5. Separation Logic

basierend auf Vorlesungsaufzeichnungen von John C. Reynolds, CMU

Ziel: Erweitere Hoare-Logik um nächtige Zusicherungen,
um die Trennung von Resourcen (inkl. Heap) zu beschreiben.

Historie: Ursprünglich gemacht für das Reasoning über
shared mutable Datenstrukturen.
deren Einträge von mehr als einem Ort
referenziert und geändert werden können.
Separation hat sich als sehr nützlich herausgestellt,
neben Memory auch Permissions für Capabilities oder Knowledge.

Problem: In Logik ist Sharing default,
in Programmen ist Separation default.
Entsprechend aufwendig ist es, in Logik Separation zu beschreiben
(man muss explizit sagen, wo es nicht auftritt).

Beispiel:
In-Place List-Reversal:

```
LREV =   j := nil;
         while i ≠ nil do
                                  // Bedeutet i -> next
           h := [i+1];
           [i+1] := j;
           j := i;
           i := h;
         od
```

Hier soll [e] bedeuten: Content an Adresse e.
Wenn an der in i gespeicherten Adresse

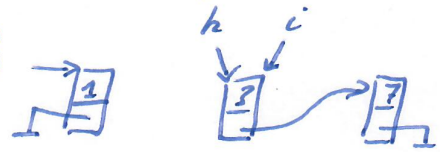
Ein Struct hat die Form:

```
Struct Node {
    Datum d;
    Node* next;
}
```

dann ist [i+1] der Wert des next-Fields und [i] das Datum.

Betrachte die Liste

Nach einem Durchlauf der Schleife haben wir:



Nach einem weiteren Durchlauf erhalten wir bei erneutem Schleifeneintritt:

Ein letzter Durchlauf liefert:

Korrektheit:

Die Invariante muss sagen:

- i und j sind Listen α, β (dargestellt als Wort).

· Die Umkehrung der initialen Liste α_0
ist die Konkatenation der Umkehrung von α mit β.

Als Formel:

$$\exists \alpha, \beta.\ list\ \alpha\ i \wedge list\ \beta\ j \wedge \alpha_0^t = \alpha^t . \beta .$$

Dafür ist das Prädikat $list\ \alpha\ i$
induktiv über die Struktur von Listen definiert:

$$list\ \varepsilon\ i := i = nil$$

$$list\ (a.\alpha)\ i := \exists j.\ \underbrace{i \hookrightarrow a, j}_{\text{points-to}} \wedge list\ \alpha\ j$$

Das wird aber nicht genügen?
Solange es Sharing zwischen α und β gibt, passiert Unfug.
Also müssen wir garantieren, dass nur nil
sowohl von i als auch von j aus erreichbar ist:

$$INV_0': \quad \exists \alpha, \beta.\ list\ \alpha\ i \wedge list\ \beta\ j \wedge \alpha_0^t = \alpha^t . \beta$$
$$\wedge \forall h.\ reachable(i,h) \wedge reachable(j,h) \longrightarrow h = nil.$$

mit

$$reachable(i,j) := \exists n \geq 0.\ reachable_n(i,j)$$
$$reachable_0(i,j) := i = j$$
$$reachable_{n+1}(i,j) := \exists a, h.\ i \hookrightarrow a, h \wedge reachable_n(h,j).$$

Sogar schlimmer, angenommen LDEV ist nur ein Teilprogramm
und es gibt eine andere Liste δ, auf die x zeigt.

Dann darf γ nicht ausversehen von unseren Änderungen betroffen sein.

INV': $\exists \alpha, \beta.\ list\ \alpha\ i \wedge list\ \beta\ j \wedge \alpha_0^+ = \alpha^+ . \beta$

$\wedge\ \forall h.\ reachfld(i,h) \wedge reachfld(j,h) \rightarrow h = nil$

$\wedge\ list\ \gamma\ x$

$\wedge\ \forall h.\ reachfld(x,h) \wedge (reachfld(i,h) \vee reachfld(j,h))$

$\rightarrow h = nil.$

Keine Chance, Reasoning mit solch schweren Invarianten zu automatisieren.

Lösung:

- Separation-Logik führt einen neuen logischen Operator ein,

$P * Q$,

genannt Separating-Conjunction, die besagt:

P und Q gelten für disjunkte Teile des Heaps.

