Lecture Notes on Generating Verification Conditions

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Lecture 7
September 17, 2024

1 Introduction

Dynamic logic is very general. Among other things, it allows us to prove program equivalence, which you explored in Assignment 2 (Task 5) and Assignment 3 (Task 1). Here is another instance. Just using the axiom

$$[\alpha\ ;\ \beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

we can formally prove that sequential composition is associative.

$[\alpha\ ;\ (\beta\ ;\ \gamma)]Q$
$\leftrightarrow [\alpha]([\beta\ ;\ \gamma])Q$
$\leftrightarrow [\alpha]([\beta]([\gamma]Q))$
$\leftrightarrow [\alpha\ ;\ \beta]([\gamma]Q)$
$\leftrightarrow [(\alpha\ ;\ \beta)\ ;\ \gamma]Q$

When we are proving safety (or even correctness) we'd like to take advantage of the special form of pre- and post-conditions that are formulated purely in the theory of arithmetic (or maybe the theory of arrays if we model memory), namely

$$P \rightarrow [\alpha]Q$$

where P is the precondition, α is the program we are trying to verify, and Q is the postcondition. This is often written as $\{P\}\ \alpha\ \{Q\}$ and called a *Hoare triple*.

For this and the next lecture, we make the following assumptions:

1. The precondition P and postcondition Q are formulas of pure arithmetic. While they may contain expressions, they may not contain programs.
2. All formulas occurring in the program α (in conditionals, loop guards, assertions, and tests) are formulas of pure arithmetic.

The formulas of *pure arithmetic* are a subset of all formulas defined by

$$\text{Pure formulas} \quad P, Q \quad ::= \quad e_1 \leq e_2 \mid e_1 = e_2 \mid P \wedge Q \mid P \vee Q \mid P \to Q \mid \neg P \mid \top \mid \bot$$

We probably should use a new notation for such formulas, but since in today's lecture we consider only pure ones (except when recalling axioms), maybe we can track this in our heads.

As we proceed, we should also consider the status of quantifiers. On the one hand we sometimes need to refer to them in pre- and post-conditions (especially when reasoning about arrays). On the other hand, they make the theorem proving problem much harder. Plus, they shouldn't appear in program formulas like conditionals or loop guards because then rather than computing a Boolean value, we'd have to crank up a theorem prover while the program is running (potentially have to solve a problem in some undecidable class).

2 Guards

Consider the scenario where you are given the program from the previous section but no loop invariant. We might be able to guess a loop invariant, but if not we are stuck. The program looks unsafe, even with the precondition $0 \leq n \leq |M|$. If we still need to run it, what can we do? One option would be *dynamic monitoring*: we track memory accesses as the program executes and abort it if it attempts to do something unsafe. Another one is to *instrument it with guards* before memory accesses. These guards abort the program if the access would be unsafe and let it go on if they are safe. Aborting programs is considered safe, because aborting is actually a well-defined operation that does no harm (except to the running program, but it is its own fault if it tries to execute an unsafe command). For this purpose we need a new command $\mathbf{test}\ P$. In the literature on dynamic logic this is called a *guard* and written as $?P$. It has the following specification.

$$\begin{array}{lll} \omega[\![\mathbf{test}\ P]\!]\nu & \text{iff} & \omega \models P \text{ and } \nu = \omega \\ \omega[\![\mathbf{test}\ P]\!]\lightning & \text{iff} & \text{false} \end{array}$$

The program $\mathbf{test}\ \bot$ will not have a poststate, but it is also safe because it aborts. As a result, based on the definition of $\omega \models [\alpha]Q$ it is the case that

$$\omega \models [\mathbf{test}\ \bot]Q$$

for any ω and Q. This in turn means that $[\mathbf{test}\ \bot]Q$ is logically valid and $\cdot \vdash [\mathbf{test}\ \bot]Q$ should be derivable.

