

Abstract Object Creation in Dynamic Logic

to be or not to be created

Wolfgang Ahrendt¹ Frank S. de Boer² Immo Grabe³

¹Chalmers University, Göteborg, Sweden

²CWI, Amsterdam, The Netherlands

³Christian-Albrechts-University Kiel, Germany

KeY Symposium Speyer, 2009

Part I

Motivation and Outline

Modeling Object Creation in Program Logics

object-oriented programming languages (like Java):

- ▶ high-level way of creating objects
- ▶ abstract away from memory allocation
- ▶ programmer has no access to non-created (pre-)objects

Modeling Object Creation in Program Logics

object-oriented programming languages (like Java):

- ▶ high-level way of creating objects
- ▶ abstract away from memory allocation
- ▶ programmer has no access to non-created (pre-)objects

this abstraction not matched by program logics (incl. KeY):

- ▶ non-created objects can be referred to in the logic
- ▶ additional artifacts (ghost fields) to distinguish created objects
- ▶ consistency conditions on *reachable* states

Modeling Object Creation in Program Logics

object-oriented programming languages (like Java):

- ▶ high-level way of creating objects
- ▶ abstract away from memory allocation
- ▶ programmer has no access to non-created (pre-)objects

this abstraction not matched by program logics (incl. KeY):

- ▶ non-created objects can be referred to in the logic
- ▶ additional artifacts (ghost fields) to distinguish created objects
- ▶ consistency conditions on *reachable* states

because of mismatch:

- ▶ loose full abstraction property
- ▶ additional complexity in formulas and proofs
- ▶ symbolic state bloated by createdness information

Approach Taken

- ▶ a logic that can only 'talk about' created objects

Approach Taken

- ▶ a logic that can only 'talk about' created objects
problem:
calculus cannot 'substitute' new objects into pre-conditions

Approach Taken

- ▶ a logic that can only ‘talk about’ created objects
problem:
calculus cannot ‘substitute’ new objects into pre-conditions
- ▶ solution:
non-standard substitution using meta-knowledge about
‘newness’

Approach Taken

- ▶ a logic that can only ‘talk about’ created objects
problem:
calculus cannot ‘substitute’ new objects into pre-conditions
- ▶ solution:
non-standard substitution using meta-knowledge about
‘newness’
- ▶ carry over to symbolic execution paradigm

In the Following

- ▶ simple object-oriented while-language
- ▶ dynamic logic for that language
- ▶ abstract object creation semantics
- ▶ backwards reasoning calculus (wp-style)
- ▶ symbolic execution with abstract object creation

Relevance

- ▶ we examine object creation in simplified setting
- ▶ but: keep simplifications orthogonal to object creation issue

Relevance

- ▶ we examine object creation in simplified setting
- ▶ but: keep simplifications orthogonal to object creation issue
- ▶ applicable to full languages featuring abstract object creation (including Java)

Part II

Syntax and Semantics

A Simple Object-Oriented While Language

- ▶ only one class: Object
- ▶ 3 types: Object, Integer, Boolean
- ▶ no methods
- ▶ variables (e.g. u, v, w) distinct from fields (e.g. x, y, z)

A Simple Object-Oriented While Language

- ▶ only one class: Object
- ▶ 3 types: Object, Integer, Boolean
- ▶ no methods
- ▶ variables (e.g. u, v, w) distinct from fields (e.g. x, y, z)

statements:

$$s ::= \text{while } e \text{ do } s \text{ od} \mid \text{if } e_1 \text{ then } s_2 \text{ else } s_3 \text{ fi} \mid s_1; s_2 \mid \\ u := e \mid e_1.x := e_2 \mid u := \text{new}$$

A Simple Object-Oriented While Language

- ▶ only one class: Object
- ▶ 3 types: Object, Integer, Boolean
- ▶ no methods
- ▶ variables (e.g. u, v, w) distinct from fields (e.g. x, y, z)

statements:

$s ::= \text{while } e \text{ do } s \text{ od} \mid \text{if } e_1 \text{ then } s_2 \text{ else } s_3 \text{ fi} \mid s_1; s_2 \mid$
 $u := e \mid e_1.x := e_2 \mid u := \text{new}$

expressions:

$e ::= u \mid e.x \mid \text{null} \mid e_1 = e_2 \mid (e_1 ? e_2 : e_3) \mid \text{op}(e_1, \dots, e_n)$

A Simple Object-Oriented While Language

- ▶ only one class: Object
- ▶ 3 types: Object, Integer, Boolean
- ▶ no methods
- ▶ variables (e.g. u, v, w) distinct from fields (e.g. x, y, z)

statements:

$s ::= \text{while } e \text{ do } s \text{ od} \mid \text{if } e_1 \text{ then } s_2 \text{ else } s_3 \text{ fi} \mid s_1; s_2 \mid$
 $u := e \mid e_1.x := e_2 \mid u := \text{new}$

expressions:

$e ::= u \mid e.x \mid \text{null} \mid e_1 = e_2 \mid (e_1 ? e_2 : e_3) \mid \text{op}(e_1, \dots, e_n)$

to separate issues object creation and aliasing:

- ▶ no native statement $e.x := \text{new}$

A Simple Object-Oriented While Language

- ▶ only one class: Object
- ▶ 3 types: Object, Integer, Boolean
- ▶ no methods
- ▶ variables (e.g. u, v, w) distinct from fields (e.g. x, y, z)

statements:

$s ::= \text{while } e \text{ do } s \text{ od} \mid \text{if } e_1 \text{ then } s_2 \text{ else } s_3 \text{ fi} \mid s_1; s_2 \mid$
 $u := e \mid e_1.x := e_2 \mid u := \text{new}$

expressions:

$e ::= u \mid e.x \mid \text{null} \mid e_1 = e_2 \mid (e_1 ? e_2 : e_3) \mid \text{op}(e_1, \dots, e_n)$

to separate issues object creation and aliasing:

- ▶ no native statement $e.x := \text{new}$
- ▶ can be simulated by $u := \text{new}; e.x := u$ (u fresh)

The Logic

- ▶ expressions may also contain logical variables (e.g., l)
- ▶ boolean expressions are formulas
- ▶ true, false are formulas
- ▶ logical connectives $\wedge, \vee, \rightarrow, \neg$
- ▶ quantified formulas $\forall l.\phi, \exists l.\phi$
- ▶ modal formulas (base cases):
 $\langle s \rangle \phi, [s] \phi, \{ \mathcal{U} \} \phi,$

with s a statement and \mathcal{U} (singular) update of form:

- ▶ $u := e$
- ▶ $e_1.x := e_2$
- ▶ $u := \text{new}$

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_{\sigma}$: create new object and assign it to u

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_{\sigma}$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_{\sigma}$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

- ▶ $\llbracket e \rrbracket_{\sigma} \in$ **current references**

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_\sigma$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

- ▶ $\llbracket e \rrbracket_\sigma \in$ **current references**
- ▶ $\llbracket \forall l. \phi \rrbracket_\sigma$: ϕ holds for *all* **current references** l

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_{\sigma}$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

- ▶ $\llbracket e \rrbracket_{\sigma} \in$ **current references**
- ▶ $\llbracket \forall l. \phi \rrbracket_{\sigma}$: ϕ holds for *all* **current references** l
- ▶ $\llbracket \exists l. \phi \rrbracket_{\sigma}$: ϕ holds for *some* **current reference** l

e, l of type Object

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_\sigma$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

- ▶ $\llbracket e \rrbracket_\sigma \in$ **current references**
- ▶ $\llbracket \forall l. \phi \rrbracket_\sigma$: ϕ holds for *all* **current references** l
- ▶ $\llbracket \exists l. \phi \rrbracket_\sigma$: ϕ holds for *some* **current reference** l

e, l of type Object

examples:

$\forall l. \langle u := \text{new} \rangle \neg (u = l)$

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_\sigma$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

- ▶ $\llbracket e \rrbracket_\sigma \in$ **current references**
- ▶ $\llbracket \forall l. \phi \rrbracket_\sigma$: ϕ holds for *all* **current references** l
- ▶ $\llbracket \exists l. \phi \rrbracket_\sigma$: ϕ holds for *some* **current reference** l

e, l of type Object

examples:

$\forall l. \langle u := \text{new} \rangle \neg (u = l)$ true in all states

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_\sigma$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

- ▶ $\llbracket e \rrbracket_\sigma \in$ **current references**
- ▶ $\llbracket \forall l. \phi \rrbracket_\sigma$: ϕ holds for *all* **current references** l
- ▶ $\llbracket \exists l. \phi \rrbracket_\sigma$: ϕ holds for *some* **current reference** l

e, l of type Object

examples:

$\forall l. \langle u := \text{new} \rangle \neg (u = l)$ true in all states

$\langle u := \text{new} \rangle \forall l. \neg (u = l)$

Semantics

informal in this talk

- ▶ $\llbracket u := \text{new} \rrbracket_\sigma$: create new object and assign it to u

terminology:

in a state σ : **current references** = **created objects** plus **null**

- ▶ $\llbracket e \rrbracket_\sigma \in$ **current references**
- ▶ $\llbracket \forall l. \phi \rrbracket_\sigma$: ϕ holds for *all* **current references** l
- ▶ $\llbracket \exists l. \phi \rrbracket_\sigma$: ϕ holds for *some* **current reference** l

e, l of type Object

examples:

$\forall l. \langle u := \text{new} \rangle \neg (u = l)$ true in all states

$\langle u := \text{new} \rangle \forall l. \neg (u = l)$ false in all states

Part III

Calculus

Sequent Calculus

rules triggered by *top-level formulas* only:

- ▶ propositional rules, first-order rules, induction
- ▶ all these are standard!

Sequent Calculus

rules triggered by *top-level formulas* only:

- ▶ propositional rules, first-order rules, induction
- ▶ all these are standard!
- ▶ in particular: **quantifier rules are standard!**

Sequent Calculus

rules triggered by *top-level formulas* only:

- ▶ propositional rules, first-order rules, induction
- ▶ all these are standard!
- ▶ in particular: **quantifier rules are standard!**

rules triggered also by *sub-formulas*:

- ▶ program rules, update application rule
- ▶ notation used:

$$\frac{[\phi']}{[\phi]}$$

meaning:

premis obtained from conclusion by replacing any ϕ with ϕ'

Sequent Calculus

rules triggered by *top-level formulas* only:

- ▶ propositional rules, first-order rules, induction
- ▶ all these are standard!
- ▶ in particular: **quantifier rules are standard!**

rules triggered also by *sub-formulas*:

- ▶ program rules, update application rule
- ▶ notation used:

$$\frac{[\phi']}{[\phi]}$$

meaning:

premis obtained from conclusion by replacing any ϕ with ϕ'
(`\find(ϕ) \replacewith(ϕ')`)

Dynamic Logic Rules

$$\textit{split} \frac{\lfloor \langle s_1 \rangle \langle s_2 \rangle \phi \rfloor}{\lfloor \langle s_1; s_2 \rangle \phi \rfloor} \quad \textit{if} \frac{\lfloor (e \rightarrow \langle s_1 \rangle \phi) \wedge (\neg e \rightarrow \langle s_2 \rangle \phi) \rfloor}{\lfloor \langle \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi} \rangle \phi \rfloor}$$

$$\textit{unwind} \frac{\lfloor \langle \text{if } e \text{ then } s; \text{ while } e \text{ do } s \text{ od else skip fi} \rangle \phi \rfloor}{\lfloor \langle \text{while } e \text{ do } s \text{ od} \rangle \phi \rfloor}$$

$$\textit{assignVar} \frac{\lfloor \{u := e\} \phi \rfloor}{\lfloor \langle u := e \rangle \phi \rfloor} \quad \textit{assignField} \frac{\lfloor \{e_1.x := e_2\} \phi \rfloor}{\lfloor \langle e_1.x := e_2 \rangle \phi \rfloor}$$

$$\textit{createObj} \frac{\lfloor \{u := \text{new}\} \phi \rfloor}{\lfloor \langle u := \text{new} \rangle \phi \rfloor}$$

