Software Engineering using Formal Methods

Reasoning about Programs with Dynamic Logic

Wolfgang Ahrendt

08 October 2015

Dynamic Logic

(JAVA) Dynamic Logic

Typed FOL

- $+$ (JAVA) programs $\mathtt{p}$

Dynamic Logic

(Java) Dynamic Logic

Typed FOL

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

Dynamic Logic

(JAVA) Dynamic Logic

Typed FOL

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

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

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

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

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

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: $\mathcal{S} \models (\Gamma \Longrightarrow \Delta)$ 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

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

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

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

General form of rule conclusions in symbolic execution calculus

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

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

Symbolic Execution of Programs Cont'd

General form of rule conclusions in symbolic execution calculus

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

- 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

$$\text{if}\ \frac{\Gamma, \texttt{b} = \mathbf{true} \Longrightarrow \langle \texttt{p;\ rest} \rangle \phi, \Delta \qquad \Gamma, \texttt{b} = \mathbf{false} \Longrightarrow \langle \texttt{q;\ rest} \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathbf{if}\ \texttt{(b) \{ p \}}\ \mathbf{else}\ \texttt{\{ q \} ; rest} \rangle \phi, \Delta}$$

Symbolic execution must consider all possible execution branches

Symbolic Execution of Programs Cont'd

Symbolic execution of conditional

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

Symbolic execution must consider all possible execution branches

Symbolic execution of loops: unwind

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

Updates for KeY-Style Symbolic Execution

Needed: a Notation for Symbolic State Changes

- 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 branching conditions and post condition)

Updates for KeY-Style Symbolic Execution

Needed: a Notation for Symbolic State Changes

- 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 branching conditions and post condition)

We use dedicated notation for state changes: updates

Explicit State Updates

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:

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

Explicit State Updates Cont'd

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

- Update semantics similar 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)

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

Computing Effect of Updates (Automated)

Rewrite rules for update followed by . . .

program variable
$$\begin{cases} \{\mathrm{x} := t\}\mathrm{x} & \rightsquigarrow \quad t \\ \{\mathrm{x} := t\}\mathrm{y} & \rightsquigarrow \quad \mathrm{y} \end{cases}$$

logical variable $\{\mathrm{x} := t\}w \rightsquigarrow w$

complex term $\{\mathrm{x} := t\}f(t_1, \ldots, t_n) \rightsquigarrow f(\{\mathrm{x} := t\}t_1, \ldots, \{\mathrm{x} := t\}t_n)$

(because f is rigid)

FOL formula
$$\begin{cases} \{\mathrm{x} := t\}(\phi \ \& \ \psi) \rightsquigarrow \{\mathrm{x} := t\}\phi \ \& \ \{\mathrm{x} := t\}\psi \\ \qquad \cdots \\ \{\mathrm{x} := t\}(\forall \tau \ y; \phi) \rightsquigarrow \forall \tau \ y; (\{\mathrm{x} := t\}\phi) \end{cases}$$

program formula No rewrite rule for $\{\mathrm{x} := t\}(\langle \mathrm{p} \rangle \phi)$ unchanged!

Update rewriting delayed until p symbolically executed

Assignment Rule Using Updates

Symbolic execution of assignment using updates

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

- Simple! No variable renaming, etc.
- Works as long as t has no side effects

Demo

`updates/assignmentToUpdate.key`

Parallel Updates

How to apply updates on updates?

Example

Symbolic execution of

```
t=x; x=y; y=t;
```

yields:

```
{t := x}{x := y}{y := t}
```

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

Parallel Updates

How to apply updates on updates?

Example

Symbolic execution of

```
t=x; x=y; y=t;
```

yields:

$$\{t := x\}\{x := y\}\{y := t\}$$

Need to compose three sequential state changes into a single one:
parallel updates

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

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)

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

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)

$$\{v_1 := r_1\}\{v_2 := r_2\} \;=\; \{v_1 := r_1 || v_2 := \{v_1 := r_1\}r_2\}$$

