Es gibt zwei weitere Regeln,
die insbesondere dazu dienen,
neue Beweisregeln abzuleiten:

$$(\forall UX) \quad \frac{\{A\}\, c\, \{B\}}{\{\exists x.\, A\}\, c\, \{\exists x.\, B\}}, \text{ wobei } x \notin free(c)$$

$$(SUBST) \quad \frac{\{A\}\, c\, \{B\}}{\{A\Theta\}\, c\, \{B\Theta\}},$$

wobei $\Theta = \{x_1/a_1, \ldots, x_n/a_n\}$ eine Substitution ist,
in der $x_1, \ldots, x_n$ freie Variablen von A, c oder B sind
und a_i wiederum Ausdrücke mit folgender Eigenschaft:

Falls x_i von c geändert wird,
also $x_i \in modifies(c)$ mit $\boxed{modifies(c) := \{x \mid \exists \text{ in } c \text{ Befehl: } x := a \text{ oder } x :=_2 [a]\}}$,

dann ist a_i eine Variable,
die in keinem a_j vorkommt.

Beispiel (Subst):

Betrachte $\{x = y\}\ x := x + y\ \{x = 2y\}$.

Wir können folgende Substitution vornehmen (Beachte $\{x\} = modifies(c)$)

$$\Theta = \{x/z,\ y/2(w-1)\}$$

und erhalten

$$\{z = 2(w-1)\}\ z := z + 2(w-1)\ \{z = 2 \cdot 2(w-1)\}.$$

Allerdings dürfen wir nicht

$$\Theta' = \{x/z,\ y/2(z-1)\} \text{ anwenden.}$$

6.2 Frame-Rule

Beobachtung:

- In einem gültigen SL-Tripel $\{A\}\ c\ \{B\}$ spezifiziert A einen Heap-Bereich, der ausreichend ist, um Programm c auszuführen.
- Wenn wir das Programm in einen Zustand starten, in dem es weitere Heap-Zellen gibt, bleiben diese unverändert.

Definition:

$$\text{(FRAME)}\quad \frac{\{A\}\ c\ \{B\}}{\{A * C\}\ c\ \{B * C\}}\ , \text{ wobei } \mathrm{modifies}(c) \cap \mathrm{free}(C) = \emptyset.$$

Bemerkung:

Die Separating-Conjunction quantifiziert nur, dass Assertion C über disjunkten Heaps spricht.

Um sicherzustellen, dass Programm c die Stack-Variablen in C nicht ändert, nutzen wir die Nebenbedingung.

Soundness der Frame-Rule ist trotzdem nicht klar.

Was, wenn c Speicher alloziert, der in C vorkommt?

Theorem (Soundness der Frame-Rule):

$$\models \{A\}\ c\ \{B\} \quad \Rightarrow \quad \models \{A * C\}\ c\ \{B * C\}.$$

Der Beweis benötigt ein Lemma über das Verhalten von Programmen/
die Semantik unserer Programmiersprache.
Wenn wir den Heap einer Berechnung einschränken,
die Berechnung bricht aber nicht ab,
dann lässt die ursprüngliche Berechnung den extra Heap unverändert.

Lemma (Yang & O'Hearn 2000) aus "A Semantic Basis for Local Reasoning"

(1) $(c, s, h_1 \circ h_2) \to^* abort \Rightarrow (c, s, h_1) \to^* abort.$

(2) $(c, s, h_1 \circ h_2) \to^* (s', h') \wedge (c, s, h_1) \not\to^* abort$

$\Rightarrow \exists h_1'. (c, s, h_1) \to^* (s', h_1') \wedge h' = h_1' \circ h_2.$

(3) $(c, s, h) \to^* (s; h') \wedge x \notin modifies(c) \Rightarrow s(x) = s'(x).$

Beweis (Soundness):

Annahme $\vdash \{A\}\, C\, \{B\}.$

Wir müssen zeigen:

$$\forall (s,h) \in Stacks \times Heaps.\ [\![A * C]\!]\, s\, h = 1 \Rightarrow$$

$$\left[\begin{array}{l} (C, s, h) \not\to^* abort \\ \wedge\ \forall s', h'.\ (C, s, h) \to^* (s', h') \Rightarrow [\![B * C]\!]\, s'\, h' = 1 \end{array}\right]$$

Sei (s,h) ein Paar aus Stack und Heap mit $[\![A * C]\!]\, s\, h = 1.$

Per Definition der Semantik von Assertions gilt $h = h_1 \circ h_2$ mit

$$[\![A]\!]\, s\, h_1 = 1 \quad \text{und} \quad [\![C]\!]\, s\, h_2 = 1.$$

Fall 1:

Annahme $(C, s, h) \to^* abort.$

Dann gilt mit Lemma (1):

$$(c, s, h_1) \to^* \underline{\text{abort}}.$$

Das ist ein Widerspruch zu $[\![A]\!]\, s\, h_1 = 1$

und $\vdash \{A\}\, c\, \{B\}$.

