Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2
Prof. Dr. Ir. Joost-Pieter Katoen

Model-checking quantifizierter Linearer Temporaler Logik über Pointerprogrammen

Model Checking Quantified Linear Temporal Logic over Pointer Programs

Tobias Hoffmann, 245 190

Betreuer: Dipl.-Inf. Jonathan Heinen
Erstgutachter: PD Dr. Thomas Noll
Zweitgutachter: Prof. Dr. Ir. Joost-Pieter Katoen
Studiengang: Diplom Informatik
Abgabetermin: 18. April 2012

Einleitung

Computer sind heute aus vielen Bereichen unseres Lebens nicht mehr wegzudenken. Ausfälle und Fehler von Computern und insbesondere der darauf laufenden Software betreffen uns oft unmittelbar. Computerprogramme kontrollieren und steuern heute Logistik, Telekommunikation und viele mehr.

Wer aber schon einmal versucht hat, selbst Programme zu schreiben, weiß, wie schwer es ist, ein in allen Fällen korrektes Programm zu schreiben. Gleichzeitig wächst der Umfang der heute verwendenten Programm immer weiter an. Ein sehr aktives Forschungsgebiet ist daher die automatische Verifikation von Implementationen und Algorithmen auf bestimmte Eigenschaften, etwa dass eine bestimmte fatale Konstellation (z. B. „alle Ampeln einer Kreuzung sind grün“) nie erreicht wird. Diese Eigenschaften müssen dabei formal spezifiziert werden, wofür heute insbesondere temporale Logiken verwendet werden.

Auf der anderen Seite ist basieren viele Programmiersprachen auf dem Konzept untereinander verknüpfter Objekte, deren Beziehungen (Pointer) durch die Programmausführung verändert werden. Wie man die wesentlichen Beziehungen effizient modelliert und unwesentliche Informationen weglassen kann, man also abstrahiert, wird auch im Juggrnaut-Framework erforscht, in dessen Rahmen diese Arbeit entstanden ist.

Mit herkömmlichen temporalen Logiken ist es nun aber nicht möglich sowohl etwas über den Lauf eines Programmes, als auch das Verhalten von bestimmten Objekten während dieses Laufes auszusagen. Gerade das sind aber Aussagen, die man, im Bezug auf Pointerprogramme, d. h. Programme, die mit Pointern und Objekten auf einem Heap umgehen, gerne treffen möchte.

Wir werden daher damit beschäftigt, wie man die bekannte temporale Logik LTL um Objektquantoren erweitern kann, um damit Pointerprogramme zu verifizieren. Insbesondere werden wir hier einen Modelchecking-Algorithmus für die allgemeinste Erweiterung einer Obermenge(!) von LTL, nämlich CTL*, um Objektquantoren, die wir Q*CTL* genannt haben, präsentieren.

Auch wenn wir aus zeitlichen Gründen unsere Logik und das Modelchecking nicht auch noch auf die in Juggrnaut verwendete Abstraktion zu erweitern konnten, so haben wir doch gerade darauf hingearbeitet und werden mit dieser Arbeit einen solide theoretische und algorithmische Basis bieten, die in Zukunft ohne große Mühe auf Abstraktionen hin erweitern werden kann.

Inhaltsverzeichnis

1 Grundlagen

Um nachfolgende Definitionen nicht mit technischen Details zu überfrachten, bedienen wir uns einiger Konventionen:

Definition 1 (Notation)

- Ist die Folge $\sigma = \alpha_1\alpha_2 ... \in A^*$, bestehend aus Elementen aus A, gegeben, dann ist $[\sigma] = \{\alpha_1, \alpha_2, ...\}$ die dazu entsprechende *Menge* der in σ vorkommenden Elemente.
- Außerdem verwenden wir $\sigma[i] = \alpha_i$, um auf das i-te Element zuzugreifen.
- Sei $f : A \longrightarrow B$ eine beliebige in dieser Arbeit verwendete Funktion. Ohne weitere Notiz verwenden wir f auch mit Mengen $X \subseteq A$ und meinen, dass f elementweise angewendet wird: $f(X) = \{f(x) \mid x \in X\}$, sowie mit Folgen $\sigma \in A^*$, so dass $f(\sigma)[i] = f(\sigma[i])$.
- Schließlich lassen wir der Übersichtlichkeit halber Indices weg, wo sie aus dem Kontext klar sind und keine Missverständnisse entstehen.

Definition 2 (Identitätsfunktion)
$\mathsf{id}_X : X \longrightarrow X$ ist die *Identitätsfunktion* auf X, die jedes $x \in X$ auf sich selbst abbildet: $x \mapsto x$.

Definition 3 (Operationen auf Funktionen)
Sei $h : X \longrightarrow Y$ eine Funktion und $A \subseteq X$.

- $h\restriction_A$ steht für die Einschränkung von h auf die Menge A. Somit ist $h\restriction_A : A \longrightarrow Y$ mit $h\restriction_A(a) = h(a)$ für alle $a \in A$.
- $h[a \mapsto y](x) = \begin{cases} y & \text{für } x = a \text{ und} \\ h(x) & \text{sonst.} \end{cases}$
  Hier wird der Wert $h(a)$, wobei $a \in X$ sein muss, in $h' = h[a \mapsto y]$ durch $y \in Y$ ersetzt. Damit bleibt $h' : X \longrightarrow Y$.

1.1 Pointerprogramme

Im Rahmen des in [6] beschriebenen Juggrnaut-Framework, auf das wir aufbauen, wird eine einfache Pointermanipulationssprache eingeführt, die wir auch hier nutzen wollen. In der Praxis arbeitet die Implementierung von Juggrnaut zwar auf Java-Bytecode, die spezifischen Details einer professionellen Programmiersprache sind für die Einführung einer Logik jedoch eher hinderlich und vor allem technischer Natur. Dem geneigten Leser sei dazu [5] empfohlen. Außerdem ist unser Ansatz nicht auf Java beschränkt; viel mehr geht es darum, eine effiziente Verifikation des vielen Programmiersprachen zugrundeliegenden *Konzeptes* der Pointermanipulation (oft auch unter dem Namen „Referenz“ statt „Pointer“) zu ermöglichen und die Forschung in diesem Bereich voranzutreiben.

Definition 4 (Pointerprogramm)
Die Sprache besteht aus *Statements* (Stm), *Blöcken* (Blk), *Bedingungen* (Cond) und *Pointerausdrücken* (Ptr). Zusätzlich müssen *Programmvariablen* (Var_Σ) und *Selektoren* (Sel_Σ) vorgegeben werden. Die Ausdrücke und Selektoren sind dabei an die u. a. aus Java bekannten Konzepte angelehnt. Gegeben ist also folgende kontextfreie Grammatik, wobei $x \in \mathit{Var}_\Sigma$ und $s \in \mathit{Sel}_\Sigma$:

$$\begin{array}{lcll} S & ::= & x = P \mid x.s = P \mid \mathsf{new}(x) \mid \mathsf{while}\ (C)\ S \mid \mathsf{if}\ (C)\ S\ [\mathsf{else}\ S] \mid B & \in \mathit{Stm} \\ B & ::= & \{S(;S)^*\} & \in \mathit{Blk} \\ C & ::= & P == P \mid P\,!= P \mid C\ \&\&\ C \mid C \mid\mid C & \in \mathit{Cond} \\ P & ::= & \mathsf{null} \mid x \mid x.s & \in \mathit{Ptr} \end{array}$$

Ein Pointerprogramm ist dann gerade $\mathit{pgm} \in \mathit{Stm}$.

1.2 Heapkonfigurationen

Um die Pointerrelationen eines Programmes auf dem Heap mathematisch zu modellieren dienen in Juggrnaut (gelabelte) Hypergraphen. Diese enthalten, statt der bei üblichen Graphen verwendeten Kanten, (gelabelte) Hyperkanten, die eine beliebige Anzahl Knoten verbinden können.

Definition 5 (geranktes Alphabet)
Sei Σ ein endliches, geranktes Alphabet, wobei $\mathit{rk} : \Sigma \longrightarrow \mathbb{N}$ jedem Symbol $a \in \Sigma$ seinen Rang $\mathit{rk}(a)$ zuweist. O. B. d. A. ist $\Sigma \cap \{\mathsf{null}, \bot\} = \emptyset$.

Definition 6 (Hypergraph über einem Alphabet)
Ein *(gelabelter) Hypergraph über* Σ ist ein Tupel $H = (V, E, \mathit{att}, \mathit{lab}, \mathit{ext})$, wobei V eine endlichen Menge von Knoten und E eine endliche Menge *Hyperkanten* ist, so dass

- $\mathit{att} : E \longrightarrow V^*$ jeder Hyperkante eine Folge von verbundenen Knoten zuweist und
- $\mathit{lab} : E \longrightarrow \Sigma$ jeder Hyperkante ein Label. Außerdem ist
- $\mathit{ext} \in V^*$ eine (möglicherweise leere) Folge von paarweise disjunkten *externen Knoten*.

Wir verlangen nun noch, dass $|\mathit{att}(e)| = \mathit{rk}(\mathit{lab}(e))$ und schreiben kurz $\mathit{rk}(e) = \mathit{rk}(\mathit{lab}(e))$. Das heißt, dass jede Hyperkante mit genau $\mathit{rk}(e)$ Knoten verbunden ist.
Außerdem bezeichnen wir die Menge aller Hypergraphen über Σ mit HG_Σ.

Unterscheiden sich Hypergraphen nur durch Umbenennung von Knoten und Hyperkanten von einander, dann nennen wir sie isomorph:

Definition 7 (Isomorphie von Hypergraphen)
Zwei Hypergraphen $H = (V_H, E_H, \mathit{att}_H, \mathit{lab}_H, \mathit{ext}_H)$ und $H' = (V_{H'}, E_{H'}, \mathit{att}_{H'}, \mathit{lab}_{H'}, \mathit{ext}_{H'})$, $H, H' \in \mathsf{HG}_\Sigma$ sind *isomorph*, $H \cong H'$, wenn es zwei Bijektionen gibt, nämlich:

- Eine Knotenumbenennung $\mathit{iso}_V : V_H \longrightarrow V_{H'}$ und
- eine Kantenumbenennung $\mathit{iso}_E : E_H \longrightarrow E_{H'}$, so dass
- $\mathit{att}_{H'} \circ \mathit{iso}_E = \mathit{iso}_V \circ \mathit{att}_H$,

- $lab_{H'} \circ iso_E = lab_H$ und
- $ext_{H'} = iso_V(ext_H)$,

wobei $(g \circ f)(x) = g(f(x))$ die Funktionsverkettungsoperation bezeichnet.

Lemma 8 (Isomorphie ist eine Äquivalenzrelation)
Die so definierte Isomorphierelation auf Hypergraphen ist eine *Äquivalenzrelation*, das heißt, sie ist *reflexiv*, *symmetrisch* und *transitiv*.

Beweis: (Skizze) Seien $H, H', H'' \in \mathsf{HG}_\Sigma$.

Reflexiv: Zu zeigen: $H \cong H$.
Das ist offenbar erfüllt mit $iso_V = \mathsf{id}_{V_H}$, $iso_E = \mathsf{id}_{E_H}$.

Symmetrisch: Zu zeigen: Es gilt $H \cong H'$ gdw. $H' \cong H$.
Da iso_V und iso_E Bijektionen sind, folgt dies mit iso_V^{-1} und iso_E^{-1}.

Transitiv: Zu zeigen: Aus $H \cong H'$ (1) und $H' \cong H''$ (2) folgt $H \cong H''$.
Seien also $iso_V^{(1)}$, $iso_V^{(2)}$, $iso_E^{(1)}$ und $iso_E^{(2)}$ gegeben. Bilde $iso_V = iso_V^{(2)} \circ iso_V^{(1)}$ und $iso_E = iso_E^{(2)} \circ iso_E^{(1)}$. Diese sind immer noch bijektiv und erfüllen die für geforderten H und H'' Identitäten. □

Für unsere Zwecke besteht zunächst das Alphabet aus den Variablen (Var_Σ) und Selektoren (Sel_Σ).

Definition 9 (Heapkonfiguration)
Eine *(konkrete) Heapkonfiguration* ist ein Hypergraph $H = (V, E, att, lab, ext) \in \mathsf{HG}_\Sigma$ mit

- $\Sigma = Var_\Sigma \dot\cup Sel_\Sigma$,
- $\forall x \in Var_\Sigma : rk(x) = 1$,
- $\forall s \in Sel_\Sigma : rk(s) = 2$,
- $\forall x \in Var_\Sigma : |\{e \in E \mid lab(e) = x\}| \leq 1$ (Variable höchstens einmal vorhanden),
- $\forall v \in V, s \in Sel_\Sigma : |\{e \in E \mid lab(e) = s, \exists w \in V : att(e) = vw\}| \leq 1$
(pro Knoten wird jeder Selektor höchstens einmal verwendet), und
- $ext = \varepsilon$, wobei ε die leere Folge bezeichnet.

Da für alle $e \in E$ mindestens $rk(e) = 1$ ist, gibt es keine Hyperkanten, die nicht mit einem Knoten verbunden sind. In dieser Arbeit ignorieren (entfernen) wir außerdem alle Knoten, die nicht von einer der Variablen aus über Hyperkanten erreichbar sind („Garbage“). Genauer gesagt: Knoten, die nicht erreichbar sind, wenn Selektoren in „Vorwärtsrichtung“, also von $att(e)[1]$ nach $att(e)[2]$ für $e \in E$ mit $lab(e) \in Sel_\Sigma$, gefolgt wird.
Mit HC_Σ bezeichnen wir dann die Menge aller (konkreten) Heapkonfigurationen.