Update Application Rule

for certain formulas $\{\mathcal{U}\}\phi$, the \mathcal{U} can be '*applied*' (resolved)

Update Application Rule

for certain formulas $\{\mathcal{U}\}\phi$, the \mathcal{U} can be ‘*applied*’ (resolved)

$$\text{applyUpd} \frac{[\phi']}{[\{\mathcal{U}\}\phi]}$$

if $\{\mathcal{U}\}\phi \rightsquigarrow \phi'$

Update Application Rule

for certain formulas $\{\mathcal{U}\}\phi$, the \mathcal{U} can be ‘*applied*’ (resolved)

$$\text{applyUpd} \frac{[\phi']}{[\{\mathcal{U}\}\phi]}$$

if $\{\mathcal{U}\}\phi \rightsquigarrow \phi'$

now define relation \rightsquigarrow , resolving updates *in a single step*

following slides: *big-step* definition of \rightsquigarrow

Part IV

Update Application

Update Application: Standard Cases I

$$\frac{\neg\{\mathcal{U}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}\}(\neg\phi) \rightsquigarrow \phi'}$$

$$\frac{\{\mathcal{U}\}\phi_1 * \{\mathcal{U}\}\phi_2 \rightsquigarrow \phi'}{\{\mathcal{U}\}(\phi_1 * \phi_2) \rightsquigarrow \phi'}$$

with $* \in \{\wedge, \vee, \rightarrow\}$

$$\frac{op(\{\mathcal{U}\}e_1, \dots, \{\mathcal{U}\}e_n) \rightsquigarrow e'}{\{\mathcal{U}\}op(e_1, \dots, e_n) \rightsquigarrow e'}$$

$$\frac{(\{\mathcal{U}\}e_1 ? \{\mathcal{U}\}e_2 : \{\mathcal{U}\}e_3) \rightsquigarrow e'}{\{\mathcal{U}\}(e_1 ? e_2 : e_3) \rightsquigarrow e'}$$

$$\{\mathcal{U}\}\alpha \rightsquigarrow \alpha$$

with $\alpha \in \{\text{true}, \text{false}, \text{null}, I\}$

this slide: \mathcal{U} matches *all* updates

Update Application: Standard Cases II

$$\{u := e\}u \rightsquigarrow e$$

$$\frac{\{u := \alpha\}v \rightsquigarrow v \quad (\{u := e_1\}e_2).x \rightsquigarrow e'}{u \neq v \quad \alpha \equiv e \mid \text{new} \quad \{u := e_1\}(e_2.x) \rightsquigarrow e'}$$

$$\frac{((\{e.x := e_1\}e_2) = e ? e_1 : (\{e.x := e_1\}e_2).x) \rightsquigarrow e'}{\{e.x := e_1\}(e_2.x) \rightsquigarrow e'}$$

$$\frac{(\{e.x := e_1\}e_2).y \rightsquigarrow e'}{\{e.x := e_1\}(e_2.y) \rightsquigarrow e'}$$
$$x \neq y$$

Update Application: Restricted Standard Cases

The standard rules for *quantifiers* and *equality* are restricted to **non-creating updates** \mathcal{U}_{nc} of the forms ' $u := e$ ', ' $e_1.x := e_2$ '. (' $u := \text{new}$ ' excluded from these rules.)

$$\frac{\forall l. \{\mathcal{U}_{nc}\} \phi \rightsquigarrow \phi'}{\{\mathcal{U}_{nc}\}(\forall l. \phi) \rightsquigarrow \phi'}$$

$$\frac{\exists l. \{\mathcal{U}_{nc}\} \phi \rightsquigarrow \phi'}{\{\mathcal{U}_{nc}\}(\exists l. \phi) \rightsquigarrow \phi'}$$

$$\frac{\{\mathcal{U}_{nc}\} e_1 = \{\mathcal{U}_{nc}\} e_2 \rightsquigarrow e'}{\{\mathcal{U}_{nc}\}(e_1 = e_2) \rightsquigarrow e'}$$

Object Creating Update Application: the Issue

recall:

- ▶ ' $\{\mathcal{U}\}\phi$ ' is the (explicit) **weakest precondition** $wp(\mathcal{U}, \phi)$
- ▶ applying \mathcal{U} to ϕ (via \rightsquigarrow) **computes** weakest precondition

Object Creating Update Application: the Issue

recall:

- ▶ ' $\{U\}\phi$ ' is the (explicit) **weakest precondition** $wp(U, \phi)$
- ▶ applying U to ϕ (via \rightsquigarrow) **computes** weakest precondition

problem:

- ▶ result of $\{u := new\}\phi$, i.e., $wp(\{u := new\}, \phi)$, cannot talk about new object because it does not exist in pre-state
- ▶ in particular: $\{u := new\}u \rightsquigarrow ?$

Object Creating Update Application: the Issue

recall:

- ▶ ' $\{U\}\phi$ ' is the (explicit) **weakest precondition** $wp(U, \phi)$
- ▶ applying U to ϕ (via \rightsquigarrow) **computes** weakest precondition

problem:

- ▶ result of $\{u := new\}\phi$, i.e., $wp(\{u := new\}, \phi)$, cannot talk about new object because it does not exist in pre-state
- ▶ in particular: $\{u := new\}u \rightsquigarrow ?$

basic approach:

- ▶ **totally avoid** ' $\{u := new\}u$ '

Object Creating Update Application: the Issue

recall:

- ▶ ' $\{U\}\phi$ ' is the (explicit) **weakest precondition** $wp(U, \phi)$
- ▶ applying U to ϕ (via \rightsquigarrow) **computes** weakest precondition

problem:

- ▶ result of $\{u := new\}\phi$, i.e., $wp(\{u := new\}, \phi)$, cannot talk about new object because it does not exist in pre-state
- ▶ in particular: $\{u := new\}u \rightsquigarrow ?$

basic approach:

- ▶ **totally avoid** ' $\{u := new\}u$ '
- ▶ observation: the **only operations on objects** are
 - ▶ de-referencing fields
 - ▶ test for equality

Object Creating Update Application: the Issue

recall:

- ▶ ' $\{U\}\phi$ ' is the (explicit) **weakest precondition** $wp(U, \phi)$
- ▶ applying U to ϕ (via \rightsquigarrow) **computes** weakest precondition

problem:

- ▶ result of $\{u := new\}\phi$, i.e., $wp(\{u := new\}, \phi)$, cannot talk about new object because it does not exist in pre-state
- ▶ in particular: $\{u := new\}u \rightsquigarrow ?$

basic approach:

- ▶ **totally avoid** ' $\{u := new\}u$ '
- ▶ observation: the **only operations on objects** are
 - ▶ de-referencing fields
 - ▶ test for equality
 - ▶ quantification

Object Creating Update Application: the Issue

recall:

- ▶ ' $\{U\}\phi$ ' is the (explicit) **weakest precondition** $wp(U, \phi)$
- ▶ applying U to ϕ (via \rightsquigarrow) **computes** weakest precondition

problem:

- ▶ result of $\{u := new\}\phi$, i.e., $wp(\{u := new\}, \phi)$, cannot talk about new object because it does not exist in pre-state
- ▶ in particular: $\{u := new\}u \rightsquigarrow ?$

basic approach:

- ▶ **totally avoid** ' $\{u := new\}u$ '
- ▶ observation: the **only operations on objects** are
 - ▶ de-referencing fields
 - ▶ test for equality
 - ▶ quantification
- ▶ in all cases, wp computation can employ meta knowledge

Object Creating Update Application: Field Access

$$\frac{(\{u := new\}e).x \rightsquigarrow e'}{\{u := new\}(e.x) \rightsquigarrow e'}$$

e neither u nor conditional

$$\{u := new\}u.x \rightsquigarrow init_{T(x)}$$

$$init_{T(x)} \equiv \text{null} \mid 0 \mid \text{false}$$

$$\frac{(\{u := new\}b ? \{u := new\}(e_1.x) : \{u := new\}(e_2.x)) \rightsquigarrow e'}{\{u := new\}((b ? e_1 : e_2).x) \rightsquigarrow e'}$$

