8. Concurrent-Separation-Logic

nach: "Concurrent Separation Logic and Operational Semantics"
Viktor Vafeiadis 2011

"Resources, Concurrency, and Local Reasoning"
Peter O'Hearn 2007

Gödel-Award 2016 für Peter O'Hearn und Stephen Brookes für die Entwicklung von CSL.

Ziel:
- Erweitere unsere Programmiersprache W (ohne Rekursion) um Nebenläufigkeit.
- Kapsele nebenläufig ausgeführte Berechnungen in atomare Blöcke
  → Coarse-grained-Concurrency
- Zeige Korrektheit formuliert als Gültigkeit von Hoare-Tripeln.

Ansatz:
- Analysiere jeden Thread in Isolation.
- Modelliere die Interaktion zwischen den Threads über eine Resourcen-Invariante.
- Zeige ein Analogon der Frame-Rule, um die Thread-lokalen Beweise zusammenzusetzen.

8.1 Programmiersprache

Definition:
Programme der Sprache ParW sind induktiv definiert wie folgt:

$$c ::= \underbrace{\ldots}_{\text{wie in } W} \mid \underbrace{c_1 \parallel c_2}_{\text{parallel}} \mid \underbrace{\text{atomic } c}_{\text{atomarer Block}} \mid \underbrace{\text{inatom } c}_{\substack{\text{nutzen wir} \\ \text{in der Semantik,} \\ \text{nicht in Programmtext.}}}$$

Die Semantik der sequentiellen Befehle bleibt unverändert.

Für atomare Blöcke gehen wir von <u>atomic</u> c zu <u>inatom</u> c über.
Dann verhalten wir uns wie c, bis <u>inatom</u> skip erreicht ist.
Dann beenden wir den atomaren Block.

$$(\text{ATOM})\ \frac{}{(\underline{\text{atomic}}\ c,\ s,h) \to (\underline{\text{inatom}}\ c,\ s,h)}$$

$$(\text{INATOMSTEP})\ \frac{(c,s,h) \to (c',s',h')}{(\underline{\text{inatom}}\ c,\ s,h) \to (\underline{\text{inatom}}\ c',s,h')}$$

$$(\text{INATOMEND})\ \frac{}{(\underline{\text{inatom}}\ \text{skip},\ s,h) \to (\text{skip},\ s,h)}$$

$$(\text{INATOMABORT})\ \frac{(c,s,h) \to \underline{\text{abort}}}{(\underline{\text{inatom}}\ c,\ s,h) \to \underline{\text{abort}}}\ .$$

Es bleibt, das Verhalten des <u>Paralleloperators</u> zu definieren.
Die Kernidee ist, dass es ein <u>globales</u> <u>Lock</u> gibt,
das sich atomare Blöcke bei ihrem Aufruf nehmen.
Threads können nur in beliebiger Reihenfolge (<u>Interleaving</u>)
Schritte ausführen, solange das Lock frei ist.
Formal ist das Lock ein Prädikat:

<u>Definition:</u>

$$\text{locked}(c) := \begin{cases} \text{true}, & \text{falls } c = \underline{\text{inatom}}\ c' \\ \text{locked}(c_1) \vee \text{locked}(c_2) & \text{, falls } c = c_1 \parallel c_2 \\ \text{locked}(c_1) & \text{, falls } c = c_1 ; c_2 \\ \text{false} & \text{, sonst} \end{cases}$$

(PAREND) $\dfrac{}{(\mathrm{skip} \parallel \mathrm{skip}, s, h) \to (\mathrm{skip}, s, h)}$

(PAR1) $\dfrac{(c_1, s, h) \to (c_1', s', h')}{(c_1 \parallel c_2, s, h) \to (c_1' \parallel c_2, s', h')}$, falls $\neg\, \mathrm{locked}(c_2)$.

(PAR2) $\dfrac{(c_2, s, h) \to (c_2', s', h')}{(c_1 \parallel c_2, s, h) \to (c_1 \parallel c_2', s', h')}$, falls $\neg\, \mathrm{locked}(c_1)$.

(PARABORT1) $\dfrac{(c_1, s, h) \to \underline{\mathrm{abort}}}{(c_1 \parallel c_2, s, h) \to \underline{\mathrm{abort}}}$, falls $\neg\, \mathrm{locked}(c_2)$

(PARABORT2) $\dfrac{(c_2, s, h) \to \underline{\mathrm{abort}}}{(c_1 \parallel c_2, s, h) \to \underline{\mathrm{abort}}}$, falls $\neg\, \mathrm{locked}(c_1)$.

8.2 Programmlogik

Ziel: - Definiere Concurrent-Separation-Logic-Judgments der Form:

$$J \vdash \{A\}\ c\ \{B\}.$$

- Dabei sind J, A, B Separation-Logik-Assertions und c ein ParW-Programm.

