

# Formal Methods for Software Development

## Reasoning about Programs with Dynamic Logic

Wolfgang Ahrendt, Raúl Pardo

5 October 2017

# Part I

**Where are we?**

# Where Are We?

**before** specification of JAVA programs with JML

**now** dynamic logic (DL) for reasoning about JAVA programs

**after that** generating DL from JML+JAVA

+ verifying the resulting proof obligations

# Motivation

Consider the method

```
public void doubleContent(int[] a) {  
    int i = 0;  
    while (i < a.length) {  
        a[i] = a[i] * 2;  
        i++;  
    }  
}
```

We want a **logic/calculus** allowing to **express/prove** properties like, e.g.:

*If*  $a \neq \text{null}$

*then* `doubleContent` terminates normally

*and* afterwards all elements of `a` are twice the old value

# Dynamic Logic (Preview)

One such logic is **dynamic logic** (DL)

The above statement can be expressed in DL as follows:  
(assuming a suitable signature)

$$\begin{aligned} & a \neq \text{null} \\ & \wedge a \neq \text{old\_a} \\ & \wedge \forall \text{int } i; ((0 \leq i \wedge i < \text{a.length}) \rightarrow a[i] = \text{old\_a}[i]) \\ \rightarrow & \langle \text{doubleContent}(a); \rangle \\ & \forall \text{int } i; ((0 \leq i \wedge i < \text{a.length}) \rightarrow a[i] = 2 * \text{old\_a}[i]) \end{aligned}$$

## Observations

- ▶ DL combines first-order logic (FOL) with programs
- ▶ Theory of DL extends theory of FOL

introducing **dynamic logic** for JAVA

- ▶ short recap first-order logic (FOL)
- ▶ dynamic logic = extending FOL with
  - ▶ **dynamic interpretations**
  - ▶ **programs** to describe state change

# Repetition: First-Order Logic

## Signature

A first-order signature  $\Sigma$  consists of

- ▶ a set  $T_\Sigma$  of type symbols
- ▶ a set  $F_\Sigma$  of function symbols
- ▶ a set  $P_\Sigma$  of predicate symbols

## Type Declarations

- ▶  $\tau \ x$ ;                                    ‘variable  $x$  has type  $\tau$ ’
- ▶  $p(\tau_1, \dots, \tau_r)$ ;           ‘predicate  $p$  has argument types  $\tau_1, \dots, \tau_r$ ’
- ▶  $\tau \ f(\tau_1, \dots, \tau_r)$ ;   ‘function  $f$  has argument types  $\tau_1, \dots, \tau_r$   
and result type  $\tau$ ’

# Recap: First-Order States

## Definition (First-Order State)

Let  $\mathcal{D}$  be a domain with typing function  $\delta$ .

For each  $f$  be declared as  $\tau f(\tau_1, \dots, \tau_r)$ ;

and each  $p$  be declared as  $p(\tau_1, \dots, \tau_r)$ ;

$\mathcal{I}(f)$  is a mapping  $\mathcal{I}(f) : \mathcal{D}^{\tau_1} \times \dots \times \mathcal{D}^{\tau_r} \rightarrow \mathcal{D}^{\tau}$

$\mathcal{I}(p)$  is a set  $\mathcal{I}(p) \subseteq \mathcal{D}^{\tau_1} \times \dots \times \mathcal{D}^{\tau_r}$

Then  $\mathcal{S} = (\mathcal{D}, \delta, \mathcal{I})$  is a **first-order state**



## Part II

# Towards Dynamic Logic

## Reasoning about Java programs requires extensions of FOL

- ▶ JAVA type hierarchy
- ▶ JAVA program variables
- ▶ JAVA heap for reference types (next lecture)

# Type Hierarchy

## Definition (Type Hierarchy)

- ▶  $T_\Sigma$  is set of **types**
- ▶ **Subtype** relation  $\sqsubseteq \subseteq T_\Sigma \times T_\Sigma$  with top element  $\top$ 
  - ▶  $\tau \sqsubseteq \top$  for all  $\tau \in T_\Sigma$

