FOO Klausurrezept

Lukas Bach - lbach@outlook.de - lukasbach.com

1 Verhaltenskonformanz

Für Klasseninvarianten (gelten vor und nach jedem Methodenaufruf) gilt:

Unterkl.-Invariante gilt stärker als Oberkl.-Invariante: $\text{Inv(U)} \Rightarrow \text{Inv(O)}$

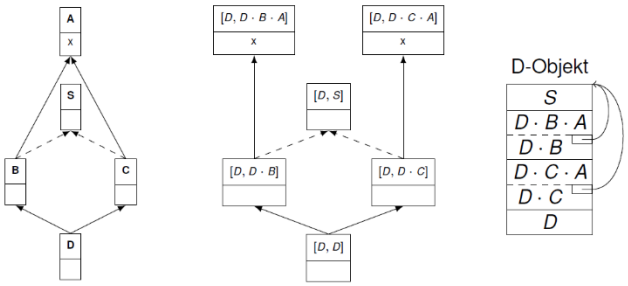
Oberklasse hat stärkere Vorbedingung (Oberklasse verlangt mehr): $\text{PRE(O.m)} \Rightarrow \text{PRE(U.m)}$

Unterklasse hat stärkere Nachbedingung (Unterklasse leistet mehr): $\text{POST(U.m)} \Rightarrow \text{POST(O.m)}$

Tipp: Bei Prüfung Bedingungen komplett ausschreiben.

2 V-Tables

VTables enthalten alle Methoden, die in der jeweiligen Klasse deklariert sind!



2.1 Dynamische Bindung

$$\text{lookup}(\sigma, m) = \min(\text{Defs}(\sigma, m))$$

$$\text{dynBind}(\sigma, f) = \min(Defs([mdc(\sigma), mdc(\sigma)], f))$$

$$= lookup([mdc(\sigma), mdc(\sigma)], f)$$

für „most-derived-class“-Fkt. $mdc([C, \alpha \cdot A]) = C$.

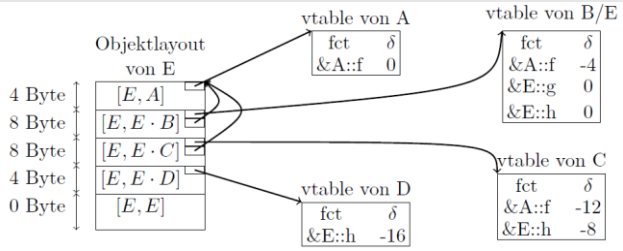
$$\text{Defs}(\sigma, m) = \{\sigma' \sqsupseteq \sigma \mid m \in Member(ldc(\sigma'))\}$$

ist die Menge aller Subobjekte, die im Subobjektgraph von σ aus erreichbar sind und eine Definition vom Member m enthalten.

Wenn $\forall \sigma' \in S: \sigma \sqsubseteq \sigma'$ (also alle Subobjekte in S über σ erreichbar sind, „Subobjekt σ dominiert Subobjekt-Menge S“) und $\sigma \in S$, dann ist $\sigma = \min(S)$.

TODO

2.1.1 Deltas



Beim Delta-Abzählen: Verwende obere Kante eines Subobjekten als Zählreferenz. Achtung: Members von betrachteter Klasse werden meist nicht in dessen Subobjekt, sondern einer geteilten vtable gespeichert.

Geteilte Vtable entlang linkester Außenkante im Subobjektgraphen.

2.1.2 Umsetzung von CPP Aufrufen

```
pbc->f(42);
```

wird umgesetzt durch:

```
register vt = &(pbc->vptr[K]);
(*vt->fct)(pbc + (vt->delta), 42);
```

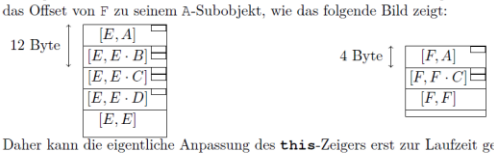
Mit K als Index der Methode in der VTable. Parameter kann auch weggelassen werden.

(d) Erläutern Sie anhand des folgenden Programmfragments, wozu die laufzeitabhängigen Offsets („Deltas“) nötig sind: **[2 Punkte]**

```
void foo(C *c) {c->f();}
...
foo(new E()); foo(new F());
```

Beispiellösung:

Der Compiler erzeugt den Code für `foo` nur einmal. Unabhängig davon, welchen Typ `c` zur Laufzeit hat, muss der Code den **`this`**-Zeiger korrekt verschieben. Im Beispiel ist das Aufrufziel bei beiden Aufrufen `A::f`, der **`this`**-Pointer muss also vom jeweiligen `C`- auf das jeweilige `A`-Subobjekt verschoben werden. Das Offset von `E` zu seinem `A`-Subobjekt ist aber ein anderes als das Offset von `F` zu seinem `A`-Subobjekt, wie das folgende Bild zeigt:



Daher kann die eigentliche Anpassung des **`this`**-Zeigers erst zur Laufzeit geschehen.