Um später die Notation zu vereinfachen, definieren wir noch folgende Funktion, die uns zu einem Label die – wegen der Regeln für Heapkonfigurationen – eindeutig bestimmte Hyperkante liefert:

Definition 10 (Hyperkante zu gegebenem Label)
Für $H = (V, E, att, lab, \varepsilon) \in \mathsf{HC}_\Sigma$, $x \in Var_\Sigma$, $v \in V$ und $s \in Sel_\Sigma$ ist:

$$edge_H(x) = \begin{cases} e & \text{für das (eindeutige!) } e \in E \text{ mit } lab(e) = x \\ \mathsf{null} & \text{sonst} \end{cases}$$

$$edge_H(v, s) = \begin{cases} e & \text{für das } e \in E \text{ mit } lab(e) = s \wedge att(e)[1] = v \\ \mathsf{null} & \text{sonst} \end{cases}$$

$$edge_H(\mathsf{null}, s) = \mathsf{null}$$

1.3 Objekte, Zustände, Pfade und Zustandsraum

Während Hypergraphen die zu einem bestimmten Zeitpunkt bestehenden Pointerrelationen eines Programmes beschreiben, besteht ein Programlauf aus vielen verschiedenen solcher Pointerrelationen, die sich über die Semantik der Programmstatements auseinander ergeben. Das Programm befindet sich also zu jedem Zeitpunkt in einem bestimmten Zustand, der durch die bestehende Pointerrelation und dem noch zu verarbeitendem Restprogramm beschrieben werden kann.

Wir verwenden zunächst diese konkrete Wahl der Repräsentation durch Hypergraphen, um einige Begriffe einzuführen:

Definition 11 (Zustand)
Ein *(Programm-)Zustand* besteht bei uns aus dem Tupel $q = (H, pgm)$, wobei $H \in \mathsf{HC}_\Sigma$ eine Heapkonfiguration ist und $pgm \in Stm \dot\cup \{\varepsilon\}$ das noch zu verarbeitende Restprogramm, wobei ε für das leere, d. h. abgearbeitete, Programm steht.
Die Menge aller möglichen Programmzustände ist dann $Q = \mathsf{HC}_\Sigma \times (Stm \dot\cup \{\varepsilon\})$.
Im Allgemeinen ist Q beliebig wählbar; Zustände werden dann gerade die Elemente $q \in Q$ genannt.

Definition 12 (Objekt)
Für $q = (H, pgm) \in Q$ mit $H = (V_H, E_H, att_H, lab_H, ext_H)$ nennen wir $v \in V_H$ *Objekt*.
Wir schreiben V_q $(= V_H)$ um die Menge aller im Zustand q enthaltenen (allgemein: mit q assoziierten) Objekte zu referenzieren.

Ziel der Arbeit ist es, Aussagen über die Entwicklung von Objekten, bzw. deren Relationen, über mehrere Zustände hinweg zu treffen. Dazu ist es notwendig, dass wir ein Objekt $v \in V_q$ aus dem Zustand q auch in einem „späteren“ Zustand q', genauer gesagt in $V_{q'}$, wieder finden – wenn es dort noch existiert.

Ein „späterer“ Zustand drückt dabei aus, dass q' durch die Programmausführung in einem oder mehreren Schritten aus q hervorgegangen ist. Wir schreiben $q \triangleright q'$, wenn $q' \in Q$ direkt, also durch Abarbeiten eines einzelnen Programmschrittes, aus $q \in Q$ hervorgeht. Bevor wir aber die sogenannte Zustandsübergangsrelation $\triangleright$ formal aus der Semantik der Programmstatements ableiten können, bedarf es noch einiger weitere Definitionen.

Zunächst wollen wir das Verfolgen von Objekten über einen einzelnen Zustandsübergang präzisieren. Dazu:

Definition 13 (Korrespondenzfunktion)
Sei Q ein beliebige Menge von Zuständen und V_q die Menge der mit $q \in Q$ assoziierten Objekte. Weiter betrachten wir die Relation $\triangleright$ als Teilmenge von $Q \times Q$.

- Für jeden Zustandsübergang $q \triangleright q'$, $q, q' \in Q$ nennen wir eine partielle injektive Funktion $kf_{q \triangleright q'} : V_q \longrightarrow V_{q'}$ *(Objekt-)Korrespondenzfunktion für* (q, q'). Sie bildet ein Objekt $v \in V_q$ im Zustand q auf das entsprechende Objekt $v' \in V_{q'}$ im Folgezustand q' ab.
- Besteht für $v \in V_q$ keine Objektkorrespondenz, so sei $kf_{q \triangleright q'}(v) = \mathsf{null}$.
- Die Menge aller Korrespondenzfunktionen über Q sei mit KF_Q bezeichnet;
- damit können wir formal eine Funktion kf definieren, die jedem Zustandsübergang, also jedem Element der Teilmenge $\triangleright$, eine Korrespondenzfunktion zuordnet: $kf : \triangleright \longrightarrow KF_Q$.
- Wir verwenden dann $kf_{q \triangleright q'} = kf(q, q')$ als Notation bei gegebenem kf, q und q'.

Wir können nun die im speziellen eingeführten Begriffe ganz allgemein definieren:

Definition 14 (Objektorientierter Zustandsraum)
Ein *objektorientierter Zustandsraum* $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$ besteht aus

- einer (opaken) Menge Q von *Zuständen*,
- den Startzuständen $Q_0 \subseteq Q$,
- der *Zustandsübergangsrelation* $\triangleright \subseteq Q \times Q$, die für jeden Zustand einen oder mehrere mögliche Nachfolgezustände angibt,
- einer Funktion obj, die für jeden Zustand $q \in Q$ die Menge der mit q assoziierten *Objekte* $V_q = obj(q)$ zurückgibt.
- Außerdem eine Abbildung $kf : \triangleright \longrightarrow KF_Q$ der Zustandsübergänge auf Korrespondenzfunktionen und damit $kf_{q \triangleright q'} : V_q \longrightarrow V_{q'}$,
- die Abbildung $var : Q \longrightarrow VAR_Q$ (vgl. kf und KF_Q), die jedem Zustand eine Auswertefunktion $var_q : Var_\Sigma \longrightarrow V_q \dot\cup \{\mathsf{null}\}$ zuordnet, wobei VAR_Q die Menge aller möglichen solchen Auswertefunktionen ist.
  $var_q(x)$ wertet im Zustand q die Programmvariable x aus, indem sie das zugehörige Objekt $v = var_q(x) \in V_q$, oder aber null (wenn x auf kein Objekt zeigt), zurückgibt.
- die Abbildung $sel : Q \longrightarrow SEL_Q$, die, analog zu $var_q \in VAR_Q$, für jedes q eine Auswertefunktion $sel_q : V_q \times Sel_\Sigma \longrightarrow V_q \dot\cup \{\mathsf{null}\}$ zur Auswertung von Selektoren bestimmt.

Wir fordern außerdem, dass $\triangleright$ *total* ist, d. h. für jedes $q \in Q$ existiert ein $q' \in Q$ mit $q \triangleright q'$. Das lässt sich durch geeignetes Hinzufügen von Zyklen, $q \triangleright q$, erreichen.

Ein Programmlauf lässt sich dann als Pfad im durch Q und $\triangleright$ gegebenen Graph beschreiben:

Definition 15 (Berechnungspfad)
Sei $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$.

- Ein *Pfad* in K ist eine maximale[†] Folge $q_1 q_2 \ldots \in Q^*$ von Zuständen mit $q_i \triangleright q_{i+1}$ für alle $i \geq 1$.

[†]Maximal drückt aus, dass (etwa) der Pfad nicht erweitert werden kann.

- $Paths_K$ sei die Menge aller Pfade in K; weiter sind
- $Paths_K(q) = \{\pi \in Paths_K \mid \pi[1] = q\}$ alle mit q beginnenden Pfade.

Für einen Pfad π und $i \geq 1$ bezeichne:

- $\pi[i\,...] = q_i q_{i+1} ... \in Paths_K$.

Außerdem weiten wir $kf_{q \triangleright q'}$ auf Pfade $\pi \in Paths_K$ aus, wobei $1 \leq i < j$:

- $kf_{\pi[i...j]} = kf_{\pi[j-1] \triangleright \pi[j]} \circ ... \circ kf_{\pi[1] \triangleright \pi[2]}$, womit $kf_{\pi[i...j]} : V_{\pi[i]} \longrightarrow V_{\pi[j]}$.

Bemerkung: Da $\triangleright$ total ist, ist jeder Pfad in K unendlich lang.

Für eine vollständige Spezifikation eines Zustandsraums über Hypergraphen müssen wir $\triangleright$ angeben können. Dafür widmen wir uns nun der Semantik unserer Pointerprogramme.

1.4 Semantik von Pointerprogrammen

Ausgehend von einer konkreten Heapkonfiguration H und einem Programm $pgm \in Stm$ beginnen wir damit, die Semantik eines Pointerausdrucks anzugeben, auf der die Semantik von Bedingungen aufbaut. Damit können wir dann letztlich auch die Semantik von Statements formalisieren, wodurch $\triangleright$ festgelegt wird.

Definition 16 (Semantik von Pointerausdrücken und Variablen)
Gegeben sei die konkrete Heapkonfiguration $H = (V, E, att, lab, \varepsilon) \in \mathsf{HC}_\Sigma$.
Die Interpretation $[\![\cdot]\!]_H : Ptr \longrightarrow V \cup \{\mathsf{null}, \bot\}$ ist:

$$[\![\mathsf{null}]\!]_H = \mathsf{null}$$

$$[\![x]\!]_H = \begin{cases} att(e)[1] & \text{wenn } e = edge(x) \neq \mathsf{null} \\ \mathsf{null} & \text{sonst} \end{cases}$$

$$[\![x.s]\!]_H = \begin{cases} att(e)[2] & \text{wenn } e = edge([\![x]\!]_H, s) \neq \mathsf{null} \\ \bot & \text{wenn } [\![x]\!]_H = \mathsf{null} \\ \mathsf{null} & \text{sonst} \end{cases}$$

Definition 17 (Semantik von Bedingungen)
Die Interpretation $[\![\cdot]\!]_H : Cond \longrightarrow \{\mathbf{tt}, \mathbf{ff}, \bot\}$, also die Bewertung mit wahr, falsch oder unzulässig, wobei $\bot$ durch Dereferenzieren eines null-Pointers verursacht wird, ist (für $P, Q \in Ptr$, $C, D \in Cond$):

$$[\![P == Q]\!]_H = \begin{cases} \bot & \text{für } [\![P]\!]_H = \bot \vee [\![Q]\!]_H = \bot \\ \mathbf{tt} & \text{für } [\![P]\!]_H = [\![Q]\!]_H \neq \bot \\ \mathbf{ff} & \text{sonst} \end{cases}$$

$$[\![C \;\&\&\; D]\!]_H = \begin{cases} \bot & \text{für } [\![C]\!]_H = \bot \vee [\![D]\!]_H = \bot \\ \mathbf{tt} & \text{für } [\![C]\!]_H = \mathbf{tt} \wedge [\![D]\!]_H = \mathbf{tt} \\ \mathbf{ff} & \text{sonst} \end{cases}$$

Analog definieren wir $[\![P\,!= Q]\!]_H$ und $[\![C \,||\, D]\!]_H$, wobei $\bot$ in einem Teilausdruck immer zu $\bot$ für den Gesamtausdruck führt.

Eine Programmausführung besteht nun darin, dass durch Statements eine Heapkonfiguration in eine neue Heapkonfiguration überführt wird. Dafür benötigen wir zunächst Operationen zur Modifikation von Heapkonfigurationen:

Definition 18 (Heapupdates)
Gegeben sei ein Heap, $(V, E, att, lab, \varepsilon) \in \mathsf{HC}_\Sigma$. Für eine Menge A verwenden wir $A \dot\cup \{f\}$ um auszudrücken, dass f ein neues Element werden soll – was möglich ist, weil es immer so gewählt werden kann, dass es nicht schon in A vorhanden ist.
Sei $e \in E$, $\sigma \in \Sigma$, $\nu \in V^*$, sowie $f \notin E$ eine neue Kante:

$$\begin{aligned} H[e \hookrightarrow \mathsf{null}] &= (V, E \setminus \{e\},\ att\restriction_{E\setminus\{e\}},\ lab\restriction_{E\setminus\{e\}}, \varepsilon) \\ H[e \hookrightarrow \nu] &= (V, E, att[e \mapsto \nu], lab, \varepsilon) \\ H[f \overset{\sigma}{\hookrightarrow} \nu] &= (V, E \dot\cup \{f\}, att[f \mapsto \nu], lab[f \mapsto \sigma], \varepsilon) \end{aligned}$$

Hier wird im Hypergraph also eine Kante entfernt ($H[e \hookrightarrow \mathsf{null}]$), geändert ($H[e \hookrightarrow \nu]$) oder mit Label σ hinzugefügt ($H[f \overset{\sigma}{\hookrightarrow} \nu]$).
Sei nun $x \in Var_\Sigma$, $s \in Sel_\Sigma$ und $w \in V$:

$$H[x \hookrightarrow \mathsf{null}] = \begin{cases} H[e \hookrightarrow \mathsf{null}] & \text{wenn } e = edge(x) \neq \mathsf{null} \\ H & \text{sonst} \end{cases}$$

$$H[x \overset{s}{\hookrightarrow} \mathsf{null}] = \begin{cases} H[e \hookrightarrow \mathsf{null}] & \text{wenn } e = edge([\![x]\!]_H, s) \neq \mathsf{null} \\ H & \text{sonst} \end{cases}$$

$$H[x \hookrightarrow w] = \begin{cases} H[e \hookrightarrow w] & \text{wenn } e = edge(x) \neq \mathsf{null} \\ H[f \overset{x}{\hookrightarrow} w] & \text{sonst} \end{cases}$$

$$H[x \overset{s}{\hookrightarrow} w] = \begin{cases} H[e \hookrightarrow vw] & \text{wenn mit } v = [\![x]\!]_H : e = edge(v, s) \neq \mathsf{null} \\ H[f \overset{s}{\hookrightarrow} vw] & \text{wenn } v = [\![x]\!]_H \neq \mathsf{null} \\ H & \text{sonst (ohne praktische Bedeutung)} \end{cases}$$

$$H[x \hookrightarrow +] = H'[x \hookrightarrow v] \quad \text{wobei } H' = (V \dot\cup \{v\}, E, att, lab, \varepsilon)$$

Hier sind bereits die Grundzüge der in unserer Sprache erlaubten Pointermanipulationen wiederzuerkennen.