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

Symbolic Execution with Updates (by Example)

$$\Longrightarrow \ \mathtt{x} < \mathtt{y} \;\mathtt{->}\; \langle \mathtt{int\ t=x;\ x=y;\ y=t;} \rangle\, \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t:=x}\}\langle \mathtt{x=y;\ y=t;} \rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\Longrightarrow\; \mathtt{x} < \mathtt{y} \;\mathtt{->}\; \langle \mathtt{int\ t=x;\ x=y;\ y=t;} \rangle\, \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \mathrel{\mathtt{->}} \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t{:=}x} \parallel \mathtt{x{:=}y}\}\{\mathtt{y{:=}t}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t{:=}x}\}\{\mathtt{x{:=}y}\}\langle \mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \;\Longrightarrow\; \{\mathtt{t{:=}x}\}\langle \mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\Longrightarrow\; \mathtt{x} < \mathtt{y} \;\mathtt{->}\; \langle \mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \ \mathtt{->}\ \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \ \Longrightarrow\ \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \ \Longrightarrow\ \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \ \Longrightarrow\ \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \ \Longrightarrow\ \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \ \Longrightarrow\ \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\Longrightarrow\ \mathtt{x} < \mathtt{y} \ \texttt{->}\ \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \implies \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \;\text{->}\; \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

Parallel Updates Cont'd

Example

symbolic execution of `x=x+y; y=x-y; x=x-y;` gives

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

Parallel Updates Cont'd

Example

symbolic execution of `x=x+y; y=x-y; x=x-y;` gives

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

Parallel Updates Cont'd

Example

symbolic execution of `x=x+y; y=x-y; x=x-y;` gives

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

Parallel Updates Cont'd

Example

symbolic execution of `x=x+y; y=x-y; x=x-y;` gives

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

Parallel Updates Cont'd

Example

symbolic execution of `x=x+y; y=x-y; x=x-y;` gives

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

KeY automatically deletes overwritten (unnecessary) updates

Demo

`updates/swap2.key`

Parallel Updates Cont'd

Example

symbolic execution of `x=x+y; y=x-y; x=x-y;` gives

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

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

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

Not allowed: $\forall\, \tau\ \mathtt{i};\ \langle \ldots \mathtt{i} \ldots \rangle \phi$
(program variables $\cap$ logical variables $= \emptyset$)

Another use of Updates

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

Not allowed: $\forall\,\tau\ \mathtt{i};\ \langle\ldots\mathtt{i}\ldots\rangle\phi$

(program variables $\cap$ logical variables $= \emptyset$)

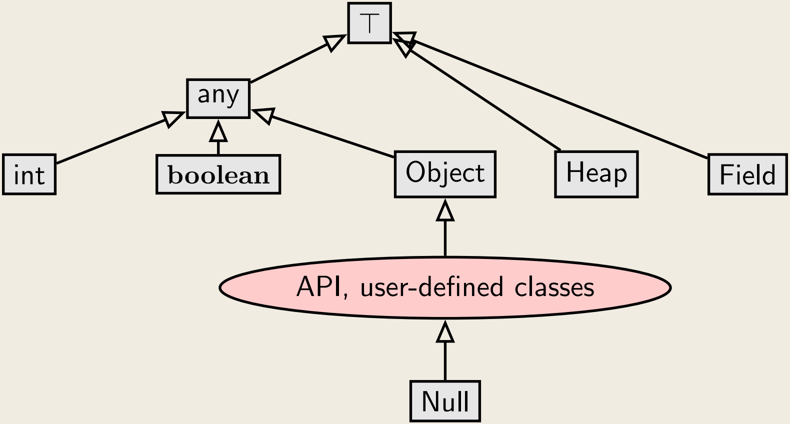
Instead

Quantify over value, and assign it to program variable:

$\forall\,\tau\ x;\ \{\mathtt{i} := x\}\langle\ldots\mathtt{i}\ldots\rangle\phi$

Modelling Java in FOL: Fixing a Type Hierarchy

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

The Java Heap

Objects are stored on (i.e., in) the heap.

- Status of heap changes during execution
- Each heap associates values to object/field pairs

Modelling the Heap in FOL

The Java Heap

Objects are stored on (i.e., in) the heap.

- Status of heap changes during execution
- Each heap associates values to object/field pairs

The Heap Model of KeY-DL

Each element of data type Heap represents a certain heap status.
Two functions involving heaps:

Modelling the Heap in FOL

The Java Heap

Objects are stored on (i.e., in) the heap.

- Status of heap changes during execution
- Each heap associates values to object/field pairs

The Heap Model of KeY-DL

Each element of data type Heap represents a certain heap status.
Two functions involving heaps:

- in F_Σ: Heap `store`(Heap, Object, Field, any) ;
  `store`(h, o, f, v) returns heap like h, but with v associated to (o, f)

Modelling the Heap in FOL

The Java Heap

Objects are stored on (i.e., in) the heap.

