

Lazy Behavioral Subtyping ¹

Einar Broch Johnsen

Dept. of Informatics, University of Oslo
einarj@ifi.uio.no

UNU-IIST, Macau
February 18, 2009



¹Based on work with Johan Dovland, Olaf Owe, and Martin Steffen

Substitutability

Problem:

When can some expression e_1 replace some other expression e_2 ?

One answer can be provided by a type system: *Subtyping*

Example 1: Assignment

$$x := e \quad \frac{\Gamma \vdash e : T \quad T \leq \Gamma(x)}{\Gamma \vdash x := e \text{ ok}}$$

Example 2: Method Calls

$$\begin{array}{l} x := m(e) \quad m : T_1 \rightarrow T_2 \\ \text{Want: } m(e) \quad T_1 \leq T'_1 \quad \downarrow \quad \uparrow \quad T'_2 \leq T_2 \\ \text{Get: } m'(e) \quad (\text{contravariance}) \quad m' : T'_1 \rightarrow T'_2 \quad (\text{covariance}) \end{array}$$



Context

- Late bound method calls in object-oriented programs
- Crucial for the incremental development principle of OOP
- Adaptable and reusable programs
- Challenge for reasoning about programs

Talk Outline

- Substitutability and behavioral subtyping
- Late binding
- Reasoning about late-bound calls
- Lazy behavioral subtyping
- Conclusions / future work



Behavioral Subtyping

Extends subtyping to behavioral properties:

“any property proved about supertype objects
also holds for subtype objects” [Liskow & Wing 94]

Example

Consider an assertion language on local state variables, a programming language, and some program logic.

Assertions $p_1, p_2, q_1, q_2, \dots$ used for pre- and postconditions

When can we replace e_1 by e_2 ?

$$\begin{array}{l} \{p_1\} e_1 \{q_1\} \quad \text{Applicability: } p_1 \Rightarrow p_2 \text{ (ref. contravariance)} \\ \{p_2\} e_2 \{q_2\} \quad \text{Predictability: } q_2 \Rightarrow q_1 \text{ (ref. covariance)} \end{array}$$



Late Binding of Method Calls

Object-oriented programming

- Incremental program development
- *Substitutability* is exploited to organize programs by means of *inheritance*
 - *object substitutability*: a subclass object may be bound to a superclass variable
 - *method substitutability* (late binding): subclass methods may be selected instead of superclass methods
- Success of OOP: More compact and better organized programs

Late binding of method calls

- The code bound to a call depends on the *actual class* of the object
- Decided at runtime
- *Not statically decidable*



Example

```
class C {
  m() {...}
  n() {...; m(); ...}
}

class D extends C {
  m() {...}
}
```

- The binding of `m()` depends on the *actual class* of the object
- *Incremental development*: the class `D` may be added later
- *Late binding* and *incremental development* pose a challenge for *program verification*



Verifying Late-bound Method Calls

- Two main approaches in the literature
- **Open world** [America 91, Liskow & Wing 94, Leavens & Naumann 06, ...]
 - Behavioral subtyping: supports incremental reasoning
 - Subtyping constraints: too restrictive in practice
- **Closed world** [Pierik & de Boer 05, ...]
 - Complete reasoning method
 - Breaks incremental reasoning
- **Gap**: programming practice vs. current verification methods
- **Lazy behavioral subtyping** [FM'08, IFM'09, BKB'09]
 - Supports incremental reasoning
 - Less restrictive than behavioral subtyping



Example: Open World Approach

```
class C {
  m(): (p1, q1) {...}
  n() {...; {p}m() {q}; ...}
}

class D extends C {
  m(): (p2, q2) {...}
}
```

Commitment (declaration site)
Requirement (call site)
 PO: $p \Rightarrow p_1, q_1 \Rightarrow q$

