(Multiple) inheritance, behavioral subtyping, and separation logic

Martin Steffen

IfI UiO

September 2008

OO, inheritance and subtyping

- general problem in OO: flexibility
  - open world / incremental program development
  - late binding /dynamic dispatch
- inheritance vs. subtyping
- *subsumption*:

  A element of a subtype can safely be used where an element of the subertype is expected

- safely = without “type error”
- note: interaction with late binding
- does (in praxis) not hold for logical properties
- behavioral subtyping :
  - porting the above principe to behavioral specs

Goals

- soundness
- modularity
- no base class code required
- breadth

Separation logic

- extension to Hoare logic
- reason about shared, mutable state on the heap
- spatial connectives
- local reasoning and frame rule
- sep. logic & OO: [Parkinson, 2005]

Formulas

$$
\begin{array}{lcl}
P, Q & ::= & B \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \\
& \mid & \mathit{empty} \mid P * Q \mid P \mathrel{-\!\!*} Q \mid E \mapsto E'
\end{array}
$$

- spatial connectives in red
- interpreted over state plus heap

Interpretation

- state/stack: from vars to values
- heap: from locations to values

$$S, H \models e_1 \mapsto e_2 \triangleq dom(H) = \{[E]_S\} \wedge H([e_1]_S) = [e_2]_S$$

$$S, H \models P * Q \triangleq \exists H_1, H_2.\ H = H_1 * H_2\ S, H_1 \models P \wedge S, H_2 \models Q$$

empty represents the empty heap

Hoare logic & heap

- as usual: pre-/post- specifications $\{P\}\ \_\ \{Q\}$
- of course: new program-constructs $\Longrightarrow$ new specific axioms
- using " small " formulas, reasoning local :

$$\frac{\{P\}\ t\ \{Q\}}{\{P \wedge R\}\ t\ \{Q \wedge R\}}$$

Hoare logic & heap

- as usual: pre-/post- specifications $\{P\} \_ \{Q\}$
- of course: new program-constructs $\Longrightarrow$ new specific axioms
- using " small " formulas, reasoning local :

$$\frac{\{P\}\, t\, \{Q\}}{\{P \wedge R\}\, t\, \{Q \wedge R\}}$$

$$\frac{\{x \mapsto -\}\, [x] := 4\, \{x \mapsto 4\}}{\{x \mapsto - \;\wedge\; y \mapsto 3\}\, [x] := 4\, \{x \mapsto 4 \;\wedge\; y \mapsto 3\}}$$

Hoare logic & heap

- as usual: pre-/post- specifications $\{P\}\ \_\ \{Q\}$
- of course: new program-constructs $\Longrightarrow$ new specific axioms
- using "small" formulas, reasoning local:

$$\frac{\{P\}\ t\ \{Q\} \qquad modifies(t) \cap fv(R) = \emptyset}{\{P \wedge R\}\ t\ \{Q \wedge R\}}$$

$$\frac{\{x \mapsto -\}\ [x] := 4\ \{x \mapsto 4\}}{\{x \mapsto -\ \wedge\ y \mapsto 3\}\ [x] := 4\ \{x \mapsto 4\ \wedge\ y \mapsto 3\}}$$

Frame rule

- generalization of the “rule of constancy”

$$\frac{\{P\}\, t\, \{Q\} \qquad \mathit{modifies}(t) \cap \mathit{fv}(R) = \emptyset}{\{P * R\}\, t\, \{Q * R\}}$$

Small axioms

$$
\begin{array}{rcl}
\Lambda;\Gamma \vdash \{E \mapsto \_\} & [E] := E' & \{E \mapsto E'\} \\
\Lambda;\Gamma \vdash \{E \mapsto n \wedge x = m\} & x := [E] & \{E[m/x] \mapsto n \wedge x = n\} \\
\Lambda;\Gamma \vdash \{E \mapsto \_\} & \mathrm{dispose}(E) & \{\mathit{empty}\} \\
\Lambda;\Gamma \vdash \{\mathit{empty} \wedge x = m\} & x := \mathrm{cons}(\vec{E}) & \{x \mapsto \vec{E}[m/x]\}
\end{array}
$$

Procedures

$$\textsc{Return}\ \frac{}{\Lambda; \Gamma \vdash \{P[x/ret]\}\ return\ x\ \{P\}}$$

$$\textsc{Proc}\ \frac{\Gamma \vdash \{P\}\ k(\vec{x})\ \{Q\}}{\Lambda, \Gamma \vdash \{P[\vec{y}/\vec{x}]\}\ y := k(\vec{y})\ \{Q[\vec{y}, y/\vec{x}, ret]\}}$$

Separation logic and inheritance

- vehicle: standard, prototypical oo language
  - single inheritance & late binding
  - inspired by C$^\sharp$
- Hoare-logic/separation logic method specification
- weakening of behavioral subtping
- incremental reasoning