Damit werden obige Invarianten zu

INV_0: $\exists \alpha, \beta.\ (list\ \alpha\ i * list\ \beta\ j) \wedge \alpha_0^+ = \alpha^+ . \beta$

INV: $\exists \alpha, \beta.\ (list\ \alpha\ i * list\ \beta\ j * list\ \gamma\ x) \wedge \alpha_0^+ = \alpha^+ . \beta.$

- Mit der Separating-Conjunction wird auch ein

lokales Reasoningprinzip

in die Programmlogik eingeführt.

Mit Hilfe von INV_0 bekommt man ein Hoare-Tripel

$\{list\ \alpha\ i\}\ LREV\ \{list\ \alpha^+\ j\}$

gezeigt.

Das Hoare-Tripel sagt nun nicht allein aus,
dass das Programm an Adresse i die Liste α findet.
Es sagt insbesondere aus, dass α die einzige Teil des Heaps ist,
die von der Ausführung von LREV betroffen ist.
Man nennt den betroffenen Teil den Footprint von LREV.

- Eine neue Beweisregel, Peter O'Hearns berühmte

  Frame-Rule,

  erlaubt uns nun, aus IV Folgendes abzuleiten:

$$\{list\ \alpha\ i * list\ \beta\ x\}\ LREV\ \{list\ \alpha^+\ j * list\ \beta\ x\}.$$

Weiterer Storage (im gesamten Programm Liste β an x)
bleibt von der Ausführung von LREV unverändert.

Motivation: Skalierbarkeit automatischer Programmanalysen.

Terminologie: Man bezeichnet sowohl die Hoare-Sprache
als auch die erweiterte Hoare-Logik
als Separation-Logik.

5.1 Programmiersprache

Ziel: Erweitere WHILE um Pointer.

Definition:
Programme der Sprache W sind durch folgende BNF definiert.

$$c ::= x := \underbrace{cons(a_1, \ldots, a_n)}_{\text{Allokation}} \mid \underbrace{x := [a]}_{\text{Lookup}} \mid \underbrace{[a_1] := a_2}_{\text{Mutation}} \mid \underbrace{dispose\ a}_{\text{Deallokation}}$$

| skip | x := a | assume b

| c_1 ; c_2 | if b then c_1 else c_2 fi // Wie bisher.

| while b do c od

Semantik:

- Explizites Memory-Management, keine Garbage-Collection.
- Dereferenzierung von Dangling-Pointern führt zu Fehlern.
- Konfigurationen haben zwei Komponenten: Stack und Heap.

  Stack: Abbildung von Variablen auf Werte (wie bisher)

  Heap: Abbildung von Adressen auf Werte.
- Adressen, Werte = alle $\mathbb{Z}$.

Definition:

Wir nutzen $\mathbb{Z}$ als Adressen mit einem ausgezeichneten Element $nil \in \mathbb{Z}$.

$$Stacks := Var \to \mathbb{Z}$$

$$Heaps := \mathbb{Z} \setminus \{nil\} \overset{fin}{\to} \mathbb{Z}$$

partielle Funktionen mit endl. Domäne.

$$Configs := Stacks \times Heaps,$$

dabei ist Var die Menge an Programmvariablen (endlich).

Die Semantik arithmetischer und Boolescher Ausdrücke ist definiert wie bisher.
Insbesondere hängen sie nur vom Stack ab und haben keinen Seiteneffekt auf den Heap.
Die Semantik der neuen Befehle machen wir an Beispielen.

Beispiel:

Stack: x : 3, y : 4
Heap: ∅

Allokation x := cons (1, 2);

⇓

Stack: x : 37, y : 4
Heap: 37 : 1, 38 : 2

Lookup y := [x];

⇓

Stack: x : 37, y : 1
Heap: 37 : 1, 38 : 2

Mutation [x+1] := 3;

⇓

Stack: x : 37, y : 1
Heap: 37 : 1, 38 : 3

Deallokation dispose (x+1);

⇓

Stack: x : 37, y : 1
Heap: 37 : 1.

Bemerkung:

- Allokation cons (a1, ..., an) aktiviert (fügt der Domäne hinzu) und initialisiert n Zellen auf dem Heap.
  Die Zellen müssen nebeneinander liegen und vorher inaktiv gewesen sein.
  Die Wahl der Adressen ist nicht-deterministisch.
- Sofern Mutation, Lookup oder Deallokation inaktive Adressen dereferenzieren oder deallozieren ergibt sich ein Fehler, angezeigt durch die Termination mit gewählter aSort.

Beispiel:

Stack: x: 3, y: 4
Heap: ∅

Allokation `x := cons(2,2);`

⇓

Stack: x: 3, y: 4
Heap: 37: 1, 38: 2

Mutation `[x+2] := 3;`

⇓

aSat