- Status of heap changes during execution
- Each heap associates values to object/field pairs

The Heap Model of KeY-DL

Each element of data type Heap represents a certain heap status.
Two functions involving heaps:

- in F_Σ: Heap `store`(Heap, Object, Field, any);
  `store`(h, o, f, v) returns heap like h, but with v associated to (o, f)
- in F_Σ: any `select`(Heap, Object, Field);
  `select`(h, o, f) returns value associated to (o, f) in h

Modelling the Heap 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 type $C \in T_\Sigma$, for example, Person

Modelling the Heap 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 type $C \in T_\Sigma$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`

Modelling the Heap 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 type $\mathsf{C} \in T_{\Sigma}$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`
- domain of all Person objects: $\mathtt{D}^{\mathsf{Person}}$

Modelling the Heap 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 type $C \in T_\Sigma$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`
- domain of all Person objects: $\mathtt{D}^{\mathsf{Person}}$
- a heap relates objects and fields to values

Modelling the Heap 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 type $C \in T_\Sigma$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`
- domain of all Person objects: D^{Person}
- a heap relates objects and fields to values

Reading Field `id` of Person `p`

FOL notation $\texttt{select}(h, \texttt{p}, \texttt{id})$

Modelling the Heap 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 type $C \in T_\Sigma$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`
- domain of all Person objects: $\mathtt{D}^{\mathsf{Person}}$
- a heap relates objects and fields to values

Reading Field `id` of Person `p`

FOL notation $\mathtt{select}(h, \mathtt{p}, \mathtt{id})$

KeY notation `p.id@`h (abbreviating $\mathtt{select}(h, \mathtt{p}, \mathtt{id})$)

Modelling the Heap 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 type $\mathsf{C} \in T_\Sigma$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`
- domain of all Person objects: $\mathtt{D}^{\mathsf{Person}}$
- a heap relates objects and fields to values

Reading Field `id` of Person `p`

FOL notation $\mathtt{select}(h, \mathtt{p}, \mathtt{id})$

KeY notation $\mathtt{p.id@}h$ (abbreviating $\mathtt{select}(h, \mathtt{p}, \mathtt{id})$)

$\mathtt{p.id}$ (abbreviating $\mathtt{select}(\mathtt{heap}, \mathtt{p}, \mathtt{id})$)[a]

[a] `heap` is special program variable for "current" heap; mostly implicit in $o.f$

Modelling the Heap 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 type $C \in T_\Sigma$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`
- domain of all Person objects: $\mathtt{D}^{\mathsf{Person}}$
- a heap relates objects and fields to values

Writing to Field `id` of Person `p`

FOL notation $\mathtt{store}(h, \mathtt{p}, \mathtt{id}, 6238)$

Modelling the Heap 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 type $\mathrm{C} \in T_\Sigma$, for example, Person
- for each field `f` there is a unique constant `f` of type Field, for example, `id`
- domain of all Person objects: $\mathtt{D}^{\mathrm{Person}}$
- a heap relates objects and fields to values

Writing to Field `id` of Person `p`

FOL notation $\mathtt{store}(h, \mathtt{p}, \mathtt{id}, 6238)$

KeY notation $h[\mathtt{p.id} := 6238]$ (notation for `store`, not update)

The Algebra of Heaps

We do *not* formalise the *structure* (implementation) of heaps.
We formalise the *behaviour*, with an algebra of heap operations:

$$\mathtt{select}(\mathtt{store}(h, o, f, v), o, f) =$$

The Algebra of Heaps

We do *not* formalise the *structure* (implementation) of heaps.
We formalise the *behaviour*, with an algebra of heap operations:

$$\mathtt{select}(\mathtt{store}(h, o, f, v), o, f) = v$$

The Algebra of Heaps

We do *not* formalise the *structure* (implementation) of heaps.
We formalise the *behaviour*, with an algebra of heap operations:

$$\mathtt{select}(\mathtt{store}(h, o, f, v), o, f) = v$$

$$(o \neq o' \vee f \neq f') \;\rightarrow\; \mathtt{select}(\mathtt{store}(h, o, f, x), o', f') =$$