Language syntax: statements

s	$::=$		statement
		$x := y$	assignment
	$\mid$	$x := \text{null}$	initialization
	$\mid$	$x := y.f$	field access
	$\mid$	$x.f := y$	field update
	$\mid$	$x := y.m(\vec{z})$	dynamic method invocation
	$\mid$	$x := y.C :: m(\vec{z})$	direct method invocation
	$\mid$	$x := (C)y$	cast
	$\mid$	$\text{if } (x = y) \text{ then } \vec{s} \text{ else } \vec{t}$	equality test
	$\mid$	$x := \text{new } C$	object creation

Language syntax: classes and specs

$$
\begin{array}{rcll}
L & ::= & \text{class } C : D\{\text{public } \vec{T}\ \vec{f}\ \vec{A}\ K\ \vec{M}\} & \text{class definition} \\
A & ::= & \text{define}\alpha_C & \text{predicate familiy entry} \\
K & ::= & \text{public } C()Sd\ Ss\ \{\vec{s}\} & \\
M & ::= & & \text{method definition} \\
 & & \text{virtual } C\ m(\vec{D}\ \vec{x})\ Sd\ Ss\ B & \text{virtual method} \\
 & \mid & \text{override } C\ m(\vec{D}\ \vec{x})\ Sd\ Ss\ B & \text{overridden method} \\
 & \mid & \text{inherit } C\ m(\vec{D}\ \vec{x})\ Sd\ Ss & \text{inherited method} \\
Sd & ::= & \text{dynamic } S & \text{dynamic specification} \\
Ss & ::= & \text{static } S & \text{static specification} \\
S & ::= & \{P\}\_\{Q\} \mid S \text{ also } \{P\}\_\{Q\} & \text{method specification} \\
B & ::= & \{\ \vec{C}\ \vec{x}; \vec{s}; \text{return } y;\ \} & \text{method body}
\end{array}
$$

Operational semantics: calls

- operational semantics
- transitions between configs: $S, H, \vec{s}$

$$\frac{mbody(C,m) = (\vec{z}'', B) \qquad B = \vec{C}\ \vec{x}; \vec{s}'\ return\ x'; \qquad \theta = [y_1, \vec{z}', \vec{x}'/this, \vec{z}'', \vec{x}] \qquad \vec{z}', \vec{x}'\ \text{fresh} \qquad S' = S[\vec{z}' \mapsto S(\vec{z})]}{S, H, y_0 := y_1.C :: m(\vec{z}); \vec{s} \rightarrow S', H, (\theta\vec{s}')y_0 := (\theta x'); \vec{s}}\ \text{CALL-DIR}$$

$$\frac{type(H(S(y))) = C \qquad S, H, x = y.C :: m(\vec{z}); \vec{s} \rightarrow S', H', \vec{s}'}{S, H, x = y.m(\vec{z}); \vec{s} \rightarrow S', H'\vec{s}'}\ \text{CALL-LATE}$$

Formulas

$$
\begin{array}{rcl}
P, Q & ::= & \forall x.P \mid P \rightarrow Q \mid \bot \mid \alpha(\vec{x}) \mid e = e' \mid c : C \\
 & \mid & x.f \mapsto e \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathit{empty} \\
e & ::= & x \mid \mathrm{null}
\end{array}
$$

- slight refinement from the previous version:
  - adaptation to the oo-language
  - abstract predicate families α

Some (fairly standard) proof rules

$$\frac{x \text{ has static type } C \qquad \Gamma \vdash C.m(\vec{x}) : \{P\}\_\{Q\}}{\Gamma; \Delta \vdash \{P[x, \vec{y}/\mathit{this}, \vec{x}] \wedge \mathit{this} \neq \mathit{null}\}\, z := x.m(\vec{y})\, \{Q[z, x, \vec{y}/\mathit{ret}, \mathit{this}, \vec{x}]\}} \text{ CAL}$$

$$\frac{x \text{ has static type } C \qquad \Gamma \vdash C :: m(\vec{x}) : \{S\}\_\{T\}}{\Gamma; \Delta \vdash \{S[x, \vec{y}/\mathit{this}, \vec{x}] \wedge \mathit{this} \neq \mathit{null}\}\, z := x.C :: m(\vec{y})\, \{T[z, x, \vec{y}/\mathit{ret}, \mathit{this}, \vec{x}]\}}$$

Aspects of inheritance

- different “uses” of inheritance
  1. “specialization” (= adding methods/members)
  2. overriding (= no change of behavior)
  3. code-reuse (= overriding with change of behavior)

