Formal Specification and Verification of Object-Oriented Programs

Dynamic Logic

TECHNISCHE UNIVERSITÄT DARMSTADT

Dynamic Logic

TECHNISCHE UNIVERSITÄT DARMSTADT

(JAVA) Dynamic Logic

Typed FOL (with JAVA-like type system)

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

Proving Validity of DL Formulas

An Example

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

Proving Validity of DL Formulas

TECHNISCHE UNIVERSITÄT DARMSTADT

An Example

$\forall$ **int** x;
  ($x \doteq$ `n` $\land\ x$ >= 0 $\rightarrow$
    [`i = 0; r = 0;`
     while `(i<n){ i = i + 1; r = r + i;}`
     `r=r+r-n;`
    ]`r` $\doteq x * x$)

Proving Validity of DL Formulas

An Example

$\forall$ **int** x;
  ($x \doteq \mathtt{n} \land x >= 0 \rightarrow$
    [`i = 0; r = 0;`
     while `(i<n){ i = i + 1; r = r + i;}`
     `r=r+r-n;`
    ]$\mathtt{r} \doteq x * x$)

How can we prove that the above formula is valid
(i.e., true in all program 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: $\mathcal{S} \models (\Gamma \Rightarrow \Delta)$ 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 $\Gamma \Rightarrow \Delta$ over DL formulas is valid iff

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

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: $\mathcal{S} \models (\Gamma \Rightarrow \Delta)$ iff $\mathcal{S} \models (\phi_1 \land \cdots \land \phi_n) \rightarrow (\psi_1 \lor \cdots \lor \psi_m)$

Define semantics of DL sequents identical to semantics of FOL sequents

Definition (Validity of Sequents over DL Formulas)

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

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

Consequence for program variables

Initial value of program variables implicitly "universally quantified"

Symbolic Execution of Programs

FOL sequent calculus decomposes top-level operator of a 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

Symbolic Execution of Programs

TECHNISCHE UNIVERSITÄT DARMSTADT

FOL sequent calculus decomposes top-level operator of a 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 for initial state $\mathtt{x} = x_0$, $\mathtt{y} = y_0$

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

Symbolic Execution of Programs Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

General form of rule conclusions in symbolic execution calculus

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

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

Symbolic Execution of Programs Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

General form of rule conclusions in symbolic execution calculus

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

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

Symbolic execution of conditional

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

Symbolic execution must consider all possible execution branches

Symbolic Execution of Programs Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Symbolic execution of conditional

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

Symbolic execution must consider all possible execution branches

Symbolic execution of loops: unwind (simplified version)

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

Updates for KeY-Style Symbolic Execution

Need to model control flow and state changes

Updates for KeY-Style Symbolic Execution

TECHNISCHE UNIVERSITÄT DARMSTADT

Need to model control flow and state changes

Explicit Notation for Symbolic State Changes: Requirements

- Symbolic execution should "walk" through program in natural forward direction
- Need succint representation of state changes effected by each symbolic execution step
- Want to simplify effects of program execution early
- Want to apply state changes late (to post condition)

Updates for KeY-Style Symbolic Execution

Need to model control flow and state changes

Explicit Notation for Symbolic State Changes: Requirements

- Symbolic execution should “walk” through program in natural forward direction
- Need succint representation of state changes effected by each symbolic execution step
- Want to simplify effects of program execution early
- Want to apply state changes late (to post condition)

We use dedicated notation for state changes: updates

Explicit State Updates

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (Syntax of Updates, Updated Terms/Formulas)

If $\mathtt{v}$ is program variable, t FOL term type-conformant to $\mathtt{v}$,
t' any FOL term, and ϕ any DL formula, then

- $\mathtt{v} := t$ is an update
- $\{\mathtt{v} := t\}t'$ is DL term
- $\{\mathtt{v} := t\}\phi$ is DL formula

Definition (Semantics of Updates)

State $\mathcal{S}$ interprets program variables v with $\mathcal{I}_{\mathcal{S}}(v)$
β variable assignment for logical variables in t, define semantics ρ as:

$\rho(\mathtt{v} := t)(\mathcal{S}) = \mathcal{S}'$ where $\mathcal{S}'$ identical to $\mathcal{S}$ except $\mathcal{I}_{\mathcal{S}'}(\mathtt{v}) = val_{\mathcal{S},\beta}(t)$

Explicit State Updates Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Facts about updates $\mathtt{v} := t$

- Update semantics almost identical to that of assignment
- Value of update also depends on $\mathcal{S}$ and logical variables in t, i.e., β
- Updates are not assignments: right-hand side is FOL term

  $\{\mathtt{x} := n\}\phi$ cannot be turned into assignment (n logical variable)

