Formal Methods for Software Development

Reasoning about Programs with Dynamic Logic

Wolfgang Ahrendt

8 October 2020

Part I

Where are we?

Where Are We?

before specification of JAVA programs with JML

now dynamic logic (DL) for resoning about JAVA programs

after that generating DL from JML+JAVA

\+ verifying the resulting proof obligations

Aim

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` $\neq$ `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:

$$
\begin{array}{l}
\quad \mathtt{a} \neq \mathtt{null} \\
\land\ \mathtt{a} \neq \mathtt{old\_a} \\
\land\ \forall \mathtt{int\ i}; ((0 \leq \mathtt{i} \land \mathtt{i} < \mathtt{a.length}) \to \mathtt{a[i]} = \mathtt{old\_a[i]}) \\
\to \langle \mathtt{doubleContent(a);} \rangle \\
\quad \forall \mathtt{int\ i}; ((0 \leq \mathtt{i} \land \mathtt{i} < \mathtt{a.length}) \to \mathtt{a[i]} = 2 * \mathtt{old\_a[i]})
\end{array}
$$

Observations

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

Connection to 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 Σ consists of

- a set T_Σ of type symbols
- a set F_Σ of function symbols
- a set P_Σ of predicate symbols

Type Declarations

- $\tau\ x$; 'variable x has type τ'
- $p(\tau_1, \ldots, \tau_r)$; 'predicate p has argument types $\tau_1, \ldots, \tau_r$'
- $\tau\ f(\tau_1, \ldots, \tau_r)$; 'function f has argument types $\tau_1, \ldots, \tau_r$ and result type τ'

First-Order States

Definition (First-Order State)

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

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

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

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

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

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

Part II

Towards Dynamic Logic

Towards Dynamic Logic

Reasoning about Java programs requires extensions of FOL

- JAVA type hierarchy
- JAVA program variables
- JAVA heap for reference types

Type Hierarchy

Definition (Type Hierarchy)

- T_Σ 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)

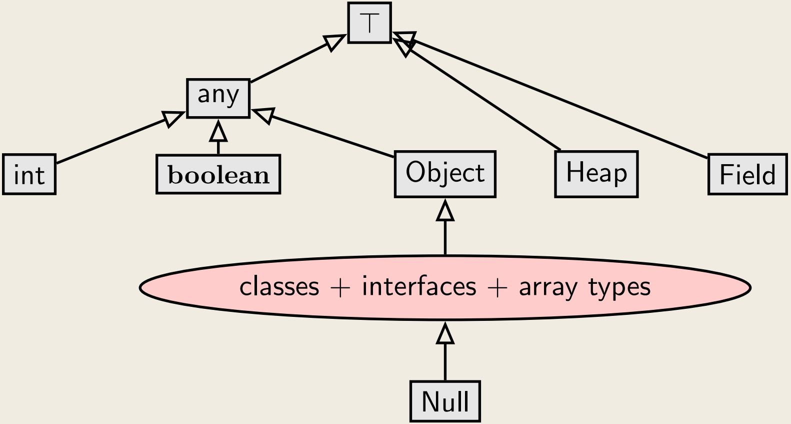
$T_\Sigma = \{\top\}$
All signature symbols have same type $\top$

Example (Type Hierarchy for Java)

(see next slide)

Modelling Java in FOL: Fixing a Type Hierarchy

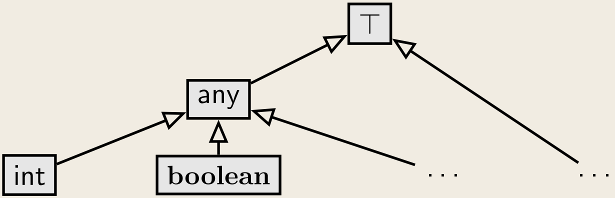
Signature based on Java's type hierarchy (sketch)



Each interface and class in libary and aplication becomes type
with appropriate subtype relation

Subset of Types

Signature based on Java's type hierarchy



We start with **int** and **boolean**, only.
Class, interfaces, arrays: later.

Modelling Dynamic Properties

Only static properties expressable 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

Dynamic Logic

(Java) Dynamic Logic

Typed FOL

- $+$ programs $\mathtt{p}$
- $+$ modalities $\langle \mathtt{p} \rangle \phi$, $[\mathtt{p}]\phi$ ($\mathtt{p}$ program, ϕ *DL* formula)
- $+ \ldots$ (later)

An Example

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

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 + ...