## Example (A Minimal Type Hierarchy)

$$T_\Sigma = \{\top\}$$

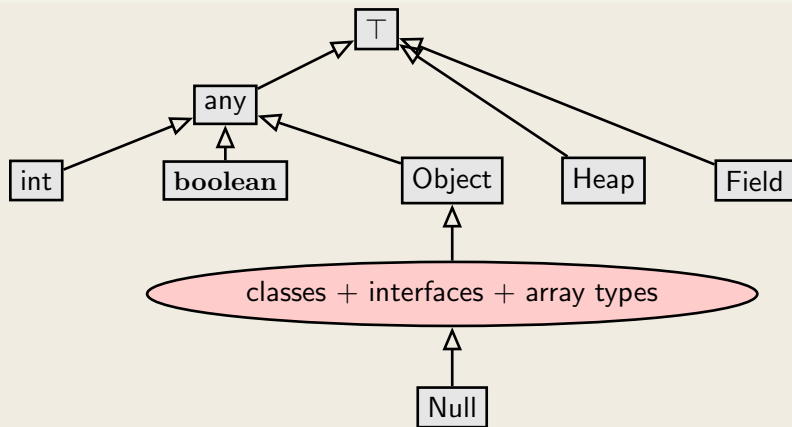
All signature symbols have same type  $\top$

## Example (Type Hierarchy for Java)

(see next slide)

# Modelling Java in FOL: Fixing a Type Hierarchy

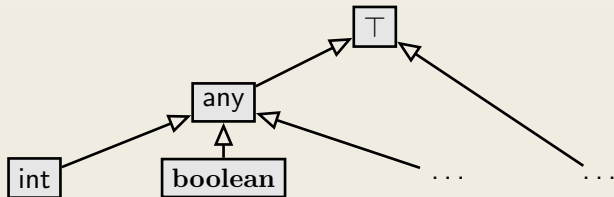
Signature based on Java's type hierarchy (sketch)



Each interface and class in API and in target program becomes type with appropriate subtype relation

# Subset of Types

Signature based on Java's type hierarchy



**int** and **boolean** are the only types for today.  
Class, interfaces, arrays: next lecture.

# Modelling Dynamic Properties

Only static properties expressible in typed FOL, e.g.,

- ▶ Values of fields in a certain range
- ▶ Invariant of a class implies invariant of its interface

Considers only one program state at a time

**Goal:** Express behavior of a program, e.g.:

**If** method `setAge` is called on an object `o` of type `Person`  
**and** the method argument `newAge` is positive  
**then afterwards** field `age` has same value as `newAge`

# Requirements

## Requirements for a logic to reason about programs

- ▶ Can relate different program states, i.e., **before** and **after** execution, within a single formula
- ▶ Program variables are represented by **constant symbols**, whose value **depend** on program **state**

Dynamic Logic meets the above requirements

## (JAVA) Dynamic Logic

### Typed FOL

- ▶ + programs  $p$
- ▶ + modalities  $\langle p \rangle \phi$ ,  $[p] \phi$  ( $p$  program,  $\phi$  DL formula)
- ▶ + ... (later)

### An Example

$$i > 5 \rightarrow [i = i + 10;] i > 15$$

### Meaning?

If **program variable**  $i$  is greater than 5 in current state, then **after** executing the JAVA statement “ $i = i + 10;$ ”,  $i$  is greater than 15



# Program Variables

Dynamic Logic = Typed FOL + ...

$$i > 5 \rightarrow [i = i + 10;]i > 15$$

Program variable  $i$  refers to different values before and after execution

- ▶ Program variables such as  $i$  are state-dependent constant symbols
- ▶ Value of state-dependent symbols changeable by a program

Three words one meaning: state-dependent, non-rigid, flexible

# Rigid versus Flexible Symbols

**Signature** of program logic defined as in FOL, but in addition, there are **program variables**

## Rigid versus Flexible

- ▶ **Rigid** symbols, meaning insensitive to program states
  - ▶ First-order variables (aka **logical variables**)
  - ▶ Built-in functions and predicates such as  $0, 1, \dots, +, *, \dots, <, \dots$
- ▶ **Flexible** (or **non-rigid**) symbols, meaning depends on state. Capture side effects on state during program execution
  - ▶ **Program variables** are flexible

Any term containing at least one flexible symbol is called flexible

# Signature of Dynamic Logic

## Definition (Dynamic Logic Signature)

$$\Sigma = (P_\Sigma, F_\Sigma, PV_\Sigma, \alpha_\Sigma), \quad F_\Sigma \cap PV_\Sigma = \emptyset$$

(Rigid) **Predicate** Symbols  $P_\Sigma = \{>, >=, \dots\}$

(Rigid) **Function** Symbols  $F_\Sigma = \{+, -, *, 0, 1, \dots\}$

**Flexible Program variables** e.g.  $PV_\Sigma = \{i, j, \text{ready}, \dots\}$

Standard typing of JAVA symbols: `boolean TRUE; <(int,int); ...`

## Dynamic Logic Signature - KeY input file

```
\sorts {  
  // only additional sorts (int, boolean, any predefined)  
}  
\functions {  
  // only additional rigid functions  
  // (arithmetic functions like +,- etc., predefined)  
}  
\predicates { /* same as for functions */ }  
  
\programVariables { // flexible  
  int i, j;  
  boolean ready;  
}
```

Empty sections can be left out

# Again: Two Kinds of Variables

Rigid:

## Definition (First-Order/Logical Variables)

Typed **logical variables** (**rigid**), declared locally in **quantifiers** as  $\exists x;$   
They may not occur in programs!