Abschließend geben wir noch die inzwischen geradezu naheliegenden, formalen Inferenzregeln an, die $q \triangleright q'$ erzeugen:

Definition 19 (Semantik von Statements)
Sei $Q = \mathsf{HC}_\Sigma \times (Stm \dot\cup \{\varepsilon\})$ und $q = H, \langle pgm \rangle \in Q$.
Für jedem Ausdruck in S aus Definition 4 geben wir nun also an, wie ein Zustand q modifiziert werden muss, um zu $q' \in Q$ zu kommen. Wir schreiben $\dfrac{Bedingung}{q \triangleright q'}$, um auszudrücken, dass, wenn die oben angegebene Bedingung erfüllt ist, ein Übergang von q nach q' möglich ist, also $q \triangleright q'$.

$$\frac{[\![P]\!]_H \neq \bot}{H, \langle x = P\rangle \ \triangleright\ H[x \hookrightarrow [\![P]\!]_H], \langle\varepsilon\rangle} \qquad \frac{[\![x]\!]_H \in V \quad \wedge \quad [\![P]\!]_H \neq \bot}{H, \langle x.s = P\rangle \ \triangleright\ H[x \overset{s}{\hookrightarrow} [\![P]\!]_H], \langle\varepsilon\rangle}$$

$$\frac{[\![C]\!]_H = \mathbf{tt}}{H, \langle \mathsf{while}\ (C)\ S\rangle \ \triangleright\ H, \langle\{S; \mathsf{while}\ (C)\ S\}\rangle} \qquad \frac{[\![C]\!]_H = \mathbf{ff}}{H, \langle \mathsf{while}\ (C)\ S\rangle \ \triangleright\ H, \langle\varepsilon\rangle}$$

$$\frac{}{H, \langle \mathsf{new}(x)\rangle \ \triangleright\ H[x \hookrightarrow +], \langle\varepsilon\rangle} \qquad \frac{[\![C]\!]_H = \mathbf{tt}}{H, \langle \mathsf{if}\ (C)\ S_1\ [\mathsf{else}\ S_2]\rangle \ \triangleright\ H, \langle S_1\rangle}$$

$$\frac{[\![C]\!]_H = \mathbf{ff}}{H, \langle \mathsf{if}\ (C)\ S_1\ \mathsf{else}\ S_2\rangle \ \triangleright\ H, \langle S_2\rangle} \qquad \frac{[\![C]\!]_H = \mathbf{ff}}{H, \langle \mathsf{if}\ (C)\ S_1\rangle \ \triangleright\ H, \langle\varepsilon\rangle}$$

Sowie:

$$\frac{H, \langle S_1\rangle \ \triangleright\ H', \langle S_1'\rangle \quad \wedge \quad S_1' \neq \varepsilon}{H, \langle\{S_1, S_2\}\rangle \ \triangleright\ H', \langle\{S_1', S_2\}\rangle} \qquad \frac{H, \langle S_1\rangle \ \triangleright\ H', \langle\varepsilon\rangle}{H, \langle\{S_1, S_2\}\rangle \ \triangleright\ H', \langle S_2\rangle}$$

Man beachte, dass keine Regel $\bot$ verarbeiten kann. Außerdem ist unsere Pointersprache rein deterministisch. Daher gibt es bis jetzt nur einen einzigen Berechnungspfad (pro Startzustand). Wie schon in Definition 14 beschrieben, ergänzen wir $\triangleright$ im Falle von ε und $\bot$ durch entsprechende Zyklen $q \triangleright q$ zu einer totalen Relation.

Beispiel 1 *(Heapkonfiguration)*
Wir geben nun in Abbildung 1.1 ein Beispiel, wie eine Heapkonfiguration aussieht. Hyperkanten werden als Rechtecke dargestellt, die Verbindungen zu den über diese Kante miteinander verknüpften Knoten werden anhand der Position ihres Vorkommens in att *durchnummeriert. Selektoren stellen wir aber der Übersichtlichkeit halber als beschriftete gerichtete Kanten dar. Sie zeigen von* 1 *nach* 2*.*

Dargestellt ist ein Baum, dessen innere Knoten aus Objekten bestehen, die zwei Pointer, left *und* right*, halten. Außerdem gibt es das Objekt* v_1*, das den ganzen Baum repräsentieren soll, und dazu* root*,* first *und* last *verwendet. Weiter ist* tree *eine Programmvariable, die auf dieses Objekt zeigt.*

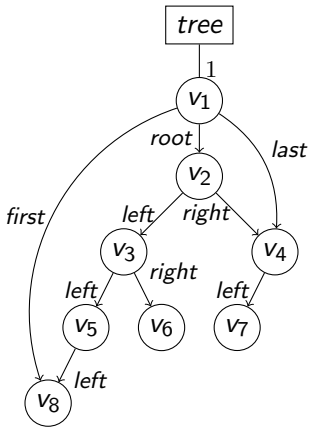


Abbildung 1.1: Heapkonfiguration aus Beispiel 1.

1.5 Zustandsraum über Hypergraphen

Unserem Ziel, Q und V_q durch Hypergraphen zu modellieren, haben wir nun erreicht: Wir können einen entsprechenden objektorientierter Zustandsraum vollständig angeben:

Definition 20 (Zustandsraum über Hypergraphen)
Für ein Programm $pgm \in Stm$ und Startheapkonfigurationen $HC_0 \subseteq \mathsf{HC}_\Sigma$ über dem Alphabet Σ ist der Zustandsraum $K^1_{pgm,HC_0} = (Q, Q_0, \triangleright, obj, kf, var, sel)$ gegeben durch:

- $Q = \mathsf{HC}_\Sigma \times (Stm \dot\cup \{\varepsilon\})$, wobei $q = (H_q, pgm_q) \in Q$, $H_q = (V_{H_q}, E_{H_q}, att_{H_q}, lab_{H_q}, ext_{H_q})$,
- $Q_0 = \{(H, pgm) \mid H \in HC_0\}$,
- $q \triangleright q'$ wie in Definition 19 (Semantik von Statements) angegeben,
- $kf(q, q') = \mathsf{id}_{V_{H_{q'}}}\big|_{V_{H_q}}$ für alle $q \triangleright q'$, da unsere Semantik von $(H_q, pgm) \triangleright (H_{q'}, pgm')$ die Objekte erhält: $V_{H_q} \subseteq V_{H_{q'}}$, [†]
- $obj(q) = V_{H_q}$,
- $var(q)(x) = \begin{cases} att_{H_q}(e)[1] & \text{wenn } e = edge_{H_q}(x) \neq \mathsf{null} \\ \mathsf{null} & \text{sonst} \end{cases}$
  für jedes $q \in Q$, $x \in Var_\Sigma$ und
- $sel(q)(v, s) = \begin{cases} att_{H_q}(e)[2] & \text{wenn } e = edge_{H_q}(v, s) \neq \mathsf{null} \\ \mathsf{null} & \text{sonst} \end{cases}$
  für jedes $q \in Q$, $v \in V_{H_q}$ und $s \in Sel_\Sigma$.

1.6 Isomorphie von Heapkonfigurationen

Um nicht zwischen isomorphen Heapkonfigurationen zu unterscheiden, da sie sich gleich verhalten, fassen wir isomorphe Heapkonfigurationen in einer sogenannten Äquivalenzklasse zusammen. Dennoch müssen wir weiterhin einzelne Objekte auch über mehrere Zustände hinweg identifizieren können. Dafür haben wir bereits die Objektkorrespondenzfunktionen eingeführt, für die wir bisher die Identitätsfunktion verwendet haben. Jetzt müssen wir darin aber die konkrete Knotenumbenennung zwischen zwei isomorphen Graphen berücksichtigen. Insbesondere wäre dazu eine eindeutige Normalform, ein einziger Repräsentant, für solche isomorphe Heapkonfiguration, also für eine Äquivalenzklasse, hilfreich, den wir als Referenz für die Umbenennungen verwenden können.

Mathematisch fassen wir also isomorphe Heapkonfigurationen in einer Äquivalenzklasse zusammen:

Definition 21 (Äquivalenzklasse, Faktormenge)
Es ist $[a]_{\cong,M} = \{x \in M \mid x \cong a\}$ die *Äquivalenzklasse*, die a enthält.
Die *Faktormenge* $M/_\cong = \{[a]_{\cong,M} \mid a \in M\}$ entsteht dabei durch die Partitionierung von M in Äquivalenzklassen bezüglich $\cong$. Diese ist bekanntlich wohldefiniert, wenn $\cong$ eine Äquivalenzrelation ist.

[†]Da wir nicht mehr erreichbare Knoten entfernen, kann es sein, dass $v \in V_{H_q}$ in $V_{H_{q'}}$ nicht mehr vorkommt. Dann bleibt die partielle Funktion kf an einer solchen Stelle $kf(q, q')(v)$ undefiniert.

Damit betrachten wir statt HC_Σ nun $\overline{\mathsf{HC}_\Sigma} = \mathsf{HC}_\Sigma/_\cong$. Dass wir für jede Äquivalenzklasse einen kanonischen Repräsentanten finden können, ermöglichen die folgenden Lemmata:

Lemma 22 (Aufzählung der Knoten einer Heapkonfiguration)
Für $H = (V, E, att, lab, ext)$ lässt sich eine von Knoten- und Kantenbennenung unabhängige Aufzählung der Knoten V erstellen, die wir mit $\texttt{enum}$ bezeichnen: $\texttt{enum}(H) = v_1 \dots v_{|V|} \in V^*$ mit $v_i \neq v_j$ für $i \neq j$.

Vorgehen: Wir können die Variablen und Selektoren unseres Alphabetes (total) ordnen.[†] Ausgehend von den Variablen wird in Juggrnaut nun durch Tiefensuche jeder (erreichbare!) Knoten in einer festen, von der usprünglichen Knotenbennenung unabhängigen, Reihenfolge aufgezählt, indem von einem Knoten aus allen Selektoren nach ihrer Ordnung gefolgt wird. Da durch Variablen und Selektoren alle Kanten abgedeckt sind, spielt die Kantenbenennung keine Rolle. □

Korollar 23
Sei $H = (V_H, E_H, att_H, lab_H, ext_H) \in \mathsf{HC}_\Sigma$ und $v_1 \dots v_{|V_H|} = \texttt{enum}(H)$ eine Aufzählung der Knoten aus V_H.
Dann induziert $\texttt{enum}$ einen zu H isomorphen Graphen H_{enum}, d. h. $H_{enum} \cong H$, wobei als Isomorphieabbildungen $iso_E = \mathsf{id}_{E_H}$ und $iso_V(i) = \texttt{enum}[i] = v_i$ für alle $1 \leq i \leq |V_H|$ verwendet werden. Es gilt: $H_{enum} = (\{1, 2, \dots, |V_H|\}, E_H, iso_V^{-1} \circ att_H, lab_H, iso_V^{-1}(ext_H))$.

Lemma 24 (Ordnung der Kanten)
Sei $H \in \mathsf{HC}_\Sigma$. Auf den Kanten E_H von H gibt es eine eindeutige, totale Ordnung und damit eine Aufzählung der Kanten, die wiederum eine Bijektion $E_H \longrightarrow \{1, 2, \dots, |E_H|\}$ induziert.

Beweis: Da auf den Knoten bereits eine totale Ordnung vermöge der Aufzählung $\texttt{enum}(H)$ besteht, vereinfachen wir zunächst das Problem, indem wir zu $H_{enum} = (V, E, att, lab, ext)$, also dem durch $\texttt{enum}(H)$ induzierten (und zu H isomorphen) Graphen übergehen.
Wie schon in Definition 9 (Heapkonfiguration) angemerkt, sind alle Kanten von mindestens einem Knoten aus erreichbar. Daher ist $att : E \longrightarrow \{1, \dots, |V|\}^+$, d. h. $att(e) = \varepsilon$ tritt nicht auf.
Dann sei $e \leq_E e'$ für $e, e' \in E$, wenn $att(e) \leq att(e')$ bezüglich der lexikalischen Ordnung [‡] auf $att(E)$. Setze außerdem $e \leq_E e'$, wenn $att(e) = att(e')$, und $lab(e) \leq lab(e')$ bezüglich der bereits für $\texttt{enum}$ verwendeten Ordnung auf dem Alphabet.
Wegen der für Heapkonfigurationen geforderten Eigenschaften ist es für $e \neq e'$ nicht möglich, dass sowohl $att(e) = att(e')$, als auch $lab(e) = lab(e')$ gleichzeitig gelten. □

Hierauf aufbauend ergibt dann $\texttt{rep}(H)$ eine zu H isomorphe Normalform:

Definition 25 (Normalform für Heapkonfigurationen)
Sei $H \in \mathsf{HC}_\Sigma$.
$\texttt{rep}(H)$ ist dann der zu H isomorphe Graph bezüglich der schon für H_{enum} verwendeten Knotenbijektion iso_V, sowie der Kantenbijektion iso_E, die sich aus der totalen Ordnung auf den Kanten von H ergibt.

[†] Für $\alpha, \beta \in \Sigma$ ist entweder $\alpha \leq \beta$ oder $\beta \leq \alpha$.

[‡] Ordne $att(e) = v_1 v_2 v_3 \dots$ und $att(e') = v_1' v_2' v_3' \dots$ anhand der ersten Stelle, an der sie sich unterscheiden, also an der nicht sowohl $v_i \leq v_i'$, als auch $v_i' \leq v_i$ ist.

Obwohl also Graphisomorphie *im Allgemeinen* nicht einfach festzustellen ist, ermöglicht uns die Normalform eben dies:

Lemma 26 (Isomorphie von Heapkonfigurationen mittels `rep`)
Sei $H, H' \in \mathsf{HC}_\Sigma$. Es ist $H \cong H'$ gdw. $\mathtt{rep}(H) = \mathtt{rep}(H')$.