  $\langle x = i++;\rangle\phi$ cannot (immediately) be turned into an update
- Updates are not equations: they change value of $\mathtt{v}$

Computing Effect of Updates (Automatic)

TECHNISCHE UNIVERSITÄT DARMSTADT

Rewrite rules for update followed by . . .

program variable	$\begin{cases} \{\mathtt{x} := t\}\mathtt{y} & \rightsquigarrow & \mathtt{y} & \mathtt{x} \neq \mathtt{y} \\ \{\mathtt{x} := t\}\mathtt{x} & \rightsquigarrow & t & \text{update happens} \end{cases}$
logical variable	$\{\mathtt{x} := t\}w \rightsquigarrow w$
complex term	$\{\mathtt{x} := t\}f(t_1, \ldots, t_n) \rightsquigarrow f(\{\mathtt{x} := t\}t_1, \ldots, \{\mathtt{x} := t\}t_n)$ (f function symbol)
FOL formula	$\begin{cases} \{\mathtt{x} := t\}(\phi \ \& \ \psi) \rightsquigarrow \{\mathtt{x} := t\}\phi \ \& \ \{\mathtt{x} := t\}\psi \\ \cdots \\ \{\mathtt{x} := t\}(\forall\, T\ y;\ \phi) \rightsquigarrow \forall\, T\ y;\ (\{\mathtt{x} := t\}\phi) \end{cases}$
program formula	No rewrite rule for $\{\mathtt{x} := t\}(\langle p \rangle \phi)$ unchanged!

Update rewriting delayed until $\mathtt{p}$ symbolically executed

Assignment Rule Using Updates

Symbolic execution of assignment using updates

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

- Simple! No variable renaming, etc.
- Works as long as `t` has no side effects
- Special cases needed for `x` $= t_1 + t_2$, etc.

Demo

`updates/assignmentToUpdate.key` (complete)

Parallel Updates

TECHNISCHE UNIVERSITÄT DARMSTADT

How to apply updates on updates?

Example

Symbolic execution of

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

yields:

```
{x := x+y}{y := x-y}{x := x-y}
```

Need to compose three sequential state changes into a single one!

Parallel Updates Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (Parallel Update)

A parallel update is an expression of the form $v_1 := r_1 || \cdots || v_n := r_n$ where each $v_i := r_i$ is simple update

- All r_i computed in old state before update is applied
- Updates of all program variables v_i executed simultaneously
- Upon conflict $v_i = v_j, r_i \neq r_j$ later update ($\max\{i, j\}$) wins

Parallel Updates Cont'd

Definition (Parallel Update)

A parallel update is an expression of the form $v_1 := r_1 || \cdots || v_n := r_n$ where each $v_i := r_i$ is simple update

- All r_i computed in old state before update is applied
- Updates of all program variables v_i executed simultaneously
- Upon conflict $v_i = v_j, r_i \neq r_j$ later update ($\max\{i, j\}$) wins

Definition (Composition Sequential Updates/Conflict Resolution)

$\{v_1 := r_1\}\{u\}\xi \rightsquigarrow \{v_1 := r_1 || \{v_1 := r_1\}u\}\xi$ (ξ is a DL formula or DL term)

$\{v_1 := r_1\}v_2 := r_2 \rightsquigarrow v_2 := \{v_1 := r_1\}r_2$

$$\{v_1 := r_1 || \cdots || v_n := r_n\}\mathtt{x} \rightsquigarrow \begin{cases} \mathtt{x} & \text{if } \mathtt{x} \notin \{v_1, \ldots, v_n\} \\ r_k & \text{if } \mathtt{x} = v_k,\ \mathtt{x} \notin \{v_{k+1}, \ldots, v_n\} \end{cases}$$

Parallel Updates Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Example

$$
\begin{array}{l}
(\{\mathtt{x} := \mathtt{x+y}\}\{\mathtt{y} := \mathtt{x-y}\})\{\mathtt{x} := \mathtt{x-y}\}\ \phi = \\
\{\mathtt{x} := \mathtt{x+y} \mid\mid \mathtt{y} := \mathtt{(x+y)-y}\}\{\mathtt{x} := \mathtt{x-y}\}\ \phi = \\
\{\mathtt{x} := \mathtt{x+y} \mid\mid \mathtt{y} := \mathtt{(x+y)-y} \mid\mid \mathtt{x} := \mathtt{(x+y)-((x+y)-y)}\}\ \phi = \\
\{\mathtt{x} := \mathtt{x+y} \mid\mid \mathtt{y} := \mathtt{x} \mid\mid \mathtt{x} := \mathtt{y}\}\ \phi = \\
\{\mathtt{y} := \mathtt{x} \mid\mid \mathtt{x} := \mathtt{y}\}\ \phi
\end{array}
$$