Example

```
class Cell {
  public int val;

  public virtual void set(int x) { this.val = x; }

  public virtual int get ()   { return this.val; }
}


class Recell; Cell {
  public int back
  public override void set(int x) {
    this.back = base.get();
    base.set(x); }
}

class DCell: Cell {
  public override void  set(int x) {
    base.set(2*x);}
}
```

Static and dynamic specs

- method specification: pre- and post-condition, method "contract"
- here: split into
  - static spec
  - dynamic spec

Example: Cell

```
class Cell {
  public int val;

  public virtual void set(int x) { this.val = x; }

  public virtual int get ()   { return this.val; }
}
```

Example: Cell

```
class Cell {
  public int val

  public virtual void set(int x)
  dynamic { (Val(this,_) } _ { Val(this,x) }
  static  { this.val > _} _ { this.val > x}
  { ... }

  public  virtual int get ()
  dynamic { Val(this,x) }   _ { Val(this,x) * ret = x }
  static  { this.val > x } _ { this.val >  * ret = x }
  { ... }
}
```

Cell (more details)

```
class Cell {
  int val;
  define Val_Cell(x, v) as x.val ↦ v

  public Cell() dynamic {true} _ {Val(this, _)}  {}

   public virtual void set(int x)
   dynamic {Val(this, _)} {Val(this, x)}
     {this.val := x}

  public virtual int get()
  dynamic {Val(this, v)} _ {Val(this, v) * ret = v}
  {return this.val; }

  public virtual void swap (Cell c)
    static {Val(this, v1) * Val(this, v2)} _ {Val(this, v2) * Val(this, v1)}
   { int t,t2; t:=this.get();
     t2 := c.get(); this.set(t2); c.set(t); }
}
```

Abstract predicate family

- abstract predicate: "hide" details (of the state)
- family: generalization for OO: entry per class
- "entry" in class `Cell`
- abstract predicates: scoped

$$Val(x, v) \triangleq x.\mathrm{val} \mapsto v \tag{1}$$

$$x : C \rightarrow (Val(x, v) \leftrightarrow x.\mathrm{val} \mapsto v) \tag{2}$$

Overriding & changing

```
class DCell : Cell{
  public override void set(int x)
    dynamic {  Val(this, _) } _ {  Val(this, x) }
     also   { DVal(this, _) } _ { DVal(this, x*2) }
  { ... }

  public inherit int get () {
    dynamic {  Val(this, v) } _ {  Val(this, v) * ret = v }
    also    { DVal(this, v) } _ { DVal(this, v) * ret = v }
}
```

DCell (more details)

Method verification

- 3 rules, covering 3 different situation, how a method can "occur" in a class
  1. defined directly, as virtual method
  2. inherited methods
  3.
- form of the judgment

$$\Gamma; \Delta \vdash M \ in \ C$$

  definition of *M* "as available in C"

Behavioral subtyping and refinement

- relation between specs
- regulates the implications between the different, involved method specs (static or dynamic)
- most straightforward definition: model inclusion [1]

$$\{P_1\}\_\{Q_1\} \text{ refines } \{P_2\}\_\{Q_2\} \textit{ if } \models \{P_1\}\,\vec{s}\,\{Q_1\} \textit{ implies } \models \{P_2\}\,\vec{s}\,\{Q_2\},$$

- notation:

$$\{P_1\}\_\{Q_1\} \Longrightarrow \{P_2\}\_\{Q_2\}$$

- speciality (for sep-logic + inheritance)

$$\{P_1\}\_\{Q_1\} \overset{this,E}{\Longrightarrow} \{P_2\}\_\{Q_2\}$$

[1] They speak the other way round, and are not consistent.

Virtual methods

$$\frac{\begin{array}{c} B = \{\vec{G}\vec{y};\vec{s};\text{return } z\} \\ Sd = \text{dynamic}\{P_E\}\,\_\,\{Q_E\} \qquad Ss = \text{static}\{S_E\}\,\_\,\{T_E\} \\ \Delta \vdash \{S_E\}\,\_\,\{T_E\} \overset{this;E}{\Longrightarrow} \{P_E\}\,\_\,\{Q_E\} \qquad \Delta;\Gamma \vdash \{S_E\}\,\vec{s}\,\{T_E[z/\text{ret}]\} \end{array}}{\Gamma;\Delta \vdash \text{virtual } C\ m(\vec{D}\vec{x}) Sd\ Ss\ in\ E}\ \text{M-VIRT}$$