Beweis: Wir verwenden $R = \mathtt{rep}(H)$ und $R' = \mathtt{rep}(H')$, sowie $V_H, V_R, V_{H'}, V_{R'}, E_H, \ldots$ für die entsprechenden Knotenmengen, Kantenmengen, etc.
Wenn $R = R'$, dann folgt wegen $H \cong R$ und $H' \cong R'$ mit der Transitivität von $\cong$, sowie der Tatsache dass $=$ nur der Sonderfall $\mathit{iso}_V = \mathsf{id}$ und $\mathit{iso}_E = \mathsf{id}$ von $\cong$ ist, sofort $H \cong H$.
Sei als nun $H \cong H'$. Wir müssen $R = R'$ zeigen:
Ohne Mühe folgt zunächst $R \cong R'$. Angenommen es wäre $R \neq R'$: Dann gäbe es $\mathit{iso}_V, \mathit{iso}_E \neq \mathsf{id}$ zwischen V_R und $V_{R'}$ bzw. E_R und $E_{R'}$. Allerdings ist sowohl R, als auch R', durch `enum` und die induzierte Ordnung auf den Kanten entstanden. Das hieße, dass (etwa) `enum` für zwei isomorphe Graphen zwei unterschiedliche Aufzählungen produziert hat, die mit iso_V ineinander überführbar sind. Da `enum` deterministisch ist und die Knoten- und Kantenbenennung nicht beachtet, muss es dazu einen Unterschied bezüglich der mit Variablen oder Selektoren modellierten Objektbeziehungen geben. Diese werden jedoch von lab und/oder att widergespiegelt und es wäre $R \ncong R'$. Also muss $\mathit{iso}_V = \mathsf{id}$ sein und es kann sich nur die Kantenaufzählung durch die induzierte Ordnung unterschieden haben. Jede Abweichung lässt sich aber auf die lexikalische Ordnung von $\mathit{att}(E)$ oder die Ordnung des Alphabetes zurückführen; beide sind aber von vorne herein gegeben und eindeutig. Daher ist $\mathit{iso}_E = \mathsf{id}$. □

Korollar 27 (Kanonischer Repräsentant)
Sei $\overline{H} \in \overline{\mathsf{HC}_\Sigma}$ und $M = \{\mathtt{rep}(H) \mid H \in \overline{H}\}$. Dann ist $|M| = 1$.
Wir schreiben daher $\mathtt{rep}(\overline{H}) = H'$ für das $H' \in M$.

Definition 28 (Zustandsraum über isomorphen Heapkonfigurationen)
Für ein Programm $\mathit{pgm} \in \mathit{Stm}$ und Startheapkonfigurationen $\mathit{HC}_0 \subseteq \mathsf{HC}_\Sigma$ über dem Alphabet Σ ist der Zustandsraum $K^2_{\mathit{pgm},\mathit{HC}_0} = (Q, Q_0, \triangleright, \mathit{obj}, \mathit{kf}, \mathit{var}, \mathit{sel})$ aufbauend auf $K^1_{\mathit{pgm},\mathit{HC}_0}$ von Definition 20 gegeben durch:

- $Q = \overline{\mathsf{HC}_\Sigma} \times (\mathit{Stm} \dot\cup \{\varepsilon\})$, wobei $q = (\overline{H}_q, \mathit{pgm}_q) \in Q$,
- $Q_0 = \{([H]_{\cong,\mathsf{HC}_\Sigma}, \mathit{pgm}) \mid H \in \mathit{HC}_0\}$,
- $q \triangleright q'$ und $\mathit{kf}(q, q')$ werden ausgehend von K^1 wie in der nachfolgenden Definition angegeben gebildet,
- und benutze mit dem Repräsentanten $H_q = \mathtt{rep}(\overline{H}_q) = (V_{H_q}, E_{H_q}, \mathit{att}_{H_q}, \mathit{lab}_{H_q}, \mathit{ext}_{H_q})$ weiterhin $\mathit{obj}(q) = V_{H_q}$, $\mathit{var}(q)(x)$ und $\mathit{sel}(q)(v, s)$ aus Definition 20.

Definition 29 (Semantik auf isomorphen Heapkonfigurationen)
Für $q, q' \in \mathsf{HC}_\Sigma \times (\mathit{Stm} \dot\cup \{\varepsilon\})$ sei $q \triangleright q'$ durch Definition 19 (Semantik von Statements) gegeben, wobei $q = H, \langle \mathit{pgm} \rangle$ usw.
Außerdem sei $\mathit{kf}^1_{q \triangleright q'} : V_q \longrightarrow V_{q'}$ die Korrespondenzfunktion aus K^1 (also $\mathsf{id}_{V_{q'}}\restriction_{V_q}$).
Wir definieren nun $\overline{q} \triangleright \overline{q}'$ für $\overline{q} \in \overline{\mathsf{HC}_\Sigma} \times (\mathit{Stm} \dot\cup \{\varepsilon\})$, indem wir (anschaulich):

$$\overline{q} = \overline{H}, \langle\cdot\rangle \quad \Rightarrow \quad q = \mathtt{rep}(\overline{H}), \langle\cdot\rangle \quad \Rightarrow \quad q \triangleright q' \quad \Rightarrow \quad \overline{q}' = [H']_{\cong,\mathsf{HC}_\Sigma}, \langle\cdot\rangle = \overline{H}', \langle\cdot\rangle$$

Und formal:

$$\frac{\mathtt{rep}(\overline{H}), \langle S\rangle \ \triangleright\ H', \langle S'\rangle}{\overline{H}, \langle S\rangle \ \triangleright\ [H']_{\cong,\mathsf{HC}_\Sigma}, \langle S'\rangle}$$

Beachte, dass nun $V_{\overline{q}} = V_{\mathtt{rep}(\overline{H})} = V_q$, jedoch i. A. $V_{H'} = V_{q'} \neq V_{\overline{q}'} = V_{\mathtt{rep}(H')} = V_{\mathtt{rep}(\overline{H}')}$. Daher nutzen wir die zur Konstruktion von $\mathtt{rep}(H')$ ($\cong H'$) in Definition 25 erstellte Bijektion $iso_V : V_{q'} \longrightarrow V_{\overline{q}'}$ und es ergibt sich $kf_{\overline{q}\triangleright\overline{q}'} : V_{\overline{q}} \longrightarrow V_{\overline{q}'}$ als $kf_{\overline{q}\triangleright\overline{q}'} = iso_V \circ kf^1_{q\triangleright q'}$. Das ist als Komposition einer Bijektion mit einer partiellen injektiven Funktion wieder einer partielle Injektion.

1.7 Isomorphie von Zustandsräumen

Außerdem können wir auch für Zustandsräume einen Isomorphiebegriff einführen:

Definition 30 (Isomorphie von Zustandsräumen)
Zwei (objektorientierte) Zustandsräume $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$ und $K' = (Q', Q'_0, \triangleright', obj', kf', var', sel')$ heißen *isomorph*, $K \simeq K'$, wenn

- es eine Bijektion $iso_Q : Q \longrightarrow Q'$ zwischen den Zuständen gibt und
- für jeden Zustand $q \in Q$ mit $q' = iso_Q(q)$ ein Bijektion $iso_{V_q} : V_q \longrightarrow V_{q'}$ zwischen den Objekten, so dass:
- $Q'_0 = iso_Q(Q_0)$,
- $q'_1 \triangleright' q'_2$ gdw. $q_1 \triangleright q_2$ für alle $q_1, q_2 \in Q$ und $q'_1 = iso_Q(q_1)$, $q'_2 = iso_Q(q_2)$,
- $obj'(iso_Q(q)) = iso_{V_q}(obj(q))$ für alle $q \in Q$,
- $kf'_{q'_1\triangleright' q'_2} \circ iso_{V_{q_1}} = iso_{V_{q_2}} \circ kf_{q_1\triangleright q_2}$ für $q_1 \triangleright q_2$, wobei $q_1, q_2 \in Q$, $q'_1 = iso_Q(q_1)$ und $q_2 = iso_Q(q_2)$; somit ist $kf' : \triangleright \longrightarrow KF_{Q'}$ definiert,
- $var'_{iso_Q(q)} = iso_{V_q} \circ var_q$, oder $var'_{iso_Q(q)}(x) = \mathsf{null}$ falls für $x \in Var_\Sigma$: $var_q(x) = \mathsf{null}$, und
- $sel'_{iso_Q(q)}(iso_{V_q}(v), s) = iso_{V_q}(sel_q(v, s))$ für $v \in V_q$, $s \in Sel_\Sigma$, bzw. $sel'_{iso_Q(q)}(v, s) = \mathsf{null}$.

Dazu ein Beispiel:

Beispiel 2
Die Isomorphie für Zustandsräume fordert nur, dass es Bijektionen zwischen den Zustände und zwischen den Objekten gibt, jedoch nicht, dass die Programme, die zu den entsprechenden Heapkonfigurationen führen, identisch sein müssen. Es gilt etwa für

$$pgm_1 = \{\mathsf{new}(x); \mathsf{while}\ (x\ !=\mathsf{null})\ \{x = x.next\}\}\ \textit{und}$$
$$pgm_2 = \{\mathsf{new}(x); \mathsf{if}\ (x.next == \mathsf{null})\ x = \mathsf{null}\}$$

mit nur der leeren Heapkonfiguration $H = (\emptyset, \ldots, \varepsilon)$ als Startkonfiguration in $HC_0 = \{H\}$, dass $K^1_{pgm_1, HC_0} \simeq K^1_{pgm_2, HC_0}$.

2 Logiken

Eigenschaften von Programmen könnnen sehr gut über temporale Logiken ausgedrückt werden, die es erlauben, sowohl Propositionen an bestimmten Zuständen auszuwerten, als auch über ganze Berechnungsläufe, also Pfade im Zustandsraum zu quantifizieren, um z. B. Aussagen wie: „Irgendwann gilt X“ zu treffen. Die bekanntesten Vertreter sind die *Linear Temporal Logic (LTL)*, sowie die *Computation Tree Logic (CTL)*, die sich in CTL* vereinigen lassen.

LTL ist zielt auf die Pfade eines Zustandsraumes ab: Es können Aussagen aufgestellt werden, die dann für *jeden möglichen* Pfade im Zustandsraum getestet werden.

CTL dagegen betrachtet den Zustandraum als Graph, bei dem pro Zustand entschieden werden kann, welche der Nachfolger und Pfade unter einer bestimmten Teilformel betrachtet werden sollen. Es ist aber nur eine Teilmenge der in LTL möglichen Aussagen über Pfaden in CTL ausdrückbar.

Im Allgemeinen modelliert man die sich verändernde Eigenschaften als atomare Propositionen, die nur in bestimmten Zuständen gelten.

Bei den von uns betrachteten Pointerprogrammen wollen wir mittels Modelchecking gerade die sich durch die Pointermanipulationen ergebenden veränderten Objektbeziehungen verifizieren.

Daher sollen Aussagen wie (für $x, y \in \mathit{Var}_\Sigma, s \in \mathit{Sel}_\Sigma$) $x.s = y$ möglich sein, was besagt, dass im aktuellen Programmzustand das Objekt, auf das die Variable x zeigt, einen Selektor hat, der auf das selbe Objekt zeigt, wie die Variable y. Das sind dann gerade unsere atomaren Propositionen, wobei wir solche Vergleiche noch weiter zerlegt haben; eine „Expression Propsition“ ist bei uns eine Vergleich oder **tt** (wahr)bzw. **ff** (falsch).

Nun ist es aber oft auch von Interesse, ob es (innerhalb eines Zustands) ein Objekt gibt, das bestimmte Bedingungen erfüllt, oder ob für alle Objekte bestimmte Eigenschaften gelten. Um solche Objekte auch über einen Programmlauf (einen Pfad im Zustandsraum) zu verfolgen, reicht es nicht, sich auf Pointeraussagen zu beschränken. Vielmehr ist es nötig, entsprechende Quantoren im „temporalen Aufbau“ von CTL* zu integrieren.

Dazu führen wir, zusätzlich zu den *Pfad*quantoren A und E aus CTL* zusätzlich die *Objekt*quantoren $\forall x$ und $\exists x$ ein, die als Grundmenge auf den Objekten $v \in V_q$ operieren. Zur besseren Unterscheidung verwenden wir für die CTL*-Konstrukte ausnahmslos Großbuchstaben (also X, A, etc.).

Vom Aufbau her werden wir zunächst CTL* formal für unsere Propositionen einführen. Dann fügen wir Objektquantoren hinzu und betrachten, welche sinnvollen Einschränkungen dafür existieren, wie auch CTL und LTL als Einschränkungen von CTL* betrachtet werden können.

2.1 CTL*

Als ersten Schritt definieren wir CTL* in einer Form, die uns Aussagen über Pointervariablen in unserem Zustandsraum ermöglicht. Dazu orientieren wir uns an Ptr unserer Pointermanipulationssprache:

Definition 31 (Syntax von CTL*)
CTL*-Formeln bilden sich nach folgender kontextfreier Grammatik, wobei $x \in \mathit{Var}_\Sigma$ zunächst für Variablen und $s \in \mathit{Sel}_\Sigma$ für Selektoren aus einem gegebenen Alphabet Σ steht:

$$\begin{array}{lcll} \mathcal{A} & ::= & \mathsf{null} \mid x \mid x.s & \in \mathsf{AP}_{\mathrm{CTL}^*} \\ \mathcal{E} & ::= & \mathcal{A} = \mathcal{A} \mid \mathcal{A} \neq \mathcal{A} \mid \mathbf{tt} \mid \mathbf{ff} & \in \mathsf{EP}_{\mathrm{CTL}^*} \\ \mathcal{S} & ::= & \mathcal{E} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \mathsf{A}\mathcal{P} \mid \mathsf{E}\mathcal{P} & \in \mathsf{SF}_{\mathrm{CTL}^*} \\ \mathcal{P} & ::= & \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid \mathsf{X}\mathcal{P} \mid \mathcal{P}\,\mathsf{U}\,\mathcal{P} \mid \mathcal{P}\,\mathsf{R}\,\mathcal{P} & \in \mathsf{PF}_{\mathrm{CTL}^*} \end{array}$$

Negationen ($\neg$) beschränken wir dabei auf „atomare Ausdrücke“, indem wir $\mathcal{E}$ unter Negation abgeschlossen haben, d. h. für $\varphi \in \mathcal{E}$ ist bereits $\neg\varphi \in \mathcal{E}$, wodurch $\neg$ in der Grammatik gar nicht auftritt. Um dennoch die volle Ausdrucksstärke von CTL* zu erhalten, benötigen wir dann allerdings den „Release“-Operator R, der „$\neg(\varphi\,\mathsf{R}\,\theta) \equiv \neg\varphi\,\mathsf{U}\,\neg\theta$“ erfüllt.
$\mathcal{S}$ sind *Stateformeln*, $\mathcal{P}$ *Pfadformeln*. Eine CTL*-Formel beginnt immer als Stateformel und wir verwenden $\psi \in \mathrm{CTL}^*$ für $\psi \in \mathsf{SF}_{\mathrm{CTL}^*}$.
Zur Erhöhung der Lesbarkeit schreiben wir im Folgenden Formeln auch mit Klammern „$(\dots)$“ obwohl dafür keine syntaktische Notwendigkeit besteht.