Just to be sure, let's recall the definition of $\omega \models [\alpha]Q$ from Lecture 5, page L5.4.

$$\begin{array}{lll} \omega \models [\alpha]Q & \text{iff} & \text{for every } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ we have } \nu \models Q \\ & & \text{and } \textbf{not}\ \omega[\![\alpha]\!]\lightning \end{array}$$

This is a *partial correctness* statement: if there is no poststate ν such that $\omega[\![\omega]\!]\nu$, then the first part of the condition is vacuously true.

What does this mean for the axiom for $[\mathbf{test}\ P]Q$? If P is true, then Q should also be true. But if the test succeeds then we know P, so we conjecture (somewhat rashly, perhaps)

$$[\mathbf{test}\ P]Q \leftrightarrow (P \to Q)$$

What if P is false? Then the program $\mathbf{test}\ P$ has no poststate, and yet it is safe. Consequently $[\mathbf{test}\ P]Q$ should be true, and by this axiom it will be becase $\bot \to Q$ is valid.

Let's prove that this axiom is valid.

Theorem 1 *The axiom*

$$[\mathbf{test}\ P]Q \leftrightarrow (P \to Q)$$

is valid.

Proof: From right to left we set up for an arbitrary ω

$\omega \models P \to Q$ (assumption)
$\ldots$
$\omega \models [\mathbf{test}\ P]Q$ (to show)

The conclusion holds if for every ν such that $\omega[\![\mathbf{test}\ P]\!]\nu$ we have $\nu \models Q$ (and **not** $\omega \models [\![\mathbf{test}\ P]\!]\lightning$, which is true).

So we assume $\omega[\![\mathbf{test}\ P]\!]\nu$ and have to show that $\nu \models Q$. By definition, this second assumption give us $\omega \models P$ and $\nu = \omega$.

By the first assumption also $\omega \models Q$ and since $\nu = \omega$ we have $\nu \models Q$

For the left to right direction, we set up for an arbitrary ω

$\omega \models [\mathbf{test}\ P]Q$ (assumption)
$\ldots$
$\omega \models P \to Q$ (to show)

So we assume $\omega \models P$ and it remains to show that $\omega \models Q$. Since $\omega \models P$, the first assumption gives us $\nu \models Q$ for any ν with $\omega[\![\mathbf{test}\ P]\!]\nu$. We can use this for $\nu = \omega$ (since $\omega \models P$) to obtain $\omega \models Q$. $\square$

We can easily turn the two directions of the axiom into right and left rules of the sequent calculus.

$$\frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash [\mathbf{test}\ P]Q, \Delta}\ [\mathbf{test}]R \qquad \frac{\Gamma \vdash P, \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, [\mathbf{test}\ P]Q \vdash \Delta}\ [\mathbf{test}]L$$

2.1 Assertions

We defined $\mathbf{test}$ so that it is never unsafe. A related type of program that appears in many languages is the $\mathbf{assert}$ command

$$\mathbf{assert}\ P$$

where P is a formula that may depend on variables. The meaning of $\mathbf{assert}\ P$ is similar to $\mathbf{test}\ P$, except that *unsafe* if P is false; otherwise it is safe but does not change the state.

$$\begin{array}{lll} \omega[\![\mathbf{assert}\ P]\!]\nu & \text{iff} & \omega \models P \text{ and } \nu = \omega \\ \omega[\![\mathbf{assert}\ P]\!]\lightning & \text{iff} & \omega \not\models P \end{array}$$

It should be clear that we preserve the property that unsafe programs have no poststate.

Among other things, we could rewrite our programs using $\mathbf{assert}$ commands. For example, if we replaced $x := \mathbf{divide}\ e_1\ e_2$ by $\mathbf{assert}\ \neg(e_2 = 0)\ ;\ x := e_1/e_2$ the two programs would have the same meaning in every state (either both unsafe, or both safe and determinate).