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

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: $\boxed{\text{state-dependent, non-rigid, flexible}}$

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,...,+,*,...,<,...`
- *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 $\quad P_\Sigma = \{>,\ >=, \ldots\}$
(Rigid) *Function* Symbols $\quad F_\Sigma = \{+,\ -,\ *,\ 0,\ 1, \ldots\}$
Flexible *Program variables* $\quad$ e.g. $PV_\Sigma = \{\mathtt{i}, \mathtt{j}, \mathtt{ready}, \ldots\}$

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 `T x;`
They must not occur in programs!

Flexible:

Program Variables

- Are *not* FO variables
- *Cannot* be quantified
- Can 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_Σ: **int** r; **int** i; **int** n;

Signature for F_Σ: **int** 0; **int** +(**int**,**int**); **int** -(**int**,**int**);

Signature for P_Σ: <(**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:

- $\langle \mathrm{p} \rangle \phi$ (diamond)
- $[\mathrm{p}]\phi$ (box)

with p a program, ϕ another DL formula

Intuitive Meaning

- $\langle \mathrm{p} \rangle \phi$: p terminates *and* formula ϕ holds in final state (total correctness)
- $[\mathrm{p}]\phi$: *If* p terminates *then* formula ϕ 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. $\texttt{i} = \texttt{old\_i} \rightarrow \langle \texttt{i = i + 1;} \rangle \texttt{i} > \texttt{old\_i}$

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

   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. $\forall\, x.\ (\ \langle prog_1 \rangle\ \texttt{i} = x\ \leftrightarrow\ \langle prog_2 \rangle\ \texttt{i} = x\ )$

   $prog_1$ 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* $\rightarrow$ $\langle$`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 ϕ a DL formula, then $\left\{\begin{matrix}\langle\mathrm{p}\rangle\phi \\ [\mathrm{p}]\phi\end{matrix}\right\}$ 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., $\langle\mathrm{p}\rangle[\mathrm{q}]\phi$

Dynamic Logic Formulas Cont'd

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

- $\forall\, \mathbf{int}\ y; (((\langle \texttt{x = 2;} \rangle \texttt{x} = y) \leftrightarrow (\langle \texttt{x = 1; x++;} \rangle \texttt{x} = y))$

  Well-formed if PV_Σ contains $\mathbf{int}\ \texttt{x;}$
- $\exists\, \mathbf{int}\ x; [x = 1;](x = 1)$

  Not well-formed, because logical variable occurs in program
- $\langle \texttt{x = 1;} \rangle ([\mathbf{while}\ (\mathbf{true})\ \texttt{\{\}}]\mathbf{false})$

  Well-formed if PV_Σ contains $\mathbf{int}\ \texttt{x;}$

  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.
  $\Rightarrow$ 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* $\rho : Program \to (States \rightharpoonup States)$

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

iff.

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

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

Semantic Evaluation of Program Formulas

Definition (Validity Relation for Program Formulas)

- $\mathcal{S} \models \langle \mathrm{p} \rangle \phi$ iff $\rho(\mathrm{p})(\mathcal{S})$ *is defined* and $\rho(\mathrm{p})(\mathcal{S}) \models \phi$

  (p *terminates* and ϕ is true in the final state after execution)
- $\mathcal{S} \models [\mathrm{p}]\phi$ iff $\rho(\mathrm{p})(\mathcal{S}) \models \phi$ whenever $\rho(\mathrm{p})(\mathcal{S})$ *is defined*

  (*If* p *terminates* then ϕ is true in the final state after execution)

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

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

More Examples

Meaning?

Example

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

Programs p and q behave equivalently on variable τ x.

Example

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

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

Semantics of Programs

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

ρ defined recursively on programs

Example (Semantics of assignment)

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

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

Very advanced task to define ρ for JAVA $\Rightarrow$ Not done in this course
We go directly to calculus for dynamic logic!