Der Fall tritt also nicht ein.

Fall 2:

Angenommen $(c, s, h) \to^* (s', h')$.

Wie in Fall 2 würde $(c, s, h_1) \to^* \underline{\text{abort}}$

zu einem Widerspruch führen.

Mit Lemma (2) gibt es h_1' so, dass

$$(c, s, h_1) \to^* (s', h_1') \wedge h' = h_1' \circledast h_2.$$

Da $[\![A]\!]\, s\, h_1 = 1$ und $\vdash \{A\}\, c\, \{B\}$, folgt

$$[\![B]\!]\, s'\, h_1' = 1.$$

Da $free(c) \cap modifies(c) = \emptyset$, folgt mit Lemma (3):

$$s'(x) = s(x)$$

für alle $x \in free(c)$.

Da $[\![C]\!]\, s\, h_2 = 1$, folgt

$$[\![C]\!]\, s'\, h_2 = 1.$$

Also $[\![B * C]\!]\, s', h_1' \circledast h_2 = 1 = [\![1) * C]\!]\, s', h'$.

□

Die Frame-Rule ist in gewissem Sinn auch vollständig.

Bemerkung (Yang 2001):

Angenommen $\vdash \{A\}c\{B\}$ impliziert $\vdash \{A'\}c\{B'\}$.
Dann lässt sich $\{A'\}c\{B'\}$ aus $\{A\}c\{B\}$ ableiten
mit

(CONSEQUENCE), (AUX), (SUBST) und (FRAME).

6.3 Small-Axioms

Ziel: Nutze die Frame-Rule,
um lokale Versionen der Axiome für Befehle zu finden.

Definition (Small-Axioms):

(LMUTV) $\{a_1 \mapsto \_\}\ [a_1] := a_2\ \{a_1 \mapsto a_2\}$

(LDISP) $\{a \mapsto \_\}$ dispose a $\{emp\}$

(LALLOCV) $\{emp \wedge x = x'\}\ x := cons(a_1, \dots, a_n)\ \{x \mapsto a_1[x/x'], \dots, a_n[x/x']\}$

(LLOOKV) $\{x = x' \wedge a \mapsto z\}\ x := [a]\ \{x = z \wedge a[x/x'] \mapsto z\}$.

Mutation $[a_1] := a_2$:

Dieses Axiom ergibt sich aus (GMUTV),
indem wir $B = emp$ wählen.
Umgekehrt ergibt sich (GMUTV) aus (LMUTV)
mittels (FRAME).
Übrigens ergibt sich auch (GMUTV) aus (GMUTV) und umgekehrt
→ siehe Übung.

Deallokation disposes:

Der Zusammenhang zwischen (GDISP) und (LDISP) ist genauso.