How do we reason about $[\mathbf{assert}\ P]Q$? If P is true, then the postcondition Q must be true. If P is false, then Q is irrelevant: the formula $[\mathbf{assert}\ P]Q$ is always false. These two conditions are neatly captured by the axiom

$$[\mathbf{assert}\ P]Q \leftrightarrow P \wedge Q$$

So why do we need $\mathbf{assert}\ P$ in our language when we already have $\mathbf{test}\ P$? First, we wouldn't necessarily want to use $\mathbf{assert}$ to check for things like whether an array index is in bounds or not; if the assertion fails, then the program is unsafe, which doesn't change anything from how normal memory accesses work.

Rather, assertions are useful as a general-purpose way of specifying what is unsafe for a program to do. For example, we could use an assertion to model that it is unsafe to read from an input stream if the current user is unauthorized to do so; or that it is not safe to send output over a network interface after the program has read from a sensitive file. These policies can be modeled using assertions by introducing additional *ghost variables* that track the current state of the policy. You will explore using ghost variables to express safety properties like this in Homework 3.

Here are the corresponding right and left rules of the sequent calculus.

$$\frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash Q, \Delta}{\Gamma \vdash [\mathbf{assert}\ P]Q, \Delta}\,[\mathbf{assert}]R \qquad \frac{\Gamma, P, Q \vdash \Delta}{\Gamma, [\mathbf{assert}\ P]Q \vdash \Delta}\,[\mathbf{assert}]L$$

Here is a little table on the differences between $\mathbf{assert}\ P$ and $\mathbf{test}\ P$.

Poststate	$\omega[\![\textbf{assert } P]\!]\nu$ iff $\omega \models P$ and $\nu = \omega$	$\omega[\![\textbf{test } P]\!]\nu$ iff $\omega \models P$ and $\nu = \omega$
Safety	$\omega[\![\textbf{assert } P]\!]\lightning$ iff $\omega \not\models P$	$\omega[\![\textbf{test } P]\!]\lightning$ never
Axiom	$[\textbf{assert } P]Q \leftrightarrow P \wedge Q$	$[\textbf{test } P]Q \leftrightarrow (P \rightarrow Q)$

3 Sandboxing

Sandboxing unsafe behavior (including memory access through the read or write commands) proceeds as follows. We replace

- every allocation $M := \textbf{alloc } e$ with the program $\textbf{test } 0 \leq e \ ; M := \textbf{alloc } e$
- every memory read $x := M[e]$ with the program $\textbf{test } 0 \leq e < |M| \ ; x := M[e]$
- every memory write $M[e_1] := e_2$ with the program $\textbf{test } 0 \leq e_1 < |M| \ ; M[e_1] := e_2$
- every division $x := \textbf{divides } e_1 \ e_2$ with the program $\textbf{test } e_2 \neq 0 \ ; x := \textbf{divides } e_1 \ e_2$.
- every assertion $\textbf{assert } P$ with the program $\textbf{test } P$

Now we can safely execute the program. Equally importantly, perhaps, we can prove the safety of the program transformed in this manner.

Theorem 2 (Safety of Sandboxed Programs) *Given a program α under precondition P, we obtain the sandboxed α' as defined in the preceding paragraph.*
Then $\cdot \vdash P \rightarrow [\alpha']\top$.

Proof: We prove safety using the loop invariant $\top$ for every loop. Since any potentially unsafe command is immediately preceded by a guard, the safety condition incorporated into the rule will be provable since it is exactly the assumption enabled by the postcondition of the guard.

More formally, this proof would be carried out by an *induction over the structure of formulas and programs*. □

There are two optimizations that come to mind. We can introduce fresh temporaries in order to avoid recomputing the value of expressions. For example, instead of $\textbf{test } 0 \leq e < |M| \ ; x := M[e]$ we would insert $t := e \ ; \textbf{test } 0 \leq t < |M| \ ; x := M[t]$ for a fresh temporary variable t.