Anschaulich besagt:

$\mathsf{X}\varphi$**:** **next**: Im nächsten Zustand gilt φ

$\varphi\,\mathsf{U}\,\theta$**:** **until**: In einem späteren Zustand gilt θ und an allen Zuständen davor, einschließlich des aktuellen Zustands, gilt φ.

$\varphi\,\mathsf{R}\,\theta$**:** **release**: Es gilt solange θ, bis φ von dieser Verpflichtung entbindet.

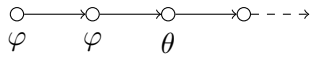


Abbildung 2.1: Erfüllender Pfad für $\varphi\,\mathsf{U}\,\theta$.

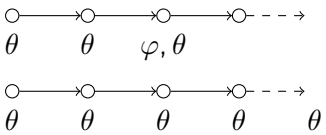


Abbildung 2.2: Erfüllende Pfade für $\varphi\,\mathsf{R}\,\theta$.

2.1.1 Semantik von CTL*

Wir formalisieren die Semantik, in dem wir die Tatsache, dass ein Pfad oder ein Zustand die durch eine Formel gegeben Eigenschaft erfüllt, durch eine Relation formalisieren:

Definition 32 (Modellrelation)
Auf $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$ drückt die Modellrelation $\pi \models_K \varphi$ aus, dass der Pfad $\pi \in Paths_K$ ein Modell für die in der Pfadformel $\varphi \in \mathsf{PF}$ gegebene Eigenschaft ist. Wir sagen: π *erfüllt* φ.
Genauso drückt $q \models_K \psi$ aus, dass der Zustand $q \in Q$ die Stateformel $\psi \in \mathsf{SF}$ erfüllt.

Definition 33 (Semantik von AP)
Mit $q \in K$, $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$ und $x \in Var_\Sigma$, $s \in Sel_\Sigma$ ist
$[\![\cdot]\!] : \mathsf{AP} \longrightarrow V_q \cup \{\mathsf{null}, \bot\}$:

$$[\![\mathsf{null}]\!]_q = \mathsf{null}$$
$$[\![x]\!]_q = var(q)(x)$$
$$[\![x.s]\!]_q = \begin{cases} sel(q)(v) & \text{wenn } v = [\![x]\!]_q \neq \mathsf{null} \\ \bot & \text{sonst} \end{cases}$$

Damit können wir die Semantik von CTL* angeben, indem wir $\models_K$ für Pfadformeln und Stateformeln induktiv definieren:

Definition 34 (Semantik von CTL*)
Für $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$ sei $q \in Q$ und $\pi \in Paths_K$.

$q \models \mu = \nu$, wenn $[\![\mu]\!]_q = [\![\nu]\!]_q \neq \bot$.
$q \models \mu \neq \nu$, wenn $q \not\models \mu = \nu$.
$q \models \mathbf{tt}$, $(q \not\models \mathbf{ff})$.
$q \models \psi \wedge \theta$, wenn $q \models \psi$ und $q \models \theta$.
$q \models \psi \vee \theta$, wenn $q \models \psi$ oder $q \models \theta$.
$q \models \mathsf{A}\varphi$, wenn für jedes $\pi \in Paths(q)$ gilt, dass $\pi \models \varphi$.
$q \models \mathsf{E}\varphi$, wenn es ein $\pi \in Paths(q)$ gibt, so dass $\pi \models \varphi$.
$\pi \models \psi$ $(\psi \in \mathsf{SF})$, wenn $\pi[1] \models \psi$.
$\pi \models \varphi \wedge \theta$, wenn $\pi \models \varphi$ und $\pi \models \theta$.
$\pi \models \varphi \vee \theta$, wenn $\pi \models \varphi$ oder $\pi \models \theta$.
$\pi \models \mathsf{X}\varphi$, wenn $\pi[2\,...] \models \varphi$.
$\pi \models \varphi\,\mathsf{U}\,\theta$, wenn es ein $i \geq 1$ gibt, so dass $\pi[i\,...] \models \theta$ und für alle $j < i$ gilt $\pi[j\,...] \models \varphi$.
$\pi \models \varphi\,\mathsf{R}\,\theta$, wenn es entweder ein $i \geq 1$ gibt, so dass $\pi[i\,...] \models \varphi$ und für alle $j \leq i$: $\pi[j\,...] \models \theta$,
oder für alle $i \geq 1$ gilt $\pi[i\,...] \models \theta$.

Alternativ legt die Entfaltung von U und R folgende Definition nahe:

$$\pi \models \varphi\,\mathsf{U}\,\theta, \text{ wenn } \pi \models \theta \vee (\varphi \wedge \mathsf{X}(\varphi\,\mathsf{U}\,\theta)),$$
$$\pi \models \varphi\,\mathsf{R}\,\theta, \text{ wenn } \pi \models \theta \wedge (\varphi \vee \mathsf{X}(\varphi\,\mathsf{R}\,\theta)),$$

aus der sich aber keine Aussage bei unendlicher Entfaltung ableiten lässt.

Dass das Verbot von Negationen an beliebigen Stellen keine Einschränkung ist, besagt das folgende Lemma:

Lemma 35

- Für jede Stateformel $\varphi \in \mathsf{SF}$ gibt es eine zu „$\neg\varphi$“ äquivalente Stateformel $\psi \in \mathsf{SF}$, so dass für alle Zustände $q \in Q$ gilt: $q \models \psi$ gdw. $q \not\models \varphi$.
- Für jede Pfadformel $\varphi \in \mathsf{PF}$ gibt es eine zu „$\neg\psi$“ äquivalente Pfadformel $\psi \in \mathsf{PF}$, so dass für alle Pfade $\pi \in \mathit{Paths}_K$ gilt: $\pi \models \psi$ gdw. $\pi \not\models \varphi$.

Beweis: (vgl. [2]) Folgt direkt aus der Dualität von $\wedge$ und $\vee$, sowie U und R. Außerdem ist X zu sich selbst dual. □

Häufig werden auch die Operatoren F und G verwendet:

$\mathsf{F}\varphi$**: finally**: In der Zukunft gilt φ.

$\mathsf{G}\varphi$**: globally**: Immer gilt φ.

Korollar 36
Auch die Operatoren $\mathsf{F}\Psi \equiv \mathbf{tt}\ \mathsf{U}\ \Psi$ und $\mathsf{G}\Psi \equiv \neg\mathsf{F}\neg\Psi$, $\Psi \in \mathsf{PF}$, sind semantisch darstellbar, d. h. es gibt eine äquivalente Formel, bei der Negationen nur als Teil von EP auftreten.

2.1.2 CTL

Definition 37
CTL ist die Einschränkung von CTL*, Temporaloperatoren (U, X, R) nur gepaart mit *Pfadquantoren* (A,E) auftreten zu lassen, d. h. (mit $\mathsf{AP}_{\mathrm{CTL}} = \mathsf{AP}_{\mathrm{CTL}^*}$ und $\mathsf{EP}_{\mathrm{CTL}} = \mathsf{EP}_{\mathrm{CTL}^*}$):

$$\begin{array}{lll} \mathcal{S} & ::= & \mathcal{E} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \mathsf{A}\mathcal{P} \mid \mathsf{E}\mathcal{P} \quad \in \mathsf{SF}_{\mathrm{CTL}} \equiv \mathsf{SF}_{\mathrm{CTL}^*} \\ \mathcal{P} & ::= & \mathsf{X}\mathcal{S} \mid \mathcal{S}\,\mathsf{U}\,\mathcal{S} \mid \mathcal{S}\,\mathsf{R}\,\mathcal{S} \quad \in \mathsf{PF}_{\mathrm{CTL}} \end{array}$$

Die Intention von CTL ist es, Pfadformeln auf einen einzigen Temporaloperator zu reduzieren, so dass sich z. B. das Modelchecking CTL nur auf *Zustände* konzentrieren muss.

2.1.3 LTL

Definition 38
Bei LTL werden dagegen innerhalb von Pfadformeln keine weiteren Pfadquantoren erlaubt; eine Folge davon ist, dass keine weiteren Stateformeln mehr auftreten können. LTL betrachtet also *Pfade*.
Der einzige Pfadquantor steht am Anfang der Formel (wenn man die Einbettung in CTL* betrachtet):

$$\begin{array}{lll} \mathcal{S} & ::= & \mathsf{A}\mathcal{P} \quad \in \mathsf{SF}_{\mathrm{LTL}} \\ \mathcal{P} & ::= & \mathcal{E} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid \mathsf{X}\mathcal{P} \mid \mathcal{P}\,\mathsf{U}\,\mathcal{P} \mid \mathcal{P}\,\mathsf{R}\,\mathcal{P} \quad \in \mathsf{PF}_{\mathrm{LTL}} \end{array}$$

Da $q \models \mathsf{E}\varphi$ gdw. $q \not\models \mathsf{A}\neg\varphi$ und eine zu $\neg\varphi$ äquivalente Pfadformel existiert, ist die Festlegung auf A in $\mathsf{SF}_{\mathrm{LTL}}$ keine Einschränkung.
Weil nun A aber keine zusätzliche Information mehr kodiert, wird üblicherweise auf $\mathcal{S}$ verzichtet, so dass LTL-Formeln aus den *Pfad*formeln der Grammatik gebildet werden: $\varphi \in \mathrm{LTL}$, wenn $\varphi \in \mathsf{PF}_{\mathrm{LTL}}$.

Beispiel 3
Ein Beispiel für eine CTL-Formel ist: $(\mathsf{EX}\mathit{head}.\mathit{next} = \mathit{last}) \wedge \mathsf{A}(\mathit{last}.\mathit{prev} \neq \mathsf{null}\,\mathsf{R}\,\mathit{head} = \mathit{last})$.
Eine LTL-Formel ist: $(\mathsf{X}\mathit{head}.\mathit{next} = \mathit{last}) \vee ((\mathit{head}.\mathit{prev} = \mathsf{null}\,\mathsf{U}\,\mathit{head}.\mathit{next} \neq \mathsf{null}) \wedge (\mathbf{tt}\,\mathsf{U}\,\mathit{head} = \mathit{last}))$.

2.2 Quantifizierte Temporale Logiken

Ziel der Arbeit ist das Einführen von Objektquantoren in CTL*. Sie sollen ermöglichen Aussagen über bestimmte Objekte formal zu modellieren, während gleichzeitig das temporale Verhalten im Zustandsraum berücksichtigt wird.

Dazu benötigen wir die mit K bereits eingeführte, aber in CTL* noch nicht verwendete Korrespondenzfunktion kf. Weiter führen wir an die bekannten Programmvariablen angelehnte Logikvariablen ein, für die jedoch durch die Objektquantoren bestimmt wird, welche Objekte sie innerhalb der Logik referenzieren.

2.2.1 Q*CTL*

Wir erweitern also unsere Definition von CTL*:

Definition 39 (Syntax von Q*CTL*)
Sei LV eine von Var_Σ disjunkte Menge von *Logikvariablen*.
Ausgehend von Definition 31 erlauben wir dann, dass $x \in Var_\Sigma \dot\cup LV$. Außerdem ergänzen wir Stateformeln und Pfadformeln um die *Objektquantoren* $\exists$ und $\forall$.

$$
\begin{array}{lcll}
\mathcal{A} & ::= & \mathsf{null} \mid x \mid x.s & \in \mathsf{AP}_{\text{Q*CTL*}} \\
\mathcal{E} & ::= & \mathcal{A} = \mathcal{A} \mid \mathcal{A} \neq \mathcal{A} \mid \mathbf{tt} \mid \mathbf{ff} & \in \mathsf{EP}_{\text{Q*CTL*}} \\
\mathcal{S} & ::= & \mathcal{E} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \mathsf{A}\mathcal{P} \mid \mathsf{E}\mathcal{P} \mid \forall_{\mathcal{S}} x.\mathcal{S} \mid \exists_{\mathcal{S}} x.\mathcal{S} & \in \mathsf{SF}_{\text{Q*CTL*}} \\
\mathcal{P} & ::= & \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid \mathsf{X}\mathcal{P} \mid \mathcal{P}\,\mathsf{U}\,\mathcal{P} \mid \mathcal{P}\,\mathsf{R}\,\mathcal{P} \mid \forall_{\mathcal{P}} x.\mathcal{P} \mid \exists_{\mathcal{P}} x.\mathcal{P} & \in \mathsf{PF}_{\text{Q*CTL*}}
\end{array}
$$

Dabei besagt:

$\exists x.\varphi(x)$**:** Es gibt im aktuellen Zustand ein Objekt $v \in V$, so dass $\varphi(v)$ gilt.

$\forall x.\varphi(x)$**:** Für alle Objekte $v \in V$ im aktuellen Zustand gilt $\varphi(v)$.

Das stellt die allgemeinste Form der Ergänzung dar. Man sieht, dass die Objektquantoren in zwei Versionen, einmal für Stateformeln und ein zweites mal für Pfadformeln vorkommen. Der Index wurde hinzugefügt, um die Mehrdeutigkeit zu vermeiden, die durch $\mathcal{S}$ in $\mathcal{P}$ entsteht:

Beispiel 4
Wird auf den Index verzichtet, dann könnte $\mathsf{A}\forall x.(x = x)$ *abgeleitet werden als* $\mathsf{A}\forall_{\mathcal{S}} x.(x = x)$*, aber auch als* $\mathsf{A}\forall_{\mathcal{P}} x.(x = x)$*. Der Unterschied besteht darin, ob die Teilformel vor oder nach dem Quantor als Stateformel gelesen wird.*

Eine Alternative zum Index wäre, die semantischen Implikationen zunächst unberücksichtigt lassend, sich darauf festzulegen, dass nach $\forall_{\mathcal{P}}$ eine „echte“ Pfadformel auftreten muss. Dazu könnte die $\mathcal{P}$-Produktion verdoppelt werden, wobei aber bei $\mathcal{P}'$ das $\mathcal{S}$ entfernt wird, so dass in $\mathcal{P}$ und $\mathcal{P}'$ jeweils das eingeschränkte $\forall x.\mathcal{P}'$, bzw. $\exists x.\mathcal{P}'$ verwendet wird. Der besseren Verständlichkeit wegen verzichten wir hier aber auf derartige rein syntaktischen (wie im Folgenden deutlich werden wird) Nuancen und indizieren die Quantoren (soweit es sinnvoll ist).