KeY automatically deletes overwritten (unnecessary) updates

Demo

`updates/swap2.key`

Parallel Updates Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Example

```
({x := x+y}{y := x-y}){x := x-y} φ =
{x := x+y || y := (x+y)-y}{x := x-y} φ =
{x := x+y || y := (x+y)-y || x := (x+y)-((x+y)-y)} φ =
{x := x+y || y := x || x := y} φ =
{y := x || x := y} φ
```

KeY automatically deletes overwritten (unnecessary) updates

Demo

`updates/swap2.key`

Parallel updates to store intermediate state of symbolic computation

Another use of Updates

If you would like to quantify over a program variable ...

Another use of Updates

TECHNISCHE UNIVERSITÄT DARMSTADT

If you would like to quantify over a program variable ...

Not allowed: $\forall\, T\ \texttt{i};\ \langle ...i... \rangle \phi$ (program $\neq$ logical variable)

Another use of Updates

TECHNISCHE UNIVERSITÄT DARMSTADT

If you would like to quantify over a program variable ...

Not allowed: $\forall\, T\; \mathtt{i};\ \langle ...i... \rangle \phi$ (program $\neq$ logical variable)

Instead

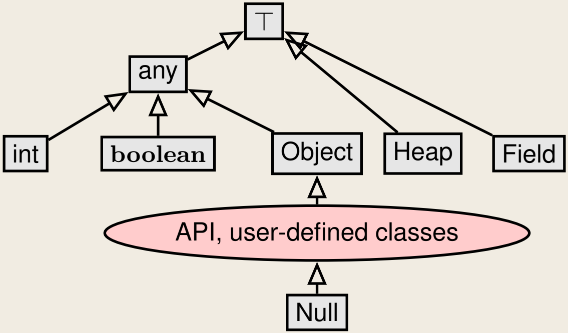
Quantify over value, and assign it to program variable:

$\forall\, T\; i_0;\ \{\mathtt{i} := i_0\} \langle ...i... \rangle \phi$

Modelling JAVA in FOL: Fixing a Type Hierarchy

TECHNISCHE UNIVERSITÄT DARMSTADT

Signature based on JAVA's type hierarchy



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

Modelling the Heap in FOL

TECHNISCHE UNIVERSITÄT DARMSTADT

The JAVA Heap

Values of reference types live on the heap

- Can dynamically change during execution
- In each program state (model) associates objects, fields, values

The JAVA Heap

Values of reference types live on the heap

- Can dynamically change during execution
- In each program state (model) associates objects, fields, values

The JAVA Heap Model of KeY

Data type Heap models content of heap in a state (model)

Write Heap `store(Heap, Object, Field, any);`
Modifies field of object to have value in 4th argument

Read any `select(Heap, Object, Field);`
Selects value of field of object

Modelling Fields in FOL

Modelling instance fields

Person
int age int id
int setAge(int newAge) int getId()

- for each JAVA reference type C there is a logical type C, for example, Person

Modelling Fields in FOL

TECHNISCHE UNIVERSITÄT DARMSTADT

Modelling instance fields

Person
int age int id
int setAge(int newAge) int getId()

- for each JAVA reference type C there is a logical type C, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `age`

Modelling Fields in FOL

Modelling instance fields

Person
`int age` `int id`
`int setAge(int newAge)` `int getId()`

- for each JAVA reference type C there is a logical type C, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `age`
- domain of all Person objects: $\mathtt{D}^{\text{Person}}$

Modelling Fields in FOL

Modelling instance fields

Person
`int age` `int id`
`int setAge(int newAge)` `int getId()`

- for each JAVA reference type C there is a logical type C, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `age`
- domain of all Person objects: $\mathtt{D}^{\text{Person}}$
- a heap relates objects and fields to values

Modelling Fields in FOL

Modelling instance fields

Person
`int age` `int id`
`int setAge(int newAge)` `int getId()`

- for each JAVA reference type C there is a logical type C, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `age`
- domain of all Person objects: D^{Person}
- a heap relates objects and fields to values

Reading Fields (Simplified)

Signature F_Σ: any `select`(Heap, Object, Field);

JAVA expression `p.age >= 0`

Typed FOL $\mathtt{select}(\mathtt{heap}, \mathtt{p}, \mathtt{age}) >= 0$

`heap` is special program variable for “current” heap

Modelling Fields in FOL—The Full Story

TECHNISCHE UNIVERSITÄT DARMSTADT

Reading Fields

Signature F_Σ: any `select(Heap, Object, Field);`

$\texttt{select}(\texttt{heap}, \texttt{p}, \texttt{age}) >= 0$ well-formed?

