15-812: Programming Language Semantics

Lecture Notes on Dynamic Logic

André Platzer

Carnegie Mellon University
Lecture 4

1 Program Contracts

Let's look at a few simple example programs and their precondition/postcondition contracts, which we continue to describe with `@requires` and `@ensures` clauses similar to what we used to do with contracts in the Principles of Imperative Computation course.

```
//@requires(x=a && y=b);
//@ensures (x=b && y=a);
{x:=x+y; y:=x-y; x:=x-y;}
```

What makes this program interesting is that, as the contract clearly expresses, it swaps the values of variables `x` and `y`, but it does so without needing any additional memory. That was sometimes important in the old days of limited memory but is also crucial for topics like reversible computation that are a prerequisite to quantum computing. For us, it's just a simple cute example of a program. But what's interesting to observe is that we need two additional logical variables `a` and `b` to even just describe the effect of the clever in-place swapping program in a contract. Indeed, this is reminiscent of the fact that a canonical implementation of swapping would first copy the value of `x` elsewhere, then copy `y` into `x` and then the value of the clone back into `y`.

2 The Logical Meaning of a Contract

Apparently, in order to have any chance at all at making sense of the above program contracts, we need to understand more than propositional logic. For one thing, our impoverished view of the world as just atomic propositions p, q, r that are true or false depending on some arbitrary interpretation I is insufficient.

For the gcd program, for example, it is not sufficient to consider $x \geq y$ as an atomic proposition p and $y > 0$ as an atomic proposition q and $x > 0$ as an atomic proposition r. If we did that, an interpretation I could happily interpret $I(p) = \mathit{true}$ and $I(q) = \mathit{true}$ but $I(r) = \mathit{false}$, which is impossible for the concrete arithmetic. For the swap program, it is not sufficient to consider $x = a$ as an atomic proposition p and $y = b$ as an atomic proposition q and $x = y$ as an atomic proposition r and $a = b$ as an atomic proposition s. For if that were the case, then nothing would prevent us from considering an interpretation I in which $I(p) = \mathit{true}$ and $I(q) = \mathit{true}$ and $I(r) = \mathit{true}$ but $I(s) = \mathit{false}$, because it is quite obviously impossible for $x = a, y = b, x = y$ to be true without $a = b$ being true in the same interpretation as well.

Consequently, it will become important for program contracts to consider logical formulas in which concrete terms like $x, x \cdot y, x + a$ or $x \mod a$ and so on occur and mean exactly what they do in integer arithmetic. Likewise, by the formula $x = y$ we exactly mean the equality comparison of terms x and y and by $x \geq y$ we exactly mean that the value of x is greater-or-equal to the value of y. These are *interpreted* because we have a fixed interpretation in mind for equality ($=$) and greater-or-equal comparison ($\geq$) and addition ($+$) and multiplication ($\cdot$) and so on.

The precise rendition of a contract for the greatest common divisor also inevitably needs a universal quantifier to say that, among all other divisors, the gcd is the greatest. Consequently, we will also find it crucial to extend propositional logic to first-order logic, which comes with universal quantifiers $\forall x\, P$ to say that P is true for all values of variable x. It also supports existential quantifiers $\exists x\, P$ to say that P is true for at least one value of variable x.

So okay, this first-order logic with arithmetic seems much more useful than propositional logic to make sense of contracts. But do they provide us with all we need to understand a program contract?

Given one particular value for each of the variables, first-order logic formulas are either true or false (much like, in an interpretation I, propositional logic formulas are either true or false). But what makes programs most interesting is that the truth of such a first-order logic formula used in, say, a postcondition will depend on the current state of the program. The postcondition may even change its truth-value. It might be false in the initial state but will become true in the final state of the program. In fact, that's often how it works in programs/ The gcd program does not start with the correct answer in the result variable `a` but merely ends up with the correct gcd answer in `a` at the end of the loop.

First-order logic is not very good at that. Just like propositional logic, first-order logic is a static logic, so its formulas will be either true or false in an state/interpretation. But they do not provide any ways of referring to what has been true before a program ran or what will be true after the program did. It is this dynamics, so behavior of change, that calls for *dynamic logic*.

