Korrekte Software: Grundlagen und Methoden
Vorlesung 9 vom 01.06.17: Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Motivation

- Weitere Basisdatentypen von C (arrays und structs)
- Noch rein funktional, keine Pointer
  - Damit auch kein *call by reference*
  - Funktion können nur globale Seiteneffekte haben
  - Was wäre C ohne Pointer?

Arrays

```
int a[1][2];

bool b[][] = { {1, 0},
               {1, 1},
               {0, 0} }; /* Ergibt Array [3][2] */

printf(b[2][1]);  /* liefert '0' */

int six[6] = {1,2,3,4,5,6};

// Allgemeine Form

typ name[groesse1][groesse2]...[groesseN] =
    { ... }
    x;
```


Struct

```
struct Point {
  int x;
  int y;
};

struct Point a = { 1, 2};
struct Point b;

b.x = a.x;
b.y = a.y;
```


Rekursive Struct

Rekursion nur über Pointer möglich:

```
struct Liste {
  int kopf;
  struct Liste *rest;
} start;

start.kopf = 10; /* start.rest bleibt undefiniert */
```

struct Liste *rest ist ein incomplete type.

Refenzen in C

- Pointer in C ("pointer type"):
  - Schwach getypt (**void** * kompatibel mit allen Zeigertypen)
  - Eingeschränkte Zeigerarithmetik (Addition, Subtraktion)
  - Felder werden durch Zeigerarithmetik implementiert
- Pointer sind *first-class-values*
- C-Standard läßt das Speichermodell relativ offen
  - Repräsentation von Objekten

Erweiterung des Zustandmodells

- Erweiterung von Zustand und Werten:

$$\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V} \qquad \mathbf{V} = \mathbf{N} + \mathbf{Loc}$$

- Was ist **Loc**?
  - Locations (Speicheradressen)
  - Man kann **Loc** axiomatisch oder modellbasiert beschreiben.

Axiomatisches Zustandsmodell

- Der Zustand ist ein abstrakter Datentyp Σ mit zwei Operationen und folgenden Gleichungen:

$$read : \Sigma \to \mathbf{Loc} \rightharpoonup \mathbf{V}$$
$$upd : \Sigma \to \mathbf{Loc} \to \mathbf{V} \rightharpoonup \Sigma$$

$$read(upd(\sigma, l, v), l) = v$$
$$l \neq m \implies read(upd(\sigma, l, v), m) = read(\sigma, m)$$
$$upd(upd(\sigma, l, v), l, w) = upd(\sigma, l, w)$$
$$l \neq m \implies upd(upd(\sigma, l, v), m, w) = upd(upd(\sigma, m, w), l, v)$$

- Diese Gleichungen sind vollständig.

Axiomatisches Speichermodell

- Es gibt einen leeren Speicher, und neue ("frische") Adressen:

$$empty : \Sigma$$
$$fresh : \Sigma \to \mathbf{Loc}$$
$$rem : \Sigma \to \mathbf{Loc} \to \Sigma$$

  - *fresh* modelliert Allokation, *rem* modelliert Deallokation
- *dom* beschreibt den Definitionsbereich:

$$dom(\sigma) = \{l \mid \exists v.\, read(\sigma, l) = v\}$$
$$dom(empty) = \emptyset$$

- Eigenschaften von *empty*, *fresh* und *rem*:

$$fresh(\sigma) \notin dom(\sigma)$$
$$dom(rem(\sigma, l)) = dom(\sigma) \setminus \{l\}$$
$$l \neq m \implies read(rem(\sigma, l), m) = read(\sigma, m)$$

Zeigerarithmetik

- Erklärt noch keine Zeigerarithmetik — dazu:

$$add : \mathbf{Loc} \to \mathbb{Z} \to \mathbf{Loc}$$

  - Wir betrachten keine Differenz von Zeigern

$$add(l, 0) = l$$
$$add(add(l, a), b) = add(l, a + b)$$

Erweiterung der Semantik

- Problem: **Loc** haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.
  - x = x+1 — Links: Addresse der Variablen, rechts: Wert an dieser Adresse
- Lösung: "Except when it is (...) the operand of the unary & oprerator, the left operand of the . operator or an assigment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)"
  C99 Standard, §6.3.2.1 (2)