Modelling Fields in FOL—The Full Story

TECHNISCHE UNIVERSITÄT DARMSTADT

Reading Fields

Signature F_Σ: any `select(Heap, Object, Field);`

$\mathtt{select}(\mathtt{heap}, \mathtt{p}, \mathtt{age}) >= 0$ well-formed?

- Return type is any—need to cast to **int**
- There can be many fields with name $\mathtt{age}$

Modelling Fields in FOL—The Full Story

TECHNISCHE UNIVERSITÄT DARMSTADT

Reading Fields

Signature F_Σ: any `select(Heap, Object, Field);`

$\mathtt{select}(\mathrm{heap}, \mathrm{p}, \mathrm{age}) >= 0$ well-formed?

- Return type is any—need to cast to $\mathbf{int}$
- There can be many fields with name age

Use function $\mathbf{int}::\mathtt{select}(\mathrm{heap}, \mathrm{p}, \text{Person}::\$\mathrm{age})$

($\mathbf{int}::\mathtt{select}$ has same meaning as $(\mathbf{int})\mathtt{select}$)

Modelling Fields in FOL—The Full Story

TECHNISCHE UNIVERSITÄT DARMSTADT

Reading Fields

Signature F_Σ: any `select(Heap, Object, Field);`

$\mathtt{select}(\mathrm{heap}, \mathrm{p}, \mathrm{age}) >= 0$ well-formed?

- Return type is any—need to cast to **int**
- There can be many fields with name age

Use function $\mathbf{int}::\mathtt{select}(\mathrm{heap}, \mathrm{p}, \text{Person}::\$\mathrm{age})$

($\mathbf{int}::\mathtt{select}$ has same meaning as $\mathtt{(}\mathbf{int}\mathtt{)select}$)

Writing to Fields

Signature F_Σ: Heap `store(Heap, Object, Field, any);`

Use function $\mathtt{store}(\mathrm{heap}, \mathrm{p}, \text{Person}::\$\mathrm{age}, 42)$

Modelling Fields in FOL—The Full Story: Semantics

`Heap` is a predefined type with predefined functions `T` :: `select`, `store` and more

Semantics

Given a Kripke structure K and a first order state $S = (D, \rho, I)$

- $D^{\mathtt{Heap}} = \{h \mid h : D^{\mathtt{Object}} \times D^{\mathtt{Field}} \to D^{\mathtt{any}}\}$
  (domain of `Heap` is set of functions associating objects and fields with a value)
- $I(\mathtt{select})(h, o, f) = h(o, f)$ with $h \in D^{\mathtt{Heap}}, o \in D^{\mathtt{Object}}, f \in D^{\mathtt{Field}}$
- $I(\mathtt{store})(h, o, f, v) = h'$ with $h \in D^{\mathtt{Heap}}, o \in D^{\mathtt{Object}}, f \in D^{\mathtt{Field}}, v \in D^{\mathtt{any}}$ and

$$h'(u, g) = \begin{cases} v & \text{if } u = o \text{ and } g = f \\ h(u, g) & \text{otherwise} \end{cases}$$

Field Updates

TECHNISCHE UNIVERSITÄT DARMSTADT

The Global Program Variable `heap`

The dynamic logic contains a distinguished program variable Heap `heap`. The heap stored in this variable is used by the Java program for read/write field accesses.

Changing the value of fields

How to translate assignment to field, for example, `p.age=17;` ?

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

Admit on left-hand side of update JAVA location expressions

Field Updates

TECHNISCHE UNIVERSITÄT DARMSTADT

The Global Program Variable `heap`

The dynamic logic contains a distinguished program variable Heap `heap`. The heap stored in this variable is used by the Java program for read/write field accesses.

Changing the value of fields

How to translate assignment to field, for example, `p.age=17;` ?

$$\text{assign}\ \frac{\Gamma \Rightarrow \{\mathtt{heap} := \mathtt{store}(\mathtt{heap}, \mathtt{p}, \mathtt{age}, 17)\}\langle \mathit{rest} \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \mathit{p.age}\ = 17; \mathit{rest} \rangle \phi, \Delta}$$

Admit on left-hand side of update JAVA location expressions

Evaluating a field access

TECHNISCHE UNIVERSITÄT DARMSTADT

Evaluation

Computing the value of a field *a* of an object *o* of type *T* in a given heap, performs a lookup in the heap using the pair (*o*, *a*) as key/index.

Example

$\mathtt{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 15), \mathtt{o}, \mathtt{f}) \leadsto$

Evaluating a field access

TECHNISCHE UNIVERSITÄT DARMSTADT

Evaluation

Computing the value of a field *a* of an object *o* of type *T* in a given heap, performs a lookup in the heap using the pair (*o*, *a*) as key/index.