Dynamic Logic

(Java) Dynamic Logic

Typed FOL

- $+$ (Java) programs p
- $+$ modalities $\langle \mathrm{p} \rangle \phi$, $[\mathrm{p}]\phi$ (p program, ϕ *DL* formula)
- $+ \ldots$ (later)

Remark on Hoare Logic and DL

In Hoare logic $\{\mathrm{Pre}\}\ \mathrm{p}\ \{\mathrm{Post}\}$ (Pre, Post must be FOL)

In DL $\mathrm{Pre} \rightarrow [\mathrm{p}]\mathrm{Post}$ (Pre, Post any DL formula)

Proving DL Formulas

An Example

$$
\begin{array}{l}
\forall\ \mathtt{int}\ x; \\
\quad (x >= 0 \land \mathtt{n} = x \rightarrow \\
\qquad [\ \mathtt{i} = 0; \mathtt{r} = 0; \\
\qquad\ \ \mathtt{while}(\mathtt{i} < \mathtt{n})\{\mathtt{i} = \mathtt{i} + 1; \mathtt{r} = \mathtt{r} + \mathtt{i};\} \\
\qquad\ \ \mathtt{r} = \mathtt{r} + \mathtt{r} - \mathtt{n}; \\
\qquad]\mathtt{r} = x * x)
\end{array}
$$

How can we prove that the above formula is valid
(i.e. satisfied in all states)?

Semantics of DL Sequents

$\Gamma = \{\phi_1, \ldots, \phi_n\}$ and $\Delta = \{\psi_1, \ldots, \psi_m\}$ sets of DL formulas where all logical variables occur bound.

Recall: $\mathcal{S} \models (\Gamma \Longrightarrow \Delta)$ iff $\mathcal{S} \models (\phi_1 \wedge \cdots \wedge \phi_n) \rightarrow (\psi_1 \vee \cdots \vee \psi_m)$

Define semantics of DL sequents identical to semantics of FOL sequents

Definition (Validity of Sequents over DL Formulas)

A sequent $\Gamma \Longrightarrow \Delta$ over DL formulas is *valid* iff

$$\mathcal{S} \models (\Gamma \Longrightarrow \Delta) \text{ in } \textit{all} \text{ states } \mathcal{S}$$

Consequence for program variables

Initial value of program variables implicitly "universally quantified"

Symbolic Execution of Programs

Sequent calculus decomposes top-level operator in formula.
What is "top-level" in a sequential program `p; q; r;` ?

Symbolic Execution

- Follow the *natural control flow* when analysing a program
- Values of some variables unknown: *symbolic state representation*

Example

Compute the final state after termination of

```
x=x+y; y=x-y; x=x-y;
```

Symbolic Execution of Programs Cont'd

Typical form of DL formulas in symbolic execution

$$\langle \texttt{stmt;}\ \mathit{rest} \rangle \phi \qquad [\texttt{stmt;}\ \mathit{rest}]\phi$$

- Rules symbolically execute *first* statement ("active statement")
- Repeated application of such rules corresponds to *symbolic program execution*

Example (`symbolicExecution/simpleIf.key`, Demo, active statement only)

```
\programVariables {
 int x; int y; boolean b;
}
\problem {
 \<{ if (b) { x = 1; } else { x = 2; } y = 3; }\> y > x
}
```

Symbolic Execution of Programs Cont'd

Symbolic execution of conditional

$$\text{if}\ \frac{\Gamma, \mathtt{b} = \mathsf{TRUE} \Longrightarrow \langle \mathtt{p;}\ \ \mathit{rest} \rangle \phi, \Delta \qquad \Gamma, \mathtt{b} = \mathsf{FALSE} \Longrightarrow \langle \mathtt{q;}\ \ \mathit{rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathbf{if}\ \ \mathtt{(b)}\ \ \mathtt{\{}\ \mathtt{p}\ \mathtt{\}}\ \ \mathbf{else}\ \ \mathtt{\{}\ \mathtt{q}\ \mathtt{\}}\ \ \mathtt{;}\ \ \mathit{rest} \rangle \phi, \Delta}$$

Symbolic execution must consider all possible execution branches

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \langle \mathbf{if}\ \ \mathtt{(b)}\ \ \mathtt{\{}\ \mathtt{p;}\ \ \mathbf{while}\ \ \mathtt{(b)}\ \ \mathtt{p}\ \mathtt{\};}\ \ \mathit{rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathbf{while}\ \ \mathtt{(b)}\ \ \mathtt{\{p\};}\ \ \mathit{rest} \rangle \phi, \Delta}$$

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`)

- *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]