Flexible:

## Program Variables

- ▶ Are **not** FO variables
- ▶ **Cannot** be quantified
- ▶ May occur in programs (and formulas)

# Dynamic Logic Programs

Dynamic Logic = Typed FOL + programs ...

Programs here: any legal sequence of JAVA statements.

## Example

Signature for  $PV_{\Sigma}$ : int r; int i; int n;

Signature for  $F_{\Sigma}$ : int 0; int +(int,int); int -(int,int);

Signature for  $P_{\Sigma}$ : <(int,int);

```
i=0;
r=0;
while (i<n) {
    i=i+1;
    r=r+i;
}
r=r+r-n;
```

Which value does the program compute in r?

# Relating Program States: Modalities

DL extends FOL with two additional operators:

- ▶  $\langle p \rangle \phi$  (diamond)
- ▶  $[p] \phi$  (box)

with  $p$  a program,  $\phi$  another DL formula

## Intuitive Meaning

- ▶  $\langle p \rangle \phi$ :  $p$  terminates **and** formula  $\phi$  holds in final state  
(total correctness)
- ▶  $[p] \phi$ : **If**  $p$  terminates **then** formula  $\phi$  holds in final state  
(partial correctness)

Attention: JAVA programs are deterministic, i.e., **if** a JAVA program terminates then exactly **one** state is reached from a given initial state.

# Dynamic Logic - Examples

Let  $i$ ,  $j$ ,  $old\_i$ ,  $old\_j$  denote program variables.  
Give the meaning in natural language:

1.  $i = old\_i \rightarrow \langle i = i + 1; \rangle i > old\_i$

If  $i = i + 1$ ; is executed in a state where  $i$  and  $old\_i$  have the same value, then the program terminates and in its final state the value of  $i$  is greater than the value of  $old\_i$ .

2.  $i = old\_i \rightarrow [\text{while}(\text{true})\{i = old\_i - 1;\}] i > old\_i$

If the program is executed in a state where  $i$  and  $old\_i$  have the same value and if the program terminates then in its final state the value of  $i$  is greater than the value of  $old\_i$ .

3.  $\forall x. (\langle prog_1 \rangle i = x \leftrightarrow \langle prog_2 \rangle i = x)$

$prog_1$  and  $prog_2$  are equivalent concerning termination and the final value of  $i$ .



# Dynamic Logic: KeY Input File

```
\programVariables { // Declares global program variables
  int i;
  int old_i;
}
```

```
\problem { // The problem to verify is stated here
  i = old_i -> \<{ i = i + 1; }\> i > old_i
}
```

## Visibility

- ▶ Program variables declared globally can be accessed anywhere
- ▶ Program variables declared inside a modality only visible therein.  
E.g., in “ $pre \rightarrow \langle \mathbf{int} \ j; \ p \rangle post$ ”,  $j$  not visible in  $post$

# Dynamic Logic Formulas

## Definition (Dynamic Logic Formulas (DL Formulas))

- ▶ Each FOL formula is a DL formula
  - ▶ If  $p$  is a program and  $\phi$  a DL formula, then  $\left\{ \begin{array}{l} \langle p \rangle \phi \\ [p] \phi \end{array} \right\}$  is a DL formula
  - ▶ DL formulas closed under FOL quantifiers and connectives
- 
- ▶ Program variables are **flexible constants**: never bound in quantifiers
  - ▶ Program variables need not be declared or initialized in program
  - ▶ Programs contain no logical variables
  - ▶ Modalities can be arbitrarily nested, e.g.,  $\langle p \rangle [q] \phi$

## Example (Well-formed? If yes, under which signature?)

- ▶  $\forall \text{int } y; ((\langle x = 2; \rangle x = y) \leftrightarrow (\langle x = 1; x++; \rangle x = y))$   
Well-formed if  $PV_{\Sigma}$  contains  $\text{int } x$ ;
- ▶  $\exists \text{int } x; [x = 1;](x = 1)$   
Not well-formed, because logical variable occurs in program
- ▶  $\langle x = 1; \rangle ([\text{while } (\text{true}) \{ \}] \text{false})$   
Well-formed if  $PV_{\Sigma}$  contains  $\text{int } x$ ;  
program formulas can be nested

# Dynamic Logic Semantics: States

First-order state can be considered as **program state**

- ▶ Interpretation of (flexible) program variables can vary from state to state
- ▶ Interpretation of **rigid** symbols is the same in all states (e.g., built-in functions and predicates)

## Program states as first-order states

We identify **first-order state**  $\mathcal{S} = (\mathcal{D}, \delta, \mathcal{I})$  with **program state**.

