Korrekte Software: Grundlagen und Methoden
Vorlesung 13 vom 16.06.16: Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Motivation: Ein kurzes Beispiel

```
void swap (int *x, int *y)
/** post \old(*x) == *y && \old(*y) == x; */
{
 int z;

 z= *x;
 *x= *y;
 *y= z;
}
```


Probleme

1. Gleichheit und Ungleichheit von Pointern:

$$read(S, \Gamma!x) \stackrel{?}{=} read(S, \Gamma!y)$$

2. Aliasing — unterschiedliche Referenzen auf das gleiche Objekt

3. Gültigkeit von Pointer und Undefiniertheit

Hoare-Regeln für Deklarationen

- Erste Näherung: skalare Typen

$$\frac{}{\Gamma \vdash \{P\}\,\{\,\}\,\{P|R\}}$$

$$\frac{\Gamma, (x : l) \vdash \{P\}\,\{ds\}\,\{Q|R\}}{\Gamma \vdash \{\lambda S.\, l = fresh(S) \wedge P(upd(S, l, init_t))\}\,\{\quad x;\ ds\}\,\{Q|R\}}$$

int

- $init_t$ ist initialer Wert für Typen t (unbestimmt)
- Aus Definition von *fresh* folgt direkt:

$$\forall l.\, l \in dom(\sigma) \longrightarrow l \neq fresh(\sigma)$$

Getypter Speicher

- Die Operation *fresh*(S) erzeugt einen frischen Speicherplatz
  - Nebenannahme: es gibt immer frischen Speicher
- Um Strukturen und Felder anzulegen, benötigen wir eine getypte Version.

$$sizeof(b) = 1 \qquad t \text{ ist skalarer Typ}$$
$$sizeof(\mathbf{struct}\ \{\}) = 0$$
$$sizeof(\mathbf{struct}\ \{\mathtt{t\ i;\ flds}\}) = sizeof(t) + sizeof(\mathbf{struct}\ \{\mathtt{flds}\})$$
$$sizeof(\mathtt{t\ id[as]}) = sizeof(t) \cdot as$$

- Damit:

$$fresh : \Sigma \rightarrow \mathbf{Type} \rightarrow \mathbf{Loc}$$
$$fresh(\sigma, t) = l \longleftrightarrow \forall 0 \leq i \leq sizeof(t).\, add(l, i) \notin dom(\sigma)$$

- Behandelt einfaches Aliasing

Erweiterungen

- Speicher wird erweitert, indem frischen Lokationen ein indeterminierter Wert zugewiesen wird. Damit sind diese Lokationen gültig, aber nicht sinnvoll lesbar.
- Um den Speicher um strukturierte Typen zu erweitern:

$$ext : \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{Type} \rightarrow \Sigma$$
$$ext(\sigma, l, t) = upd(\sigma, l, init_t) \qquad t \text{ ist skalarer Typ}$$
$$ext(\sigma, l, \mathbf{struct}\ \{\}) = \sigma$$
$$ext(\sigma, l, \mathbf{struct}\ \{\mathtt{t\ i;\ flds}\}) = ext(ext(\sigma, l, t), add(l, sizeof(t)), \mathbf{struct}\ \{\mathtt{flds}\})$$
$$ext(\sigma, l, \mathtt{t\ id[0]}) = \sigma$$
$$ext(\sigma, l, \mathtt{t\ id[n]}) = ext(ext(\sigma, l, t), add(l, sizeof(t)), \mathtt{t\ id[n-1]})$$

Erweiterte Hoare-Regeln für Deklarationen

$$\frac{}{\Gamma \vdash \{P\}\,\{\,\}\,\{P|R\}}$$

$$\frac{\Gamma, (x : l) \vdash \{P\}\,\{ds\}\,\{Q|R\}}{\Gamma \vdash \{\lambda S.\, l = fresh(S, t) \wedge P(ext(S, l, t))\}\,\{t\ x;\ ds\}\,\{Q|R\}}$$

Totale Korrektheit

- Partielle Korrektheit: wenn das Programm terminiert, erfüllt es die Nachbedingung.

Wie sinnvoll ist diese Aussage?

Mein Programm wäre richtig gewesen, wenn es nicht vorher abgestürtzt wäre.

