15-812: Programming Language Semantics

Lecture Notes on Proving Programs

André Platzer

Carnegie Mellon University

Lecture 4

1 Introduction

This lecture will focus on developing systematic logical reasoning principles for sequential programs. Writing programs with correctness specifications is one thing. But proving them to be correct is a different matter. Both are exceedingly useful, because the clear expression of our expectations on a program often already make it more correct as it will more likely occur to us if our expectations and the program's realization are out of sync. But, of course, we might still fail to notice that a program does not meet its correctness specification if all we do is look at them.

The fact that we unambiguously rendered program contracts in logic now plays to our advantage. Not only did this make it clear what a precondition and postcondition of a program really means. But logic also provides ways of reasoning logically (go figure) about the programs by systematically transforming one logical formula into a simpler logical formula to find out whether it is true. This will lead us to discover a very systematic logical way of reasoning about the correctness of sequential programs. More information on the topic of axioms for reasoning about the behavior of programs in dynamic logic can also be found in the literature [HKT00, Pla17b].

2 Semantical Considerations on Programs

Recall the dynamic logic formula for the program swapping two variables x and y in place:

$$x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a) \tag{1}$$

Its meaning, and thus the meaning of the program contract that it came from, are now mathematically defined precisely. What can we do with its mathematical semantics?

Well, we could, for example, follow the definitions of the semantics to find out how a specific initial state ω changes as the program is executing. Consider the initial state ω with $\omega(x) = 5$ and $\omega(y) = 7$. For this state to satisfy the preconditions, it also needs to have the following values $\omega(a) = 5$ and $\omega(b) = 7$ for variables a and b. Thus,

$$\omega \in I[\![x = a \land y = b]\!]$$

Since the swap program only changes the variables x and y, we only need to track their values, since everything else stays unchanged. After running the first assignment $x := x + y$, the program reaches state a μ_1 with $\mu_1(x) = 12, \mu_1(y) = 7$. After running the second assignment $y := x - y$; from state μ_1 the program reaches a state μ_2 with $\mu_2(x) = 12, \mu_2(y) = 5$. After running the third assignment $x := x - y$; from state μ_2 the program reaches a state ν with $\nu(x) = 7, \nu(y) = 5$. Let's write the respective program statements in the first row and the states in between these in the next rows:

$$\begin{array}{ccccccc}
 & x := x + y; & & y := x - y; & & x := x - y & \\
\omega(x) = 5 & & \mu_1(x) = 12 & & \mu_2(x) = 12 & & \nu(x) = 7 \\
\omega(y) = 7 & & \mu_1(y) = 7 & & \mu_2(y) = 5 & & \nu(y) = 5
\end{array}$$

All those states agree that a has the value 5 and b the value 7. So indeed, the (only) final state ν satisfies the postcondition:

$$\omega \in I[\![x = b \land y = a]\!]$$

Well that's nice. We followed the semantics of program execution from the particular initial state ω with $\omega(x) = 5$ and $\omega(y) = 7$ and found out that all its final states (well ν is the only one) satisfy the postcondition that formula (1) claims. This justifies that (1) is true in state ω:

$$\omega \in I[\![x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a)]\!]$$

In fact, since we just saw there is a final state ν in which the postcondition is true, this also justifies the diamond modality case is true in state ω:

$$\omega \in I[\![x = a \land y = b \rightarrow \langle x := x + y; y := x - y; x := x - y\rangle(x = b \land y = a)]\!]$$

Lovely. Now all we need to do to justify that DL formula (1) is not just true in this particular initial state ω but is valid in all states, is to consider one state at a time and follow the semantics to show the same.

The only downside of that approach of following the semantics through concrete states is that it will keep us busy till the end of the universe because there are infinitely many different states. Even among those initial states that satisfy the precondition $x = a \land y = b$ (otherwise there is nothing to show for (1) since implications are true if their left hand sides are false), there are still infinitely many such states. That's not very practical for such a simple program nor, in fact, for any other interesting program with input.

3 Axioms for Programs

Our approach to understanding programs with logic is to design one reasoning principle for each program operator that describes its effect in logic with simpler logical operators. If we succeed doing that for every operator that a program can have, then we will understand even the most complicated programs just by repeatedly making use of the respective logical reasoning principles.