Object Creating Update Application: Equality

$$\frac{(\{u := new\}e_1) = (\{u := new\}e_2) \rightsquigarrow e'}{\{u := new\}(e_1 = e_2) \rightsquigarrow e'}$$

e_1, e_2 neither u nor conditional

$$\{u := new\}(u = e) \rightsquigarrow \text{false}$$

e neither u nor conditional

$$\{u := new\}(u = u) \rightsquigarrow \text{true}$$

$$\frac{(\{u := new\}b ? \{u := new\}(e_1 = e_3) : \{u := new\}(e_2 = e_3)) \rightsquigarrow e'}{\{u := new\}((b ? e_1 : e_2) = e_3) \rightsquigarrow e'}$$

Object Creating Update Application: Quantifiers

$$\frac{(\{u := \text{new}\}\phi[l/u]) \wedge \forall l. (\{u := \text{new}\}\phi) \rightsquigarrow \phi'}{\{u := \text{new}\}\forall l. \phi \rightsquigarrow \phi'}$$

Object Creating Update Application: Quantifiers

$$\frac{(\{u := new\}\phi[l/u]) \wedge \forall l. (\{u := new\}\phi) \rightsquigarrow \phi'}{\{u := new\}\forall l. \phi \rightsquigarrow \phi'}$$

$$\frac{(\{u := new\}\phi[l/u]) \vee \exists l. (\{u := new\}\phi) \rightsquigarrow \phi'}{\{u := new\}\exists l. \phi \rightsquigarrow \phi'}$$

Example Proof 1

$$\begin{array}{c} \text{closeFalse} \frac{*}{\text{false} \Rightarrow} \\ \text{notRight} \frac{\text{false} \Rightarrow}{\Rightarrow \neg \text{false}} \\ \text{applyUpd} \frac{\Rightarrow \neg \text{false}}{\Rightarrow \{u := \text{new}\} \neg (u = c)} \\ \text{assignVar} \frac{\Rightarrow \{u := \text{new}\} \neg (u = c)}{\Rightarrow \langle u := \text{new} \rangle \neg (u = c)} \\ \text{allRight} \frac{\Rightarrow \langle u := \text{new} \rangle \neg (u = c)}{\Rightarrow \forall l. (\langle u := \text{new} \rangle \neg (u = l))} \end{array}$$

Example Proof 2

$$\begin{array}{l} \text{closeTrue} \frac{\quad *}{\forall l. \neg \text{false} \Rightarrow \text{true}} \\ \text{notLeft} \frac{\forall l. \neg \text{false} \Rightarrow \text{true}}{\neg(\text{true}), \forall l. \neg \text{false} \Rightarrow} \\ \text{andLeft} \frac{\neg(\text{true}), \forall l. \neg \text{false} \Rightarrow}{\neg(\text{true}) \wedge \forall l. \neg \text{false} \Rightarrow} \\ \text{applyUpd} \frac{\neg(\text{true}) \wedge \forall l. \neg \text{false} \Rightarrow}{\{u := \text{new}\} \forall l. \neg(u = l) \Rightarrow} \\ \text{assignVar} \frac{\{u := \text{new}\} \forall l. \neg(u = l) \Rightarrow}{\langle u := \text{new} \rangle \forall l. \neg(u = l) \Rightarrow} \\ \text{notRight} \frac{\langle u := \text{new} \rangle \forall l. \neg(u = l) \Rightarrow}{\Rightarrow \neg \langle u := \text{new} \rangle \forall l. \neg(u = l)} \end{array}$$

(applyUpd step in Example Proof 2)

$$\frac{\frac{\{u := new\}(u = u) \rightsquigarrow true}{\{u := new\}\neg(u = u) \rightsquigarrow \neg(true)} \quad \frac{\frac{\{u := new\}(u = l) \rightsquigarrow false}{\{u := new\}\neg(u = l) \rightsquigarrow \neg(false)}}{\forall l.\{u := new\}\neg(u = l) \rightsquigarrow \forall l.\neg false}}{\frac{\{u := new\}\neg(u = u) \wedge \forall l.\{u := new\}\neg(u = l) \rightsquigarrow \neg(true) \wedge \forall l.\neg false}{\{u := new\}\forall l.\neg(u = l) \rightsquigarrow \neg(true) \wedge \forall l.\neg false}}$$