Bemerkung: Das Bild war nicht gefordert, sondern dient nur zur Illustration.

2.1.3 Umsetzung von Upcasts

2.1.3.1 Nicht-virtuelle Mehrfachvererbung

```
B pb = pc; // pc is type C; wird zu:
B pb = (B*) (((char*)pc) + delta(C,B));
```

$delta(C, B) = sizeof(A)$ ist statisch zur Kompilierzeit bekannt.

2.1.3.2 Virtuelle Mehrfachvererbung

```
B pb = pc; // pc is type C; wird zu:
B pb = *((B**) (((char*)pc)+offset(B_ptr)));
```

B_ptr ist der Offset des B-Subobjektzeigers im C-Objkt.

3 Generics

`? extends C` im Typparameter: Der Typ liegt unterhalb C. Ich kann davon nur *als C oder höher lesen* und Object-Typen darin schreiben. Schreiben von C-Typen ist nicht möglich.

`? super C` im Typparameter: Der Typ liegt oberhalb C. Ich kann davon nur als Object lesen, nicht als C, dafür *C-Objekte oder höher darin schreiben*.

3.1 Klausurtypische Aufgabe

todo

Nicht akzeptierte Beispiele – Aufgabe: 1-3 Punkte => eine zu restriktive Deklaration, 3-5 Punkte => zwei zu restriktive Deklarationen.

3.2 PECS Example

Note how the source list `src` (the producing list) uses `extends`, and the destination list `dest` (the consuming list) uses `super`:

```
public class Collections {
 public static <T> void copy(List<? super T> dest,
                             List<? extends T> src) {
   for (int i = 0; i < src.size(); i++)
    dest.set(i, src.get(i));
 }
}
```

```
interface Fun<S,T> {
    T apply(S x);
}

public static <S,T> void map(Fun<S,T> f, List<S> l, List<T> r) {
    for (S s : l) {
            r.add(f.apply(s));
    }
}

class Integer implements Number {}
class Byte implements Number {}

List<Number> nlist;      List<Integer> ilist;      List<Byte> blist;
List<Number> nresults;   List<Integer> iresults;   List<Object> oresults;
Fun<Byte,Integer> b2i;   Fun<Integer,Integer> i2i;  Fun<Number,Integer> n2i;
```

Die generische Deklaration von `map` ist zwar gültig[1], aber zu restriktiv. Beispielsweise wird der Aufruf `map(b2i,blist,oresults)` nicht akzeptiert, obwohl er sinnvoll ist (Ausführung würde zu keinem Fehler führen).

(a) Geben Sie für die folgenden alternativen Deklarationen an, ob diese ungültig, gültig aber zu restriktiv, oder gültig und allgemein verwendbar sind. **[10 Punkte]**

i.) `<S,T> void map(Fun<? super S, Object> f, List<S> l, List<T> r)`
☑ ungültig ☐ zu restriktiv ☐ gültig und allgemein

ii.) `<S,T> void map(Fun<? super S,? extends T> f, List<S> l, List<T> r)`
☐ ungültig ☐ zu restriktiv ☑ gültig und allgemein

iii.) `<S,T> void map(Fun<S,? extends T> f, List<? super S> l, List<T> r)`
☑ ungültig ☐ zu restriktiv ☐ gültig und allgemein

iv.) `<S,T> void map(Fun<? super S,? extends T> f, List<S> l, List<Object> r)`
☐ ungültig ☑ zu restriktiv ☐ gültig und allgemein

v.) `<S,T> void map(Fun<S,T> f, List<S> l, List<? super T> r)`
☐ ungültig ☑ zu restriktiv ☐ gültig und allgemein

(b) Geben Sie für jede gültige, aber zu restriktive Deklaration einen sinnvollen Beispielaufruf[2] an, der unter dieser Deklaration nicht akzeptiert wird. **[5 Punkte]**

Beispiellösung:

iv.) `map(b2i,blist,iresults)`

v.) `map(n2i,blist,iresults)`

4 Typsystem

$CONST\ \frac{c \in Const}{\Gamma \vdash c : \tau_c}$	
$VAR\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$	
$ABS\ \frac{\Gamma, x:\sigma \vdash t:\tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$	
$APP\ \frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$	
$NEW\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash new\ t : Cell\ \tau}$	Allokation einer Speicherzelle.
$!\ \frac{\Gamma \vdash t : Cell\ \tau}{\Gamma \vdash !\,t : \tau}$	Lesen einer Speicherzelle.
$:=\ \frac{\Gamma \vdash t_1 : Cell\ \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 := t_2 : unit}$	Schreiben auf eine Speicherzelle.
$\{\}I\ \frac{\Gamma \vdash t_1 : \tau_1 \ \dots \ \Gamma \vdash t_n : \vdash \tau_n}{\Gamma \vdash \{m_1 = t_1, \dots, m_n = t_n\} : \{m_1 : \tau_1, \dots \backslash m_n : \tau_n\}}$ *Typregel für Objekte als Records*	
$\{\}E\ \frac{\Gamma \vdash o : \{\dots, m:\tau, \dots\}}{\Gamma \vdash o.m : \tau}$	Typregel für Objekte als Records
$O1\ \frac{\sigma_1 \le \tau_1 \dots \sigma_n \le \tau_n}{\{m_1:\sigma_1, \dots, m_n;\sigma_n\} \le \{m_1:\tau_1, \dots, m_n:\tau_n\}}$ *Objektkonversion*	
$O2\ \frac{}{\{m_1:\tau_1, \dots, m_n:\tau_n, m_{n+1}:\tau_{n+1}, \dots, m_{n+k}:\tau_{n+k}\} \le \{m_1:\tau_1, \dots, m_n:\tau_n\}}$ *Objekterweiterung*	
$SUB\ \frac{\Gamma \vdash t:\sigma \quad \sigma \le \tau}{\Gamma \vdash t:\tau}$	Sumsumption, Typkonformanz.
$\to SUB\ \frac{\sigma' \le \sigma \quad \tau \le \tau'}{\sigma \to \tau \le \sigma' \to \tau'}$ Subtyp-Beziehungen auf Funktionen. Kontravarianz im Parameter, Kovarianz im Ergebnis *Dh. Parameter erben in die gegensätzliche Richtung, Rückgabetypen in dieselbe!*	
$refl\ \frac{}{\sigma \le \sigma}$	Reflexiv. ZB für Trivialumformung {..}=Klasse
$transitiv\ \frac{\tau \le \tau' \quad \tau' \le \tau''}{\tau \le \tau''}$	
$antisym\ \frac{\tau \le \tau' \quad \tau' \le \tau}{\tau = \tau'}$	
$\forall SUB\ \frac{\sigma' \le \sigma \quad \text{für alle } p \le \sigma' \text{ gilt } \tau[\alpha \leftarrow p] \le \tau'[\alpha \leftarrow p]}{\forall \alpha \le \sigma.\tau \le \alpha \le \sigma'.\tau'}$ Also: Polymorphe Typen erben, wenn die Typparameter in dieselbe Richtung erben. Zwischen monomorphe und polymorphen Typen kann keine Erben-Beziehung bestehen.	

5 Programmanalyse

5.1 Points-to-Analyse nach Andersen

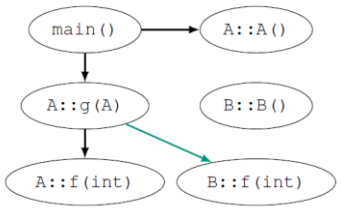
Fluss-sensitiv und Kontext-sensitiv.

Zuweisung neues Objekt `O x = new O();`	$\{o_1\} \subseteq PT(x)$
Zuweisung Variable `p = q;`	$PT(q) \subseteq PT(p)$
Return einer Funktion `class O { R func() {return x;}}`	$PT(x) \subseteq PT(ret_f^O)$
Aufruf dynamische Funktion `q=o.m(a)` $\forall C: Class$ $\frac{o_i \in PT(o) \quad type(o_i) = C \quad lookup(C,m) = R \quad D :: m(T\ p)}{PT(a) \subseteq PT(p) \quad \{o_i\} \subseteq PT(this_m^D) \quad PT(ret_m^D) \subseteq PT(q)}$ `mit statischem Lookup von typkorrekten o_i!`	
Attribut Lesen `x = p.f;`	$\frac{o_i \in PT(p)}{PT(o_i.f) \subseteq PT(x)}$
Attribut Schreiben `p.f = y;`	$\frac{o_i \in PT(p)}{PT(y) \subseteq PT(o_i.f)}$

5.2 Rapid-Type Analysis

Analysiere Aufrufstruktur als Graphen. Knoten sind Methodensignaturen, Kanten geben an ob innerhalb einer Methodensignatur eine andere aufgerufen werden *kann*. Wegen dynamischer Bindung müssen alle potentielle Ziele berücksichtigt werden.

```
1  class A {
2    void f(int x) { }
3    void g(A p) {p.f(42);}
4  }
5  class B extends A {
6    void f(int x) { }
7  }
8  void main() {
9    A p = new A();
10   p.g(p);
11 }
```



Umgang mit dynamischer Bindung: Wegen konservativer Approximation, Bestimme Menge Z der potentiellen Aufrufziele: finde aufgerufene Methode durch Static Lookup (Aufwärtssuche), füge aufgerufene Funktion und *alle* Methoden mit passender Signatur abwärts der Hierarchie dazu.

Reduktion der Call-Größe: Siehe Zusammenfassung.

5.2.1 RTA als Constraint Problem

Finde Mengen $R = LebendigeMethoden$ und $S = LebendigeKlassen$ mit:

$$\frac{}{main \in R} \qquad \frac{}{Main \in S} \qquad \frac{M \in R \quad new\ C() \in body(M)}{C \in S}$$

$$\frac{M \in R \wedge e.m(x) \in body(M) \wedge C \le staticType(e) \wedge C \in S \wedge lookup(C,m) = M'}{M' \in R}$$