Umgebung

- Für Funktionen brauchten wir eine Umgebung (Environment):

$$Env = Id \rightharpoonup [\![\mathbf{FunDef}]\!]$$
$$= Id \rightharpoonup \mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)$$

- Diese muss erweitert werden für Variablen:

$$Env = Id \rightharpoonup ([\![\mathbf{FunDef}]\!] \uplus \mathbf{Loc})$$

  - Insbesondere: gleicher Namensraum für Funktionen und Variablen (*C99 Standard*, §6.2.3)

Ausdrücke

Syntaktische Klasse von Ausdrücken, die eine Location bezeichnen (**Lexp**):

$$\begin{aligned}
\textbf{Lexp}\ l &::= Id \mid l\ [a] \mid l.Id \mid *a \\
\textbf{Aexp}\ a &::= \mathbf{N} \mid l \mid \&l \mid a_1 + a_2 \mid a_1 - a_2 \mid \\
&\quad a_1 * a_2 \mid a_1 / a_2 \mid Id(a^*) \\
\textbf{Bexp}\ b &::= \mathbf{0} \mid \mathbf{1} \mid a_1 == a_2 \mid a_1 != a_2 \mid \\
&\quad a_1 <= a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\
\textbf{Exp}\ e &:= a \mid b
\end{aligned}$$

Statements

$$\begin{aligned}
\textbf{Type} &::= \textbf{PointerType} \mid \textbf{BasicType} \mid \textbf{StructType} \mid \textbf{ArrayType} \\
\textbf{BasicType} &::= int \\
\textbf{StructType} &::= struct\ \textbf{name}\ \{\textbf{Puredecl}*\} \\
\textbf{ArrayType} &::= \textbf{Type}[\textbf{n}?] \\
\textbf{PointerType} &::= \textbf{Type}* \\
\textbf{ExtDecl} &::= \textbf{LogSpec} \mid \textbf{Decl} \mid \textbf{FunDef} \\
\textbf{Decl} &::= \textbf{Type}\ Id\ (=\ e|\{el\})?; \\
\textbf{FunDef} &::= \textbf{Type}\ Id(\textbf{PureDecl}^*)\ \textbf{FunSpec}^+\ \textbf{Blk} \\
\textbf{Blk} &::= \{\textbf{Decl}^*\ \textbf{Stmt}\} \\
\textbf{Puredecl} &::= \textbf{Type}\ Id \\
\textbf{Stmt} &::= \textbf{Lexp} = \textbf{Exp};\ |\ \textbf{if}\ (\ b\ )\ c_1\ \textbf{else}\ c_2 \\
&\quad |\ \textbf{while}\ (\ b\ )\ c\ |\{c^*\} \\
&\quad |Id(\textbf{AExp}*)\ |\ \textbf{return}\ (\textbf{AExp}?)
\end{aligned}$$

Erweiterung der Semantik: Lexp

$$\mathcal{L}[\![-]\!] : Env \to \textbf{Lexp} \to \Sigma \rightharpoonup \textbf{Loc}$$

$$\begin{aligned}
\mathcal{L}[\![x]\!]\ \Gamma &= \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\} \\
\mathcal{L}[\![lexp[a]]\!]\ \Gamma &= \{(\sigma, add(l, i \cdot sizeof(\tau))) \mid (\sigma, l) \in \mathcal{L}[\![lexp]\!]\ \Gamma, (\sigma, i) \in \mathcal{A}[\![a]\!]\ \Gamma\} \\
&\quad type(\Gamma, lexp) = \tau \text{ ist der Basistyp des Feldes} \\
\mathcal{L}[\![lexp.f]\!]\ \Gamma &= \{(\sigma, l.f) \mid (\sigma, add(l, fld_off(\tau, f))) \in \mathcal{L}[\![lexp]\!]\ \Gamma\} \\
&\quad type(\Gamma, lexp) = \tau \text{ ist der Typ der Struktur} \\
\mathcal{L}[\![*e]\!]\ \Gamma &= \mathcal{A}[\![e]\!]\ \Gamma
\end{aligned}$$

- $type(\Gamma, e)$ ist der Typ eines Ausdrucks
- $fld\_off(\tau, f)$ ist der Offset des Feldes f in der Struktur τ
- $sizeof(\tau)$ ist die Größe von Objekten des Typs τ