The other optimization is a bit trickier. At first one might think that if we can *prove* P when we encounter $\mathbf{test}\ P$ during the verification of safety we can replace it by **skip** (or $\mathbf{assert}\ \top$, which should be equivalent). However, in conditionals the postcondition is replicated:

$$[\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q \leftrightarrow (P \to [\alpha]Q) \wedge (\neg P \to [\beta]Q)$$

When the postcondition contains a program (which arises from sequential composition, for example), this program may be proved twice, once in each branch. A similar remark applies if we unfold loops because the program α is replicated.

So we can only replace $\mathbf{test}\ P$ with **skip** only if for all occurrences of $[\mathbf{test}\ P]Q$ in a safety proof we can prove P.

Returning to our earlier example, we can sandbox

$$n \leq |M| \to [i := 0\ ;\ \mathbf{while}\ (i < n)\{\ M[i] := i\ ;\ i := i + 1\ \}]\top$$

as

$$n \leq |M| \to [i := 0\ ;\ \mathbf{while}\ (i < n)\{\ \mathbf{test}\ 0 \leq i < |M|\ ;\ M[i] := i\ ;\ i := i + 1\ \}]\top$$

Without a loop invariant and stronger precondition we can't eliminate the guard.

4 Weakest Liberal Precondition

If we are given a program α and a postcondition Q, then a *sufficient precondition* is a pure formula P such that $P \to [\alpha]Q$. For example, $P = \bot$ is a sufficient precondition for any α and Q since $\bot \to [\alpha]Q$ is valid. We would like the *weakest* such precondition which means that it is implied by any other precondition. We call this the weakest *liberal* precondition if we consider *partial correctness*, which is exactly what $[\alpha]Q$ captures. As has become common practice we may drop the adjective *liberal* since we essentially never consider *total correctness* (that is, require termination to be proved).

In order to explain the term "weakest": we say P is *stronger than* Q if P implies Q. Then $P = \bot$ is the strongest formula (it implies everything), while $P = \top$ is the weakest: it only implies Q if Q is already true without the help of P.

Writing $\mathsf{wlp}\ \alpha\ Q$ for the weakest liberal precondition we want the following two properties:

1. $\mathsf{wlp}\ \alpha\ Q \to [\alpha]Q$ (it is a precondition)
2. If $P \to [\alpha]Q$ then $P \to \mathsf{wlp}\ \alpha\ Q$ (it the weakest among them).

It is easy to check that $\mathsf{wlp}\ \alpha\ Q \leftrightarrow [\alpha]Q$ because $[\alpha]Q$ satisfies both conditions. Sadly, we cannot just define $\mathsf{wlp}\ \alpha\ Q = [\alpha]Q$ because the right-hand side is not pure, and we therefore cannot just hand it off to a theorem prover for arithmetic.

When thinking about the desired property it quickly becomes clear that loops are a major obstacle to computing the weakest liberal precondition algorithmically. So we require the programmer to supply a *loop invariant* J for every loop and then construct a weakest liberal precondition *with respect to the given loop invariants*. We'll come back to this point in Section 6. For all the other constructs, we can derive an algorithm for computing it from the axioms.

When given a problem $P \to [\alpha]Q$, then we call $P \to \mathsf{wlp}\ \alpha\ Q$ the *verification condition*. If it can be shown valid by an arithmetic prover, then the original dynamic logic formula $P \to [\alpha]Q$ is valid.

5 Programs Without Loops

The function $\mathsf{wlp}\ \alpha\ Q$ is defined by *induction* over α. That is, we can make recursive calls to wlp, but only on constituents programs for α. The result, P, should be a pure formula as defined before (and, in particular, it should not contain any programs).

We go through the program constructs one by one, reminding ourselves of the axioms and then deriving from that a case in the definition of wlp. The key here is $\mathsf{wlp}\ \alpha\ Q \leftrightarrow [\alpha]Q$.