3.1 Conditionals

The first case we choose to look at is what we need to prove in order to show the formula $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$, which expresses that formula P always holds after running the if-then-else conditional $\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta$ that runs program α if formula Q is true and runs β otherwise. In order to understand it from a logical perspective, how could we express $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$ in easier ways?

If Q holds then $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$ says that P always holds after running α. If Q does not hold then the same formula $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$ says that P always holds after running β. It is easy to say with a logical formula that P always holds after running α, which is precisely what $[\alpha]P$ is good for. Likewise $[\beta]P$ directly expresses in logic that P always holds after running β. Both of those formulas $[\alpha]P$ as well as $[\beta]P$ are simpler than the original formula $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$. But, of course, they express something else, because the program $\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta$ only runs the respective programs conditionally depending on the truth-value of Q.

Yet, there still is a way of expressing $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$ in logic in easier ways with the help of other logical operators. Implications are perfect at expressing the conditions that an if-then statement states in a program. Indeed, if Q holds then $[\alpha]P$ needs to be true and if Q does not hold then $[\beta]P$ for $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$ to hold. Indeed, $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$ is true if and only if $(Q \to [\alpha]P) \land (\neg Q \to [\beta]P)$ is true. We capture this argument once and for all in the if-then-else axiom R1:

$$\text{R1}\quad [\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P \leftrightarrow (Q \to [\alpha]P) \land (\neg Q \to [\beta]P)$$

From now on, every time we want to make use of this equivalence, we just refer to it by name: R1. And, indeed, this axiom tells us everything we need to know about if-then-else statements. When using the equivalence R1 from left to right, we can use it to simplify every question about an if-then-else statement of the form $[\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P$ by a corresponding structurally simpler formula $(Q \to [\alpha]P) \land (\neg Q \to [\beta]P)$ that does not use the if-then-else statement any more but is logically equivalent. The axiom will enable us, for example to conclude this equivalence:

$$[\mathsf{if}(x{\geq}0)\,y := x\,\mathsf{else}\,y := -x]y{=}|x| \leftrightarrow (x{\geq}0 \to [y := x]y{=}|x|) \land (\neg x{\geq}0 \to [y := -x]y{=}|x|)$$

This formula uses $|x|$ as notation for the absolute value of x.

Whether the right hand side of axiom R1 is really seriously simpler than its left hand side needs a moment's thought because it is longer. But the point is that, even if it may be textually longer, the right hand side is structurally simpler, because it does not use the if-then-else statement anymore but subprograms and simpler logical operators.

Also, since axiom R1 justifies this equivalence, we will be able to reduce a question about whether its left hand side is valid with axiom R1 to the question whether its corresponding right hand side is valid. In sequent calculus proofs, we will, thus, mark the use of such an axiom by giving its name R1:

$$\text{R1}\ \frac{\vdash (x{\geq}0 \rightarrow [y := x]y{=}|x|) \land (\neg x{\geq}0 \rightarrow [y := -x]y{=}|x|)}{\vdash [\mathsf{if}(x{\geq}0)\, y := x\ \mathsf{else}\ y := -x]y{=}|x|}$$

Almost always will we take care to only use axioms for reducing its left hand side to the structurally simpler right hand side in order to make sure the proof makes progress toward simpler formulas.

This proof step with axiom R1 beautifully took care of the if-then-else statement in the conclusion and reduced it to a formula without if-then-else statements. That took us closer to a proof, but we still need other axioms, e.g., for assignments to complete the proof of the remaining formula even if the propositional logical proof rules such as $\land$R and $\rightarrow$R will already excel at handling the $\land$ and $\rightarrow$ operators in the premise. But observe how nicely the R1 axiom allows us to reduce a proof of an if-then-else program to a logical combination of questions about subprograms. We will try to identify similar axioms that reduce a property of a composed program to a logical combination of properties of subprograms also for all the other statements in a program. That way we will obtain a compositional reasoning technique that reduces the correctness of any arbitrary big program to a number of questions about smaller and smaller subprograms, of which there are only finitely many.

3.2 Test

The if-then-else statement branches execution of the program depending on the truth-value of its condition in the current state. The test statement $?Q$ also checks a condition on the current state. The difference is that it has no effect on the state if Q is indeed true, but aborts and discards the execution if Q is not true. How can we express $[?Q]P$ in logic in structurally simpler ways? In fact, let's preferably express $[?Q]P$ equivalently in simpler ways, because that equivalence principle worked so well in axiom R1.