2.2.2 Semantik von Q*CTL*

Um die Semantik von Q*CTL* einzuführen, müssen wir über die aktuelle Belegung der Logikvariablen Buch führen.

Definition 40 (Logikvariablen)
Die partiellen Funktionen $vl_q : LV \longrightarrow V_q$ ordnen einer Logikvariable im Zustand q ein Objekt zu; definiert vl_q keine Zuordnung für $x \in LV$, dann sei $vl_q(x) = \mathsf{null}$.
Die Menge aller möglicher Variablenbelegungen im Zustand q bezeichnen wir dann mit Vl_q.

Damit erweitern wir die Modellrelation:

Definition 41 (Modellrelation)
Die Modellrelation $q, vl \models_K \psi$ drückt aus, dass der Zustand $q \in Q$ die Stateformel $\psi \in \mathsf{SF}$ mit der (Logik-)Variablenbelegung $vl \in Vl_q$ erfüllt.
Ebenso gibt $\pi, vl \models_K \varphi$ an, dass der Pfad $\pi \in Paths_K$ die Pfadformel $\varphi \in \mathsf{PF}$ mit der in $\pi[1]$ als vl startenden Variablenbelegung $vl \in Vl_{\pi[1]}$ erfüllt.
Außerdem sagen wir: $K \models \psi$, wenn für alle Startzustände $q_0 \in Q_0$ gilt: $q_0, vl_0 \models_K \psi$. Dabei enthält die initiale Variablenbelegung $vl_0 : LV \longrightarrow V_{q_0}$ keine Zuordnungen, d. h. $vl_0(x) = \mathsf{null}$ für alle $x \in LV$.

Ebenso erweitern wir $[\![\cdot]\!]_q$:

Definition 42 (Semantik von AP)
Sei $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$, $q \in Q$ und $vl \in Vl_q$.
$[\![\cdot]\!]_{q,vl} : \mathsf{AP} \longrightarrow V_q \cup \{\bot, \mathsf{null}\}$ ist gegeben durch:

$$[\![\mathsf{null}]\!]_{q,vl} = \mathsf{null}$$

$$[\![x]\!]_{q,vl} = \begin{cases} var(q)(x) & \text{für } x \in Var_\Sigma, \text{ und} \\ vl(x) & \text{für } x \in LV. \end{cases}$$

$$[\![x.s]\!]_{q,vl} = \begin{cases} sel(q)(v) & \text{wenn } v = [\![x]\!]_{q,vl} \neq \mathsf{null} \\ \bot & \text{sonst} \end{cases}$$

Definition 43 (Semantik von Q*CTL*)
Sei $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$, $q \in Q$ und $\pi \in Paths_K$, womit wir, ausgehend von der Semantik von CTL*, $\models_K$ definieren.
Für Stateformeln ist ($vl \in Vl_q$):

$q, vl \models \mu = \nu$, wenn $[\![\mu]\!]_{q,vl} = [\![\nu]\!]_{q,vl} \neq \bot$.
$q, vl \models \mu \neq \nu$, wenn $q, vl \not\models \mu = \nu$.
$q, vl \models \mathbf{tt}$, ($q, vl \not\models \mathbf{ff}$).
$q, vl \models \psi \wedge \theta$, wenn $q, vl \models \psi$ und $q, vl \models \theta$.
$q, vl \models \psi \vee \theta$, wenn $q, vl \models \psi$ oder $q, vl \models \theta$.
$q, vl \models \mathsf{A}\varphi$, wenn für jedes $\pi \in Paths(q)$ gilt, dass $\pi, vl \models \varphi$.
$q, vl \models \mathsf{E}\varphi$, wenn es ein $\pi \in Paths(q)$ gibt, so dass $\pi, vl \models \varphi$.
$q, vl \models \forall_{\mathcal{S}} x.\psi$, wenn für jedes $v \in V_q$ gilt, dass $q, vl[x \mapsto v] \models \psi$
$q, vl \models \exists_{\mathcal{S}} x.\psi$, wenn es ein $v \in V_q$ gibt, so dass $q, vl[x \mapsto v] \models \psi$.

Und für Pfadformeln ($vl \in Vl_{\pi[1]}$):

$\pi, vl \models \psi$ ($\psi \in \mathsf{SF}$), wenn $\pi[1], vl \models \psi$.
$\pi, vl \models \varphi \wedge \theta$, wenn $\pi, vl \models \varphi$ und $\pi, vl \models \theta$.
$\pi, vl \models \varphi \vee \theta$, wenn $\pi, vl \models \varphi$ oder $\pi, vl \models \theta$.
$\pi, vl \models \mathsf{X}\varphi$, wenn $\pi[2\ldots], (kf_{\pi[1]\triangleright\pi[2]} \circ vl) \models \varphi$.
$\pi, vl \models \varphi \,\mathsf{U}\, \theta$, wenn es ein $i \geq 1$ gibt, so dass $\pi[i\ldots], (kf_{\pi[1\ldots i]} \circ vl) \models \theta$ und für alle $1 \leq j < i$ gilt $\pi[j\ldots], (kf_{\pi[1\ldots j]} \circ vl) \models \varphi$.
$\pi, vl \models \varphi \,\mathsf{R}\, \theta$, wenn es entweder ein $i \geq 1$ gibt, so dass $\pi[i\ldots], (kf_{\pi[1\ldots i]} \circ vl) \models \varphi$ und für alle $1 \leq j \leq i : \pi[j\ldots], (kf_{\pi[1\ldots j]} \circ vl) \models \theta$, oder für alle $i \geq 1$ gilt $\pi[i\ldots], (kf_{\pi[1\ldots i]} \circ vl) \models \theta$.
$\pi, vl \models \forall_{\mathcal{P}} x.\varphi$, wenn für jedes $v \in V_{\pi[1]}$ gilt, dass $\pi, vl[x \mapsto v] \models \varphi$
$\pi, vl \models \exists_{\mathcal{P}} x.\varphi$, wenn es ein $v \in V_{\pi[1]}$ gibt, so dass $\pi, vl[x \mapsto v] \models \varphi$.

Dann ist $(kf_{\pi[1\ldots i]} \circ vl) : LV \longrightarrow V_{\pi[i]}, \in Vl_{\pi[i]}$; außerdem handelt es sich als Komposition von partiellen Funktionen selbst wieder um eine partielle Funktion.

Um auch hier sicherzustellen, dass das Verbot von Negationen semantisch nichts einschränkt, erweitern wir Lemma 35 auf ganz Q*CTL*.

Lemma 44

- Für jede Stateformel $\varphi \in \mathsf{SF}_{\mathrm{Q^*CTL^*}}$ gibt es eine zu „$\neg\varphi$“ äquivalente Stateformel $\psi, vl \in \mathsf{SF}_{\mathrm{Q^*CTL^*}}$, so dass für alle Zustände $q \in Q$ und Variablenbelegungen $vl \in Vl_q$ gilt: $q, vl \models \psi$ gdw. $q, vl \not\models \varphi$.
- Für jede Pfadformel $\varphi \in \mathsf{PF}$ gibt es eine zu „$\neg\psi$“ äquivalente Pfadformel $\psi \in \mathsf{PF}_{\mathrm{Q^*CTL^*}}$, so dass für alle Pfade $\pi, vl \in \mathit{Paths}(K)$ und Variablenbelegungen $vl \in Vl_{\pi[1]}$ gilt: $\pi, vl \models \psi$ gdw. $\pi, vl \not\models \varphi$.

Beweis: Zunächst bemerken wir, dass Negationen auf vl keinen Einfluss haben, wenn von $\forall$ und $\exists$ abgesehen wird. Weiter stellen wir fest, dass $\forall x.\Psi = \neg\exists x.\neg\Psi$ ist, wobei sich vl dabei wie erwartet verhält. Zusammen mit Lemma 35 lassen sich also alle Q*CTL*-Formel umschreiben. □

Wir betrachten nun, was passiert, wenn Objektquantoren nur entweder zu $\mathcal{P}$ oder zu $\mathcal{S}$ hinzufügt werden. Insbesondere der Objektquantor im Pfadkontext ist zunächst etwas schwer zu fassen; daher gibt es hierzu nach der nächsten Definition einer Logik ohne diesen Quantor ein Beispiel.

2.2.3 Q$^+$CTL*

Werden also Objektquantoren nur in Stateformeln zugelassen, so ergibt sich:

Definition 45 (Syntax vom Q$^+$CTL*)
Bei Q$^+$CTL* werden Objektquantoren nur zu $\mathcal{S}$ hinzugefügt (also nur $\forall_{\mathcal{S}}$ und $\exists_{\mathcal{S}}$, jedoch ohne Index), so dass (zusammen mit $\mathsf{AP}_{\mathrm{Q^+CTL^*}} = \mathsf{AP}_{\mathrm{Q^*CTL^*}}$ und $\mathsf{EP}_{\mathrm{Q^+CTL^*}} = \mathsf{EP}_{\mathrm{Q^*CTL^*}}$):

$$\mathcal{S} \in \mathsf{SF}_{\mathrm{Q^+CTL^*}} \equiv \mathsf{SF}_{\mathrm{Q^*CTL^*}}$$
$$\mathcal{P} \in \mathsf{PF}_{\mathrm{Q^+CTL^*}} \equiv \mathsf{PF}_{\mathrm{CTL^*}}$$

Beispiel 5
Wir geben zunächst einige Beispiele, wie Formeln aus Q^+CTL* aussehen können:

$$\begin{aligned} &\forall x.(\mathsf{AX}\varphi \wedge \mathsf{AX}\theta) && \in Q^+CTL* \\ &\mathsf{A}\forall x.(\mathsf{X}\varphi \wedge \mathsf{X}\theta) && \notin Q^+CTL* \\ &\mathsf{AX}\forall x.\varphi && \in Q^+CTL* \end{aligned}$$

Dass es sich bei Q$^+$CTL* um eine echte Einschränkung der Ausdrucksstärke handelt wird deutlich, wenn wir $\exists_{\mathcal{P}}$ exemplarisch genauer betrachten:

Beispiel 6
Gegeben sei $\varphi = \mathsf{AF}\exists_{\mathcal{P}} x.(x.next = cur \wedge \mathsf{X}(x = x.next)) \in Q*CTL*$, mit insb. der Programmvariable cur und dem Selektor next, wie in Zustandsraum in Abbildung 2.3 zu sehen.
Die mit φ modellierte Eigenschaft kann (im allgemeinen Fall) offenbar mit $\exists_{\mathcal{S}}$ (oder auch $\forall_{\mathcal{S}}$) nicht geprüft werden.

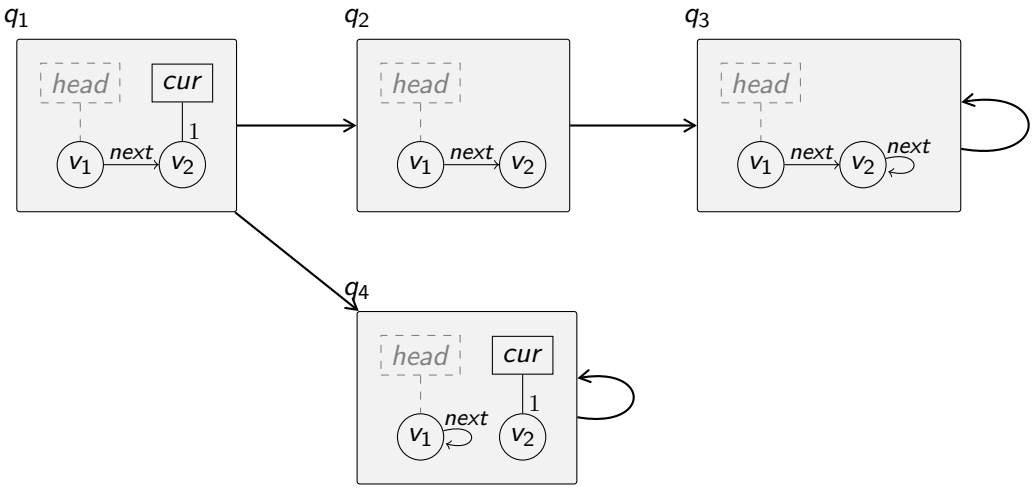


Abbildung 2.3: Zustandsraum für Beispiel 6; $\varphi = \mathsf{AF}\exists_{\mathcal{P}} x.(x.next = cur \wedge \mathsf{X}(x = x.next))$.

Wird bei Q*CTL* dagegen $\forall_{\mathcal{S}}$ und $\exists_{\mathcal{S}}$ weggelassen, dann könnte wegen $\mathcal{S}$ als Startproduktion (etwa) $\forall$ nicht ohne vorhergehendes A auftreten:

Beispiel 7
Sei y eine Programmvariable. Die Formel $\forall x.(x = y)$ sagt aus, dass es nur ein Objekt im aktuellen Zustand gibt: das Objekt auf das y zeigt. Sie ist ohne „$\forall_{\mathcal{S}}$“ nicht ableitbar.

Dennoch ist das, im Gegensatz zu Q$^+$CTL*, keine Einschränkung:

Lemma 46 ($\forall_{\mathcal{P}}$ und $\exists_{\mathcal{P}}$ sind ausreichend für Q*CTL*)
Für jedes $\varphi \in$ Q*CTL* gibt es eine äquivalente $\forall_{\mathcal{S}}$- und $\exists_{\mathcal{S}}$-freie Formel $\theta \in$ Q*CTL*, so dass für alle q, vl: $q, vl \models \varphi$ gdw. $q, vl \models \theta$.