Sequential Composition. Recall from earlier in this lecture

$$[\alpha\ ;\ \beta]Q \leftrightarrow [\alpha]([\beta]Q)$$

Blindly using equivalences:

$$\mathsf{wlp}\ (\alpha\ ;\ \beta)\ Q = \mathsf{wlp}\ \alpha\ (\mathsf{wlp}\ \beta\ Q)$$

This actually works! $\mathsf{wlp}\ \beta\ Q$ will be the weakest liberal precondition of Q with respect to β, and we can use this as the postcondition for the next recursive call on α because it must be pure.

An interesting aspect of this clause is that we proceed through the program from right to left: when computing the weakest precondition of $\alpha\ ;\ \beta$ we first compute it for β and then for α. This is characteristic of this approach. Going in the other direction is also possible, but it either leads us to the *strongest postcondition* (which we will not discuss) or the closely related *symbolic evaluation* (which is the subject of Lecture 8.

Assignment. For assignment, we will take particular advantage of the purity of the postcondition. First, let's recall the axiom:

$$[x := e]Q(x) \leftrightarrow \forall x'.\, x' = e \to Q(x') \quad (x' \text{ fresh})$$

where x' is chosen so it does not already occur in e or $Q(x)$. We need this side condition for two reasons. (1) if we have some previous knowledge about x (like:

$x = 2$) then after an assignment (like: $x := 3$) we would reach inconsistent assumptions because they the two values of x would be in conflict. (2) We cannot just substitute e for x in $Q(x)$ because $Q(x)$ might contain programs. For example, in $[x := 5]([\mathbf{while}\ x > 0\ x := x - 1])$ the variable x in the program successively refers to $5, 4, 3, 2, 1, 0$, so substituting in the initial value 5 ist just plain incorrect.

Fortunately, when calculating the weakest precondition we know that $Q(x)$ is a formula of pure arithmetic. Significantly, it does not contain any programs. Because of that, substituting e for x is actually not problematic. We write this as $Q(e)$. So we define

$$\mathsf{wlp}\ (x := e)\ Q(x) = Q(e)$$

Let's run through an example to see the two clauses in the definition of wlp in action. The program $z := x\ ;\ x := y\ ;\ y := z$ swaps the contents of variables x and y using the auxiliary variable z. Let's say the postcondition is $x = b \wedge y = a$. We expect the weakest precondition to imply that before the execution of the program, $x = a \wedge y = b$ must be true.

$$\begin{array}{rl} & \mathsf{wlp}\ (z := x\ ;\ x := y\ ;\ y := z)\ (x = a \wedge y = b) \\ = & \mathsf{wlp}\ (z := x)\ (\mathsf{wlp}\ (x := y\ \ ;\ y := z)\ (x = a \wedge y = b)) \\ = & \mathsf{wlp}\ (z := x)\ (\mathsf{wlp}\ (x := y)\ (\mathsf{wlp}\ (y := z)\ (x = a \wedge y = b))) \end{array}$$

At this point in rightmost call will use the rule for assignment, substituting z for y in the postcondition. The new constructed postcondition then feeds into the prior call to wlp, and so on.

$$\begin{array}{rl} = & \mathsf{wlp}\ (z := x)\ (\mathsf{wlp}\ (x := y)\ (x = a \wedge z = b)) \\ = & \mathsf{wlp}\ (z := x)\ (y = a \wedge z = b) \\ = & y = a \wedge x = b \end{array}$$

In this case, we get essentially *exactly* the precondition we expected. It does not mention z, since z is written to in the first assignment so its value prior to execution is irrelevant.

Conditionals. Recall the axiom

$$[\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta]Q \leftrightarrow (P \rightarrow [\alpha]Q) \wedge (\neg P \rightarrow [\beta]Q)$$

Again, we can use this straightforwardly, making two recursive calls on subprograms.

$$\mathsf{wlp}\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta)\ Q = (P \rightarrow \mathsf{wlp}\ \alpha\ Q) \wedge (\neg P \rightarrow \mathsf{wlp}\ \beta\ Q)$$

The postcondition does not change in both calls and is therefore pure, while P is a program condition and therefore pure by our general assumption (2) from this lecture. Therefore, the result of wlp is pure.