The formula $[?Q]P$ is true iff formula P holds always after running the test $?Q$, which can only run if Q is true. What happens if the test program $?Q$ cannot run because Q is false? Well in that case nothing needs to be shown, because $[?Q]P$ merely expresses that P holds after all runs of the program $?Q$, which is vacuously true for any postcondition if there simply isn't a run of $?Q$ at all because Q is false in the current state.

Consequently P holds after all runs of the program $?Q$ iff postcondition P is true if the test Q is. That is iff the test formula Q implies the postcondition P. This is captured in the test axiom [?]:

$$[?]\quad [?Q]P \leftrightarrow (Q \rightarrow P)$$

3.3 Assignments

The next case to look into is what we need to prove in order to show the formula $[x := e]p(x)$, which expresses that the formula $p(x)$ holds after the assignment $x := e$ that assigns the value of term e to variable x. How could we reduce this to another logical formula that is simpler?

If we want to show that the formula $p(x)$ holds after assigning the new value e to variable x then we might as well show $p(e)$ right away. And, in fact, p is true of x after assigning e to x if and only if p is true of its new value e. That is, the formula $[x := e]p(x)$ is equivalent to the formula $p(e)$. We capture this argument once and for all in the assignment axiom [:=]:

$$[:=] \quad [x := e]p(x) \leftrightarrow p(e)$$

In the assignment axiom [:=], the formula $p(e)$ has the term e everywhere in place of where the formula $p(x)$ has the variable x. Of course, it is important for this substitution of e for x to avoid capture of variables and not make any replacements under the scope of a quantifier or modality binding an affected variable [Pla17a]. For example, the following formula is an instance of [:=]:

$$[x := x^2 - 1]x(x + 1) \geq x + y \leftrightarrow (x^2 - 1)(x^2 - 1 + 1) \geq (x^2 - 1) + y$$

But the following is not because it would capture the replacement y that is used for x:

$$[x := y](x \geq 0 \land \forall y\, (x \geq y)) \leftrightarrow (y \geq 0 \land \forall y\, (y \geq y))$$

Instead, if we first rename $\forall y$ to $\forall z$ then the substitution works:

$$[x := y](x \geq 0 \land \forall z\, (x \geq z)) \leftrightarrow (y \geq 0 \land \forall z\, (y \geq z))$$

The axioms we saw so far already enable us to do a first proof:

$$\text{R1}\;\cfrac{\land\text{R}\;\cfrac{\to\text{R}\;\cfrac{[:=]\;\cfrac{\mathbb{R}\;\cfrac{*}{x\geq0 \vdash x=|x|}}{x\geq0 \vdash [y := x]\, y=|x|}}{\vdash x\geq0 \to [y := x]\, y=|x|} \qquad \to\text{R}\;\cfrac{[:=]\;\cfrac{\mathbb{R}\;\cfrac{*}{\neg x\geq0 \vdash -x=|x|}}{\neg x\geq0 \vdash [y := -x]\, y=|x|}}{\vdash \neg x\geq0 \to [y := -x]\, y=|x|}}{\vdash (x\geq0 \to [y := x]\, y=|x|) \land (\neg x\geq0 \to [y := -x]\, y=|x|)}}{\vdash [\mathsf{if}(x\geq0)\, y := x \;\mathsf{else}\; y := -x]\, y=|x|}$$

This proof shows validity of the following formula, which says that the given program correctly implements the absolute value function $|\cdot|$ from mathematics:

$$[\mathsf{if}(x\geq0)\, y := x \;\mathsf{else}\; y := -x]\, y=|x|$$

The proof refers to propositional logic sequent calculus rules such as $\land$R and $\to$R as well as the dynamic logic axioms R1 and [:=]. The proof is developed starting with the desired conclusion at the bottom and working with proof rules to the top as usual in sequent calculus. The proof also makes use of integer arithmetic reasoning (marked by $\mathbb{R}$) to show that, indeed, if x is nonnegative then x equals the absolute value of x (on the left branch). Likewise, integer arithmetic reasoning is needed to show that if x is negative then $-x$ equals the absolute value of x (on the right branch). It is quite common for nontrivial arithmetic to be needed during program verification.