Beweis: Wir weisen darauf hin, dass $q, vl \models \forall_{\mathcal{S}} x.\psi$ gdw. $q, vl \models \mathsf{A}\forall_{\mathcal{P}} x.\psi$ gdw. $q, vl \models \mathsf{E}\forall_{\mathcal{P}} x.\psi$, wobei wegen $\forall_{\mathcal{S}}$ festgelegt ist, dass $\psi \in \mathsf{SF}$.
Das heißt: A (bzw. E) hat überhaupt keine Auswirkung; analoges gilt für $\exists_{\mathcal{S}}$.
Daher können wir θ bilden, indem wir in φ alle $\forall_{\mathcal{S}}$ und $\exists_{\mathcal{S}}$ durch $\mathsf{A}\forall_{\mathcal{P}}$, bzw. $\mathsf{A}\exists_{\mathcal{P}}$ ersetzen. □

2.2.4 QCTL*

Im Gegensatz zu den Pfadquantoren, die jeweils nur bis zum nächsten Pfadquantor gelten, sind Objektquantoren, z. B. $\exists x. \ldots \exists y. \ldots$ für $x \neq y$, $x, y \in LV$ voneinander unabhängig. Doch auch das kann lässt sich einschränken:

Definition 47 (Syntax von QCTL*)
In QCTL* $\subseteq$ Q$^+$CTL* muss immer über alle Logikvariablen gleichzeitig quantifiziert werden. Mit den Variablen $x_1, \ldots, x_{|LV|} \in LV$ ist:

$$\begin{aligned} \mathcal{Q} &::= \forall \mid \exists \\ \mathcal{S} &::= \mathcal{E} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid \mathsf{A}\mathcal{P} \mid \mathsf{E}\mathcal{P} \mid \mathcal{Q}x_1 \ldots \mathcal{Q}x_{|LV|}.\mathcal{S} \quad \in \mathsf{SF}_{\mathrm{QCTL}^*} \end{aligned}$$

Beispiel 8
Eine Formeln aus QCTL:*

$$\forall x \forall y.\mathsf{A}(\mathsf{X}\varphi \wedge \forall x \forall y \mathsf{EX}\theta) \in QCTL^*$$

2.2.5 Q$^+$CTL und QCTL

Aus Q$^+$CTL* ergibt sich analog zu CTL die Logik Q$^+$CTL, d. h. durch Beschränkung von Pfadformeln auf jeweils einen einzige Temporaloperator. Weiter lässt sich QCTL als Schnitt von Q$^+$CTL und QCTL* darstellen.

Der Versuch eine Logik „Q*CTL“ zu definieren scheitert allerdings, da das zustandsbasierte Vorgehen von CTL bereits zu stark mit dem Pfadkonzept von LTL (bzw. CTL*) vermischt ist. Entweder erhält man Q*CTL $\equiv$ Q*CTL*, oder es bleibt Q*CTL $\equiv$ Q$^+$CTL.

Auch wenn QCTL also am ehesten CTL entspricht, ist es durch die Kopplung aller Logikvariablen schon deutlich eingeschränkter als etwa Q$^+$CTL, womit nur Q$^+$CTL eine sinnvolle und zugleich mächtige Einschränkung von Q*CTL* im Geiste von CTL ergibt.

Beispiel 9
Beispiele für Formeln aus Q$^+$CTL und QCTL:

$$\begin{aligned} &\forall x.\mathsf{E}(\varphi\ \mathsf{U}\ (\forall y.\theta)) && \in Q^+CTL \\ &\forall x \exists y.(\psi \vee (\mathsf{AX}\varphi \wedge \exists x \forall y.\mathsf{EX}\theta)) && \in QCTL \end{aligned}$$

2.2.6 Q*LTL, Q$^+$LTL und QLTL

Im Gegensatz zu LTL orientiert sich LTL ganz klar an Pfadformeln, die sich durch ein (implizites) A vor der Formel auf einen einzigen Pfadquantor beschränken.

So ergeben sich Q*LTL und QLTL, wobei bei letzterer Logik der Syntax wohl leicht modifiziert werden sollte, um A nicht schreiben zu müssen (z. B. indem von Q$^+$CTL* statt QCTL* abgeleitet und auf LTL eingeschränkt wird); auf $\forall$, und insbesondere $\exists$, kann dagegen (im Gegensatz zu E) nicht verzichtet werden, ohne Ausdruckskraft zu verlieren.

Q$^+$LTL fällt allerdings aus der Reihe: Um von einem $\forall_{\mathcal{S}}$ (bzw. $\exists_{\mathcal{S}}$) wieder auf Pfaden zu kommen, muss nach einer Gruppe von Quantoren (die eine Stateformel darstellen) ein neues A hinzukommen (wenn man LTL folgt). Außerdem muss wieder erlaubt werden, dass innerhalb von Pfadformeln $\forall_{\mathcal{S}}$ auftreten kann. Das heißt aber, da sich (etwa) $\forall x.\mathsf{A}\varphi$ für nicht von x abhängiges φ semantisch auf $\mathsf{A}\varphi$ reduziert, dass Q$^+$LTL nahezu genauso mächtig ist wie AQ$^+$CTL*, also die Einschränkung von Q$^+$CTL* auf die Verwendung von nur A als

Pfadquantor. Erlaubt man A oder E nach einem Objektquantor zu wählen (also anzugeben), so hätte man wieder ganz Q$^+$CTL*. Insbesondere ist Q$^+$LTL $\not\subseteq$ Q*LTL.

Somit sind hier sowohl Q*LTL als auch QLTL sinnvolle Varianten.

Beispiel 10
*Beispiele für Formeln aus Q*LTL und QLTL:*

$$\exists y.(\mathsf{X}(\exists x.\varphi) \wedge (\mathbf{tt}\,\mathsf{U}\,\forall y.(\psi \vee \mathsf{X}\theta))) \qquad \in \mathit{Q*LTL}$$
$$\forall x \exists y.(\mathsf{X}\varphi \wedge (\psi\,\mathsf{R}\,\theta)) \qquad \in \mathit{QLTL}$$

In Abbildung 2.4 haben wir schließlich dargestellt, welche Logiken beim Hinzufügen von Objektquantoren entstehen können und wie deren Teilmengenbeziehung untereinander aussieht.

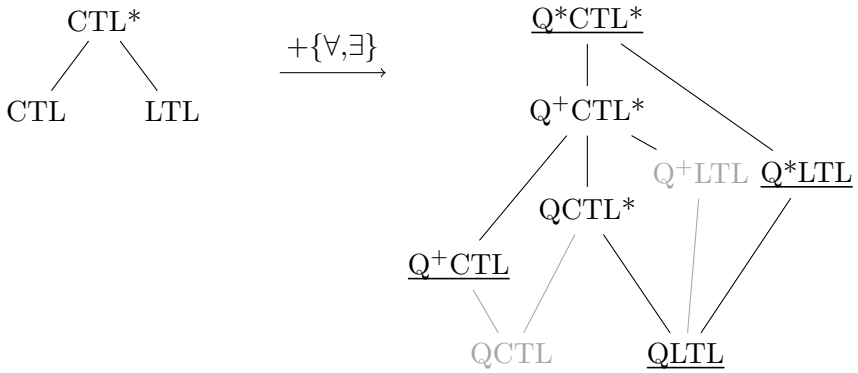


Abbildung 2.4: Hierarchie von Logiken durch Hinzufügen von Objektquantoren.
Wenig relevate Logiken sind in grau, besonders wichtige unterstrichen.

2.2.7 Q$^-$CTL* und Q$^-$LTL

Dem aufmerksamen Leser mag aufgefallen sein, dass die Möglichkeit nur über alle Logikvariablen gleichzeitig zu quantifizieren, wie in QCTL*, aber ohne die Einschränkung von Q$^+$CTL*, nur $\forall_{\mathcal{S}}$ ($\exists_{\mathcal{S}}$) zu verwenden, bisher unberücksichtigt geblieben ist.

Der Vollständigkeit halber sei die so eingeschränke Logik mit Q$^-$CTL* bezeichnet. Aus ihr ergibt sich schließlich dann noch die Logik Q$^-$LTL (auch hier wäre ein „Q$^-$CTL“ $\equiv$ QCTL), die allerdings eher akademischer denn praktischer Natur ist und sich wie in Abbildung 2.5 gezeigt einfügen.

2.3 Ergänzendes

Definition 48 (Q*CTL*-Äquivalenz)
Zwei Zustandsräumen K und K' heißen *durch Q*CTL* nicht unterscheidbar*, wenn für jedes $\psi \in$ Q*CTL* gilt: $K \models \psi$ gdw. $K' \models \psi$.

2.3.1 Isomorphe Zustandsräume

Satz 49
Isomorphe Zustandsräume sind in Q*CTL* nicht unterscheidbar.

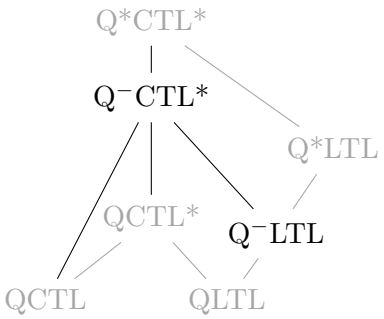


Abbildung 2.5: Hierarchie mit Q⁻CTL* und Q⁻LTL.

Beweis: Wir können q, vl und π, vl via iso_Q und iso_{V_q} nach q', vl', bzw. π', vl' übertragen. In der Semantik von Q*CTL* können dann auch bei $\exists$ und $\forall$ entsprechende Objekte gefunden, bzw. nicht gefunden werden. Damit ändert sich $\models$ nicht. □

2.3.2 Erweiterungen

In Anbetracht unserer konkreten Modellierung der Zustände, nämlich $q = (H, pgm) \in Q$, bietet es sich an, CTL* und damit auch Q*CTL* usw. um Propositionen ε, $\bot$ zu erweitern (genauer: EP zu erweitern, so dass es unter Negationen abgeschlossen bleibt), die Endzustände identifizieren und anhand von $pgm = \varepsilon$ und $\bot$ (vgl. Definition 19 (Semantik von Statements)) danach zu unterscheiden, ob es sich ein ordnungsgemäßes Programmende oder einen Abbruch wegen null-pointer-dereferenzierung handelt. Formal muss dazu die Signatur des Zustandsraumes um zwei disjunkte Teilmengen $bot, eps \subseteq Q$ erweitert werden, die die jeweiligen Endzustände identifizieren. Auch die Semantik der Logik lässt sich ohne Anstrengung entsprechend erweitern, genauso wie das im nächsten Kapitel eingeführte Modelchecking. Für den Anwender ist dabei anzumerken, dass $x.sel$ für eine Logikvariable $x = \mathsf{null}$ natürlich nicht zu einem „Abbruch“ ($\bot$) führt, sondern für alle möglichen y, auch $y = \mathsf{null}$ immer $x.sel \neq y$ und $x.sel \neq y.sel$ gilt.

Im folgenden Kapitel geben wir nun an, wie man Q*CTL*-Formeln auf objektorientierten Zustandsräumen verifiziert.

3 Modelchecking

Beim Modelchecking wird versucht auf algorithmische, also automatisierbare Weise, eine Logikformel auf einem gegebenen Zustandsraum zu verifizieren, d. h. zu zeigen, dass sie überall gilt, oder anders ausgedrückt: nirgends verletzt wird.

Für verschiedene Logiken wurden dabei unterschiedliche Herangehensweisen entwickelt. Auf die grundlegenden Prinzipien wird in [1] im Detail eingegangen. Einen Überblick über die Techniken für LTL gibt etwa [11]. Unterschiede bestehen unter anderem darin, ob der Zustandsraum von hinten, also den Endzuständen oder von den Startzuständen aus untersucht wird.

Eine effiziente Technik für ganz CTL* haben Bhat, Cleaveland und Grumberg im 1995 erschienen Paper [2] angegeben. Dabei wird der Zustandsraum von den Startzuständen her untersucht und kann sogar erst während dieser Erforschung aufgebaut werden; diese Eigenschaft wird „On-the-Fly“ genannt.

Diesen Algorithmus werden wir auf Q*CTL* erweitern. Er basiert auf einer sogenannten Beweisstruktur, die als Kreuzprodukt von Zuständen mit Teilformeln entsteht. Genauer gesagt ist eine Beweisstruktur ein Graph, dessen Knoten sogenannten Forderungen sind (Zustände mit Teilformeln) und dessen Kanten angeben, welche Forderungen durch den Beweis anderer Forderungen (Teilziele) bereits komplett bewiesen sind.

Die Beweisstruktur wird dann mittels Tiefensuche erforscht, wobei für R und U auch Aussagen über unendliche Pfade im Zustandsraum wichtig werden. Diese können (müssen aber nicht) zu unendlichen Pfaden in der Beweisstruktur führen.

Wir werden zunächst die wichtigen Begriffe einführen, dann die zugrundeliegende Theorie behandeln, um dann einen Algorithmus zunächst für Q*LTL anzugeben, den wir schließlich auf Q*CTL* erweitern. Dieses Vorgehen, d. h. mit einem Algorithmus für LTL (Q*LTL) zu beginnen, haben wir, genauso wie vieles andere, aus [2] übernommen.

Wir beziehen uns im Folgenden auf den Zustandsraum $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$, auch ohne weitere Hinweise. Außerdem verlangen wir, dass dieser endlich ist ($|Q| < \infty$). Das kann z. B. dadurch sichergestellt werden, dass das zu testende Programm terminiert. Durch das Zusammenfassen isomorpher Heapkonfigurationen gibt es aber sogar nicht-terminierende Programme mit endlichem Zustandsraum. Weiter gebe es nur endlich viele Objekte in jedem Zustand, also insbesondere $\sum_{q \in Q} |obj(q)| < \infty$.

Zum besseren Verständnis des nachfolgenden Abschnittes wollen wir zunächst unsere Modellrelation etwas anders darstellen:

Definition 50 (Alternative Schreibweise)
Für Pfadformeln können wir statt $\pi, vl \models \varphi$ genauso gut auch $\pi \models \varphi^{vl}$ schreiben, da $\models \subseteq (Q^+ \times Vl) \times \mathsf{PF}, \cong Q^+ \times (\mathsf{PF} \times Vl)$.
Hierbei handelt es sich also alleine um eine Änderung der Notation, die keine Auswirkung auf die Semantik hat.