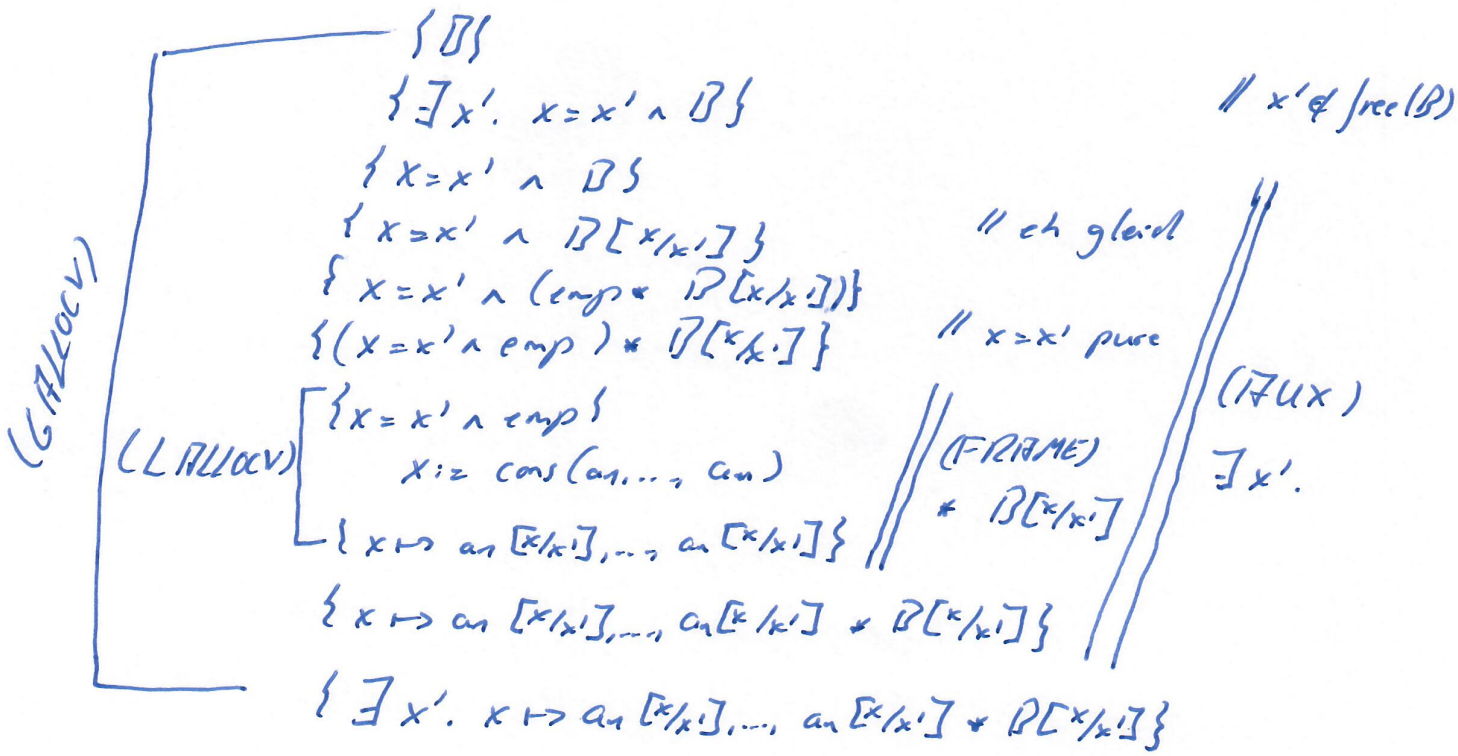
Allokation $x := cons(a_1 \dots a_n)$:

Lemma:

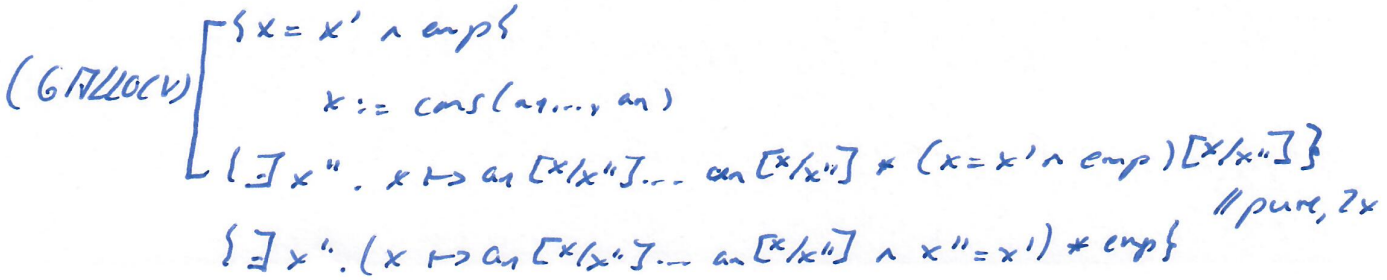
(LALLOCV) und (GALLOCV) lassen sich aus dem jeweils anderen Axiom herleiten.

Beweis:

(GALLOCV) aus (LALLOCV):



(LALLOCV) aus (GALLOCV):



(LALLOCV) $\left\{\begin{array}{l} \{\exists x''.\ x \mapsto a_n[x/x'']\ldots a_n[x/x''] \wedge x''=x'\} \\ \{\exists x''.\ x \mapsto a_1[x/x']\ldots a_n[x/x'] \wedge x''=x'\} \\ \{x \mapsto a_n[x/x']\ldots a_n[x/x']\} \end{array}\right.$

□

Lookup $x := [a]$:

Auch hier kann mit (LLOOKV) aus (GLOOKV) abgeleitet und umgekehrt.