On a Specification-oriented Model for Object-orientation

Ana Cavalcanti and David Naumann
naumann@cs.stevens-tech.edu

University of Kent, Canterbury, UK
and Stevens Institute of Technology, New Jersey, USA
Co-op project (CNPq-NSF) participants:
Paulo Borba (UFPE), Ana Cavalcanti (UFPE/Kent),
David Naumann (Stevens Tech), Uday Reddy
(Birmingham), Leila Silva (UFS), Augusto Sampaio
(UFPE)
Project goals:

- formalize refactoring transformations
- behavioral interface specification incl. callbacks
- tools for correct transformation: refactoring,
  compilation, development by stepwise refinement

Goal of talk: introduce predicate transformer
semantics and discuss alternatives
**Approach: refinement**

- mixed specifications and code:
  
  **specification statement** $y: [x \geq 0, \ x = y]$ means
  
  requires $x \geq 0$, ensures $x = y$, modifies only $y$
Approach: refinement

- mixed specifications and code:
  specification statement $y: [x>0, x=y]$ means requires $x>0$, ensures $x=y$, modifies only $y$

- correctness as refinement:
  $y: [x>0, x=y] \sqsubseteq y:=x$
Approach: refinement

- mixed specifications and code:
  specification statement \( y: [x \geq 0, \ x = y] \) means requires \( x \geq 0 \), ensures \( x = y \), modifies only \( y \)

- correctness as refinement:
  \( y: [x \geq 0, \ x = y] \subseteq y := x \)

- specify callbacks using code with method calls; avoid over-specification using spec statements
Approach: refinement

• mixed specifications and code:
  specification statement $y: [x>=0, x=y]$ means
  requires $x>=0$, ensures $x=y$, modifies only $y$

• correctness as refinement:
  $y: [x>=0, x=y] \subseteq y:=x$

• specify callbacks using code with method calls;
  avoid over-specification using spec statements

• refactoring as refinement
  class Person{int phone; String street;...}
  $\subseteq$
  class Addr{int phone; String street;...}
  class Person{Addr a;...}
Approach: refinement

- mixed specifications and code:
  specification statement \( y : [x \geq 0, \ x = y] \) means requires \( x \geq 0 \), ensures \( x = y \), modifies only \( y \)

- correctness as refinement:
  \( y : [x \geq 0, \ x = y] \subseteq y := x \)

- specify callbacks using code with method calls;
  avoid over-specification using spec statements

- refactoring as refinement
  class Person{int phone; String street;...}
  \( \subseteq \)
  class Addr{int phone; String street;...}
  class Person{Addr a;...}

- normal-form compilation source \( \subseteq \ldots \subseteq \) object
Organization

- semantics ...used to prove:
- basic laws (algebraic semantics) and simulation theorem ...used to prove:
- laws for refactoring, for compilation, etc.
- tools: can implement basic laws; refactoring etc. can be strategy for using derived laws

Language (first phase of project, this talk): imperative (sequential) commands, specification statements, classes and inheritance, visibility control, dynamic dispatch, and recursive classes and methods – but copy semantics, not pointers.

Current phase: adding pointers, interfaces, etc.
Operational semantics

\[(x := e): \sigma \longrightarrow \sigma \oplus \{x \mapsto [e]_\sigma\}\]

\[
\frac{c_1 : \sigma \longrightarrow \sigma' \quad c_2 : \sigma' \longrightarrow \sigma''}{c_1; c_2 : \sigma \longrightarrow \sigma''}
\]

Not compositional, so hard to use in proving laws; doesn’t handle spec statements; but easy to handle many language features [Nipkow et al,...]
Operational semantics

\[(x := e) : \sigma \longrightarrow \sigma \oplus \{x \mapsto [e]_{\sigma}\}\]

\[
\frac{c_1 : \sigma \longrightarrow \sigma' \quad c_2 : \sigma' \longrightarrow \sigma''}{c_1; c_2 : \sigma \longrightarrow \sigma''}
\]

Not compositional, so hard to use in proving laws; doesn’t handle spec statements; but easy to handle many language features [Nipkow et al,...]

Denotational I: state transformers

\[[c] : states \rightarrow states\]

\[[x := e]_{\sigma} = \sigma \oplus \{x \mapsto [e]_{\sigma}\}\]

\[[c_1; c_2]_{\sigma} = ([c_2] \circ [c_1])_{\sigma}\]

Clear operational interpretation, compositional [Jacobs, Poll, et al,...]; doesn’t handle spec statements; weakest preconditions are needed for program verification.
Predicate transformer semantics

\([c] : \mathcal{P} \text{states} \rightarrow \mathcal{P} \text{states}\)

\([c] \psi \) is weakest precondition to ensure \(\psi\)

inverse image: \(\sigma \in [c] \psi \) iff \(\{c\}\sigma \in \psi \) (for all \(\sigma\))
Predicate transformer semantics

\[ [c] : \mathcal{P} \text{states} \rightarrow \mathcal{P} \text{states} \]

\([c] \psi \text{ is weakest precondition to ensure } \psi\]

inverse image: \( \sigma \in [c] \psi \text{ iff } \{c\} \sigma \in \psi \) (for all \( \sigma \))