Dynamic logic crucially provides modalities that talk about what is true after a program runs. The modal formula $[\alpha]P$ expresses that the formula P is true after all runs of program α. That formula $[\alpha]P$ is true in a state if it is indeed the case that all states reached after running program α satisfy the postcondition P. We can use it to rigor-

ously express what contracts mean. But let's first officially introduce the language of dynamic logic.

3 Dynamic Logic

3.1 Syntax

Definition 1 (Terms)**.** Terms are defined by the following grammar (θ, η are terms, x a variable, q a number literal and f a function symbol):

$$\theta, \eta \ ::= \ x \mid q \mid f(\theta_1, \ldots, \theta_n) \mid \theta + \eta \mid \theta \cdot \eta$$

Definition 2 (Nondeterministic programs)**.** Programs are defined by the following grammar (α, β are programs, x a variable, θ a term possibly containing x, and Q a formula of first-order logic of real arithmetic):

$$\alpha, \beta \ ::= \ x := \theta \mid ?Q \mid \alpha; \beta \mid \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta \mid \texttt{while}(Q)\, \alpha \mid \alpha \cup \beta \mid \alpha^*$$

Definition 3 (DL formula)**.** The *formulas of dynamic logic* (DL) are defined by the grammar (where ϕ, ψ are DL formulas, θ_1, θ_2 terms, p a predicate symbol, x a variable, α a program):

$$\phi, \psi \ ::= \ \theta_1 = \theta_2 \mid \theta_1 \geq \theta_2 \mid p(\theta_1, \ldots, \theta_n) \mid \neg\phi \mid \phi \wedge \psi \mid \forall x\, \phi \mid \exists x\, \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

3.2 Dynamic Semantics

A state is a mapping from variables to $\mathbb{R}$. The set of states is denoted $\mathcal{S}$. Let ν_x^r denote the state that agrees with state ν except for the interpretation of variable x, which is changed to $r \in \mathbb{R}$.

For later use, additional function symbols f and predicate symbols p are already allowed in the logic. For now, consider unary minus $f(x)$ with the fixed interpretation as $I(f)(d) = -d$ and arithmetic comparison operators $p(x, y)$ with the fixed interpretation $(c, d) \in I(p)$ iff $c < d$. But this device of extending DL with additional function symbols and predicate symbols will prove useful later.

Definition 4 (Semantics of terms)**.** The *semantics of a term* θ in a state $\nu \in \mathcal{S}$ is its value. It is defined inductively as follows

- $I\nu[\![x]\!] = \nu(x)$ for variable x
- $I\nu[\![q]\!] = q$ for number literals q
- $I\nu[\![f(\theta_1, \ldots, \theta_k)]\!] = I(f)\big(I\nu[\![\theta_1]\!], \ldots, I\nu[\![\theta_k]\!]\big)$
- $I\nu[\![\theta + \eta]\!] = I\nu[\![\theta]\!] + I\nu[\![\eta]\!]$
- $I\nu[\![\theta \cdot \eta]\!] = I\nu[\![\theta]\!] \cdot I\nu[\![\eta]\!]$

Definition 5 (Transition semantics of programs)**.** Each program α is interpreted semantically as a binary reachability relation $I[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ over states, defined inductively by

