Incremental Reasoning for Multiple Inheritance

Johan Dovland and Einar Broch Johnsen
Olaf Owe and Martin Steffen

Institutt for Informatikk Universitet i Oslo

iFM, Düsseldorf

17. February 2009

Context

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

Talk Outline

- substitutability and behavioral subtyping
- late binding
- reasoning about late-bound calls
- lazy behavioral subtyping
- introducing multiple inheritance
- conclusions / future work

Substitutability and subtype polymorphism

Problem:

When can some expression e_1 replace some other e_2?

classical answer: subtyping

Example 1: Assignment

`x := e`

$$\frac{\Gamma \vdash e : T \qquad T \leq \Gamma(x)}{\Gamma \vdash \texttt{x := e} : \textbf{ok}}$$

Example 2: Method Calls

`x := m(e)`

$\texttt{m}: \quad \texttt{T}_1 \rightarrow \texttt{T}_2$

$\texttt{T}_1 \leq \texttt{T}_1' \qquad \Downarrow \qquad \Uparrow \qquad \texttt{T}_2' \leq \texttt{T}_2$

(contra-variance) $\texttt{m}': \quad \texttt{T}_1' \rightarrow \texttt{T}_2'$ (covarianc

Want: `m(e)`
Get: `m′(e)`

Behavioral subtyping

Extend subtyping to behavioral properties:

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

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

assertions $p_1, p_2, q_1, q_2, \ldots$ used for pre- and post-conditions

When can we replace e_1 by e_2?

$\{p_1\}$ `e1` $\{q_1\}$

$\{p_2\}$ `e2` $\{q_2\}$

contra-variance: $p_1 \Rightarrow p_2$

co-variance: $q_2 \Rightarrow q_1$

Late Binding of Method Calls

Object-oriented programming

- incremental program development
- *substitutability* is exploited to organize programs by means of *inheritance*
  - "inheritance implies subtyping"
  - *object substitutability*:
    a subclass object may be bound to a superclass variable
  - *late binding*:
    subclass methods may be selected instead of superclass methods

Late binding of method calls

- 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: 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
- **Closed world** [Pierik & de Boer 05, . . .]
  - Complete reasoning method
  - Breaks incremental reasoning
- **Open world** [America 91, Liskow & Wing 94, Leavens & Naumann 06, . . .]
  - Behavioral subtyping: supports incremental reasoning
  - Subtyping constraints: too restrictive in practice
- **Lazy behavioral subtyping** [1]
  - supports incremental reasoning
  - less restrictive than behavioral subtyping