Example

$$\mathtt{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 15), \mathtt{o}, \mathtt{f}) \rightsquigarrow 15$$

$$\mathtt{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 15), \mathtt{o}, \mathtt{g}) \rightsquigarrow$$

Evaluating a field access

Evaluation

Computing the value of a field *a* of an object *o* of type *T* in a given heap, performs a lookup in the heap using the pair (*o*, *a*) as key/index.

Example

$$\mathtt{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap},\mathtt{o},\mathtt{f},15),\mathtt{o},\mathtt{f}) \rightsquigarrow 15$$

$$\mathtt{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap},\mathtt{o},\mathtt{f},15),\mathtt{o},\mathtt{g}) \rightsquigarrow \mathtt{int}::\mathtt{select}(\mathtt{heap},\mathtt{o},\mathtt{g})$$

$$\mathtt{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap},\mathtt{o},\mathtt{f},15),\mathtt{u},\mathtt{f}) \rightsquigarrow$$

Evaluating a field access

TECHNISCHE UNIVERSITÄT DARMSTADT

Evaluation

Computing the value of a field *a* of an object *o* of type *T* in a given heap, performs a lookup in the heap using the pair (*o*, *a*) as key/index.

Example

$$\texttt{int::select}(\texttt{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 15), \texttt{o}, \texttt{f}) \rightsquigarrow 15$$

$$\texttt{int::select}(\texttt{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 15), \texttt{o}, \texttt{g}) \rightsquigarrow \texttt{int::select}(\texttt{heap}, \texttt{o}, \texttt{g})$$

$$\texttt{int::select}(\texttt{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 15), \texttt{u}, \texttt{f}) \rightsquigarrow \text{if } (\texttt{o} \doteq \texttt{u}) \text{ then } (15) \text{ else } (\texttt{int::select}(\texttt{heap}, \texttt{u}, \texttt{f}))$$

Evaluating a field access

TECHNISCHE UNIVERSITÄT DARMSTADT

Evaluation

Computing the value of a field *a* of an object *o* of type *T* in a given heap, performs a lookup in the heap using the pair (o, a) as key/index.

Example

$$\mathtt{int::select}(\mathtt{store}(\mathtt{heap},\mathtt{o},\mathtt{f},15),\mathtt{o},\mathtt{f}) \rightsquigarrow 15$$

$$\mathtt{int::select}(\mathtt{store}(\mathtt{heap},\mathtt{o},\mathtt{f},15),\mathtt{o},\mathtt{g}) \rightsquigarrow \mathtt{int::select}(\mathtt{heap},\mathtt{o},\mathtt{g})$$

$$\mathtt{int::select}(\mathtt{store}(\mathtt{heap},\mathtt{o},\mathtt{f},15),\mathtt{u},\mathtt{f}) \rightsquigarrow$$
$$\text{if } (\mathtt{o} \doteq \mathtt{u}) \text{ then } (15) \text{ else } (\mathtt{int::select}(\mathtt{heap},\mathtt{u},\mathtt{f}))$$

Pretty Printing

$\mathtt{T} :: \mathtt{select}(\mathtt{heap},\mathtt{o},\mathtt{f})$ is shown as $\mathtt{o.f}$

$\mathtt{select}(\mathtt{store}(\mathtt{heap},\mathtt{o},\mathtt{f},17),\mathtt{u},\mathtt{f})$ is shown as $\mathtt{u.f@heap[o.f} := 17]$

Evaluating a field access

Evaluation

Computing the value of a field *a* of an object *o* of type *T* in a given heap, performs a lookup in the heap using the pair (o, a) as key/index.

Example

On the following slides we often use the pretty printed version and omit the T :: prefix for ease of presentation.

$\mathtt{int::select(store(heap, o, f, 15), u, f)} \rightsquigarrow$

$$\text{if } (\mathtt{o} \doteq \mathtt{u}) \text{ then } (15) \text{ else } (\mathtt{int::select(heap, u, f)})$$

Pretty Printing

$\mathtt{T :: select(heap, o, f)}$ is shown as $\mathtt{o.f}$

$\mathtt{select(store(heap, o, f, 17), u, f)}$ is shown as $\mathtt{u.f@heap[o.f := 17]}$

Dynamic Logic: KeY input file


```
\javaSource "path to source code referenced in problem";

\programVariables { Person p; }

\problem {
       \<{   p.age = 18;  }\> p.age = 18
}
```

KeY reads in all source files and creates automatically
the necessary signature (types, program variables, field constants)

Dynamic Logic: KeY input file

TECHNISCHE UNIVERSITÄT DARMSTADT

```
\javaSource "path to source code referenced in problem";

\programVariables { Person p; }

\problem {
       \<{   p.age = 18;  }\> p.age = 18
}
```

