy--;
            apple += 2 }
     else { apple-- }
}
{* true *}
```

Termination metrics

- In practice, for most terminating loops, the termination metric is pretty obvious
- In general, finding such a metric can't be automated

```
{* n > 0 *}
 while n > 1  do {
    if n mod 2 = 0
       then { n := n / 2}
       else { n := 3 * n + 1}
 }
{* true *}
```

A bigger example

- The following program claims to check if the array segment `a[0..n)` consists of only `0`'s:

```
{* 0≤n *}
   i := 0 ; r := true ;
   while  i < n  do  {
      r := r ∧ (a[i]=0);
      i := i+1 }
{* r  = (∀k : 0≤k<n : a[k]=0) *}
```

- We need to propose an **invariant** and a **termination metric**.
- This time, the proof will be quite involved. The rule to handle loops also generates multiple conditions.

Invariant and termination metric

```
{* 0≤n *}
i := 0 ; r := true ;
 while  i < n  do  {
    r := r ∧ (a[i] = 0);
    i++ }
{*  r  = (∀k : 0≤k<n : a[k] = 0) *}
```

- Let's go with this choice of invariant I :

$$\underbrace{(r = (\forall k : 0 \le k < i : a[k] = 0))}_{I_1} \quad \wedge \quad \underbrace{0 \le i \le n}_{I_2}$$

- The termination metric m is quite obvious, namely: $m = n - i$

It comes down to proving these

1. Exit Condition:

   $I \wedge \neg g \Rightarrow Q$

2. Initialization Condition:

   {* given-pre-cond *} `i := 0 ; r := true` {* I *},

3. Invariance:

   {* $I \wedge g$ *} body {* I *}

   Or equivalently, prove: $I \wedge g \Rightarrow \boldsymbol{wp}$ body I

4. Termination Condition 1:

   {* $I \wedge g$ *} (C :=m ; body) {* $m <$ C *}

5. Termination Condition 2:

   $I \wedge g \Rightarrow m > 0$

Reformulating them in terms of wp

If the components contain no loops, we can convert the Hoare triples into implications towards ***wp***:

1. Exit Condition $I \wedge \neg g \Rightarrow Q$
2. Initialisation Condition
   given-pre-cond $\Rightarrow$ ***wp*** (`i := 0 ; r := true`) I
3. Invariance $I \wedge g \Rightarrow$ ***wp*** $body\ I$
4. Termination Condition $I \wedge g \Rightarrow$ ***wp*** $(C{:=}m\ ;\ body)\ \ (m<C)$
5. Termination Condition $I \wedge g \Rightarrow\ m>0$

Now we can use the previous proof system for predicate logic to prove those implications.

Proof of init

given precon. $\Rightarrow$ ***wp*** (`i := 0 ; r := true`) I

PROOF PInit
[A1] `0≤n`
[G](`true` = (`∀k:0≤k<0:a[k]=0`))`∧0≤0≤n`
BEGIN
1. { follows from A1 } `0≤0≤n`
2. { see subproof } `true` = (`∀k:0≤k<0: a[k]=0`)

EQUATIONAL PROOF
`(∀k : 0≤k<0 : a[k]=0)`
= { the domain is false }
`(∀k : false : a[k]=0))`
= {∀ over empty domain }
`true`
END

3. { conjunction of 1 and 2 } G
END

calculated ***wp***

```
{* 0 ≤ n *}
i := 0 ; r := true ;
 while  i < n  do  {
    r := r ∧ (a[i] = 0);
    i++ }
{* r = (∀k : 0≤k<n : a[k] = 0) *}
```

$I = (\mathtt{r} = (\forall k : 0 \le k < \mathtt{i} : a[k] = 0)) \wedge \quad 0 \le \mathtt{i} \le n$

Proof of I's invariance

$$I \wedge g \Rightarrow \mathit{wp}\ \mathit{body}\ I$$

PROOF PIC (proof of the invariance condition)

[A1] `r = (∀k:0≤k<i: a[k]=0)`

[A2] `0≤i≤n`

[A3] `i<n`

[G1] `r ∧ (a[i]=0)= (∀k:0≤k<i+1: a[k]=0)`

[G2] `0≤i+1≤n`

I_1

I_2

Loop guard

calculated from wp

BEGIN

1. { see the subproof below } G1

EQUATIONAL PROOF

```
(∀k : 0≤k<i+1 : a[k]=0)
= { dom. merge , PIC.A2 }
(∀k : 0≤k<i ∨ k=i : a[k]=0)
= { ∀ domain-split , justified by 0≤i (PIC.A2) }
(∀k : 0≤k<i: a[k]=0) ∧ (∀k:k=i: a[k]=0)
= { PIC.A1 }
r ∧ (∀k : k=i : a[k]=0)
= { quant. over singleton }
r ∧ (a[i]=0)
```

END

2. { A2 implies 0≤i+1 and A3 implies i+1≤n } G2

END

$I = (\mathtt{r = (\forall k : 0 \le k < i : a[k] = 0)}) \wedge 0 \le i \le n$