Using formulas to describe sets of states (\([Q] = \psi\))

\[ [x := e] Q = Q[e/x] \]
\[ [c1; c2] Q = ([c1] \circ [c2]) Q \]
Predicate transformer semantics

\[ [c] : \mathbb{P} \text{ states} \rightarrow \mathbb{P} \text{ states} \]

\[ [c] \psi \text{ is weakest precondition to ensure } \psi \]

inverse image: \( \sigma \in [c] \psi \iff \{[c]\} \sigma \in \psi \) (for all \( \sigma \))

Using formulas to describe sets of states (\( [Q] = \psi \))

\[ [x := e] Q = Q[e/x] \]

\[ [c_1; c_2] Q = ([c_1] \circ [c_2]) Q \]

\[ [x : [P, R]] Q = P \land (\forall x \bullet R \implies Q) \]
Denotational II

Predicate transformer semantics
\[
[c] : \mathcal{P} \text{ states } \rightarrow \mathcal{P} \text{ states}
\]
\[\boxed{c} \psi \text{ is weakest precondition to ensure } \psi\]

inverse image: \( \sigma \in [c] \psi \iff \{c\} \sigma \in \psi \) (for all \( \sigma \))

Using formulas to describe sets of states \( [Q] = \psi \)
\[
[x := e] Q = Q[e/x]
\]
\[
[c_1;c_2] Q = ([c_1] \circ [c_2]) Q
\]
\[
[x : [P,R]] Q = P \land (\forall x \cdot R \Rightarrow Q)
\]
Challenges for extending previous work:
- not a fixed global state space
- recursive classes: state of Person may involve state of other Person objects, e.g., children
- dynamic binding
- previous work mostly semi-formal
Formalizing complex language

Judgements for well formed commands:
\[ \Gamma, \Sigma, N \triangleright c : \text{com} \]

- \( N \) is class declaring method containing \( c \)
- \( \Gamma \) is symbol table describing all available classes
- \( \Sigma \) declares variables (locals, method parameters, and visible attributes of \( N \))
Formalizing complex language

Judgements for well formed commands:
\[ \Gamma, \Sigma, N \triangleright c : \text{com} \]

- \( N \) is class declaring method containing \( c \)
- \( \Gamma \) is symbol table describing all available classes
- \( \Sigma \) declares variables (locals, method parameters, and visible attributes of \( N \))

Semantics, first approximation
\[ [\Gamma, \Sigma, N \triangleright c : \text{com}] : \mathcal{P} [\Gamma, \Sigma, N] \rightarrow \mathcal{P} [\Gamma, \Sigma, N] \]

where \([\Gamma, \Sigma, N]\) is the set of appropriate states.
Formalizing complex language

Judgements for well formed commands:
\[ \Gamma, \Sigma, N \triangleright c : \text{com} \]
- \( N \) is class declaring method containing \( c \)
- \( \Gamma \) is symbol table describing all available classes
- \( \Sigma \) declares variables (locals, method parameters, and visible attributes of \( N \))

Semantics, first approximation
\[ [\Gamma, \Sigma, N \triangleright c : \text{com}] : \mathbb{P} [\Gamma, \Sigma, N] \rightarrow \mathbb{P} [\Gamma, \Sigma, N] \]
where \([\Gamma, \Sigma, N]\) is the set of appropriate states.

Actual semantics
\[ [\Gamma, \Sigma, N \triangleright c : \text{com}] : \text{MethodEnv} \rightarrow \mathbb{P} [\Gamma, \Sigma, N] \rightarrow \mathbb{P} [\Gamma, \Sigma, N] \]
A method environment \( \eta \) provides, for each \( N \) and \( m \), the meaning \( \eta N m \) of method \( m \) for objects of dynamic type \( N \).
**Semantic definitions**

\[
[\Gamma, \Sigma, N \triangleright e : T'] = f \\
\sigma \in [\Gamma, \Sigma, N \triangleright x := e : \text{com}] \eta \psi \iff f \sigma \neq \text{error} \land \sigma \in \psi[f \sigma // x]
\]
Semantic definitions

\[ [\Gamma, \Sigma, N \triangleright e : T'] = f \]
\[ \sigma \in [\Gamma, \Sigma, N \triangleright x := e : \text{com}] \eta \psi \leftrightarrow f \sigma \neq \text{error} \land \sigma \in \psi[f \sigma \parallel x] \]

\[ [\Gamma, \Sigma, N \triangleright c1 : \text{com}] \eta = f1 \quad [\Gamma, \Sigma, N \triangleright c2 : \text{com}] \eta = f2 \]
\[ [\Gamma, \Sigma, N \triangleright c1; c2 : \text{com}] \eta \psi = f1(f2 \psi) \]
Semantic definitions

\[
\begin{align*}
[\Gamma, \Sigma, N \triangleright e : T'] &= f \\
\sigma \in [\Gamma, \Sigma, N \triangleright x := e : \text{com}] &\quad \eta \psi \iff f \sigma \neq \text{error} \land \sigma \in \psi[f \sigma / x]
\end{align*}
\]

\[
\begin{align*}
[\Gamma, \Sigma, N \triangleright c1 : \text{com}] &\eta = f1 \\
[\Gamma, \Sigma, N \triangleright c2 : \text{com}] &\eta = f2 \\
[\Gamma, \Sigma, N \triangleright c1; c2 : \text{com}] &\eta \psi = f1(f2 \psi)
\end{align*}
\]

These are relatively simple, because the typing context determines the state space.