KeY reads in all source files and creates automatically
the necessary signature (types, program variables, field constants)

Demo

`updates/firstAttributeExample.key`

Semantics of Abrupt Termination

TECHNISCHE UNIVERSITÄT DARMSTADT

Does abrupt termination count as normal termination?
No! Need to distinguish normal and exceptional termination

Semantics of Abrupt Termination

Does abrupt termination count as normal termination?
No! Need to distinguish normal and exceptional termination

- $\langle p \rangle \phi$: $\mathtt{p}$ terminates normally and formula ϕ holds in final state (total correctness)

Semantics of Abrupt Termination

Does abrupt termination count as normal termination?
No! Need to distinguish normal and exceptional termination

- $\langle p \rangle \phi$: $\mathtt{p}$ terminates normally and formula ϕ holds in final state (total correctness)
- $[p]\phi$: If $\mathtt{p}$ terminates normally then formula ϕ holds in final state (partial correctness)

Semantics of Abrupt Termination

Does abrupt termination count as normal termination?
No! Need to distinguish normal and exceptional termination

- $\langle p \rangle \phi$: $\mathtt{p}$ terminates normally and formula ϕ holds in final state (total correctness)
- $[p]\phi$: If $\mathtt{p}$ terminates normally then formula ϕ holds in final state (partial correctness)

Abrupt termination counts as non-termination!

Example Reconsidered: Exception Handling


```
\javaSource "path to source code";

\programVariables {
  ...
}

\problem {
       p != null -> \<{   p.age = 18;  }\> p.age = 18
}
```

Only provable when no top-level exception thrown

Demo

`updates/secondAttributeExample.key`

Null Pointers

Null pointer exceptions

There are no “exceptions” in FOL: $\mathcal{I}$ total on FSym

Need to model possibility that $\mathtt{o} \doteq \mathbf{null}$ in $\mathtt{o.a}$

- KeY branches over $\mathtt{o}$ $!\!= \mathbf{null}$ upon each field access

Null Pointers

Null pointer exceptions

There are no "exceptions" in FOL: $\mathcal{I}$ total on FSym

Need to model possibility that `o` $\doteq$ $\mathbf{null}$ in `o.a`

- KeY branches over `o` != $\mathbf{null}$ upon each field access

Assignment Rule for Fields

$$\text{assign}_{\text{Field}} \quad \frac{\begin{array}{l}\Gamma, \{u\}(o \mathrel{!=} \mathbf{null}) \Rightarrow \{u\}\{\mathtt{heap} := \mathtt{store}(\mathtt{heap}, o, f, v)\}\langle rest\rangle\phi, \Delta \\ \Gamma, \{u\}(o \doteq \mathbf{null}) \Rightarrow \{u\}\langle \mathbf{throw\ new}\ \mathtt{NullPointerException();}\, rest\rangle\phi, \Delta\end{array}}{\Gamma \Rightarrow \{u\}\langle o.f \;=\; v;\, rest\rangle\phi, \Delta}$$

o, *v* schemavariables matching prog. variables; *f* schemavariable matching fields

Aliasing

Demo

`aliasing/attributeAlias1.key`

Aliasing

Demo

`aliasing/attributeAlias1.key`

Reference Aliasing

Naive alias resolution causes proof split (on $\mathtt{o} \doteq \mathtt{u}$) at each access

$$\Rightarrow \mathtt{o.age} \doteq 1 \quad \text{-->} \quad \langle \mathit{u.age} = 2; \rangle \mathtt{o.age} \doteq \mathtt{u.age}$$

Aliasing Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Unnecessary case analyses

$$\Rightarrow \mathtt{o.age} \doteq 1 \;\; \texttt{->} \;\; \langle u.age = 2; o.age = 2; \rangle \mathtt{o.age} \doteq \mathtt{u.age}$$

$$\Rightarrow \mathtt{o.age} \doteq 1 \;\; \texttt{->} \;\; \langle u.age = 2; \rangle \mathtt{u.age} \doteq 2$$

Aliasing Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Unnecessary case analyses

$$\Rightarrow \texttt{o.age} \doteq 1 \;\; \texttt{->} \;\; \langle u.age = 2; o.age = 2; \rangle \texttt{o.age} \doteq \texttt{u.age}$$

$$\Rightarrow \texttt{o.age} \doteq 1 \;\; \texttt{->} \;\; \langle u.age = 2; \rangle \texttt{u.age} \doteq 2$$

Avoiding case analyses— Demo

`aliasing/avoidingCaseAnalysis2.key`

- Delayed state computation until clear what is required
- Simplification of heap terms

Modeling Static Fields in FOL

TECHNISCHE UNIVERSITÄT DARMSTADT

Modeling class (static) fields which are not compile-time constants

For each class C with static field `a` of type T that is not a compile time constant, there is

- a unique constant $C.a$ of type Field
- value v is stored on heap as $\mathtt{store}(\mathtt{heap}, \mathtt{null}, \mathtt{C.a}, \mathtt{v})$
- C is the fully qualified class name

Value stored on the heap.