```
{* 0 ≤ n *}
i := 0 ; r := true ;
 while  i < n  do  {
    r := r ∧ (a[i] = 0);
    i++ }
{* r = (∀k : 0≤k<n : a[k] = 0) *}
```

Top level structure of the proofs of termination

TC1: $I \wedge g \Rightarrow \boldsymbol{wp}$ (C:= n − i; body) (n − i < C)

PROOF TC1 (proof of the 1st termination condition)

[A1] r = (∀k:0≤k<i:a[k]=0)
[A2] 0≤i≤n
[A3] i < n
[G] n−(i+1) < n − i

....
END

I = (r =(∀k:0≤k<i:a[k]=0)) ∧ 0≤i≤n

TC2 = $I \wedge g \Rightarrow$ n−i > 0

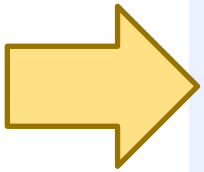
PROOF TC2
[A1] r = (∀k : 0≤k<i : a[k]=0)
[A2] 0≤i≤n
[A3] i < n
[G] n − i > 0
....
END

```
{* 0 ≤ n *}
i := 0 ; r := true ;
 while  i < n  do  {
    r := r ∧ (a[i] = 0);
    i++ }
{* r = (∀k : 0≤k<n : a[k] = 0) *}
```

Loop with breaks, LN Sec. 9.3

- There is no `break` statement in uPL, but we can still look at the concept behind it, namely to stop a loop earlier because we know that it has achieved its objective.

```
i := 0 ; r := true ;
while  i<n do  {
  r := r ∧ (a[i]=0);
  i++
}
```



```
i := 0 ; r := true ;
while i<n ∧ ¬r do  {
  r := r ∧ (a[i]=0);
  i++
}
```

- How to prove that breaking the loop early is indeed safe?

Loop with breaks

Consider the following loop, which has been proven correct using an invariant I and a termination metric m:

(1) $\{^* P ^*\}$ **while** g **do** S $\{^* Q ^*\}$

Suppose now we optimise the loop by adding a "break" to let it terminate early in some situations

(2) $\{^* P ^*\}$ **while** $g \wedge h$ **do** S $\{^* Q ^*\}$

The validity of (1) implies

- (2) will also terminate
- (2) will also maintain I's invariance
- If g becomes false, (2) will terminate in Q

Loop with breaks

That is, we need to prove the premises of this rule:

$$\frac{\{* g \wedge h \wedge I *\}\ \ S\ \ \{* I *\} \qquad I \wedge \neg(g \wedge h) \Rightarrow Q \qquad \{* I \wedge g \wedge h *\}\ \ C{:=}m; S\ \ \{* m{<}C *\} \qquad I \wedge g \wedge h \Rightarrow m > 0}{\{* I *\}\ \ \textbf{while}\ (g \wedge h)\ \textbf{do}\ S\ \{* Q *\}}$$

when we already showed that:

1. $\{* g \wedge I *\}\ \ S\ \ \{* I *\}$
2. $I \wedge \neg(g) \Rightarrow Q$
3. $\{* I \wedge g *\}\ \ C{:=}m; S\ \ \{* m{<}C *\}$
4. $I \wedge g \Rightarrow m > 0$

The only premise which is stronger than what we have is: $I \wedge \neg(g \wedge h) \Rightarrow Q$

Since we've already shown that $I \wedge \neg g \Rightarrow Q$

all that's left to prove is $I \wedge g \wedge \neg h \Rightarrow Q$

Example