Assertions and Tests. Again, the axioms:

$$\begin{array}{l}[\textbf{assert } P]Q \leftrightarrow (P \wedge Q)\\ [\textbf{test } P]Q \leftrightarrow (P \rightarrow Q)\end{array}$$

So:

$$\begin{array}{lcl}\mathsf{wlp}\ (\textbf{assert } P)\ Q & = & P \wedge Q\\ \mathsf{wlp}\ (\textbf{test } P)\ Q & = & P \rightarrow Q\end{array}$$

As in the case of conditionals, these results are pure because P is a program condition.

This leads us to the case of loops.

6 Loops

First, let's recall the right rule for $[\textbf{while}]R$. Roughly, it states that the loop invariant J must hold initially, that it must be preserved by one trip around the loop, and that it must imply the postcondition.

$$\frac{\Gamma \vdash J, \Delta \quad J, P \vdash [\alpha]J \quad J, \neg P \vdash Q}{\Gamma \vdash [\textbf{while } P\ \alpha]Q, \Delta}\ [\textbf{while}]R$$

In applying this rule we have to choose the right loop invariant J. Since this a very difficult problem (both in theory and in practice), we will assume for the remainder of this lecture and part of the next lecture that the programmer has supplied it—maybe they were lucky enough to have taken 15-122 *Principles of Imperative Computation*! Our notation here is just $\textbf{while}_J\ P\ \alpha$ where J is the loop invariant.

Another particularly tricky aspect of this rule is that neither Γ nor Δ are allowed to be used in the second and third premise. That's because Γ and Δ hold formulas that reference variables *in their state as we enter the loop*. However, the loop invariant must be preserved no matter how many times we go around the loop, so we cannot rely on any assumptions that holds as enter it. That's the same reason we cannot substitute an expression for a variable that appears in the loop.

So what to do? It turns out that we need a new logical connective to model this as a formula. We write $\Box P$ (pronounced "*white box P*") which is true exactly if P is valid (which means: true in every state, as we have defined when proving validity of our axioms). The right rule for $\Box P$ then wipes out any knowledge we might have about the current state.

$$\frac{\cdot \vdash P}{\Gamma \vdash \Box P, \Delta}\ \Box R$$

Semantically, it is also easy to define

$$\omega \models \Box P \quad \text{iff} \quad \nu \models P \quad \text{for all states } \nu$$

It is a useful exercise to show the soundness of the $\Box R$ rule given this definition. Unfortunately, it is not invertible. For example, we have $\bot \vdash \Box(\bot)$, but we cannot prove the premise of the rule $\cdot \vdash \bot$.

We won't discuss the left rule or axioms for $\Box P$, because we don't need them in the context of this course. The particular modal logic we need here is called S4, and if you are curious you can read more about it in the Stanford Encyclopedia of Philosophy [Garson, 2000].

With this in hand, we can turn the $[\textbf{while}]R$ rule into an axiom.

$$\begin{array}{lcll}[\textbf{while}_J\ P\ \alpha]Q & \leftrightarrow & J & \text{(true initially)} \\ & \land & \Box(J \land P \rightarrow [\alpha]J) & \text{(preserved)} \\ & \land & \Box(J \land \neg P \rightarrow Q) & \text{(implies postcondition)}\end{array}$$

We have packaged up the *allowed antecedents* (like J and P) together with the succedent and then stashed under a white box in order to "erase" any other antecedents or succedents we might have.