3.4 Sequential Compositions

The axioms we investigated so far already handle some programs, but sequential compositions are missing quite noticeably and we won't get very far in programs without them. So how can we equivalently express $[\alpha;\beta]P$ in simpler logic without sequential compositions? This formula expresses that P holds after all runs of $\alpha;\beta$, which first runs α and then runs β. How can this be expressed in an easier way in logic, again using just the subprograms α as well as β of $\alpha;\beta$ then?

In order to express $[\alpha;\beta]P$ what we need to say is that after all runs of α it is the case that P holds after all runs of β. It is comparably easy to say that P holds after all runs of β just with the formula $[\beta]P$. But where does this formula need to hold? After all runs of α. In particular, all we need to say is that $[\beta]P$ holds after all runs of α, which is exactly what the formula $[\alpha][\beta]P$ says. We capture these insights in the sequential composition axiom [;]:

$$[;] \quad [\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$$

Indeed, after all runs of $\alpha;\beta$ does P hold if and only if after all runs of α it is the case that after all runs of β does P hold.

These axioms already enable us to prove the correctness of the integer-based swapping function

$$x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a)$$

All we need to do is turn it into a sequent and start with this as the desired conclusion at the bottom of a sequent proof and successively apply axioms until the proof completes:

$$
\begin{array}{rl}
 & \qquad\qquad * \\ \hline
\mathbb{R} & x{=}a \land y{=}b \vdash y = b \land x = a \\ \hline
 & x{=}a \land y{=}b \vdash x + y - (x + y - y) = b \land x + y - y = a \\ \hline
[:=] & x{=}a \land y{=}b \vdash [x := x + y](x - (x - y) = b \land x - y = a) \\ \hline
[:=] & x{=}a \land y{=}b \vdash [x := x + y][y := x - y](x - y = b \land y = a) \\ \hline
[:=] & x{=}a \land y{=}b \vdash [x := x + y][y := x - y][x := x - y](x = b \land y = a) \\ \hline
[;] & x{=}a \land y{=}b \vdash [x := x + y][y := x - y; x := x - y](x = b \land y = a) \\ \hline
[;] & x{=}a \land y{=}b \vdash [x := x + y; y := x - y; x := x - y](x = b \land y = a) \\ \hline
\rightarrow\!R & \qquad \vdash x = a \land y = b \rightarrow [x := x + y; y := x - y; x := x - y](x = b \land y = a)
\end{array}
$$

Remember how we mark the use of arithmetic reasoning as $\mathbb{R}$. In practice, this is where verification tools use SMT solvers to handle satisfiability modulo theories, especially certain classes of arithmetical formulas. In our paper proofs we will take care to make sure we have a good reason why an arithmetic fact is true in all states and make a note of it below the proof. Here, for example, we might say:

the arithmetic proves since $x + y$ cancels $-(x + y)$ and y cancels $-y$

Note how this is now a proof of correctness of the swap program from (1) that, in a finite amount of work, justifies correctness for all states and, thus, implies its validity:

$$\vDash x = a \land y = b \to [x := x + y; y := x - y; x := x - y](x = b \land y = a)$$

The above sequent calculus proof used the assignment axiom inside out, so starting with handling the last assignment first. It would also have been possible to start outside in handling the first assignment first. That would have led to the following proof step:

$$
\begin{array}{ll}
 & \qquad\qquad \dots \\ \hline
{[:=]} & x=a \land y=b \vdash [y := x + y - y][x := x + y - y](x = b \land y = a) \\ \hline
{[:=]} & x=a \land y=b \vdash [x := x + y][y := x - y][x := x - y](x = b \land y = a) \\ \hline
{[;]} & x=a \land y=b \vdash [x := x + y][y := x - y; x := x - y](x = b \land y = a) \\ \hline
{[;]} & x=a \land y=b \vdash [x := x + y; y := x - y; x := x - y](x = b \land y = a) \\ \hline
{\to}\mathrm{R} & \qquad\qquad \vdash x = a \land y = b \to [x := x + y; y := x - y; x := x - y](x = b \land y = a)
\end{array}
$$

3.5 Loop the Loop

The final and most difficult case is that of the loop. How can we prove $[\mathsf{while}(Q)\,\alpha]P$ in another way by rephrasing it equivalently in logic? What the loop $\mathsf{while}(Q)\,\alpha$ does is to test whether formula Q is true and, if so, run α, and then repeating that process until Q is false (if it ever is, otherwise the loop just keeps running α until the end of time).