Example: Closed world approach

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

Commitment (declaration site)
Requirement (call site)
PO: $p \Rightarrow p_1 \land p_2,\ q_1 \lor q_2 \Rightarrow q$

```
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
- redo proofs if a new class is added to the program
- breaks with incremental development principle (proof reuse.)

Example: Open World Approach

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

commitment (decl. site)
requirement (call site)
PO: $p \Rightarrow p_1$, $q_1 \Rightarrow q$

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

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

Behavioral subtyping

- (p_1, q_1) acts as 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): strong requirement for redefinitions

Example: Lazy Behavioral Subtyping

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

Commitment (declaration site)
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*, not on commitments (contracts)
- irrelevant parts of old commitments may be ignored
- more flexible than behavioral subtyping approach
- incremental: proof reuse when program is extended

Lazy Behavioral Subtyping

- Distinguish method use and method declarations
- track call site requirements and declaration site commitments
- Proof reuse: Impose these 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

  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

Proof Environment for Program Analysis

The proof environment consists of three mappings, which capture

- the class hierarchy
- method commitments
  - $S(C, B.m)$: commitment of a method `m` (defined in B) in `C`
  - Concerned with the *declaration* of methods
  - Commitment of a particular implementation
- method requirements
  - $R(C, B\#m)$: requirements towards `m` made by `C`
    - C: imposes the requirements
    - B: call-site class, where calling method is defined.
  - *use* of methods
  - requirements on several implementations

Example: Lazy Behavioral Subtyping

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

$(p_1, q_1) \in S(C, m)$
$(p, q) \in R(C, m)$
$PO: S(C, m) \Rightarrow R(C, m)$

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

$(p_2, q_2) \in S(D, m)$
$PO: S(D, m) \Rightarrow R{\uparrow}(C, m)$

Analysis uses and modifies a proof environment

- Analysis uses and updates the proof environment
- Collect information from mappings; e.g., $R{\uparrow}(C, m)$, $S{\uparrow}(C, m)$
- Context-dependent commitments:
  New proof outlines for old method declarations

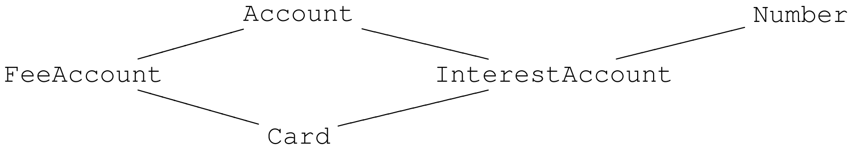
Glimpse of the calculus

$$
\begin{array}{rcll}
P & ::= & \overline{L}\ \{t\} & \text{program} \\
L & ::= & \textbf{class}\ C\ \textbf{extends}\ \overline{C}\ \{\overline{f}\ \overline{M}\} & \text{class definitio} \\
M & ::= & m\ (\overline{x})\{t\} & \text{method} \\
e & ::= & \textbf{new}\ C \mid b \mid v \mid \textbf{this} \mid e.m(\overline{e}) \mid m(\overline{e}) \mid m(\overline{e})@C & \text{expression} \\
v & ::= & f \mid f@C & \text{values} \\
t & ::= & v := e \mid \textbf{return}\ e \mid \textbf{skip} & \\
 & \mid & \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ t\ \textbf{fi} \mid t; t &
\end{array}
$$

- variant of Featherweight Java
- with multiple inheritance
- static calls (as generalization of super-calls)

Multiple inheritance

- inheritance hiearchy = directed acyclic graph ($\neq$ tree)



Example

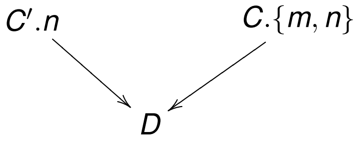
```
class Account { int bal = 0;
 deposit(int x) {...;update(x)}
 withdraw(int x) {...;update(-x)}
 update(int y) {...; bal=bal+y;...}}

class Number { int num;
 update(int x) {num = x }
 increase(int x) {update(num+x)}}

class InterestAccount extends Account Number { int fee;
 addInterest(int x y) {...; deposit(x); increase(y)}
 withdraw(int x) {withdraw(x)@Account; if bal<0 then update(-fee) fi}}

class FeeAccount extends Account { int fee;
 withdraw(int x) {withdraw(x)@Account; update(-fee) }
 update(int y) {...; bal=bal+y;...}}