- Wir wollen mindestens ausschließen, dass Laufzeitfehler ("undefined behaviour" *C99 Standard*, §3.4.3) auftreten.
- Problem: wenn Pointer als Parameter übergeben werden müssen sie dereferenzierbar sein.
- Dazu neue Annotationen: valid und array.

Neue Annotationen

- valid (l): l ist eine gültige Lokation

$$[\![\text{valid (l)}]\!]\Gamma \stackrel{def}{=} \lambda S.\, \{add([\![l]\!]\Gamma, x) \mid 0 \le x < sizeof(\mathbf{Type}(l))\} \subset dom(S)$$

- array (l, n): l ist eine gültige Lokation für ein Feld der Größe n.

$$[\![\text{array (a, n)}]\!]\Gamma \stackrel{def}{=} \lambda S.\, \{add([\![a]\!]\Gamma, x) \mid 0 \le x < n * sizeof(\mathbf{Type}(a))\} \subset dom(S)$$

- separated (a, m, b, n): Felder a[m] und b[n] sind disjunkt.

$$[\![\text{separated (a, m, b, n)}]\!]\Gamma \stackrel{def}{=} (\{add([\![a]\!]\Gamma, x) \mid 0 \le x < m * sizeof(\mathbf{Type}(a))\} \cap \{add([\![b]\!]\Gamma, x) \mid 0 \le x < n * sizeof(\mathbf{Type}(b))\}) = \emptyset$$

Funktionsparameter und Frame Conditions

- Problem: Funktionen können beliebige Änderungen im Speicher vornehmen.

```
int x, y, z;

z= x+ y;
swap(&x, &y);
/** { z = \old(x)+ \old(y) } */
```

- Vor/Nach dem Funktionsaufruf (hier swap) muss die Nachbedingung/Vorbedingung noch gelten.

Frame Rule

- Konstanzregel (Rule of Constancy):

$$\frac{\vdash \{P\}\, c\, \{Q\}}{\vdash \{P \wedge R\}\, c\, \{Q \wedge R\}}$$

- Problem: gilt mit Pointern nur eingeschränkt, da c eventuell Teile des Zustands verändert, über den R Aussagen macht.

Modification Sets

- Idee: Spezifiziere, welcher Teil des Zustands verändert werden darf.
  - ... denn wir können nicht spezifizieren, was gleich bleibt.
- Syntax: modifies **Mexp**

$$\mathbf{Mexp} ::= \mathbf{Loc} \mid \mathbf{Mexp}\,[*] \mid \mathbf{Mexp}\,[i:j] \mid \mathbf{Mexp}\,.\,\mathbf{name}$$

  - Mexp sind Lexp, die auch Teile von Feldern bezeichnen.
- Semantik: $[\![-]\!] : Env \to \mathbf{Mexp} \to \Sigma \to \mathbb{P}(\mathbf{Loc})$
- Modification Sets werden in die Hoare-Tripel integriert.

Semantik mit Modification Sets

- Hoare-Tripel mit Modification Sets:

$$\Lambda \models \{P\}\, c\, \{Q\} \longleftrightarrow \forall \sigma.\, P(\sigma) \wedge \exists \sigma'.\, \sigma' = c(\sigma) \longrightarrow Q(\sigma') \wedge \sigma \cong_\Lambda \sigma'$$

- wobei $\sigma \cong_L \tau \longleftrightarrow \forall l \in dom(\sigma) \cup dom(\tau) \setminus L.\, \sigma(l) = \tau(l)$
- oder alternativ $\sigma \cong_L \tau \longleftrightarrow \forall l.\, \sigma(l) \neq \tau(l) \longrightarrow l \in L$

Regeln mit Modification Sets

- Regeln werden mit Modification Set annotiert:

$$\Gamma, \Lambda \vdash \{P\}\, c\, \{Q_1 | Q_2\}$$

- Modification Set wird durchgereicht, aber:

$$\frac{}{\Gamma, \Lambda \vdash \{\lambda\sigma. [\![l]\!]\Gamma \in dom(\sigma) \wedge [\![l]\!]\Gamma \in \Lambda \wedge Q(upd(\sigma, [\![l]\!]\Gamma, [\![e]\!]\Gamma))\}\; l = e\; \{Q|R\}}$$

Das Beispiel vom Anfang

```
void swap(int *x, int *y)
/** modifies *x, *y;
    pre \valid(*x) && \valid(*y);
    post *x == \old(*y) && *y == \old(*x); */
{
  int z;

  z= *x;
  *x= *y;
  *y= z;
}
```

Brauchen wir pre x != y?