The Algebra of Heaps

We do *not* formalise the *structure* (implementation) of heaps.
We formalise the *behaviour*, with an algebra of heap operations:

$$\mathtt{select}(\mathtt{store}(h, o, f, v), o, f) = v$$

$$(o \neq o' \vee f \neq f') \;\rightarrow\; \mathtt{select}(\mathtt{store}(h, o, f, x), o', f') = \mathtt{select}(h, o', f')$$

The Algebra of Heaps

We do *not* formalise the *structure* (implementation) of heaps.
We formalise the *behaviour*, with an algebra of heap operations:

$$\texttt{select}(\texttt{store}(h, o, f, v), o, f) = v$$

$$(o \neq o' \vee f \neq f') \;\rightarrow\; \texttt{select}(\texttt{store}(h, o, f, x), o', f') = \texttt{select}(h, o', f')$$

Example

$\texttt{select}(\texttt{store}(h, \texttt{o}, \texttt{f}, 15), \texttt{o}, \texttt{f}) \leadsto 15$
$\texttt{select}(\texttt{store}(h, \texttt{o}, \texttt{f}, 15), \texttt{o}, \texttt{g}) \leadsto \texttt{select}(h, \texttt{o}, \texttt{g})$
$\texttt{select}(\texttt{store}(h, \texttt{o}, \texttt{f}, 15), \texttt{u}, \texttt{f}) \leadsto$
$\quad \text{if } (\texttt{o} = \texttt{u}) \text{ then } (15) \text{ else } (\texttt{select}(h, \texttt{u}, \texttt{f}))$

Pretty Printing

Shorthand Notations for Heap Operations

$\mathtt{o.f@h}$	is	$\mathtt{select}(\mathtt{h}, \mathtt{o}, \mathtt{f})$
$\mathtt{h}[\mathtt{o.f} := \mathtt{v}]$	is	$\mathtt{store}(\mathtt{h}, \mathtt{o}, \mathtt{f}, \mathtt{v})$

Pretty Printing

Shorthand Notations for Heap Operations

$\mathtt{o.f@h}$	is	$\mathtt{select}(\mathtt{h}, \mathtt{o}, \mathtt{f})$
$\mathtt{h}[\mathrm{o.f} := \mathrm{v}]$	is	$\mathtt{store}(\mathtt{h}, \mathtt{o}, \mathtt{f}, \mathtt{v})$

therefore:

$\mathtt{u.f@h}[\mathrm{o.f} := \mathrm{v}]$	is	$\mathtt{select}(\mathtt{store}(\mathtt{h}, \mathtt{o}, \mathtt{f}, \mathtt{v}), \mathtt{u}, \mathtt{f})$
$\mathtt{h}[\mathrm{o.f} := \mathrm{v}][\mathrm{o}'.\mathrm{f}' := \mathrm{v}']$	is	$\mathtt{store}(\mathtt{store}(\mathtt{h}, \mathtt{o}, \mathtt{f}, \mathtt{v}), \mathtt{o}', \mathtt{f}', \mathtt{v}')$

Pretty Printing

Shorthand Notations for Heap Operations

$\mathtt{o.f@h}$	is	$\mathtt{select}(\mathtt{h}, \mathtt{o}, \mathtt{f})$
$\mathtt{h}[\mathtt{o.f} := \mathtt{v}]$	is	$\mathtt{store}(\mathtt{h}, \mathtt{o}, \mathtt{f}, \mathtt{v})$

therefore:

$\mathtt{u.f@h}[\mathtt{o.f} := \mathtt{v}]$	is	$\mathtt{select}(\mathtt{store}(\mathtt{h}, \mathtt{o}, \mathtt{f}, \mathtt{v}), \mathtt{u}, \mathtt{f})$
$\mathtt{h}[\mathtt{o.f} := \mathtt{v}][\mathtt{o}'.\mathtt{f}' := \mathtt{v}']$	is	$\mathtt{store}(\mathtt{store}(\mathtt{h}, \mathtt{o}, \mathtt{f}, \mathtt{v}), \mathtt{o}', \mathtt{f}', \mathtt{v}')$

Very-Shorthand Notations for Current Heap

Current heap always in special variable $\mathtt{heap}$.

$\mathtt{o.f}$	is	$\mathtt{select}(\mathtt{heap}, \mathtt{o}, \mathtt{f})$
$\{\mathtt{o.f} := \mathtt{v}\}$	is update	$\{\mathtt{heap} := \mathtt{heap}[\mathtt{o.f} := \mathtt{x}]\}$

Modelling the Heap in FOL—The Full Story

Is formula $\texttt{select}(h, \texttt{p}, \texttt{id}) >= \texttt{0}$ type-safe?