Modeling class (static) fields which are compile-time constants

For each class C with final static field `a` of type T that is a compile time constant, there is

- a constant $C.a$ of type T whose interpretation $I(C.a)$ is fixed, e.g., $I(\mathtt{java.util.Byte.MAX\_VALUE}) = 127$

Value not stored on the heap

`this` **and** `self`

Modeling reference **this** to the receiving object

Special name for the object whose JAVA code is currently executed:

in JML: `Object` **this**;

in JAVA: `Object` **this**;

in KeY: Arbitrary, but by convention `Object self;`
Name is arbitrary as long as unique: only a program variable

Default assumption in JML-KeY translation: `self` != **null**

Extending Dynamic Logic to Java

KeY admits any syntactically correct JAVA with some extensions:

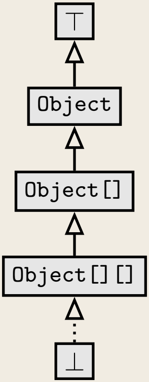
- Needs not be compilable unit
- Permit externally declared, non-initialized variables
- All referenced class definitions loaded in background

And some limitations . . .

- Limited concurrency (some support on side-branches)
- No generics
- No I/O
- No floats
- No dynamic class loading or reflexion
- API method calls: need either JML contract or implementation

JAVA Features in Dynamic Logic: Arrays

Arrays



- JAVA type hierarchy includes array types that occur in given program (for finiteness)
- Types ordered according to JAVA subtyping rules
- Value of entry in array `T[] a;` defined in class `C` depends on reference `a` to array in `C` and index
- Function $\mathtt{arr} : \mathrm{int} \to \mathrm{Field}$ injective mapping from indices to fields
- Store array elements on heap, e.g., the value of $\mathtt{a}[\mathtt{i}]$ on the heap $\mathtt{store}(\mathtt{heap}, \mathtt{a}, \mathtt{arr}(\mathtt{i}), 17)$ is 17
- Arrays `a` and `b` can refer to same object (aliases)
- KeY implements simplification and evaluation rules for array locations

JAVA Features in Dynamic Logic: Complex Expressions

TECHNISCHE UNIVERSITÄT DARMSTADT

Complex expressions with side effects

- JAVA expressions may contain assignment operator with side effect
- JAVA expressions can be complex, nested, have method calls
- FOL terms have no side effect on the state

Example (Complex expression with side effects in JAVA)

`int i = 0; if ((i=2)>= 2) i++;` value of `i` ?

Complex Expressions Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Decomposition of complex terms by symbolic execution

Follow the rules laid down in JAVA Language Specification

Local code transformations

$$\text{evalOrderIteratedAssgnmt}\quad \frac{\Gamma \Rightarrow \langle y = t; x = y;\ \omega\rangle\phi, \Delta}{\Gamma \Rightarrow \langle x = y = t;\ \omega\rangle\phi, \Delta}\quad \texttt{t}\ \text{simple}$$

Temporary variables store result of evaluating subexpression

$$\text{ifEval}\quad \frac{\Gamma \Rightarrow \langle \mathbf{boolean}\ v0; v0\ = b; \mathbf{if}\ (v0)\ p; \omega\rangle\phi, \Delta}{\Gamma \Rightarrow \langle \mathbf{if}\ (b)\ p;\ \omega\rangle\phi, \Delta}\quad \texttt{b}\text{complex}$$

Guards of conditionals/loops always evaluated (hence: side effect-free) before conditional/unwind rules applied

JAVA Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

Redirection of control flow via **return**, **break**, **continue**, exceptions

$$\langle\pi\ \mathbf{try}\ \{p\}\ \mathbf{catch}(e)\ \{q\}\ \mathbf{finally}\ \{r\}\ \omega\rangle\phi$$

Rules ignore inactive prefix, work on **active statement**, leave postfix

Abrupt Termination: Exceptions and Jumps

Redirection of control flow via $\mathbf{return}$, $\mathbf{break}$, $\mathbf{continue}$, exceptions

$$\langle \pi\ \mathbf{try}\ \{p\}\ \mathbf{catch}(e)\ \{q\}\ \mathbf{finally}\ \{r\}\ \omega\rangle\phi$$