  Assertion J heißt Ressourcen-Invariante.

- Die Bedeutung des CSL-Judgments ist Folgende:

  Sofern c von einem Zustand ausgeführt wird, der $A * J$ erfüllt:

  es erreicht das Programm nicht abort,

  es gilt J die gesamte Ausführung über und

  es sollte das Programm terminieren, gilt bei Terminierung $B * J$.

Die formale Definition der Gültigkeit von CSL-Judgments betrachten wir, wenn wir Soundness der Programmlogik zeigen.

Wir hatten gesagt, wir wollten die Korrektheit eines jeden Threads in Isolation zeigen.

Nur die Ressourcen-Invariante sollte der Kommunikation zwischen den Threads dienen.

Wo findet sich das in den CSL-Judgments wieder?

Nicht in der Semantik, aber in den Beweisregeln zum Herleiten der CSL-Judgments.

<u>Ownership</u>:

Die Beweisregeln implementieren folgendes <u>Reasoningprinzip</u>

Bei einem Judgment

$$J \vdash \{A\}\, c\, \{B\}$$

<u>owned</u> das Programm c den von A beschriebenen State.

Das Programm c kann diesen State ändern, kann sich aber darauf verlassen, dass <u>kein paralleler</u> <u>Thread</u> den State ändert.

Der State, der von J beschrieben wird, kann jederzeit von anderen Threads geändert werden.

Allerdings haben wir die Garantie, dass der geänderte Zustand auch wieder J erfüllt.

Zunächst die sogenannten Structural-Rules, die auf den Assertions arbeiten:

$$\frac{A \to A' \qquad J \vdash \{A'\}\, c\, \{B'\} \qquad B' \to B}{J \vdash \{A\}\, c\, \{B\}} \quad \text{(CONSEQUENCE)}$$

$$\frac{J \vdash \{A_1\}\, c\, \{B\} \qquad J \vdash \{A_2\}\, c\, \{B\}}{J \vdash \{A_1 \vee A_2\}\, c\, \{B\}} \quad \text{(DISJ)}$$

$$\frac{J \vdash \{A\}\, c\, \{B\}}{J \vdash \{\exists x.\, A\}\, c\, \{B\}}, \text{ falls } x \notin free(c, B) \quad \text{(EX)}$$

$$\frac{J \vdash \{A\}\, c\, \{B_1\} \qquad J \vdash \{A\}\, c\, \{B_2\}}{J \vdash \{A\}\, c\, \{B_1 \wedge B_2\}}, \text{ falls } J \text{ precise} \quad \text{(CONJ)}$$

Bemerkung:

(DISJ) = Case-Split auf die Vorbedingung

(CONJ) = Kombiniere zwei Beweise

→ nicht sound für beliebige Ressourcen-Invarianten.

Für das sequentielle Fragment sind die Regeln gegenüber Hoare/Separation-Logik unverändert, allerdings führt die Ressourcen-Invariante Nebenbedingungen ein.

(SKIP) $\dfrac{}{J \vdash \{A\}\ skip\ \{A\}}$ $\qquad \dfrac{}{J \vdash \{A[x/a]\}\ x := a\ \{A\}}$, falls $x \notin free(J)$ (ASSIGN)

(ASSUME) $\dfrac{}{J \vdash \{A\}\ \underline{assume}\ b\ \{A \wedge b\}}$ // Dieses Axiom hatten wir vergessen, als wir Hoare-Logik eingeführt haben.

$$\text{(IF)}\quad \frac{J \vdash \{A \wedge b\}\ c_1\ \{B\} \qquad J \vdash \{A \wedge \neg b\}\ c_2\ \{B\}}{J \vdash \{A\}\ \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}\ \{B\}}$$

$$\frac{J \vdash \{A \wedge b\}\ c\ \{A\}}{J \vdash \{A\}\ \text{while } b \text{ do } c \text{ od}\ \{A \wedge \neg b\}}\quad \text{(WHILE)}$$

$$\text{(LMUTV)}\quad \frac{}{J \vdash \{a_1 \mapsto \_\}\ [a_1] := a_2\ \{a_1 \mapsto a_2\}}$$

$$\text{(LDISP)}\quad \frac{}{J \vdash \{a \mapsto \_\}\ \text{dispose } a\ \{\text{emp}\}}$$

$$\text{(LALLOCV)}\quad \frac{}{J \vdash \{\text{emp}\}\ x := \text{cons}(\bar{a})\ \{x \mapsto \bar{a}\}},\ \text{falls } x \notin \text{free}(J, \bar{a})$$

$$\text{(LLOOKV)}\quad \frac{}{J \vdash \{a_1 \mapsto a_2\}\ x := [a_1]\ \{a_1 \mapsto a_2 \wedge x = a_2\}},\ \text{falls } x \notin \text{free}(J, a_1, a_2)$$