Modelling the Heap in FOL—The Full Story

Is formula $\texttt{select}(h, \texttt{p}, \texttt{id}) >= 0$ type-safe?

1. Return type is any—need to 'cast' to **int**
2. There can be many fields with name `id`

Modelling the Heap in FOL—The Full Story

Is formula $\mathtt{select}(h, \mathrm{p}, \mathtt{id}) >= 0$ type-safe?

1. Return type is any—need to 'cast' to **int**
2. There can be many fields with name `id`

Real Field Access

$\mathtt{int::select}(h, \mathrm{p}, \mathsf{Person::}\mathtt{\$id}) >= 0$ is type-safe

- `int::select` is a function name, not a cast
- can be understood *intuitively* as (**int**)`select`

Modelling the Heap in FOL—The Full Story

Is formula $\texttt{select}(h, \texttt{p}, \texttt{id}) >= 0$ type-safe?

1. Return type is any—need to 'cast' to $\mathbf{int}$
2. There can be many fields with name `id`

Real Field Access

$\texttt{int::select}(h, \texttt{p}, \text{Person::}\texttt{\$id}) >= 0$ is type-safe

- `int::select` is a function name, not a cast
- can be understood *intuitively* as $(\mathbf{int})\texttt{select}$

General

For each `T` typed field `f` of class `C`, F_Σ contains

- a constant declared as Field `C::$f`
- a function declared as `T` `T::select`(Heap, `C`, Field)

Modelling the Heap in FOL—The Full Story

Is formula $\texttt{select}(h, \texttt{p}, \texttt{id}) >= 0$ type-safe?

1. Return type is any—need to 'cast' to **int**
2. There can be many fields with name id

Real Field Access

$\texttt{int::select}(h, \texttt{p}, \text{Person::\$id}) >= 0$ is type-safe

- int::select is a function name, not a cast
- can be understood *intuitively* as (**int**)select

General

For each T typed field f of class C, F_Σ contains

- a constant declared as Field C::\$f
- a function declared as T T::select(Heap, C, Field)

Everything blue is a function name

Modelling the Heap in FOL—The Full Story

Writing to Fields

We stick to the above:

Declaration: Heap `store`(Heap, Object, Field, any)`;`

Usage: `store`(h, `p`, Person::`$id`, 42)

Field Update Assignment Rule

Changing the value of fields

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

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathtt{o.f} := t\}\langle \mathtt{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathtt{o.f\ =\ t;\ rest}\rangle\phi, \Delta}$$

Admit on left-hand side of update Java location expressions

Field Update Assignment Rule

Changing the value of fields

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

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

Admit on left-hand side of update JAVA location expressions

Field Update Assignment Rule

Changing the value of fields

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

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathtt{p.age} := 17\}\langle\mathtt{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\mathtt{p.age\ =\ 17;\ rest}\rangle\phi, \Delta}$$