Erweiterung der Semantik: Aexp(1)

$$\mathcal{A}[\![-]\!] : \mathit{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
\mathcal{A}[\![n]\!]\ \Gamma &= \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbf{N} \\
\mathcal{A}[\![e]\!]\ \Gamma &= \{(\sigma, \mathit{read}(\sigma, l)) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\quad e \text{ is } \mathbf{LExp} \text{ und } \mathit{type}(\Gamma, e) \text{ kein Array-Typ} \\
\mathcal{A}[\![e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\quad e \text{ is } \mathbf{LExp} \text{ und } \mathit{type}(\Gamma, e) \text{ Array-Typ} \\
\mathcal{A}[\![\&e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
\mathcal{A}[\![p + e]\!]\ \Gamma &= \{(\sigma, \mathit{add}(l, n \cdot \mathit{sizeof}(\tau)))) \mid (\sigma, l) \in \mathcal{L}[\![p]\!]\ \Gamma \wedge (\sigma, n) \in \mathcal{A}[\![e]\!]\ \Gamma\} \\
&\quad \mathit{type}(\Gamma, p) = * \tau,\ \mathit{type}(\Gamma, e) \text{ Integer-Typ} \\
\mathcal{A}[\![e + p]\!]\ \Gamma &= \mathcal{A}[\![p + e]\!]\Gamma \\
&\quad \mathit{type}(\Gamma, e) \text{ Integer-Typ und } \mathit{type}(\Gamma, p) = * \tau
\end{aligned}$$

Erweiterung der Semantik: Aexp(2)

$$\mathcal{A}[\![-]\!] : \mathit{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
\mathcal{A}[\![a_0 + a_1]\!]\ \Gamma &= \{(\sigma, n_0 + n_1 \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
&\quad \text{für } a_0, a_1 \text{ arithmetische Typen} \\
\mathcal{A}[\![a_0 - a_1]\!]\ \Gamma &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
\mathcal{A}[\![a_0 * a_1]\!]\ \Gamma &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
\mathcal{A}[\![a_0 / a_1]\!]\ \Gamma &= \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma \\
&\quad \wedge n_1 \neq 0\}
\end{aligned}$$

Explizite Zustandsprädikate

- Erweiterung der **Aexp** um *read*, neue Sorte **St** mit Operation *upd*:

$$\begin{aligned}
\mathbf{Bexp} &::= \ldots \text{(wie vorher)} \\
\mathbf{Aexp} &::= \mathit{read}(\mathbf{St}, \mathbf{Lexp}) \mid \mathbf{N} \mid \mathbf{Lexp} \mid \&\mathbf{Lexp} \mid \ldots \mid \backslash\mathbf{old}(e) \mid \ldots \\
\mathbf{St} &::= \mathit{StateVar} \mid \mathit{upd}(\mathbf{St}, \mathbf{Aexp}, \mathbf{Bexp})
\end{aligned}$$

- Zustandsvariablen *StateVar*: Aktueller Zustand σ, Vorzustand ρ
- Damit Semantik:

$$\begin{aligned}
\mathcal{B}_{sp}[\![.]\!] &: \mathit{Env} \to \mathbf{Bexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{T} \\
\mathcal{A}_{sp}[\![.]\!] &: \mathit{Env} \to \mathbf{Aexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}
\end{aligned}$$

- Explizite Zustandsprädikate enthalten kein $*$ oder &

Hoare-Triple

$$\Gamma \models \{P\}\, c\, \{Q|R\}$$

- P, Q, R sind explizite Zustandsprädikate
- Deklarationen (**Decl**) allozieren für jede Variable eine Location, und ordnen diese in der Umgebung zu.
- Restriktion: keine dynamische Allokation von Variablen (malloc und Freunde)
- Gültigkeit wie vorher

Floyd-Hoare-Kalkül mit expliziten Zustandsprädikaten

$$\overline{\Gamma \vdash \{Q[\mathit{upd}(\sigma, x, e)/\sigma]\}\, x = e\, \{Q|R\}}$$

- Ein **Lexp** l auf der rechten Seite e wird durch $\mathit{read}(\sigma, l)$ ersetzt.[1]
- & dient lediglich dazu, diese Konversion zu verhindern.
- $*$ erzwingt diese Konversion, auch auf der linken Seite x.
- Beispiel: $*a = *\&b;$.