Let's try to understand that by cases. If Q holds then $[\mathsf{while}(Q)\,\alpha]P$ runs α and then runs the while loop afterwards yet again. If Q does not hold then the loop has no effect and just stops right away. That is why $\mathsf{while}(Q)\,\alpha$ is equivalent to $\mathsf{if}(Q)\,\{\alpha; \mathsf{while}(Q)\,\alpha\}$, because both have no effect if Q is false but repeat α as long as Q is true. We can capture these thoughts in the following axiom:

$$[\text{unwind}]\quad [\mathsf{while}(Q)\,\alpha]P \leftrightarrow [\mathsf{if}(Q)\,\{\alpha; \mathsf{while}(Q)\,\alpha\}]P$$

By applying the R1 axiom and the composition axiom [;] on the right hand side of axiom [unwind], we obtain the following minor variation of axiom [unwind] which we call [unfold]. But on paper we might just as well accept either name, because both axioms follow essentially the same idea and one can easily tell which one we refer to:

$$[\text{unfold}]\quad [\mathsf{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\mathsf{while}(Q)\,\alpha]P) \land (\neg Q \to P)$$

Both the unwinding axiom [unwind] and the closely related unfolding axiom [unfold] have a slight deficiency that we will get back to. Can you spot it already?

4 Soundness

Writing down axioms is one thing. Making use of them for proofs is quite helpful, too. But if the axioms are wrong, then that would not help making the programs any more correct. Consequently, it is imperative that all axioms we adopt are indeed sound, so only allow us to prove formulas that are actually valid.

Without any further delay, let us immediately make up for our mistake of using axioms that we have not proved correct yet by proving them sound before the lecture is over. An axiom is sound iff all its instances are valid formulas, so true in all states.

Lemma 1. *The test axiom* [?] *is sound, i.e. all its instances are valid:*

$$[?] \quad [?Q]P \leftrightarrow (Q \rightarrow P)$$

Proof. This lemma is so deserving of a proof that you simply must solve assignment 2 to find out how it works. □

Lemma 2. *The sequential composition axiom* [;] *is sound, i.e. all its instances are valid:*

$$[;] \quad [\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$$

Proof. Recall the semantics of sequential composition:

$$I[\![\alpha;\beta]\!] = I[\![\alpha]\!] \circ I[\![\beta]\!] = \{(\omega,\nu) \;:\; (\omega,\mu) \in I[\![\alpha]\!], (\mu,\nu) \in I[\![\beta]\!]\}$$

In order to show that the formula $[\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$ is valid, i.e. $\vDash [\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$, consider any state ω and show that $\omega \in I[\![[\alpha;\beta]P \leftrightarrow [\alpha][\beta]P]\!]$. We prove this biimplication by separately proving both implications.

"$\leftarrow$" Assume the right hand side $\omega \in I[\![[\alpha][\beta]P]\!]$ and show $\omega \in I[\![[\alpha;\beta]P]\!]$. To show the latter, consider any state ν with $(\omega,\nu) \in I[\![\alpha;\beta]\!]$ and show that $\nu \in I[\![P]\!]$. By the semantics of sequential composition, $(\omega,\nu) \in I[\![\alpha;\beta]\!]$ implies that there is a state μ such that $(\omega,\mu) \in I[\![\alpha]\!]$ and $(\mu,\nu) \in I[\![\beta]\!]$. The assumption implies with $(\omega,\mu) \in I[\![\alpha]\!]$ that $\mu \in I[\![[\beta]P]\!]$. This, in turn, implies with $(\mu,\nu) \in I[\![\beta]\!]$ that $\nu \in I[\![P]\!]$ as desired.