Admit on left-hand side of update Java location expressions

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

```
\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`

Refined Semantics of Program Modalities

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

Refined Semantics of Program Modalities

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

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

Refined Semantics of Program Modalities

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

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

Refined Semantics of Program Modalities

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

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

Abrupt termination on top-level 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`

The Self Reference

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: `Object self;`

Default assumption in JML-KeY translation: $\mathtt{self} \mathrel{!=} \mathbf{null}$

Which Objects do Exist?

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

Which Objects do Exist?

How to model object creation with **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 **new**()

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

Object Creation

Realizing Constant Domain Assumption

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

Object Creation

Realizing Constant Domain Assumption

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

ObjectCreation(simplified)

$$\frac{\begin{array}{l}\Gamma,\ \{u\}(\mathtt{select}(\mathtt{heap},\mathtt{ob},\mathtt{Object}::\mathtt{<created>}) = \mathtt{FALSE}) \Longrightarrow \\ \quad \{u\}(\{\mathtt{heap}:=\mathtt{create}(\mathtt{heap},\mathtt{ob})\}\{\mathtt{o}:=\mathtt{ob}\}\langle\pi\,\mathtt{o.<init>(}\mathit{param}\mathtt{)};\omega\rangle\phi),\ \Delta\end{array}}{\Gamma \Longrightarrow \{u\}(\langle\pi\ \ \mathtt{o} = \mathtt{new}\ \ \mathtt{T}(\mathit{param});\ \ \omega\rangle\phi), \Delta}$$

`ob` is a fresh program variable

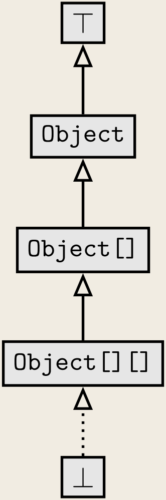
Dynamic Logic to (almost) full Java

KeY supports full sequential Java, with some limitations:

- Limited concurrency
- 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 a[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

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

Decomposition of complex terms by symbolic execution

Follow the rules laid down in JAVA Language Specification

Local code transformations

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

Temporary variables store result of evaluating subexpression

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

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

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

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

Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

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

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

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

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

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

Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

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

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

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

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

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

Demo

`exceptions/try-catch.key`

Java Features in Dynamic Logic: Aliasing

Demo

`aliasing/attributeAlias1.key`

Java Features in Dynamic Logic: Aliasing

Demo

`aliasing/attributeAlias1.key`

Reference Aliasing

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

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

Java Features in Dynamic Logic: Method Calls

Method Call

First evaluate arguments, leading to:

$$\{arg_0 := t_0 \,||\, \cdots \,||\, arg_n := t_n \,||\, c := t_c\}\langle c.\mathtt{m}(arg_0, \ldots, arg_n)\mathtt{;}\rangle\phi$$

Actions of rule methodCall

- for each formal parameter $\mathtt{p_i}$ of $\mathtt{m}$:
  declare and initialize new local variable $\tau_{\mathtt{i}}\ \mathtt{p\#i} = arg_i;$
- look up implementation class C of $\mathtt{m}$ and split proof
  if implementation cannot be uniquely determined
- create concrete method invocation $c.\mathtt{m}(\mathtt{p\#0}, \ldots, \mathtt{p\#n})\mathtt{@}C$

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\tau_{\mathtt{i}}\ \mathtt{p\#i} = arg_i;$
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \mathtt{method\text{-}frame(source=C,\ this=c)\{}\ \mathtt{body}\ \mathtt{\}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \mathtt{c.m(p\#0,\ldots,p\#n)@C;}\ \omega\rangle\phi, \Delta}$$

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\tau_{\mathtt{i}}$ $\mathtt{p\#i}$ $=arg_i$;
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi \ \mathtt{method\text{-}frame(source=C,\ this=c)\{\ body\ \}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi \ \mathtt{c.m(p\#0,\ldots,p\#n)@C;}\ \omega\rangle\phi, \Delta}$$

Demo

`methods/instanceMethodInlineSimple.key`

A Round Tour of Java Features in DL Cont'd

Localisation of Fields and Method Implementation

JAVA has complex rules for localisation of fields and method implementations

- Polymorphism
- Late binding
- Scoping (class vs. instance)
- Context (static vs. runtime)
- Visibility (private, protected, public)

Proof split into cases when implementation not statically determined

A Round Tour of Java Features in DL Cont'd

Null pointer exceptions

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

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

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

A Round Tour of Java Features in DL Cont'd

Formal specification of Java API

How to perform symbolic execution when JAVA API method is called?

1. API method has reference implementation in JAVA
   Call method and execute symbolically

   Problem Reference implementation not always available

   Problem Breaks modularity
2. Use JML contract of API method:
   - 2.1 Show that requires clause is satisfied
   - 2.2 Obtain postcondition from ensures clause
   - 2.3 Delete updates with modifiable locations from symbolic state

A Round Tour of Java Features in DL Cont'd

Formal specification of Java API

How to perform symbolic execution when JAVA API method is called?

1. API method has reference implementation in JAVA
   Call method and execute symbolically

   Problem Reference implementation not always available

   Problem Breaks modularity

2. Use JML contract of API method:
   - 2.1 Show that requires clause is satisfied
   - 2.2 Obtain postcondition from ensures clause
   - 2.3 Delete updates with modifiable locations from symbolic state

Java Card API in JML or DL

DL version available in KeY, JML work in progress See W. Mostowski

`http://limerick.cost-ic0701.org/home/verifying-java-card-programs-with-key`

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
- Proving loops requires user to provide invariant
  - Automatic invariant generation sometimes possible
- Symbolic execution paradigm lets you use KeY
  w/o understanding details of logic

Literature for this Lecture

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, 3.6.5, 3.6.7