```
{* 0≤n *}
i := 0 ; r := true ;
while  i<n ∧ r do  {
   r := r ∧ (a[i]=0)  ;
   i++
}
{* r = (∀k : 0≤k<n : a[k]=0) *}
```

We have proven the basic form of this program (without the break with ∧ r) before. Now prove that with the break the program is still correct.

$I \wedge g \wedge \neg h \Rightarrow Q$

I : (r = (∀k : 0≤k<i: a[k]=0)) ∧ 0≤i≤n

I g $\neg h$

(r = (∀k : 0≤k<i: a[k]=0)) ∧ 0≤i≤n ∧ i<n ∧ ¬r

⇒ r = (∀k : 0≤k<n : a[k]=0)

Q

Dealing with more constructs

LN Section 6.10 -- 6.12

Simultaneous assignment

$$x,y := e_1,e_2$$

- Simultaneous assignment $x,y := e_1,e_2$ evaluates e_1 and e_2 in the current state to values v_1 and v_2, then assign these to x and y respectively.

$$wp\ (x,y := e_1,e_2)\ Q = Q[e_1,e_2/x,y]$$

- Simultaneous substitution $Q[e_1,e_2/x,y]$ replaces any free occurrence of x in Q with e_1, and y with e_2. In particular, we should **not** do the substitution of y ***after*** the substitution of x, nor the other way around.

The order of assignment matters

- The order in which you assign variables matters (you already know this). E.g. these three have different behaviour:

wp (x:=x∗y; y:=x+y) (x=2 ∧ y=1) = x∗y=2 ∧ x∗y+y=1

equivalent to: x=−2 ∧ y=−1

wp (y:=x+y; x:=x∗y) (x=2 ∧ y=1) = x(x+y)=2 ∧ x+y=1

equivalent to: x=2 ∧ y=−1

wp (x,y := x*y, x+y) (x=2 ∧ y=1) = x*y=2 ∧ x+y=1

which has no integer solution.

Assignment into an array

```
(c → e1 | e2)
=
if c then e1 else e2
```

- Recall :

$$wp\ (x := e)\ Q \quad = \quad Q[e/x]$$

- Can we extend the rule to $a[e_1] := e_2$?
- Let's try and see:

(valid pre-cond) `{* y = 10 *}` `a[i] := y` `{* a[i] = 10 *}`

(not a valid pre-cond) `{* a[0] = 10 *}` `a[i] := y` `{* a[0] =10 *}`

Fix : `{* (i=0 → y | a[0]) = 10 *}`

The formalism

- Let `a` be an array. Introduce this notation to mean the same array as `a`, except its `i`-th element, which is `e`:

```
a (i  repby  e)
```

- Formally defined via this equation:

```
a(i repby e)[k]  =  (i=k  →  e  |  a[k])
```

- Treat array assignments like `a[i] := e` as the assignment

```
a  :=  a(i repby e)
```

This has the same effect, but now we can safely use the old ***wp*** rule for assignment.

Example

- Prove: {* i≠k *} a[k]:= 0 ; a[i]:=1 {* a[k] = 0 *}
- Let's first calculate the ***wp***

```
{* (i=k → 1 | 0)                          = 0 *}     // def. repby
{* (i=k → 1 | a(k repby 0)[k])  = 0 *}     // wp
a[k]:=0                                  //  a :=  a(k repby 0)
{* (i=k → 1 | a[k])  = 0 *}                          // def. repby
{* a(i repby 1)[k]   = 0 *}                          // wp
a[i]:=1     // a := a(i repby 1)
{* a[k] = 0 *}
```

- Now we need to prove that i≠k implies the ***wp***.

Example

4. What is the **weakest** pre-condition of the following statement with respect to the given post-condition?

$$\{* \; ? \; *\} \quad \texttt{a[k]} := \texttt{a[0]}+\texttt{a[k]} \quad \{* \; \texttt{a[0]}=\texttt{a[k]} \; *\}$$

(a) $\texttt{a[0]} = \texttt{a[0]} + \texttt{a[k]}$

(b) $\texttt{a(0 }\mathbf{\text{repby}}\texttt{ a[0]+a[k])[0]} = \texttt{a[0]} + \texttt{a[k]}$

(c) $\texttt{a(k }\mathbf{\text{repby}}\texttt{ a[0]+a[k])[0]} = \texttt{a[0]} + \texttt{a[k]}$

(d) $\texttt{a[0]}\ \mathbf{\text{repby}}\ \texttt{a[0]+a[k]} = \texttt{a[k]}\ \mathbf{\text{repby}}\ \texttt{a[0]+a[k]}$

Example

2. What is the **weakest** pre-condition of the following statement with respect to the given post-condition?

$$\{*\ ?\ *\} \quad \mathtt{a}[0] := \mathtt{a}[0] - \mathtt{a}[\mathtt{k}] \quad \{*\ \mathtt{a}[\mathtt{k}]{=}0\ *\}$$

(a) $\mathtt{a}[\mathtt{k}] = 0$

(b) $\mathtt{k} = 0$

(c) $(\mathtt{k}{=}0 \rightarrow \mathtt{a}[0]{-}\mathtt{a}[\mathtt{k}] \mid \mathtt{a}[\mathtt{k}]) = 0$

(d) $\mathtt{a}(0\ \mathbf{repby}\ (\mathtt{a}\ \mathbf{repby}\ 0) - (\mathtt{a}\ \mathbf{repby}\ \mathtt{k}))[\mathtt{k}] = 0$

Next up

- The semantics of program calls
- How to find loop invariants
- Tail recursion and loops
- Using program calls

Reducing a program spec to a statement spec

- So far, we only looked Hoare logic at the statement level
- Let's now look at the program level
- How can we show that a specification of a program *P* is correct?

$\{* R *\}$ `P(x : int, …)` *body* $\{* Q *\}$

Reducing a program spec to a statement spec

- ***uPI*** supports simple program definitions
  - program name: *P*
  - formals parameters (more about these later): `x, y, z, …`
  - sequence of statements: S_1; S_2; S_3; …
  - a single return statement at the end (optional)