Bemerkung:

- Die Versionen von (LALLOCV) und (LLOOKV) sind einfacher als zuvor, weil wir uns auf die non-overwriting Versionen beschränken, bei denen x nicht in den Ausdrücken vorkommen darf.
- Die non-overwriting Einschränkung gilt auch für J. Damit ist x eine Thread-lokale Variable. Wenn x in J enthalten wäre, wüssten wir nicht, ob die Resource-Invariante nach der Allokation / dem Lookup noch gilt.
- Beachte, dass (LMUTV), (LDISP) und (LLOOKV) fordern, dass a_1/a allokiert ist. Wie im sequentiellen Fall garantiert das, dass der Speicherzugriff keinen Fehler liefert.

Allerdings garantiert die Vorbedingung nun noch mehr.
Sie garantiert <u>außerdem</u>, dass kein anderer Thread
die Adresse zugreift.
Folgt aus der Regel für parallele Komposition.

- Die Beweisregel für parallele Komposition erlaubt es,
  zwei Threads zu komponieren, falls
  - sowohl die Heaps disjunkt sind (ausgedrückt durch $*$)
  - als auch die Programmvariablen disjunkt sind (ausgedrückt durch eine Nebenbedingung).

$$\text{(PAR)}\quad \frac{J \vdash \{A_1\}\, c_1\, \{B_1\} \qquad J \vdash \{A_2\}\, c_2\, \{B_2\}}{J \vdash \{A_1 * A_2\}\, c_1 \| c_2\, \{B_1 * B_2\}}, \quad \text{falls } \begin{array}{l} \mathrm{free}(J, A_1, c_1, B_1) \cap \mathrm{modifies}(c_2) = \emptyset \\ \text{und } \mathrm{free}(J, A_2, c_2, B_2) \cap \mathrm{modifies}(c_1) = \emptyset. \end{array}$$

Die Nebenbedingung gilt in der Regel,
weil die Threads auf lokalen Variablen arbeiten.

<u>Bemerkung</u>:

- Die Regel (PAR) garantiert insbesondere,
  dass alle korrekt nachgewiesenen Programme
  (irgendein CSL-Judgment kann gezeigt werden)
  <u>keine</u> <u>Data-Races</u> haben.
- Ein <u>Data-Race</u> ist eine Situation
  in der zwei parallel ausgeführte Befehle
  auf dieselbe Variable/Adresse zugreifen wollen,
  mindestens einer davon schreibend.

$$\underbrace{x := y \parallel z := x}_{DR} \qquad \underbrace{x := y \parallel x := z}_{DR} \qquad \underbrace{y := x \parallel z := x}_{\text{kein } DR.}$$

- Sofern ein Programm Data-Race-frei ist, garantieren Programmiersprache und Prozessorarchitektur, dass sich das Programm wie in der Interleaving-Semantik verhält
  (Sequentielle Konsistenz)

Interleaving / SC $\longrightarrow$ Illusion / Interface / Abstraktion des Memory-Layers.

- Die Regel für atomare Blöcke erlaubt uns
  - ↳ die Invariante zu nutzen, während wir uns in dem Block befinden.
    - $\longrightarrow$ sonst gäbe es tatsächlich keine Möglichkeit der Kommunikation
  - ↳ die Invariante kaputt zu machen, während wir uns in dem Block befinden,
  - ↳ solange wir die Invariante bei Verlassen des Blocks wieder herstellen.

$$(ATOM)\ \frac{\mathrm{emp} \vdash \{A * J\}\ c\ \{B * J\}}{J \vdash \{A\}\ \underline{\mathrm{atomic}}\ c\ \{B\}}$$

Bemerkung:

- Mit Blick auf die Data-Race-Bemerkung gibt es also Zugriffe auf gemeinsame Adressen.
  Diese Zugriffe sind also durch atomare Blöcke getrennt und führen daher nicht zu Data-Races.
- Gemeinsame Variablen werden nur gelesen.

- Mit (SHARE) können wir die Ressourcen-Invariante erweitern, indem wir Teile unseres lokalen States weggeben:

$$\frac{J * R \vdash \{A\}\, c\, \{B\}}{J \vdash \{A * R\}\, c\, \{B * R\}} \qquad \text{(SHARE)}$$

- Mit (FRAME) können wir Teile des lokalen States, den <u>Frame</u>, die wir für den Beweis nicht brauchen, außer Acht lassen.

$$\text{(FRAME)} \quad \frac{J \vdash \{A\}\, c\, \{B\}}{J \vdash \{A * R\}\, c\, \{B * R\}}, \text{ falls } free(R) \cap modifies(c) = \emptyset.$$