Part V

Abstract Object Creation in Symbolic Execution

KeY-style Symbolic Execution

up to here, backwards reasoning only

KeYapproach:

forward symbolic execution using update parallelisation

KeY-style Symbolic Execution

up to here, backwards reasoning only

KeYapproach:

forward symbolic execution using update parallelisation

$$\begin{array}{l} \text{close} \frac{*}{u < v \Rightarrow u < v} \\ \text{applyUpd} \frac{}{u < v \Rightarrow \{w := u \mid u := v \mid v := u\} v < u} \\ \text{mergeUpd} \frac{}{u < v \Rightarrow \{w := u \mid u := v\} \{v := w\} v < u} \\ \text{assignVar} \frac{}{u < v \Rightarrow \{w := u \mid u := v\} \langle v := w \rangle v < u} \\ \text{mergeUpd, assignVar} \frac{}{u < v \Rightarrow \{w := u\} \langle u := v \rangle \langle v := w \rangle v < u} \\ \text{split, assignVar} \frac{}{u < v \Rightarrow \{w := u\} \langle u := v; v := w \rangle v < u} \\ \text{split, assignVar} \frac{}{u < v \Rightarrow \langle w := u; u := v; v := w \rangle v < u} \end{array}$$

Problem: Parallelising Object Creating Updates

no natural way of merging $\{u := \text{new}\}$ with other updates

consider the two formulas (one true, one false):

$$\langle u := \text{new}; v := u \rangle (u = v) \quad \langle u := \text{new}; v := \text{new} \rangle (u = v)$$

symbolic execution generates:

$$\{u := \text{new}\}\{v := u\}(u = v) \quad \{u := \text{new}\}\{v := \text{new}\}(u = v)$$

merging updates, both result in:

$$\{u := \text{new} \mid v := \text{new}\}(u = v)$$

cannot be true and false

Solution

- ▶ **not merge** object creation with other updates

Solution

- ▶ **not merge** object creation with other updates
- ▶ split $\{u := \text{new}\}$ into creation and (mergable) assignment to u

Solution

- ▶ **not merge** object creation with other updates
- ▶ split $\{u := \text{new}\}$ into creation and (mergable) assignment to u

new object creation rule:

$$\text{createObj} \frac{[\{a := \text{new}\}\{u := a\}\phi]}{[\langle u := \text{new} \rangle \phi]}$$

a a fresh program variable

Solution

- ▶ **not merge** object creation with other updates
- ▶ split $\{u := \text{new}\}$ into creation and (mergable) assignment to u

new object creation rule:

$$\text{createObj} \frac{[\{a := \text{new}\}\{u := a\}\phi]}{[\langle u := \text{new} \rangle \phi]}$$

a a fresh program variable

facilitate merging of all non-creating updates by shifting creation

$$\text{shiftCreation} \frac{[\{u := \text{new}\}\{\mathcal{U}_{nc}\}\phi]}{[\{\mathcal{U}_{nc}\}\{u := \text{new}\}\phi]}$$