"$\leftarrow$" Conversely, assume the left hand side $\omega \in I[\![[\alpha;\beta]P]\!]$ and show $\omega \in I[\![[\alpha][\beta]P]\!]$. To show $\omega \in I[\![[\alpha][\beta]P]\!]$, consider any state μ with $(\omega,\mu) \in I[\![\alpha]\!]$ and show $\mu \in I[\![[\beta]P]\!]$. To show the latter, consider any state ν with with $(\mu,\nu) \in I[\![\beta]\!]$ and show $\nu \in I[\![P]\!]$. Now $(\omega,\mu) \in I[\![\alpha]\!]$ and $(\mu,\nu) \in I[\![\beta]\!]$ imply $(\omega,\nu) \in I[\![\alpha;\beta]\!]$ by the semantics of sequential composition. Consequently, the assumption $\omega \in I[\![[\alpha;\beta]P]\!]$ implies $\nu \in I[\![P]\!]$ as desired.

In a nutshell, we can also just observe that the semantics implies that: $\omega \in I[\![[\alpha;\beta]P]\!]$ iff $\nu \in I[\![P]\!]$ for all $(\omega,\nu) \in I[\![\alpha;\beta]\!] = I[\![\alpha]\!] \circ I[\![\beta]\!]$ iff $\mu \in I[\![[\beta]P]\!]$ for all $(\omega,\mu) \in I[\![\alpha]\!]$ iff $\omega \in I[\![[P][\beta]P]\!]$. □

Now of course all other axioms we use first need to be proved sound as well, but that is an excellent exercise.

The [unfold] axiom can be justified to be sound in another way. Rather than arguing by semantics, which would work, too, we can derive it with a sequent calculus proof from the other axioms. After all other axioms are proved to be sound the *derived axiom* [unfold] is thus sound too.

Lemma 3. *The following axiom is a* derived axiom, *so can be proved from the other axioms in sequent calculus, and is, thus, sound:*

$$\text{[unfold]}\quad [\mathsf{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\mathsf{while}(Q)\,\alpha]P) \land (\neg Q \to P)$$

Proof. The axiom [unfold] can be proved from the other axioms by using some of them in the backwards implication direction:

$$\dfrac{\dfrac{\dfrac{*}{\vdash [\mathsf{while}(Q)\,\alpha]P \leftrightarrow [\mathsf{if}(Q)\,\{\alpha;\mathsf{while}(Q)\,\alpha\}]P}\text{[unwind]}}{\vdash [\mathsf{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha;\mathsf{while}(Q)\,\alpha]P) \land (\neg Q \to P)}\text{R1}}{\vdash [\mathsf{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\mathsf{while}(Q)\,\alpha]P) \land (\neg Q \to P)}\text{[;]}$$

□

Every time we need the derived axiom [unfold], we could instead write down this sequent proof to prove it. It just won't be very efficient, so instead we will settle for deriving axiom [unfold] in the sequent calculus once and then just believing it from then on.

This gives us two ways of establishing the soundness of an axiom. Either by a mathematical proof from the semantics of the operators. Or as a derived axiom by a formal proof in sequent calculus from other axioms and proof rules that have already been proved to be sound. Of course, the first time a new operator is mentioned in any of our axioms, we cannot derive it yet but have to work from its semantics. But the second time, it may become possible to argue as in a derived axiom.

5 Summary

The axioms introduced in this lecture are summarize in Fig. 1.

References

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, 2000.

[Pla17a] André Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, 59(2):219–265, 2017. doi:10.1007/s10817-016-9385-1.

$$
\begin{aligned}
&[:=] && [x := e]p(x) \leftrightarrow p(e) \\
&[?] && [?Q]P \leftrightarrow (Q \to P) \\
&\text{R1} && [\mathsf{if}(Q)\,\alpha\,\mathsf{else}\,\beta]P \leftrightarrow (Q \to [\alpha]P) \land (\neg Q \to [\beta]P) \\
&[;] && [\alpha;\beta]P \leftrightarrow [\alpha][\beta]P \\
&[\text{unwind}] && [\mathsf{while}(Q)\,\alpha]P \leftrightarrow [\mathsf{if}(Q)\,\{\alpha;\mathsf{while}(Q)\,\alpha\}]P \\
&[\text{unfold}] && [\mathsf{while}(Q)\,\alpha]P \leftrightarrow (Q \to [\alpha][\mathsf{while}(Q)\,\alpha]P) \land (\neg Q \to P)
\end{aligned}
$$

Figure 1: Axioms of the day

[Pla17b] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Switzerland, 2017. URL: `http://www.springer.com/978-3-319-63587-3`.