- ▶ Interpretation  $\mathcal{I}$  only changes on program variables.  
⇒ Enough to record values of variables  $\in PV_{\Sigma}$
- ▶ Set of all states  $\mathcal{S}$  is called *States*

# Kripke Structure

## Definition (Kripke Structure)

Kripke structure or Labelled transition system  $K = (States, \rho)$

- ▶ States  $\mathcal{S} = (\mathcal{D}, \delta, \mathcal{I}) \in States$
- ▶ Transition relation  $\rho : Program \rightarrow (States \rightarrow States)$

$$\rho(p)(\mathcal{S}_1) = \mathcal{S}_2$$

iff.

program  $p$  executed in state  $\mathcal{S}_1$  terminates **and** its final state is  $\mathcal{S}_2$ ,  
**otherwise** undefined.

- ▶  $\rho$  is the **semantics** of programs  $\in Program$
- ▶  $\rho(p)(\mathcal{S})$  can be undefined (' $\rightarrow$ '):  $p$  may **not terminate** when started in  $\mathcal{S}$
- ▶ JAVA programs are **deterministic** (unlike PROMELA):  $\rho(p)$  is a function (at most one value)

# Semantic Evaluation of Program Formulas

## Definition (Validity Relation for Program Formulas)

- ▶  $\mathcal{S} \models \langle p \rangle \phi$  iff  $\rho(p)(\mathcal{S})$  is defined and  $\rho(p)(\mathcal{S}) \models \phi$   
( $p$  terminates and  $\phi$  is true in the final state after execution)
- ▶  $s \models [p]\phi$  iff  $\rho(p)(\mathcal{S}) \models \phi$  whenever  $\rho(p)(\mathcal{S})$  is defined  
(If  $p$  terminates then  $\phi$  is true in the final state after execution)

A DL formula  $\phi$  is **valid** iff  $\mathcal{S} \models \phi$  for all states  $\mathcal{S}$ .

- ▶ **Duality:**  $\langle p \rangle \phi$  iff  $\neg[p]\neg\phi$   
Exercise: justify this with help of semantic definitions
- ▶ **Implication:** if  $\langle p \rangle \phi$  then  $[p]\phi$   
Total correctness implies partial correctness
  - ▶ converse is false
  - ▶ holds only for deterministic programs

# More Examples

Meaning?

## Example

$$\forall \tau y; ((\langle p \rangle x = y) \leftrightarrow (\langle q \rangle x = y))$$

Programs  $p$  and  $q$  behave equivalently on variable  $\tau x$ .

## Example

$$\exists \tau y; (x = y \rightarrow \langle p \rangle \text{true})$$

Program  $p$  terminates if initial value of  $x$  is suitably chosen.

# Semantics of Programs

In labelled transition system  $K = (\text{States}, \rho)$ :  
 $\rho : \text{Program} \rightarrow (\text{States} \rightarrow \text{States})$  is **semantics** of programs  $p \in \text{Program}$

$\rho$  defined recursively on programs

## Example (Semantics of assignment)

States  $\mathcal{S}$  interpret program variables  $v$  with  $\mathcal{I}_{\mathcal{S}}(v)$

$$\rho(x=t;)(\mathcal{S}) = \mathcal{S}' \quad \text{where} \quad \mathcal{I}_{\mathcal{S}'}(y) := \begin{cases} \mathcal{I}_{\mathcal{S}}(y) & y \neq x \\ \text{val}_{\mathcal{S}}(t) & y = x \end{cases}$$

Very advanced task to define  $\rho$  for JAVA  $\Rightarrow$  Not done in this course  
**Next lecture**, we go directly to calculus for program formulas!



# Literature for this Lecture

**KeYbook** *W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors.*

**Deductive Software Verification - The KeY Book**

Vol 10001 of *LNCS*, Springer, 2016

(E-book at [link.springer.com](http://link.springer.com))

- ▶ *W. Ahrendt, S. Grebing, Using the KeY Prover*  
Chapter 15 in [KeYbook]

further reading:

- ▶ *B. Beckert, V. Klebanov, B. Weiß, Dynamic Logic for Java*  
Chapter 3 in [KeYbook]