We have written this as a bi-implication. If the loop invariant J were not specified in the syntax of the program, it would only be a right-to-left implication. Keeping this in mind, we can now turn this axiom into a definition of the weakest precondition.

$$\begin{array}{lcll}\mathsf{wlp}\ (\textbf{while}_J\ P\ \alpha)\ Q & = & J & \text{(true initially)} \\ & \land & \Box(J \land P \rightarrow \mathsf{wlp}\ \alpha\ J) & \text{(preserved)} \\ & \land & \Box(J \land \neg P \rightarrow Q) & \text{(implies postcondition)}\end{array}$$

We should check a few things. First, are all postconditions in calls to wlp pure? There is only one recursive call, and its postcondition J must be pure because J appears in the program and was therefore assumed to be pure. Also, the formula returned by wlp is pure *if we allow* $\Box P$ *as a pure formula*. This seems reasonable since it does not contain any program. So we revise:

$$\begin{array}{llcl}\text{Pure formulas} & P, Q & ::= & e_1 \leq e_2 \mid e_1 = e_2 \mid P \land Q \mid P \lor Q \mid P \rightarrow Q \mid \neg P \mid \top \mid \bot \\ & & \mid & \Box P\end{array}$$

The theorem provers we use don't understand the white box modality, so we have to take care to turn these pure formulas into their language. We comment on that in Section 8.

We can also see that if the loop invariant is **not** preserved or does **not** imply the postcondition, the weakest liberal precondition is equivalent to falsehood ($\bot$) because the white boxed formula is always either true or false.

7 A Loop Example

Let's reconsider an example from Lecture 4 and Lecture 5.

$$[\textbf{while}_{x \geq 0}\ (x > 1)\ x := x - 2]\,(0 \leq x \leq 1)$$

We have already written in the loop invariant we discovered the first time. Our precondition was $x \geq 6$, but we purposely omit it here to see what the the weakest liberal precondition might be. We expect that whatever it is should be *implied by* $x \geq 6$.

So we start to calculate (with $J = (x \geq 0)$).

$$\begin{aligned}
&\mathsf{wlp}\ (\mathbf{while}_{x\geq 0}\ (x > 1)\ x := x - 2)\ (0 \leq x \leq 1) \\
&= x \geq 0 \\
&\quad \wedge \Box\,(x \geq 0 \wedge x > 1 \rightarrow \mathsf{wlp}\ (x := x - 2)\ (x \geq 0)) \\
&\quad \wedge \Box\,(x \geq 0 \wedge \neg\,(x > 1) \rightarrow 0 \leq x \leq 1)
\end{aligned}$$

There is one recursive call to wlp, which we can work out by substitution (since it is an assignment):

$$\mathsf{wlp}\ (x := x - 2)\ (x \geq 0) = (x - 2 \geq 0)$$

Plugging this back in we get

$$\begin{aligned}
&\mathsf{wlp}\ (\mathbf{while}_{x\geq 0}\ (x > 1)\ x := x - 2)\ (0 \leq x \leq 1) \\
&= x \geq 0 \\
&\quad \wedge \Box\,(x \geq 0 \wedge x > 1 \rightarrow x - 2 \geq 0) \\
&\quad \wedge \Box\,(x \geq 0 \wedge \neg\,(x > 1) \rightarrow 0 \leq x \leq 1)
\end{aligned}$$

Since we can verify the two boxed formulas as being valid, the weakest liberal precondition is equivalent to $x \geq 0$. As expected, this is implied by $x \geq 6$.

8 White Box[1]

First a note on substitution. When we write $Q(x)$ where Q may contain formulas $\Box P$, then occurrence of x in P are allowed, but excluded from substitution. That's because P has to be *valid*, which means true in every state. Therefore any occurrence of x in P does not refer to the same x as outside the white box. For example, if

$$Q(x) = (x > 1 \wedge \Box(x < 0 \rightarrow -x > 0))$$

then

$$Q(5) = (5 > 1 \wedge \Box(x < 0 \rightarrow -x > 0))$$

We might say that "substitution is blocked by white boxes". This is related to the fact that substitution into $[\alpha]P$ may be prohibited, in particular if α contains loops or assignments.

