

10. Lock-Coupling-List

Ziel: Verifiziere mit R6tep eine fine-grained Concurrent-List, die eine Set-Datenstruktur implementiert.

Zur Implementierung:

Set-Datenstruktur:

Es gibt Operationen add und remove,

die Elemente hinzufügen bzw. entfernen.

Es werden beim Hinzufügen keine Duplikate erzeugt.

Nicht vorhandene Elemente werden halt nicht entfernt.

Fine-grained-Concurrency:

Kein globales Lock auf der Liste sondern

ein Lock pro Knoten (Locks speichern Thread-IDs \rightarrow später).

So ist paralleles Hinzufügen und Entfernen möglich.

Lock-Coupling:

Die Liste wird als geordnet angenommen.

Außerdem gibt es Sentinel-Nodes mit den Werten $-\infty$ und ∞ .

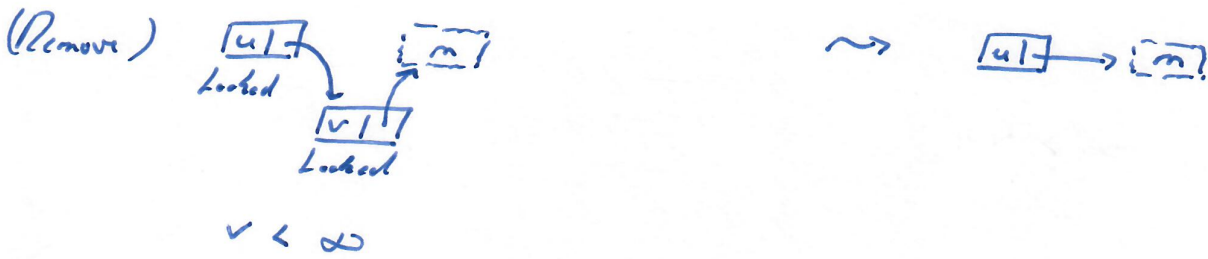
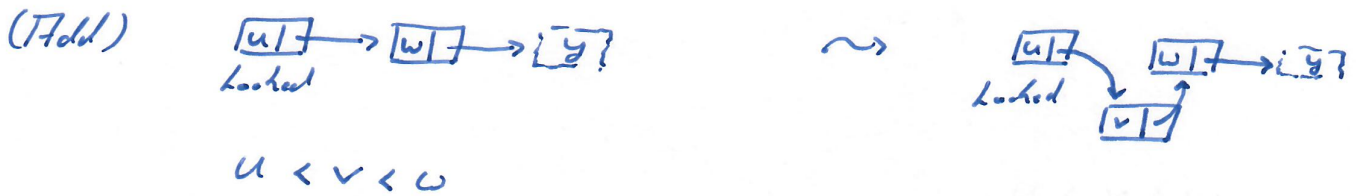
Die Liste wird unter Verwendung von Lock-Coupling durchlaufen:

Ein Knoten wird nicht geunlocked :),

bevor nicht der nächste Knoten geunlocked ist.

Wie beim Seilklettern, man hat immer eine Hand am Seil.

Die Methoden lassen sich graphisch darstellen:



10.1 Präzise

Knoten:

$$N_s(x, v, y) := x \mapsto \underbrace{y}_{\rightarrow \text{next}}, \underbrace{v}_{\rightarrow \text{value}}, \underbrace{s}_{\rightarrow \text{lock}} \quad // \text{Knoten, mit } s \in \{1, 2, 3\} \cup \mathcal{N}$$

$$U(x, v, y) := N_{-2}(x, v, y) \quad // \text{Knoten unlocked}$$

$$L(x, v, y) := N_{\neq}(x, v, y) \wedge E \neq 0 \quad // \text{Knoten locked.}$$

List-Segments:

$$L_s(x, \overline{P}, y) := (x=y \wedge \overline{P} = E \wedge \text{emp})$$

$$\underbrace{\text{Content}}_{= \text{Wert}} \quad \vee \quad \underbrace{(\exists v, z, \beta. x \neq y \wedge \overline{P} = v.\beta * N_-(x, v, z) * L_s(z, \beta, y))}_{\text{Klammerung unbedeutlich, links Stack (hier auf Heap), rechts Heap.}}$$

// \overline{P} zyklisches Listensegment von x zu y mit Content \overline{P} .

// Schreibe E für das leere Wort, $*$ für Konkatination.

Sortedness:

$$\text{sorted}(P) := \exists B. P = -\infty.B.\infty \wedge s(P) \wedge \text{emp}$$

$$s(B) := \begin{cases} \text{true}, & \text{falls } B = \varepsilon \text{ oder } B = a \\ a \leq b \wedge s(b.C) & \text{falls } B = a.b.C \end{cases}$$

10.2 Locks

Wir definieren eine Menge an Locks $A(T)$,

die parametrisiert ist in eine Menge an Thread-IDs, $T \subseteq \mathcal{N}$:

$$A(T) := \{ \text{Lock}(T), \text{Unlock}(T), \text{Add}(T), \text{Rem}(T) \}.$$

Damit ist die Guarantee von Thread $h \in \mathcal{N}$:

$$G := A(\{h\}).$$

Das Rely von Thread $h \in \mathcal{N}$ ist:

$$R := A(\mathcal{N} \setminus \{h\}).$$

Die Verwendung eines Rely-Guarantee-Beweises,

der Bezug nimmt auf die Thread-ID,

ist der eigentliche Grund für die Verwendung von Locks über Mutex .

Der Algorithmus funktioniert auch mit normalen Locks.

Die Locks sind:

$$(\text{Lock}(T)) x.v.n.\ell \in T \wedge U(x.v.n) \quad \rightsquigarrow L_\ell(x.v.n)$$

$$(\text{Unlock}(T)) x.v.n.\ell \in T \wedge L_\ell(x.v.n) \quad \rightsquigarrow U(x.v.n)$$

$$(Add(T)) \quad u, v, w, x, m, n, y. \quad t \in T \wedge u \leq v \leq w \wedge (L_e(x, u, n) \neq N_-(n, w, y)) \\ \leadsto L_e(x, u, m) \neq \cup(m, v, n) \neq N_-(n, w, y)$$

$$(Rem(T)) \quad u, v, x, n, m. \quad t \in T \wedge v \leq \infty \wedge (L_e(x, u, n) \neq L_e(n, v, m)) \\ \leadsto L_e(x, u, m).$$

Warum atomic {...} beim Zugriff auf den Shared-Stack?

Die Regel für primitive Befehle erlaubt keinen Zugriff auf den Shared-Stack:

$$(COM) \quad \frac{\vdash_{SL} \{A\} \text{ com } \{B\} \quad \text{ modifies } (com) \wedge \text{ free } (R, G) = \emptyset}{com : (A, R, G, B)}.$$

Daher fügen wir atomic hinzu, um auf den Shared-Stack zugreifen zu können.

Wie sind die freien Variablen bei Aktionen definiert?

$$\text{free } (\bar{x}. A \rightarrow B) := (\text{free } (A) \cup \text{free } (B)) \setminus \bar{x}.$$