Commitment (declaration site)
 PO: $p_1 \Rightarrow p_2, q_2 \Rightarrow q_1$

Behavioral subtyping

- (p_1, q_1) acts as a commitment (contract) for declarations of `m`
- Redefinitions relate to the contract, not to the call site
- Incremental: Proof reuse when the program is extended
- Restriction: (p_1, q_1) too strong requirement for redefinitions
- **Commitment biased**



Example: Closed World Approach

```
class C {
  m(): (p1, q1) {...}           Commitment (declaration site)
  n() {...; {p}m() {q}; ...}      Requirement (call site)
}
```

PO: $p \Rightarrow p_1 \wedge p_2, q_1 \vee q_2 \Rightarrow q$

```
class D extends C {
  m(): (p2, q2) {...}           Commitment (declaration site)
}
```

Closed world approach

- Assumes all commitments of a method known at reasoning time
- Sufficiently expressive: *complete* reasoning system
- Need to redo proofs if a new class is added to the program
- Breaks with incremental development principle (proof reuse)
- Requirement biased**

CREDO

Example: Lazy Behavioral Subtyping

```
class C {
  m(): (p1, q1) {...}           Commitment (declaration site)
  n() {...; {p}m() {q}; ...}      Requirement (call site)
}
```

PO: $p \Rightarrow p_1, q_1 \Rightarrow q$

```
class D extends C {
  m(): (p2, q2) {...}           Commitment (declaration site)
}
```

PO: $p \Rightarrow p_2, q_2 \Rightarrow q$

Lazy behavioral subtyping

- POs depend on *requirements*, but are triggered by *commitments*
- Irrelevant parts of old commitments may be ignored
- Code reuse**: More flexible than the BS approach
- Incremental**: proof reuse when program is extended

CREDO

Lazy Behavioral Subtyping

- Distinguish *method use* and *method declarations*
- Track *call site requirements* and *declaration site commitments*
- Proof reuse**: Impose use site requirements on method overridings in new subclasses to ensure that old proofs remain valid
- Declaration site proof obligations wrt. superclass' requirements
 - Many, but weaker POs than with behavioral subtyping
- Context-dependent proof outlines** for superclass declarations
- Formalize how commitments and requirements propagate as subclasses and proof outlines are added
 - Proof environment** tracks commitments and requirements
 - Syntax-driven inference system** for program analysis
 - Independent of a particular program logic

CREDO

Proof Environment for Program Analysis

The proof environment consists of *three mappings*, which capture

- The class hierarchy**
- Method commitments**
 - $S(C, m)$: Commitments of a method m in C
 - Concerned with the *declaration* of methods
 - Commitments of a particular implementation
- Method requirements**
 - $R(C, m)$: Requirements towards m made by C
 - Use* of methods
 - Requirements on several implementations

Program analysis uses and updates the proof environment

CREDO

Example: Lazy Behavioral Subtyping

```

class C {
  m(): (p1, q1) {...}           (p1, q1) ∈ S(C, m)
  n() {...; {p}m() {q}; ...}      (p, q) ∈ R(C, m)
}                                  PO: S(C, m) ⇒ R(C, m)
    
```

```

class D extends C {
  m(): (p2, q2) {...}           (p2, q2) ∈ S(D, m)
}                                  PO: S(D, m) ⇒ R↑(C, m)
    
```

Analysis Uses and Modifies the Proof Environment

- Lift entailment to sets of pre/postconditions
- Collect information from mappings; e.g., $R↑(C, m)$, $S↑(C, m)$
- Context-dependent commitments:
New proof outlines for old method declarations



Program Analysis

- **Module**: a set of classes which form a *unit of analysis*
- Analysis happens in modules
- **Incremental development**: a sequence (or stream) of modules
- Proof environment carries over from one module to the next

Modules

$$\frac{\mathcal{E} \vdash [\epsilon; \bar{L}] \cdot \mathcal{A}}{\mathcal{E} \vdash \text{module}(L) \cdot \mathcal{A}} \text{ (NEWMODULE)} \quad \frac{\mathcal{E} \vdash \mathcal{A}}{\mathcal{E} \vdash [\epsilon; \emptyset] \cdot \mathcal{A}} \text{ (EMPMODULE)}$$