Can we translate a formula with white boxes into arithmetic without boxes? This is what we need in order to pass it to a theorem prover for arithmetic. It turns

[1]not covered in lecture, but important for the first lab

out to be quite easy: we "pull out" all white boxed formulas to the top level and replace them by $\top$ or $\bot$, depending on whether they can be verified or not.

In the example from the previous section (not with the precondition), we'd like to verify

$$x \geq 6 \rightarrow [\mathbf{while}_{x \geq 0}\ (x > 1)\ x := x - 2]\,(0 \leq x \leq 1)$$

We calculate the weakest liberal precondition,

$$\begin{array}{l} \mathsf{wlp}\ (\mathbf{while}_{x \geq 0}\ (x > 1)\ x := x - 2)\,(0 \leq x \leq 1) \\ = x \geq 0 \\ \quad \wedge \Box\,(x \geq 0 \wedge x > 1 \rightarrow x - 2 \geq 0) \\ \quad \wedge \Box\,(x \geq 0 \wedge \neg\,(x > 1) \rightarrow 0 \leq x \leq 1) \end{array}$$

form the implication from the precondition, and pull out the white boxed formulas. We thus have to ask our theorem prover if all of the following are valid and plug in the results:

$$\begin{array}{lcl} p_1 & = & \mathsf{valid}\ (x \geq 0 \wedge x > 1 \rightarrow x - 2 \geq 0) \\ p_2 & = & \mathsf{valid}\ (x \geq 0 \wedge \neg\,(x > 1) \rightarrow 0 \leq x \leq 1) \\ \multicolumn{3}{l}{\mathsf{valid}\ (x \geq 6 \rightarrow x \geq 0 \wedge p_1 \wedge p_2)} \end{array}$$

If we determine the validity of p_1 and p_1 first (which will be true or false), we can then replace p_1 and p_2 with $\top$ or $\bot$ in the third line.

Fortunately, in this example, all of these are quite easy and valid. Note that we cannot just signal an error if p_1 or p_2 are invalid. For example, in

$$\mathsf{wlp}\ (\mathbf{if}\ \top\ \mathbf{then}\ x := x + 1\ \mathbf{else}\ (\mathbf{while}_{x=0}\ (x \geq 0)\ x := x - 1))\,(x = 5)$$

the loop invariant $x = 0$ is not preserved, but, logically, the weakest liberal precondition should be $x = 4$ since the implication $\neg\top \rightarrow \mathsf{wlp}\ (\mathbf{while} \ldots)\,(x = 5)$ is valid, even if $\mathsf{wlp}\ (\mathbf{while} \ldots)\,(x = 5) = \bot$.

Alternatively we could stipulate that loop invariants must be loop invariants wherever they occur and just abort with an error when given a program such as the one above.

9 Summary

We summarize the definition of $\mathsf{wlp}\ \alpha\ Q$.

$$\begin{array}{lcl} \mathsf{wlp}\ (\alpha\ ;\ \beta)\ Q & = & \mathsf{wlp}\ \alpha\ (\mathsf{wlp}\ \beta\ Q) \\ \mathsf{wlp}\ (x := e)\ Q(x) & = & Q(e) \\ \mathsf{wlp}\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta)\ Q & = & (P \rightarrow \mathsf{wlp}\ \alpha\ Q) \wedge (\neg P \rightarrow \mathsf{wlp}\ \beta\ Q) \\ \mathsf{wlp}\ (\mathbf{assert}\ P)\ Q & = & P \wedge Q \\ \mathsf{wlp}\ (\mathbf{test}\ P)\ Q & = & P \rightarrow Q \\ \mathsf{wlp}\ (\mathbf{while}_J\ P\ \alpha)\ Q & = & J \\ & & \wedge \Box(J \wedge P \rightarrow \mathsf{wlp}\ \alpha\ J) \\ & & \wedge \Box(J \wedge \neg P \rightarrow Q) \end{array}$$

References

James Garson. Modal logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Spring 2024 edition edition, 2000. URL `https://plato.stanford.edu/archives/spr2024/entries/logic-modal/`.