```
P(x, y, z, …) { S1; S2; S3; …; return e} ;
```

Reducing a program spec to a statement spec

- Consider the program below with the given specification:

```
{* x > 0 *}  P(x) { x++ ; return x } {* return  = x+1 *}
```

- We first need to *reduce* such a program-level specification to a specification in terms of its body.

- Such reduction raises several issues:

  - What is the logic of a **`return`** statement ?

  - Should `x` in the post condition refer to its initial or final value?

  - How to refer to the old value of a variable?

Some restrictions imposed on uPL

- Restrictions imposed on uPL:
  - the **`return`** statement should only appear as the *last statement in the program*.
  - formulas in the pre- and post-conditions of a program-level specification of a program **P** can only refer to **P**'s parameters or to *auxiliary variables* (explained later).
  - parameters are either ***passed by value***, or ***passed by copy-restore***. In particular uPL does not allow references/pointers to be passed to a program.

Pass by copy-restore

- Arrays are by default passed by copy-restore. A parameter with **OUT** marker is passed by copy-restore.
- Example:

```
P(OUT  x, OUT y) { x := true ; y := false }
```

- The behaviour of a call e.g. `P(a,b)` is as follows:

  The values of `a` and `b` are copied to `P`'s local variables `x` and `y`. The the body of `P` is executed. Then the final values of `x` and `y` are copied to `a` and `b`, in that order.

Pass by copy-restore, why?

- Why not pass-by-reference?
  - this could create **aliases** (different variables pointing to the same data-cell). This complicates the logic of assignment. Recall this logic:

$$\{*\ P\ *\}\ \texttt{x:=}\ e\ \{*\ Q\ *\} \quad = \quad P \Rightarrow Q[e/x]$$

    This assignment will now also affect the meaning of x's aliases! The above logic is too simplistic to handle this.

- In this course we want to avoid this complication, to focus more on the basic Hoare logic.

- Pass-by-copy-restore still allows us to simulate programs with side effect, which is an interesting class to consider.

Meaning of variables in the post-condition

- If `y` is a pass-by-copy-restore parameter, in the post-condition it refers to its final value. Example:

```
{* true *} P(OUT y) { y := 1 } {* y > 0 *}
```

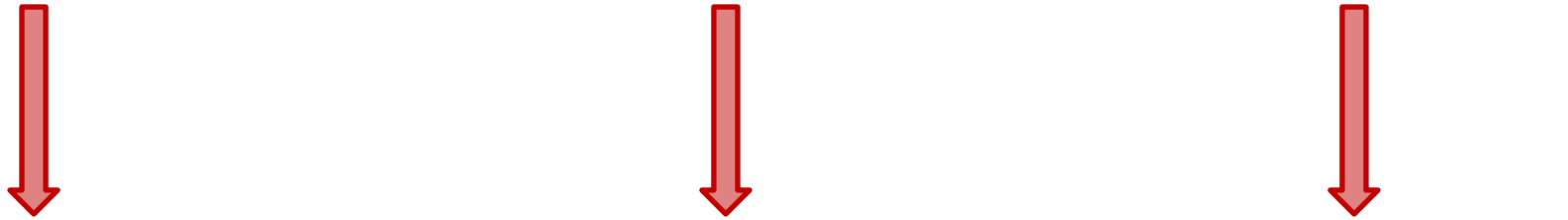
- If `x` is a pass-by-value parameter, in the post-condition it refers to its **initial value.** Example:

```
{* true *}P(x, OUT y){y := x; x:=0} {* y = x *}
```

Example of the reduction

- Consider again the previous example. To reduce the specification to the statement level we will do the following transformation:

```
{* x > 0 *} P(x:int){ x++ ; return x }{* return = x+1 *}
```



Introduce auxiliary variable(s) to remember the initial value of the parameters,

Replace pass-by-value parameters with reference to their initial values.

How to refer to parameters' old value?

- Consider a program **P(OUT x)** that increases the value of **x** by one. How to write a specification of this **P**? Again, we will use an auxiliary variable to remember **x**'s old value:

```
{* true *}
   X:=x; P(OUT x){x := x+1}
{* x = X+1 *}
```

Here, **X** is an **auxiliary variable**. It is not part of the program P. It is introduced just for the purpose of expressing P's specification.

```
{* true *} X:=x; x := x+1 {* x = X+1 *}
```

The corresponding statement-level specification.