Here, \bar{L} are classes, and \mathcal{A} are remaining modules



The Analysis Rules

$$\frac{\begin{array}{l} C \notin \mathcal{E} \quad D \neq \text{nil} \Rightarrow D \in \mathcal{E} \\ \mathcal{E} \oplus (\text{class } C \text{ extends } D \{ \bar{f} \bar{M} \}) \\ \vdash [\langle C : \text{analyzeMtds}(\bar{M}) \rangle ; S] \cdot \mathcal{A} \end{array}}{\mathcal{E} \vdash [\epsilon; \{\text{class } C \text{ extends } D \{ \bar{f} \bar{M} \} \cup S \}] \cdot \mathcal{A}} \text{ (NEWCLASS)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \text{verify}(m, \{(p, q)\} \cup R↑_{\mathcal{E}}(P_{\mathcal{E}}(C).inh, m)) \cdot \mathcal{O} \rangle ; S] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \text{analyzeMtds}(m(\bar{x}) : (p, q) \{ \{t\} \}) \cdot \mathcal{O} \rangle ; S] \cdot \mathcal{A}} \text{ (NEWMTD)}$$

$$\frac{S↑_{\mathcal{E}}(C, m) \Rightarrow (p, q) \quad \mathcal{E} \vdash [\langle C : \mathcal{O} \rangle ; S] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \text{verify}(m, (p, q)) \cdot \mathcal{O} \rangle ; S] \cdot \mathcal{A}} \text{ (REQDER)}$$

$$\frac{\begin{array}{l} \vdash_{PL} m(\bar{x}) : (p, q) \{ \langle \text{body}_{\mathcal{E}}(C, m) \rangle \} \\ \mathcal{E} \oplus \text{extS}(C, m, (p, q)) \\ \vdash [\langle C : \text{analyzeOutline}(\langle \text{body}_{\mathcal{E}}(C, m) \rangle) \cdot \mathcal{O} \rangle ; S] \cdot \mathcal{A} \end{array}}{\mathcal{E} \vdash [\langle C : \text{verify}(m, (p, q)) \cdot \mathcal{O} \rangle ; S] \cdot \mathcal{A}} \text{ (REQNOTDER)}$$

(in addition: 3 rules for calls and some structural rules)



Properties of the Inference System

- **A sound proof environment**
 1. Enough requirements reflecting the use of methods
 2. All requirements follow from commitments
- **Preservation of environment soundness**
The inference system maintains the soundness of proof environments at the *level of modules*
- **Soundness of the proof system**
Assuming soundness for the given program logic, the proof outline system with LBS is sound
Proof: by induction on the depth of call graphs, we show the correctness of the proof outlines
- **Minimality of proof environments**
No “junk” in the proof environment



Methodological Perspective

Pure LBS

- Incremental proof system for program development [FM'08]
- Focus on allowing code reuse, closer to programming practice
- External calls may require new proofs for analyzed classes
- Breaks with behavioral *substitutability* principle for subclassing
- LBS works also for *multiple inheritance* hierarchies [FM'09]

Modular LBS [BKB'09]

- Use *behavioral interfaces* for typing pointers
- Fully separate class inheritance and subtyping
- Restrict use of external calls to *interface contracts*
- Still support flexible code reuse for class inheritance
- Substitutability is recovered at the interface level



Conclusion

LBS: strategy for reasoning about late-bound method calls

- **Comparison to previous approaches**
 - *Behavioral subtyping*: incremental, but too restrictive
 - *Closed world*: complete, but not incremental
 - *LBS*: incremental, less restrictive than BS
- **Lazy behavioral subtyping strategy**
 - Method commitments (declarations) vs. requirements (use)
 - Context-dependent proof outlines
 - Proof reuse: requirements inherited by need
 - Formalized as syntax-driven inference system
 - Combines with invariant reasoning and interfaces
- **Ongoing / Future work**
 - Integrate LBS in programming and analysis environment
 - Empirical evidence