For method call \(\Gamma, \Sigma, N \triangleright x.m(e)\) with \(x : N'\), the environment gives a meaning \(\eta N' m\) in the state space of \(N'\), not in \(\Sigma, N\).
**Self call without parameters**

Environment has method meaning that acts on state of target object.

**State-transformer semantics:**

\[
\{[\Gamma, ((\text{attr } \Gamma N); \Sigma), N \triangleright \text{self}.m() : \text{com}]\} \eta \sigma \\
= (st (\Sigma \triangleleft \sigma)) \oplus (\Sigma \mapsto \sigma(\Sigma))
\]

where \(N' = \sigma \text{myclass}\) and \(st = \eta N' m\)

**Note:** \(\Sigma\) is parameters and locals, \((\text{vattr } \Gamma N)\) are attributes.
Self call without parameters

Environment has method meaning that acts on state of target object.

State-transformer semantics:

\[
\{ [\Gamma, ((\text{attr } \Gamma N); \Sigma), N \triangleright \text{self.m}() : \text{com}] \eta \sigma
= (\text{st } (\Sigma \leftrightarrow \sigma)) \oplus (\Sigma \mapsto \sigma(\Sigma))
\]

where \( N' = \sigma \text{ myclass} \) and \( \text{st} = \eta N' \ m \)

Note: \( \Sigma \) is parameters and locals, \( (\text{vattr } \Gamma N) \) are attributes.

Predicate-transformer semantics:

\[
\sigma \in [\Gamma, ((\text{vattr } \Gamma N); \Sigma), N \triangleright \text{self.m}() : \text{com}] \eta \psi
\Rightarrow \sigma \in \text{lift } \Sigma \ \text{pt } \psi
\]

where \( N' = \sigma \text{ myclass} \) and \( \text{pt} = \eta N' \ m \)
Using formulas

\[
[\Gamma, \ldots, N \triangleright \text{self}.m() : \text{com}] \eta Q
= \exists N' \leq N \bullet \text{self isExactly } N' \land \eta N' \land m \land Q
\]
Using formulas

\[ [\Gamma, \ldots, N \triangleright \text{self}.m() : \text{com}] \eta Q \]
\[ = \exists N' \leq N \bullet \text{self isExactly } N' \land \eta N' \ m \ Q \]

Value parameter and non-self call in “Oxford style”:
\[ [\ldots x.m(e)] \]
\[ = [\ldots \text{var } p : T \bullet \text{self} := x; \ p := e; \ \text{body}; \ x := \text{self}] \]
Using formulas

$$[[\Gamma, \ldots, N \triangleright \text{self}.m() : \text{com}] \eta \ Q]$$

\[= \exists N' \leq N \bullet \text{self isExactly } N' \wedge \eta N' \ m \ Q\]

Value parameter and non-self call in “Oxford style”:

$$[[\ldots \ x.\text{m}(e)]]$$

\[= [[\ldots \ \text{var} \ p : T \bullet \text{self} := x; \ p := e; \ \text{body}; \ x := \text{self}]]\]

$$[[\Gamma, \ldots, N \triangleright x.\text{m}(e) : \text{com}] \eta \ Q]$$

\[= \exists N' \leq N \bullet x \text{ isExactly } N' \wedge [[\Gamma, \ldots \triangleright (\eta N' \ m)(x, e)]] \eta Q\]
Conclusions

- Initial approach: formulas, to extend earlier work and make proofs easier.
Conclusions

- Initial approach: formulas, to extend earlier work and make proofs easier.
- Syntactic transformation for parameter-passing: leads to method body (syntax) in environment, and special typing rules for body in context of call.
Conclusions

- Initial approach: formulas, to extend earlier work and make proofs easier.
- Syntactic transformation for parameter-passing: leads to method body (syntax) in environment, and special typing rules for body in context of call.
- To formalize simulation proofs, need formulas over two state spaces; complicated formula language for technical purposes.
Conclusions

- Initial approach: formulas, to extend earlier work and make proofs easier.
- Syntactic transformation for parameter-passing: leads to method body (syntax) in environment, and special typing rules for body in context of call.
- To formalize simulation proofs, need formulas over two state spaces; complicated formula language for technical purposes.
- Predicates-as-sets successfully used to prove soundness of simulation for data refinement.
Conclusions

- Initial approach: formulas, to extend earlier work and make proofs easier.
- Syntactic transformation for parameter-passing: leads to method body (syntax) in environment, and special typing rules for body in context of call.
- To formalize simulation proofs, need formulas over two state spaces; complicated formula language for technical purposes.
- Predicates-as-sets successfully used to prove soundness of simulation for data refinement.
- How to simplify? Extend to pointers and other language constructs?