10. Lock-Coupling-List

Ziel: Verifizieren mit Rblep 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 -> später).
So ist parallelles Hinzufügen und Entfernen möglich.

Lock-Coupling:

Die Liste wird als gerichtet 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 gelocked ist.

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

Die Methoden lassen sich graphisch darstellen:

(Lock)

(Unlock)

(Add)

$u < v < w$

(Remove)

$v < \infty$

10.1 Prädikate

Knoten:

$N_s(x, v, y) := x \mapsto \underbrace{y}_{\to next} , \underbrace{v}_{\to value} , \underbrace{s}_{\to lock}$ // Knoten, mit $s \in \{-1\} \cup \mathbb{N}$

$U(x, v, y) := N_{-1}(x, v, y)$ // Knoten unlocked

$L_\varepsilon(x, v, y) := N_\varepsilon(x, v, y) \wedge \varepsilon \geq 0$ // Knoten locked.

List-Segments:

$ls(x, \underbrace{\Pi}_{\substack{\text{Content} \\ = \text{Wort}}}, y) := (x = y \wedge \Pi = \varepsilon \wedge emp)$

$\vee \left(\exists v, z, \beta.\ \underline{x \neq y \wedge \Pi = v.\beta} * N_-(x, v, z) * ls(z, \beta, y)\right)$

Klammerung unschließlich,
links Stack (hier auf Heap), rechts Heap.

// Rekursives Listensegment von x zu y mit Content Π.

// Schreibe ε für das leere Wort, $\cdot$ für Konkatenation.

Sortedness:

$$sort(\Pi) := \exists B.\ \Pi = -\infty.B.\infty \wedge s(\Pi) \wedge emp$$

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

10.2 Relations

Wir definieren eine Menge an Relations $A(T)$,
die parametrisiert ist in eine Menge an Thread-IDs $T \subseteq \mathbb{N}$:

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

Damit ist die Guarantee von Thread $tid \in \mathbb{N}$:

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

Das Rely von Thread $tid \in \mathbb{N}$ ist:

$$R := A(\mathbb{N} \setminus \{tid\}).$$

Die Verwendung eines Rely-Guarantee-Beweises,
die Bezug nimmt auf die Thread-ID,
ist der eigentliche Grund für die Verwendung von Locks über Mut-Fl
Der Algorithmus funktioniert auch mit normalen Locks.
Die Relations sind:

$(Lock(T))\ x.v.n. \in T \wedge U(x,v,n) \rightsquigarrow Lk(x,v,n)$

$(Unlock(T))\ x.v.n. \in T \wedge Lk(x,v,n) \rightsquigarrow U(x,v,n)$

(Add(T)) $u,v,w,x,m,n,y.\ t \in T \wedge u<v<w \wedge (L_t(x,u,n) * N_-(n,w,y))$

$\rightsquigarrow L_t(x,u,m) * \mathcal{U}(m,v,n) * N_-(n,w,y)$

(Rem(T)) $u,v,x,n,m.\ t \in T \wedge v < \infty \wedge (L_t(x,u,n) * L_t(n,v,m))$

$\rightsquigarrow L_t(x,u,m).$

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

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

$$(COM)\quad \frac{\vdash_{SL} \{\Pi\}\ com\ \{\Pi'\} \qquad modifies(com) \wedge free(R,G) = \emptyset}{com : (\Pi, R, G, \Pi')}$$

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

Wie sind die freien Variablen bei Actions definiert?

$$free(\overline{x}.\ \Pi \rightsquigarrow \Pi') := (free(\Pi) \cup free(\Pi')) \setminus \overline{x}.$$