- $I[\![x := \theta]\!] = \{(\nu, \omega) \ : \ \omega = \nu \text{ except that } I\omega[\![x]\!] = I\nu[\![\theta]\!]\}$
- $I[\![?Q]\!] = \{(\nu, \nu) \ : \ \nu \in I[\![Q]\!] \text{ i.e. } \nu \in I[\![Q]\!]\}$
- $I[\![\alpha; \beta]\!] = I[\![\alpha]\!] \circ I[\![\beta]\!] = \{(\nu, \omega) : (\nu, \mu) \in I[\![\alpha]\!], (\mu, \omega) \in I[\![\beta]\!]\}$
- $I[\![\texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta]\!] = I[\![Q]\!] \circ I[\![\alpha]\!] \cup I[\![Q]\!]^{\complement} \circ I[\![\beta]\!] =$
  $\{(\nu, \omega) : (\nu, \omega) \in I[\![\alpha]\!] \text{ and } \nu \in I[\![Q]\!]\} \cup \{(\nu, \omega) : (\nu, \omega) \in I[\![\beta]\!] \text{ and } \nu \notin I[\![Q]\!]\}$
- $I[\![\alpha^*]\!] = \big(I[\![\alpha]\!]\big)^* = \bigcup_{n \in \mathbb{N}} I[\![\alpha^n]\!] = \big\{(\nu, \omega) \ : \ \text{there are an } n \text{ and states } \mu_0 = \nu, \mu_1, \mu_2, \ldots, \mu_n = \omega \text{ such that } (\mu_i, \mu_{i+1}) \in I[\![\alpha]\!] \text{ for all } 0 \leq i < n\big\}$
  with $\alpha^{n+1} \equiv \alpha^n; \alpha$ and $\alpha^0 \equiv ?true$ and where ρ^* is the reflexive transitive closure of a relation $\rho \subseteq \mathcal{S} \times \mathcal{S}$.
- $I[\![\texttt{while}(Q)\, \alpha]\!] = \big\{(\nu, \omega) \ : \ $ there are an n and states $\mu_0 = \nu, \mu_1, \mu_2, \ldots, \mu_n = \omega$ such that for all $0 \leq i < n$: ① the loop condition is true $\mu_i \in I[\![Q]\!]$ and ② from state μ_i is state μ_{i+1} reachable by running α so $(\mu_i, \mu_{i+1}) \in I[\![\alpha]\!]$ and ③ the loop condition is false $\mu_n \notin I[\![Q]\!]$ in the end$\big\} = \big(I[\![Q]\!] \circ I[\![\alpha]\!]\big)^* \circ I[\![Q]\!]^{\complement}$

Definition 6 (DL semantics)**.** The *semantics of a DL formula* ϕ, for each interpretation I with a corresponding set of states $\mathcal{S}$, is the subset $I[\![\phi]\!] \subseteq \mathcal{S}$ of states in which ϕ is true. It is defined inductively as follows

1. $I[\![p(\theta_1, \ldots, \theta_k)]\!] = \{I, \nu \in \mathcal{S} \ : \ (I\nu[\![\theta_1]\!], \ldots, I\nu[\![\theta_k]\!]) \in I(p)\}$
2. $I[\![\theta_1 \geq \theta_2]\!] = \{I, \nu \in \mathcal{S} \ : \ I\nu[\![\theta_1]\!] \geq I\nu[\![\theta_2]\!]\}$
3. $I[\![\neg\phi]\!] = (I[\![\phi]\!])^{\complement}$
4. $I[\![\phi \wedge \psi]\!] = I[\![\phi]\!] \cap I[\![\psi]\!]$
5. $I[\![\exists x\, \phi]\!] = \{I, \nu \in \mathcal{S} \ : \ \nu_x^r \in I[\![\phi]\!] \text{ for some } r \in \mathbb{R}\}$
6. $I[\![\langle\alpha\rangle\phi]\!] = I[\![\alpha]\!] \circ I[\![\phi]\!] = \{\nu \ : \ \omega \in I[\![\phi]\!] \text{ for some } \omega \text{ such that } (\nu, \omega) \in I[\![\alpha]\!]\}$
7. $I[\![[\alpha]\phi]\!] = I[\![\neg\langle\alpha\rangle\neg\phi]\!] = (I[\![\alpha]\!] \circ (I[\![\phi]\!])^{\complement})^{\complement} =$
   $\{\nu \ : \ \omega \in I[\![\phi]\!] \text{ for all } \omega \text{ such that } (\nu, \omega) \in I[\![\alpha]\!]\}$

A DL formula ϕ is *valid in* I, written $I \models \phi$, iff $I[\![\phi]\!] = \mathcal{S}$. Formula ϕ is *valid*, $\models \phi$, iff $I \models \phi$ for all interpretations I.