[1] Außer l ist ein Array-Typ.

Formal: Konversion in Zustandsprädikate

$$\begin{aligned}
(-)^{\dagger} &: \mathbf{Lexp} \to \mathbf{Lexp} \\
v^{\dagger} &= v \quad (v \text{ Variable}) \\
l.id^{\dagger} &= l^{\dagger}.id \\
l[e]^{\dagger} &= l^{\dagger}[e^{\#}] \\
*l^{\dagger} &= l^{\#}
\end{aligned}$$

$$\begin{aligned}
(-)^{\#} &: \mathbf{Aexp} \to \mathbf{Aexp} \\
e^{\#} &= \mathit{read}(\sigma, e^{\dagger}) \quad (e \in \mathbf{Lexp}) \\
n^{\#} &= n \\
v^{\#} &= v \quad (v \text{ logische Variable}) \\
\&e^{\#} &= e^{\dagger} \\
e_1 + e_2{}^{\#} &= e_1{}^{\#} + e_2{}^{\#} \\
\backslash\mathbf{result}^{\#} &= \backslash\mathbf{result} \\
\backslash\mathbf{old}(e)^{\#} &= \backslash\mathbf{old}(e)
\end{aligned}$$

$$\overline{\Gamma \vdash \{Q[\mathit{upd}(\sigma, x^{\dagger}, e^{\#})/\sigma]\}\, x = e\, \{Q|R\}}$$

Zwei kurze Beispiele

```
void foo(){
 int x, y, z;
 /** { True } */
 z= x;
 x= 0;
 z= 5;
 y= x;
 /** { y == 0 } */
}
```

```
void foo(){
 int x, y, *z;
 /** { True } */
 z= &x;
 x= 0;
 *z= 5;
 y= x;
 /** { y == 5 } */
```


Weiteres Beispiel: Strukturen

```
struct Point {
  int x;
  int y;
 };
struct Point a = { 1, 2};
struct Point b;
b.x = a.x;
b.y = a.y;
{ b.x == a. x }
```

Weitere Beispiele: Felder

```
#include <limits.h>
#define N 10
int a[N];
int findmax()
  /** post \forall int i; 0 <= i && i < 10
                          ==> a[i] <= \result; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< N) {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```

Voller Beweis auf der Webseite (Quellen, `findmax-annotated.c`)

Felder und Zeiger revisited

- In C sind Zeiger und Felder schwach spezifiziert
- Insbesondere:
  - a[j]= *(a+j) für a Array-Typ
  - Derefenzierung von *x nur definiert, wenn x "gültig" ist (d.h. auf ein Objekt zeigt) *C99 Standard*, §6.5.3.2(4)
- Bisher in den Hoare-Regeln ignoriert — partielle Korrektheit.
- Ist das sinnvoll? Nein, bekannte Fehlerquelle

Spezifikation von Zeigern und Feldern

Das Prädikat \valid(x)

\valid($*x$) für x Pointer-Typ $\iff$ $*x$ ist definiert.

- Felder als Parameter werden Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger "in Wirklichkeit" ein Feld ist.
- **\array**(a, n) bedeutet: a ist ein Feld der Länge n, d.h.

$$\textbf{\array}(a, n) \iff (\forall i.\, 0 \leq i < n \implies \textbf{\valid}(a[i]))$$

- Validität kann abgeleitet werden:

$$\frac{\texttt{x= \&e}}{\texttt{\valid(*x)}} \qquad \frac{\texttt{\array(a, n)} \quad 0 \leq i \quad i < n}{\texttt{\valid(a[i])}}$$

Zusammenfassung

- Um Referenzen (Pointer) in C behandeln zu können, benötigen wir ein Zustandsmodell
- Referenzen werden zu Werten wie Zahlen oder Zeichen.
  - Arrays und Strukturen sind keine first-class values.
  - Großes Problem: aliasing
- Erweiterung der Semantik und der Hoare-Tripel nötig:
  - Vor/Nachbedingungen werden zu expliziten Zustandsprädikaten.
  - Zuweisung wird zu Zustandsupdate.
  - Problem:
    - Zustände werden sehr groß
    - Rückwärtsrechnung erzeugt schnell sehr große „unbestimmte" Zustände, die nicht vereinfacht werden können
  - Daher: Verifikationsbedingungen berechnen