# Exercises 4: Dynamic Logic

TECHNISCHE UNIVERSITÄT DARMSTADT

**The solutions to the exercises will be discussed on Thursday, 11th June.**

## Problem 1 Interpreting Dynamic Logic Formulas

Let $p$ denote a Java program.

Formalize the following statements using DL. You are allowed to introduce new (uninterpreted) function symbols or program variables. When using an additional symbol state its kind and type. Let $p$ denote an arbitrary Java program.

```
\programVariables {
  int i, j;
  int[] a;
  boolean b;
}
```

a) program $p$ terminates and in its final state program variable `i` has value 0 and program variable `j` has not changed its value.

**Solution:** $c \doteq \mathtt{j} \rightarrow \langle \mathtt{p} \rangle (\mathtt{i} \doteq 0 \wedge \mathtt{j} = c)$ ($c$ is a rigid constant symbol of type `int`)

b) if $p$ is executed in a state where program variable `i` has a non-negative value then $p$ terminates and in its final state the value of `j` has been increased by the initial value of `i`.

**Solution:** $(\mathtt{i} \geq 0 \wedge \mathtt{i} = c \wedge \mathtt{j} = d) \rightarrow \langle \mathtt{p} \rangle (\mathtt{j} \doteq c + d)$ ($c, d$ are rigid constant symbols of type `int`)

c) program $p$ terminates and might only change the value of `o.f` (where o is a program variable of a class type `A` and `f` a unique constant of type `Field`.

1. $p$ must not create new objects **Solution:**
   $\{\mathtt{heapAtPre} := \mathtt{heap}\} \langle \mathtt{p} \rangle \forall\, \mathtt{Object}\ u; \forall\, \mathtt{Field}\ g; ($
   $\mathtt{singleton}(u, g) \subseteq \mathtt{singleton}(o, f)$
   $\vee\ \mathtt{select}(heap, u, g) \doteq \mathtt{select}(heapAtPre, u, g))$
   (`heapAtPre` is a program variable of type `Heap`)
2. $p$ might create new objects
   $\{\mathtt{heapAtPre} := \mathtt{heap}\} \langle \mathtt{p} \rangle \forall\, \mathtt{Object}\ u; \forall\, \mathtt{Field}\ g; ($
   $\mathtt{singleton}(u, g) \subseteq \mathtt{singleton}(o, f)$
   $\vee\ \mathtt{select}(\mathtt{heap}, u, g) \doteq \mathtt{select}(\mathtt{heapAtPre}, u, g)$
   $\vee\ \mathtt{select}(\mathtt{heapAtPre}, u, <\mathtt{created}>) = \mathtt{FALSE})$

d) if executed in an initial state where program variable `i` has non-negative value, then program $p$ terminates in a state where `j` has the same value as `i` and the only heap locations it is allowed to change are the array elements `a[0..i]`.

**Solution:** (similar to others above) locset for array range: $\mathtt{arrayRange}(\mathtt{heap}, \mathtt{a}, 0, \mathtt{i})$
$\mathtt{i} \geq 0 \rightarrow \{\mathtt{heapAtPre} := \mathtt{heap}\} \langle \mathtt{p} \rangle \forall\, \mathtt{Object}\ u; \forall\, \mathtt{Field}\ g; ($
$\mathtt{singleton}(u, g) \subseteq \mathtt{arrayRange}(\mathtt{a}, 0, \mathtt{i})$
$\vee\ \mathtt{select}(\mathtt{heap}, u, g) \doteq \mathtt{select}(\mathtt{heapAtPre}, u, g)$
$\vee\ \mathtt{select}(\mathtt{heapAtPre}, u, <\mathtt{created}>) = \mathtt{FALSE}) \wedge \mathtt{j} \doteq \mathtt{i}$

## Problem 2 Decomposition of Complex Terms

There are many dynamic logic calculus rules integrated in the KeY system. However, to keep the set of rules as small as possible, some complex terms are decomposed into simple terms that already have rules in KeY. For instance, KeY supports rules for variable declaration, simple assignment and addition.

$$\text{assign}\ \frac{\Gamma \Rightarrow \{\mathtt{x} := t\}\langle \mathtt{rest}\rangle \phi, \Delta}{\Gamma \Rightarrow \langle \mathtt{x = t; \ rest}\rangle \phi, \Delta}$$

$$\text{assignAddition}\ \frac{\Gamma \Rightarrow \{\mathtt{x} := s + t\}\langle \mathtt{rest}\rangle \phi, \Delta}{\Gamma \Rightarrow \langle \mathtt{x = s + t; \ rest}\rangle \phi, \Delta}$$

Design rules to decompose the following complex terms, such that the resulting terms are supported in KeY.

1. `y = ++ x;`

   **Solution:**

$$\text{assignPreIncrement}\ \frac{\Gamma \Rightarrow \langle \mathtt{x = x + 1; \ y = x; \ rest}\rangle \phi, \Delta}{\Gamma \Rightarrow \langle \mathtt{y = ++x; \ rest}\rangle \phi, \Delta}$$

2. `y = x ++;`

$$\text{assignPostIncrement}\ \frac{\Gamma \Rightarrow \langle \mathtt{t = x; \ x = x + 1; \ y = t; \ rest}\rangle \phi, \Delta}{\Gamma \Rightarrow \langle \mathtt{y = x++; \ rest}\rangle \phi, \Delta}$$

What changes if `x` is an expression (attribute access) rather than a program variable?

## Problem 3 Playing around with the prover

The downloaded archive contains a selection of DL problems to be proven with KeY. Play around with KeY and get to know how KeY works to verify dynamic logic problems. If some programs cannot be verified, think about the reasons and fix them.

Highlighted exercises are: 1, 3 and 8.

## Problem 4 KeY Quicktour

To get familiar with KeY. Read and play through the quicktour

`http://www.key-project.org/download/quicktour/quicktour-2.0.zip`

## Problem 5

Try to verify that method `replaceIfGreater` of class `ArrayHelper` in directory `verify` ensures its postconditions. Explain how you can read from the open goal(s) and the path to an open goal, what is going on. Then fix the bug in the specification or implementation and verify the method again.

## Problem 6

The directory `verify` contains a few classes. One of them is the class `Person`. Imagine the class belongs to a database of the authority responsible for issuing driver's licenses. Further, each person in this database has already successfully passed all tests. So each person already has a driver's license, but some are still not old enough to actually be allowed to drive. A person must be at least 18 years old to be allowed to drive. The allowance is issued by activating the license of a person.

a) Specify the normal and exceptional behavior of method `activateLicense` of class `Person`.

b) Verify that both contracts ensure their respective postcondition and preserve all invariants of the class `Person`.

If one of the above proofs cannot be closed, try to find out why by looking at the proof. To which actual situation(s) do the open goals belong? What kind of hints can you read from the sequent of an open goal?

## Problem 7

In the directory `contracts` you find a class `Coll.java` and a class `CollClient.java`. The latter class contains a method `containsZero` using the `contains` method of class `Coll`. Specify the method in `Coll` as precisely as possible and verify that the method satisfies its specification. Then prove that method `containsZero` of `CollClient` is correct. Perform this proof twice:

a) using only method inlining/expand

b) using method contract (only for the invocation of `contains`)

and apply the method invocation rules manually. What can you observe concerning the proof size?

Assume you would change the implementation of `contains` in `Coll` in such a way that the for loop is exited as soon as the element has been found and eliminate the need for the local variable `found` completely. Which of the previous proofs have to be redone? Justify your answer.