- 2 relevant proof obligations
  1. dynamic dispatch
  2. body verification

Inherited methods

$$\frac{\begin{array}{c} E \leq^1 F \qquad Sd = \text{dynamic}\{P_E\}\_\{Q_E\} \qquad Ss = \text{static}\{S_E\}\_\{T_E\} \\ \Gamma \vdash F.m(\vec{x}) : \{P_f\}\_\{Q_F\} \qquad \Gamma \vdash F :: m(\vec{x}) : \{S_E\}\_\{T_E\} \\ \Delta \vdash \{P_E\}\_\{Q_E\} \Longrightarrow \{P_F\}\_\{Q_F\} \qquad \Delta \vdash \{S_F\}\_\{T_F\} \Longrightarrow \{S_E\}\_\{T_E\} \\ \Delta \vdash \{S_E\}\_\{T_E\} \overset{this;E}{\Longrightarrow} \{P_E\}\_\{Q_E\} \end{array}}{\Gamma; \Delta \vdash \text{inherit } C\ m(\vec{D}\vec{x}) Sd\ Ss\ in\ E} \text{ M}$$

- 2 relevant proof obligations
  1. behavioral subtyping
  2. inheritance
  3. body verification

Overridden methods

$$\frac{\begin{array}{c} E \leq^1 F \qquad Sd = \text{dynamic}\{P_E\}\_\{Q_E\} \qquad Ss = \text{static}\{S_E\}\_\{T_E\} \\ B = \{\vec{G}\ \vec{y}; \vec{s}\ \text{return} z; \} \qquad \Gamma \vdash F.m(\vec{x}) : \{P_F\}\_\{Q_F\} \\ \Delta \vdash \{P_E\}\_\{Q_E\} \Longrightarrow \{P_F\}\_\{Q_F\} \\ \Delta \vdash \{S_E\}\_\{T_E\} \overset{this;E}{\Longrightarrow} \{P_E\}\_\{Q_E\} \\ \Delta; \Gamma \vdash \{S_E\}\ \vec{s}\,\{T_E[z/\text{ret}]\} \end{array}}{\Gamma; \Delta \vdash \text{override } C\ m(\vec{D}\vec{x}) Sd\ Ss\ in\ E} \text{ M-OVE}$$

- 3 relevant proof conditions
  1. behavioral subtyping
  2. dynamic dispatch
  3. body verification

Recell

Abstract function definition

$$\frac{\Lambda, \Lambda'; \Gamma \vdash \{P'\}\, C'\, \{Q'\} \qquad \Lambda;\ \Gamma, \{P'\}\, k(\vec{x})\, \{Q'\} \vdash \{P\}\, C\, \{Q\}}{\Lambda; \Gamma \text{ in} \vdash \{P\}\, \text{let} k\ \vec{x} = C' \text{ in } C\, \{Q\}} \text{ ABST-FUN-D}$$

Scoping

$$\frac{\Lambda; \Gamma \vdash \{P\}\, C\, \{Q\}}{\Lambda, \Lambda'; \Gamma \vdash \{P\}\, C\, \{Q\}} \text{ A-W}{\small\text{EAK}}$$

$$\frac{\Lambda, \Lambda'; \Gamma \vdash \{P\}\, C\, \{Q\} \qquad \mathit{dom}(\Lambda') \text{ not used in } P, Q, \Gamma, \Lambda}{\Lambda; \Gamma \vdash \{P\}\, C\, \{Q\}} \text{ A-E}{\small\text{LIM}}$$

```
let
  consPool s =
    { newvar p;
      p := cons(null,s);
      return p }

  getConn x =
    ( newvar n,c,l,p;
      l := [x];
      if   (l=null)
      then p:=[x+1];c:=consConn(p)
      else (c:=[l]; n:=[n+1]; dispose(l);
            dispose(l+1); [x] := n);
      return c
    )

  freeConn x y =
    ( newvar t,n;
      t  := [x];
```

References I

[Chin et al., 2008] Chin, W.-N., David, C., Nguyen, H.-H., and Qin, S. (2008).
Enhancing modular oo verification with separation logic.
In [POPL'08, 2008], pages 87–99.

[Dovland et al., 2007] Dovland, J., Johnsen, E. B., and Owe, O. (2007).
A calculus for incremental specification and reasoning about late binding and inheritance.
In preparation.

[Dovland et al., 2008] Dovland, J., Johnsen, E. B., Owe, O., and Steffen, M. (2008).
Incremental reasoning for multiple inheritance.
Research Report 373, Department of Informatics, University of Oslo.

[Parkinson, 2005] Parkinson, M. (2005).
Local Reasoning for Java.
PhD thesis, Cambridge University.

[Parkinson and Biermann, 2005] Parkinson, M. J. and Biermann, G. M. (2005).
Separation logic and abstraction.
In *Proceedings of POPL '05*, pages 247–258. ACM.

[Parkinson and Biermann, 2008] Parkinson, M. J. and Biermann, G. M. (2008).
Separation logic, abstraction, and inheritance.
In [POPL'08, 2008].

[POPL'08, 2008] POPL'08 (2008).
37th Annual Symposium on Principles of Programming Languages (POPL). ACM.