Rules ignore inactive prefix, work on **active statement**, leave postfix

Rule tryThrow matches $\mathbf{try}$–$\mathbf{catch}$ in pre-/postfix and active $\mathbf{throw}$

$$\frac{\Rightarrow \langle \pi\ \mathrm{if}(e\ \mathrm{instanceof}\ T)\{\mathrm{try}\{x = e; q\}\mathrm{finally}\{r\}\}\mathrm{else}\{r; \mathbf{throw}\ e;\}\ \omega\rangle\phi}{\Rightarrow \langle \pi\ \mathbf{try}\ \{\mathbf{throw}\ e;\ p\}\ \mathbf{catch}(T\ x)\ \{q\}\ \mathbf{finally}\ \{r\}\ \omega\rangle\phi}$$

Abrupt Termination: Exceptions and Jumps

Redirection of control flow via **return**, **break**, **continue**, exceptions

$$\langle \pi\ \mathbf{try}\ \{p\}\ \mathbf{catch}(e)\ \{q\}\ \mathbf{finally}\ \{r\}\ \omega \rangle \phi$$

Rules ignore inactive prefix, work on **active statement**, leave postfix

Rule tryThrow matches **try**–**catch** in pre-/postfix and active **throw**

$$\frac{\Rightarrow \langle \pi\ \mathbf{if}(e\ \mathbf{instanceof}\ T)\{\mathbf{try}\{x = e; q\}\mathbf{finally}\{r\}\}\mathbf{else}\{r; \mathbf{throw}\ e;\}\ \omega \rangle \phi}{\Rightarrow \langle \pi\ \mathbf{try}\ \{\mathbf{throw}\ e;\ p\}\ \mathbf{catch}(T\ x)\ \{q\}\ \mathbf{finally}\ \{r\}\ \omega \rangle \phi}$$

Demo: `exceptions/try-catch.key`

Which Objects do Exist?

TECHNISCHE UNIVERSITÄT DARMSTADT

How to model object creation with **new** ?

Which Objects do Exist?

How to model object creation with $\mathbf{new}$?

Constant Domain Assumption

Assume that domain $\mathcal{D}$ is the same in all states of LTS $K = (S, \rho)$

Desirable consequence:
Validity of rigid FOL formulas unaffected by programs containing $\mathbf{new}$()

$$\models \forall\, T\ x;\ \phi \ {->}\ [p](\forall\, T\ x;\ \phi) \qquad \text{is valid for rigid } \phi$$

Object Creation

Realizing Constant Domain Assumption

- Implicitly declared field `boolean <created>` in class `java.lang.Object`
- Equal to **true** iff argument object has been created
- Object creation modeled as $\{$`heap` := `create(heap, o)`$\}$ for next “free” o (essentially sets `<created>` field of o to **true**)
- Normal heap function *store* “cannot” set value of field `<created>`

Object Creation

TECHNISCHE UNIVERSITÄT DARMSTADT

Realizing Constant Domain Assumption

- Implicitly declared field **boolean** `<created>` in class `java.lang.Object`
- Equal to **true** iff argument object has been created
- Object creation modeled as $\{\texttt{heap := create(heap, o)}\}$ for next "free" `o` (essentially sets `<created>` field of `o` to **true**)
- Normal heap function *store* "cannot" set value of field `<created>`

ObjectCreation(simplified)

$$\frac{\Gamma,\ \{u\}(\texttt{select}(\texttt{heap},\texttt{newObj},\texttt{Object}::\texttt{<created>}) = \texttt{FALSE}) \Rightarrow \{u\}(\{\texttt{heap} := \texttt{create}(\texttt{heap},\texttt{newObj})\}\{\texttt{o}:=\texttt{newObj}\}\langle\pi\ \texttt{o.<init>(}\mathit{selist}\texttt{);}\ \omega\rangle\phi),\ \Delta}{\{u\}(\langle\pi\ \texttt{o = } \mathbf{new}\ \texttt{T(}\mathit{selist}\texttt{);}\ \omega\rangle\phi)}$$

`newObj` is a fresh program variable

Summary

- Most JAVA features covered in KeY
- Several of remaining features available in experimental version
  - Simplified multi-threaded JMM
  - Floats
- Degree of automation for loop-free programs is very high
- Symbolic execution paradigm lets you use KeY w/o understanding details of logic

Remaining Questions

- How to deal with method calls?
  - Inlining method bodies vs. using contracts
  - Proof obligations from method contracts
- Proving loops

Literature for this Lecture

TECHNISCHE UNIVERSITÄT DARMSTADT

Essential

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 10: Using KeY

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 3: Dynamic Logic, Sections 3.1, 3.2, 3.4, 3.5, 3.6.1, 3.6.2, 3.6.3, 3.6.4, 3.6.7