Swap (Annahme: &x != &y)

```
int swap(int *x, int *y) {
  /** { &x!=&y, *y == \old(*y), *x == \old(*x) } */
  int z;
  /** { &x!=&y, &z!=x, &z!=y, *y == \old(*y), *x == \old(*x) } */
  /** Beweis:
      [read(s,x)!=read(s,y), read(s,read(s,y)) == read(sold,read(sold,y)),
       read(s,z) == read(sold,read(sold,x))] upd(s,z,read(s,read(s,x)))
     <==> read(s,x)!=read(s,y), read(s,read(s,y)) == read(sold,read(sold,y)),
          read(s,read(s,x)) == read(sold,read(sold,x))
          : Da z!=read(s,x), z!=read(s,y) */
  z = *x;
  /** { &x!=&y, &z!=x, &z!=y, *y == \old(*y), z == \old(*x) } */
  /** { &x!=&y, *y == \old(*y), z == \old(*x) } */
  /** Beweis:
      [read(s,x)!=read(s,y), read(s,read(s,x)) == read(sold,read(sold,y)),
       read(s,z) == read(sold,read(sold,x))] upd(s,read(s,x),read(s,read(s,y)))
     <==> read(s,x)!=read(s,y), read(s,read(s,y)) == read(sold,read(sold,y)),
          read(s,z) == read(sold,read(sold,x))
        : Da z!=read(s,x), x!=read(s,x) und y!=read(s,x)  */
  *x = *y;
  /** { &x!=&y, *x == \old(*y), z == \old(*x) } */
  /** Beweis:
      [read(s,read(s,x)) == read(sold,read(sold,y)),read(s,read(s,y)) ==
       read(sold,read(sold,x))] upd(s,read(s,y),read(s,z))
     <==> read(s,read(s,x)) == read(sold,read(sold,y)), read(s,z) ==
          read(sold,read(sold,x)) : Da &x!=&y (read(s,x)!=read(s,y)) */
  *y = z;
  /** { &x!=&y, *x == \old(*y), *y == \old(*x) } */
  /** { *x == \old(*y), *y == \old(*x) } */
}
```


Swap (Annahme: &x == &y)

```
int swap(int *x, int *y) {
  /** { &x==&y, *x == \old(*x), *y == \old(*y) } */
  int z;
  /** { &x==&y, *x == \old(*x), *y == \old(*y) } */
  /** Beweis:
      [read(s,x) == read(s,y),
      read(s,z) == read(sold,read(sold,x)),
      read(s,z) == read(sold,read(sold,y))] upd(s,z,read(s,read(s,x)))
     <==> read(s,x)!=read(s,y), read(s,read(s,x)) == read(sold,read(sold,x)),
          read(s,read(s,y)) == read(sold,read(sold,y))
         -- da z!=&x, z != &y */
  z = *x;
  /** { &x==&y, z == \old(*x), z == \old(*y) } */
  /** Beweis:
     [read(s,x) == read(s,y),
      read(s,z) == read(sold,read(sold,x)),
      read(s,z) == read(sold,read(sold,y))](upd(s, read(s, x),read(s,read(s,y))))
   <==> read(s,x) == read(s,y), read(s,z) == read(sold,read(sold,y)),
          read(s,z) == read(sold,read(sold,x)) -- da z != &x, z != &y */
  *x = *y;
  /** { &x==&y, z == \old(*x), z == \old(*y) } */
  /** Beweis:
      [read(s, x) == read(s,y), read(s, read(s,x)) == read(sold,read(sold,x)),
       read(s, read(s, y)) == read(sold,read(sold,y))] (upd(s,read(s,y),read(s,z)))
     <==> read(s,x) == read(s,y), read(s,z) == read(sold,read(sold,x)),
          read(s,z) == read(sold,read(sold,y)) : Da read(s,x)==read(s,y) */
  *y = z;
  /** { &x==&y, *x == \old(*y), *y == \old(*x) } */
}
```


Zusammenfassung

- Herleitung von Gleichheit, Ungleichheit und Validität von Pointern ist schwierig.
- Dazu: kürzere Beschreibung des Zustands, Separation Logic
- Der Zustand ist immer noch sehr groß.
  - Wir können inbesondere keine Beweisverpflichtung zwischendurch erledigen.
- Dazu: Vorwärtsrechnung.