u not appearing in (non-creating) \mathcal{U}_{nc}

Symbolic Execution Proof

$$\begin{array}{l}
 \text{notRight, closeFalse} \xrightarrow{*} \Rightarrow \neg \text{false} \\
 \text{applyUpd} \xrightarrow{\quad} \Rightarrow \{a := \text{new}\} \neg (v = a) \\
 \text{applyUpd} \xrightarrow{\quad} \Rightarrow \{a := \text{new}\} \{u := v \mid v := a \mid w := u\} \neg (w = v) \\
 \text{assignVar, mergeUpd} \xrightarrow{\quad} \Rightarrow \{a := \text{new}\} \{u := v \mid v := a\} \langle w := u \rangle \neg (w = v) \\
 \text{mergeUpd} \xrightarrow{\quad} \Rightarrow \{a := \text{new}\} \{u := v \mid v := a\} \langle w := u \rangle \neg (w = v) \\
 \text{shiftCreation} \xrightarrow{\quad} \Rightarrow \{a := \text{new}\} \{u := v\} \{v := a\} \langle w := u \rangle \neg (w = v) \\
 \text{createObj} \xrightarrow{\quad} \Rightarrow \{u := v\} \{a := \text{new}\} \{v := a\} \langle w := u \rangle \neg (w = v) \\
 \text{split, assignVar, split} \xrightarrow{\quad} \Rightarrow \{u := v\} \langle v := \text{new} \rangle \langle w := u \rangle \neg (w = v) \\
 \text{split, assignVar, split} \xrightarrow{\quad} \Rightarrow \langle u := v; v := \text{new}; w := u \rangle \neg (w = v)
 \end{array}$$

Part VI

Object Creation vs. Object Activation

Abstract Object Creation Proof

reconsider proof from above

$$\begin{array}{l} \text{closeFalse} \frac{*}{\text{false} \Rightarrow} \\ \text{notRight} \frac{\text{false} \Rightarrow}{\Rightarrow \neg \text{false}} \\ \text{applyUpd} \frac{\Rightarrow \{u := \text{new}\} \neg (u = c)}{\Rightarrow \langle u := \text{new} \rangle \neg (u = c)} \\ \text{assignVar} \frac{\Rightarrow \langle u := \text{new} \rangle \neg (u = c)}{\Rightarrow \forall l. (\langle u := \text{new} \rangle \neg (u = l))} \\ \text{allRight} \end{array}$$

Object Activation Proof

$$\begin{array}{c}
 \text{close} \frac{\quad}{c.\text{cre}, \text{obj}(\text{next}) = c \Rightarrow c.\text{cre}} \quad * \\
 \text{equality} \frac{\quad}{c.\text{cre}, \text{obj}(\text{next}) = c \Rightarrow \text{obj}(\text{next}).\text{cre}} \\
 \text{notLeft} \frac{\quad}{\neg \text{obj}(\text{next}).\text{cre}, c.\text{cre}, \text{obj}(\text{next}) = c \Rightarrow} \\
 (\approx 2 \text{ rules}) \frac{\quad}{(\text{obj}(\text{next}).\text{cre} \leftrightarrow \text{next} < \text{next}), c.\text{cre}, \text{obj}(\text{next}) = c \Rightarrow} \\
 \text{allLeft} \frac{\quad}{\forall n. (\text{obj}(n).\text{cre} \leftrightarrow n < \text{next}), c.\text{cre}, \text{obj}(\text{next}) = c \Rightarrow} \\
 \text{inReachableState} \frac{\quad}{c.\text{cre}, \text{obj}(\text{next}) = c \Rightarrow} \\
 \text{notRight} \frac{\quad}{c.\text{cre} \Rightarrow \neg(\text{obj}(\text{next}) = c)} \\
 \text{applyUpd} \frac{\quad}{c.\text{cre} \Rightarrow \{u := \text{obj}(\text{next}); u.\text{cre} := \text{true}; \text{next} := \text{next} + 1\} \neg(u = c)} \\
 \text{createObj} \frac{\quad}{c.\text{cre} \Rightarrow \langle u := \text{new} \rangle \neg(u = c)} \\
 \text{impRight} \frac{\quad}{\Rightarrow c.\text{cre} \rightarrow \langle u := \text{new} \rangle \neg(u = c)} \\
 \text{allRight} \frac{\quad}{\Rightarrow \forall l. (l.\text{cre} \rightarrow \langle u := \text{new} \rangle \neg(u = l))}
 \end{array}$$

Part VII

Reflections

Reflections

- ▶ abstraction level of logic matches programming language
- ▶ changes to standard treatment very local
 - ▶ additional update type,
not mergable with others, but shiftable to the front
 - ▶ update application differs only in few cases
- ▶ formulas and proofs are simpler
- ▶ symbolic state representation:
 - ▶ not diluted by createdness bookkeeping
 - ▶ separates out
 1. newly created objects (shifted forward)
 2. symbolic value of fields and variables