class Card extends FeeAccount InterestAccount {
 withdraw(int x) {withdraw(x)@InterestAccount; update(-fee@FeeAccount)}}
```

Name conflicts and healthiness

- name "conflicts"
  - vertical
  - horizontal
- resolved by binding strategy
- 2 classes C_1 and C_2 related: $C_1 \leq C_2$ or vice versa
- healthiness: general condition on binding strategies when methods are inherited

$\Rightarrow$ "do not bind to unrelated classes"

  - self-calls in C: must bind to a class related to C
  - remote call $x.m$, with x's declared type C: bind to class related to C



Program Analysis

- module : a set of classes which form a *unit of analysis*
- analysis happens in modules
- incremental development : a sequence/stream of modules
- proof environment carries over from one module to the next-

Modules

$$\frac{\mathcal{E} \vdash [\epsilon\,;\,\overline{L}] \cdot \mathcal{A}}{\mathcal{E} \vdash \mathit{module}(\overline{L}) \cdot \mathcal{A}} \quad (\textsc{NewModule}) \qquad \frac{\mathcal{E} \vdash \mathcal{A}}{\mathcal{E} \vdash [\epsilon\,;\,\emptyset] \cdot \mathcal{A}} \quad (\textsc{EmpModule})$$

Here, $\overline{L}$ are classes, and $\mathcal{A}$ are remaining modules.

Tracking constraints

- formalized by a derivation system
- analyzing a $m(\vec{x}) : (p, q)\{body(B, m)\}$ in class C:

$\Rightarrow$

- add (p, q) to the commitments $S(C, B.m)$
- analyze the annotated method: for each call $\{r\}\ n()\ \{s\}$
  1. (r, s) is analyzed wrt. implementation of $B\#n$ found when starting the search in C: proof obligation $S \uparrow (C, E.n) \implies (r, s)$ must be established, where $E = bind(C, B\#n)$.
  2. (r, s) is remembered in requirements $R(C, B\#n)$

Analysis rules

$$\frac{C \notin \mathcal{E} \qquad \overline{D} \in \mathcal{E} \qquad \overline{E} = commSup_{\mathcal{E}}(C) \qquad \mathcal{E} \oplus extP(C, \overline{D}, \overline{f}, \overline{M}) \vdash [\langle C : analyzeMtds(\overline{M}) \cdot supCls(\overline{E})\rangle \;;\; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\epsilon \;;\; \{\texttt{class}\ C\ \texttt{extends}\ \overline{D}\ \{\overline{f}\ \overline{M}\}\} \cup \mathcal{S}] \cdot \mathcal{A}} \text{ CLASS}$$

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

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

$$\frac{\vdash_{\mathrm{PL}} m : (p,q)\ \{body_{\mathcal{E}}(D,m)\} \qquad \mathcal{E} \oplus extS(C, D, m, (p,q)) \vdash [\langle C : analyzeOutline(D, body_{\mathcal{E}}(D,m)) \cdot \mathcal{O}\rangle \;;\; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : verify(D, m, (p,q)) \cdot \mathcal{O}\rangle \;;\; \mathcal{S}] \cdot \mathcal{A}}$$

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 soundness of the proof environment at the *level of modules*
- Soundness of the proof system
  Assuming soundness for the given program logic, the proof outline system is sound

  Proof: by induction on the depth of derivation, we show the correctness of the proof outlines
- **Minimality of proof environments**
  No "junk" in the proof environment

Conclusion

sound, incremental strategy for reasoning about late-bound method calls

- *Comparison to previous approaches*
  - behavioral subtyping: incremental, but too restrictive
  - closed world: complete, but not incremental
  - behavioral subtyping plus separation logic (and multiple inheritance)
  - LBS: incremental, less restrictive than BS
- *Lazy behavioral subtyping strategy for multiple inheritance*
  - Method commitments (declarations) vs. requirements (use)
  - Proof reuse: requirements inherited by need
  - soundness condition for multiple inheritance
  - Formalized as syntax-driven inference system
- *Future work*
  - Combination with invariant reasoning and interfaces
  - Integration in programming and analysis environment

References I

[1] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen.
Lazy behavioral subtyping.
In *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 2008.

[2] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen.
Incremental reasoning for multiple inheritance.
In *Proceedings of the 7th International Conference on integrated Formal Methods (iFM'09), Düsseldorf, Germany, 16 - 19 February, 2009*, Lecture Notes in Computer Science. Springer-Verlag, Feb. 2009.
To appear.

[3] E. B. Johnsen, O. Owe, and I. C. Yu.
Creol: A type-safe object-oriented model for distributed concurrent systems.
Theoretical Computer Science, 365(1–2):23–66, Nov. 2006.