4 Contracts in Dynamic Logic

Requiring `x>=y` in the following gcd function is just for sake of simplicity.

```
//@requires x>=y && y>0;
//@ensures x mod a = 0 && y mod a = 0;
//@ensures \forall s (s>0 && x mod s = 0 && y mod s = 0 -> a mod s = 0);
{
    a := x;
    b := y;
    while (b!=0)
    {
        t := a mod b;
        a := b;
        b := t;
    }
}
```

What is not just for the sake of simplicity is the need in the second postcondition to not just say that the resulting value `a` divides the two inputs `x` and `a` but that it also is the greatest such divisor. So for all other divisors `s`, `a` is greater or equal `s` or, in fact, even the other divisor `s` divides the greatest common divisor resulting from `a` in the end.

Since the box modality in $[\alpha]P$ expresses that formula P holds after all runs of program α, we can use it directly to express the `@ensures` postconditions of the gcd program. Let `gcd` be the gcd program from above and `postdiv` as well as `postgrt` its two conditions from the two `@ensures` clauses:

$$\begin{aligned}
\texttt{gcd} &\equiv a := x; b := y; \texttt{while}(p \neq 0)\,\{t := a \mod b; a := b; b := t\}\\
\texttt{postdiv} &\equiv x \mod a = 0 \land y \mod a = 0\\
\texttt{postgrt} &\equiv \forall s\,(s > 0 \land x \mod s = 0 \land y \mod s = 0 \rightarrow a \mod s = 0)
\end{aligned}$$

With these abbreviations and the box modalities of dynamic logic it suddenly is a piece of cake to express that the first `@ensures` postcondition holds after all program runs:

$$[\texttt{gcd}]\texttt{postdiv}$$

It is also really easy to express the second `@ensures` postcondition:

$$[\texttt{gcd}]\texttt{postgrt}$$

Well, maybe it would have been better if we had expressed both `@ensures` clauses at once. How do we do that again?

Well, if we want to say that both postconditions are true after running `gcd` and the logic is closed under all operators including conjunction, we can simply use the conjunction of both formulas for the job:

$$[\texttt{gcd}]\texttt{postdiv} \land [\texttt{gcd}]\texttt{postgrt}$$

This formula means that `postdiv` is true after all runs of `gcd` and that `postgrt` is also true after all runs of `gcd`. Maybe it would have been better to simultaneously state both postconditions at once? Well, that would have been the formula

$$[\texttt{gcd}](\texttt{postdiv} \land \texttt{postgrt})$$

which says that the conjunction of `postdiv` and `postgrt` is true after all runs of `gcd`. Which formula is better now?

Well that depends. For one thing, both are perfectly equivalent, because that is what it means for a formula to be true after all runs of a program. That means the following biimplication in dynamic logic is valid so true in all states:

$$[\texttt{gcd}]\texttt{postdiv} \land [\texttt{gcd}]\texttt{postgrt} \leftrightarrow [\texttt{gcd}](\texttt{postdiv} \land \texttt{postgrt})$$

Now that we have worried so much about how to state the postcondition in a lot of different equivalent ways, the question is whether the following formula or any of its equivalent forms is actually always true?

$$[\texttt{gcd}](\texttt{postdiv} \land \texttt{postgrt})$$

Well, of course not, because we forgot to take the program's precondition from the `@requires` clause into account, which the program assumes to hold in the initial state. But that is really easy in logic because we can simply use implication for the job of expressing such an assumption:

$$x \geq y \land y > 0 \rightarrow [\texttt{gcd}](\texttt{postdiv} \land \texttt{postgrt})$$

And, indeed, this formula will now turn out to be valid, so true in all states. In particular, in every initial state it is true that if that initial state satisfies the `@requires` preconditions $x \geq y \land y > 0$, then all states reached after running the `gcd` program will satisfy the `@ensures` postconditions `postdiv` $\land$ `postgrt`. If the initial state does not satisfy the precondition, then the implication does not claim anything, because it makes an assumption about the initial state that apparently is not presently met.

Expressing the contract for the swapping program as a formula in dynamic logic yields:

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

Notice how these dynamic logic formulas make it absolutely precise what the meaning of a program contract is. Well, at least after we define the semantics of dynamic logic formulas, which is our next challenge.

References