3.1 Beweisstrukturen für Q*LTL

Ein Algorithmus muss, um eine Formel zu verifizieren, einen Beweis für die Formel finden. Dafür führen wir nun einen Pendanten zur semantischen Modellrelation ein:

Definition 51 (Forderung, Formelmenge)
Einen Ausdruck der Form $q \vdash \mathsf{A}\Phi$ nennen wir *Forderung*.
Gefordert ist dann einen Beweis für eine der Formeln aus der Menge von Pfadformeln mit Variablenbelegungen $\Phi = \{\varphi_1^{vl_1}, \ldots, \varphi_n^{vl_n}\}$ für das entsprechende $vl_i \in Vl_q$ im Zustand $q \in Q$ zu zeigen (abzuleiten). Später werden wir erreichen, dass unsere Beweisstrategie für $\vdash$ genau dann zu einem Beweis führen, wenn für ein solches $\varphi_i^{vl_i}$ gilt: $q, vl_i \models \mathsf{A}\varphi_i$.
Die Menge aller Forderungen sei $\Gamma = \{q \vdash A\Phi \mid q \in Q, \Phi \subseteq \mathsf{PF}_{\mathrm{Q^*CTL^*}} \times Vl_q\}$; wir betrachten aber, wie schon angedeutet, zunächst nur die Teilmenge, für die $\Phi \subseteq \mathsf{PF}_{\mathrm{Q^*LTL}}$.

Definition 52 (Notation)
Wir verwenden außerdem: $\mathsf{A}(\Phi, \varphi_1^{vl_1}, \ldots, \varphi_n^{vl_n}) = \mathsf{A}(\Phi \cup \{\varphi_1^{vl_1}, \ldots, \varphi_n^{vl_n}\})$
Ist $\gamma \equiv q \vdash \mathsf{A}\Phi$, dann schreiben wir q_γ für das in der Forderung enthaltene q, sowie $\varphi^{vl} \in \gamma$, wenn $\varphi^{vl} \in \Phi$ ist.
Mit $\Phi = \{\varphi_1^{vl}, \ldots, \varphi_n^{vl}\}$ schreiben wir $q \models \mathsf{A}\Phi$, wenn für (mindestens) ein $1 \leq i \leq n$ gilt: $q, vl_i \models \mathsf{A}\varphi_i$.

Das Grundprizip dieses Modelcheckingansatzes sind top-down Beweisregeln, die eine Formel, ein Beweisziel, in einzelne Teilziele zerlegen, die durch die Semantik motiviert sind. Wenn alle Teilziele gezeigt werden, dann ist auch das Beweisziel komplett bewiesen.

Definition 53 (Beweisregeln)
Sei $\mu \in \mathsf{EP}_{\mathrm{Q^*CTL^*}}$, also die Propositionen von Q*CTL*, die auf einem einzelnen Zustand (und einer Variablenbelegung) auswertbar sind.
Beweisregeln bestehen aus dem *Beweisziel*, das oben steht, und allen dafür zu zeigenden Teilzielen, die auf der nächsten Seite angegeben sind (Abbildung 3.1).

$R^{\forall x}$ hat also eine an das $\wedge$ angelehnte Struktur, $R^{\exists x}$ ist $\vee$ ähnlich, wobei aber zusätzlich vl geändert wird. R^{X} geht erst dann zum nächsten Zustand über, wenn alle im aktuellen Zustand auswertbaren Teilformeln „abgearbeitet“ sind. Die Schwierigkeit von LTL (Q*LTL, etc.) im Vergleich zu CTL beim Modelchecking ist, dass verschiedene Teilformeln auf dem *selben* Pfad getestet werden müssen. R^{X} ermöglicht nun gerade das, im Zusammenspiel mit der disjunktiv verknüpften Formelmenge Φ, indem immer nur für alle $\vee$-Stücke gemeinsam ein Schritt, für alle Formeln zu dem selben Zielzustand, gegangen wird. Bei $\wedge$ besteht dieses Problem nicht, da das A am Anfang die in der Teilformel formulierte Eigenschaft immer noch für *alle* vom Punkt der Auswertung des $\wedge$ weiterführenden Pfade getestet werden muss.

Das Ziel ist nun zu zeigen, dass $q \vdash \mathsf{A}\Phi$ genau dann beweisbar ist, wenn $q \models \mathsf{A}\Phi$. Dazu stellen wir eine erste Verbindung zwischen der Semantik von Q*CTL* und den Beweisregeln her:

Lemma 54
Sei $\gamma \equiv q \vdash \mathsf{A}\Phi$ eine Forderung.

- Wenn die Teilziele nach Anwendung einer Regel auf γ die Form $q_1 \vdash \mathsf{A}\Phi_1, \ldots, q_n \vdash \mathsf{A}\Phi_n$ haben, dann gilt $q \models \mathsf{A}\Phi$ gdw. $q_i \models \mathsf{A}\Phi_i$ für alle i.
- Entsteht nach Anwendung einer Regel auf γ das Teilziel **tt**, dann gilt $q \models \mathsf{A}\Phi$.

$$R^{\models}: \frac{q \vdash \mathsf{A}(\Phi, \mu^{vl})}{\mathbf{tt}}, \text{ wenn } q, vl \models \mu, \qquad R^{\not\models}: \frac{q \vdash \mathsf{A}(\Phi, \mu^{vl})}{q \vdash \mathsf{A}\Phi}, \text{ wenn } q, vl \not\models \mu.$$

$$R^{\vee}: \frac{q \vdash \mathsf{A}(\Phi, (\varphi \vee \theta)^{vl})}{q \vdash \mathsf{A}(\Phi, \varphi^{vl}, \theta^{vl})} \qquad R^{\wedge}: \frac{q \vdash \mathsf{A}(\Phi, (\varphi \wedge \theta)^{vl})}{q \vdash \mathsf{A}(\Phi, \varphi^{vl}) \qquad q \vdash \mathsf{A}(\Phi, \theta^{vl})}$$

$$R^{\mathsf{U}}: \frac{q \vdash \mathsf{A}(\Phi, (\varphi \,\mathsf{U}\, \theta)^{vl})}{q \vdash \mathsf{A}(\Phi, \varphi^{vl}, \theta^{vl}) \qquad q \vdash \mathsf{A}(\Phi, \theta^{vl}, (\mathsf{X}(\varphi \,\mathsf{U}\, \theta))^{vl})}$$

$$R^{\mathsf{R}}: \frac{q \vdash \mathsf{A}(\Phi, (\varphi \,\mathsf{R}\, \theta)^{vl})}{q \vdash \mathsf{A}(\Phi, \theta^{vl}) \qquad q \vdash \mathsf{A}(\Phi, \varphi^{vl}, (\mathsf{X}(\varphi \,\mathsf{R}\, \theta))^{vl})}$$

$$R^{\mathsf{X}}: \frac{q \vdash \mathsf{A}((\mathsf{X}\varphi_1)^{vl_1}, \ldots, (\mathsf{X}\varphi_n)^{vl_n})}{q_1 \vdash \mathsf{A}(\varphi_1^{kf_{q \triangleright q_1} \circ vl_1}, \ldots, \varphi_n^{kf_{q \triangleright q_1} \circ vl_n}) \quad \ldots \quad q_m \vdash \mathsf{A}(\varphi_1^{kf_{q \triangleright q_m} \circ vl_1}, \ldots, \varphi_n^{kf_{q \triangleright q_m} \circ vl_n})}$$

wobei $\{q_1, \ldots, q_m\} = \{q' \mid q \triangleright q'\}$ alle Nachfolgezustände von q sind.

$$R^{\forall x}: \frac{q \vdash \mathsf{A}(\Phi, (\forall x.\varphi)^{vl})}{q \vdash \mathsf{A}(\Phi, \varphi^{vl[x \mapsto v_1]}) \quad \ldots \quad q \vdash \mathsf{A}(\Phi, \varphi^{vl[x \mapsto v_m]})} \qquad \text{für jedes } x \in LV$$

wobei $\{v_1, \ldots, v_m\} = V_q$ die Objekte im Zustand q sind.

$$R^{\exists x}: \frac{q \vdash \mathsf{A}(\Phi, (\exists x.\varphi)^{vl})}{q \vdash \mathsf{A}(\Phi, \varphi^{vl[x \mapsto v_1]}, \ldots, \varphi^{vl[x \mapsto v_m]})} \qquad \text{für jedes } x \in LV$$

wobei $\{v_1, \ldots, v_m\} = V_q$ die Objekte im Zustand q sind.

Abbildung 3.1: Beweisregeln für Q*LTL

- Kann keine Regel mehr auf γ angewendet werden (weil $\Phi = \emptyset$), dann $q \not\models \mathsf{A}\Phi$.

Beweis: (vgl. [2]) Folgt direkt aus der Semantik von Q*CTL* und der Pfadcharakterisierung von R und U (siehe auch Definition 34). Außerdem sind die verwendeten Variablenbelegungen bei $\vdash$ und $\models$ gleich. □

Weiter setzen wir Beweis(-teil-)ziele miteinander in Beziehung und erhalten einen Graph:

Definition 55 (Beweisstruktur)
Sei $\Gamma' = \Gamma \cup \{\mathbf{tt}\}$, $V \subseteq \Gamma'$, $E \subseteq V \times V$ und $\gamma \in V$.
Dann ist (V, E) eine *Beweisstruktur* für γ ein maximaler Graph, bei dem für jedes $\delta \in V$ gilt: δ ist von γ aus erreichbar und die Menge $\{\delta' \mid (\delta, \delta') \in E\}$ ist das Ergebnis der Anwendung einer Beweisregel auf δ.

Anschaulich besteht nun ein Beweis darin, zu zeigen, dass alle Pfade in diesem Graphen zu **tt** führen. Durch die oder-Semantik der Formelmenge wird nämlich $\mathsf{A}\Phi$ erst erreicht, wenn bereit alle Alternativen abgearbeitet sind.

Da es aber **tt** nicht mehr weiter geht, wäre nur endliche Pfade beweisbar, wir sagen auch: „erfolgreich“. Allerdings können in (V, E) auch unendliche Pfade entstehen, nämlich durch R und U (und nicht anders!). Laut Semantik ist aber alleine R erfüllbar; daher sollte auch nur das erfolgreich sein.

Wir fassen das eben gesagte in eine Definition darüber, wann eine Beweisstruktur ein Beweis für eine Formel ist:

Definition 56 (Blatt, (partiell) erfolgreich)
Sei (V, E) eine Beweisstruktur.

- $\gamma \in V$ ist ein *Blatt*, wenn es dafür keinen Nachfolger gibt, also kein $\gamma' \in V$, so dass $(\gamma, \gamma') \in E$.
  Ein Blatt nennen wir *erfolgreich*, wenn $\gamma \equiv \mathbf{tt}$.

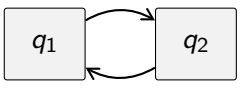
- Ein unendlicher Pfad $\pi = \gamma_1\gamma_2 \ldots$ in (V, E) heißt *erfolgreich*, wenn für ein der Forderungen, etwa γ_i gilt, dass $(\varphi \,\mathsf{R}\, \theta)^{vl} \in \gamma_i$ und für alle $j \geq i$ ist $\theta^{kf_{\pi[i\ldots j]} \circ vl} \notin \gamma_j$. Hier basiert $kf_{\pi[i..j]}$ natürlich auf $kf_{q_{\gamma_i} \triangleright q_{\gamma_{i+1}}}$, etc, wobei $kv_{q \triangleright q} = \mathrm{id}_{V_q}$, wenn $q = q_{\gamma_i} = q_{\gamma_{i+1}}$.
  Auf dem Pfad wird also bei der Anwendung von Regel R^{R} immer wieder versucht, die rechte Forderung (das Teilziel, das wir bei der Definition von Regel R^{R} rechts geschrieben haben) zu beweisen. Diese Forderung muss aber durch X und damit R^{X} irgendwann wieder zu R^{R} kommen (wenn nicht die linke Seite irgendwann **tt** liefert).
  Würde die linke Forderung irgendwann wieder vorkommen, dann wäre der Pfad nicht wegen diesem R unendlich lange – und sollte daher auch nicht wegen diesem R erfolgreich heißen.
  Diese Definition ist für den Aufbau des Algorithmus entscheidend. Und hier ist auch schon zu ahnen, dass das Einführen von $\exists$ und $\forall$ in die Logik keine *direkten* Auswirkungen auf den Algorithmus haben wird. Die dafür notwendigen Variablenbelegungen und insbesondere der Korrespondenzfunktion machen weitaus größere Probleme, da R^{R} – wegen der Korrespondenzfunktion! – nicht immer mit der selben Variablenbelegung auf dem Pfad vorkommen muss.

- (V, E) heißt *partiell erfolgreich*, wenn jedes Blatt erfolgreich ist;

- (V, E) heißt *erfolgreich*, wenn es sowohl partiell erfolgreich ist, als auch jeder unendliche Pfad in (V, E) erfolgreich ist.

Um die nachfolgenden Beweise nicht ohne Anschauungsmaterial zu lassen, geben wir nun zwei Beispiele dafür, wie Beweisstrukturen praktisch aussehen:

Beispiel 11
Gegeben sei die Formel $\psi = \mathsf{A}((cur \neq last)\,\mathsf{U}\,(head = last))$*, sowie ein Zustandsraum, der nur aus zwei Zuständen,* q_1 *und* q_2 *besteht (wir gehen zunächst davon aus, das weder* $cur \neq last$ *noch* $head = last$ *in einem der Zustände gilt):*



Dadurch entsteht die in Abbildung 3.2 gezeigte Beweisstruktur, wobei $vl' = kf_{q_1 \triangleright q_2} \circ vl$ *und* $vl'' = kf_{q_2 \triangleright q_1} \circ vl'$ *verwendet werden. Die gestrichelten Linien deuten an, welche Übergänge entstehen würden, wenn eine der Propositionen zu* **tt** *statt zu* **ff** *ausgewertet würde, also* $R^{\models}$

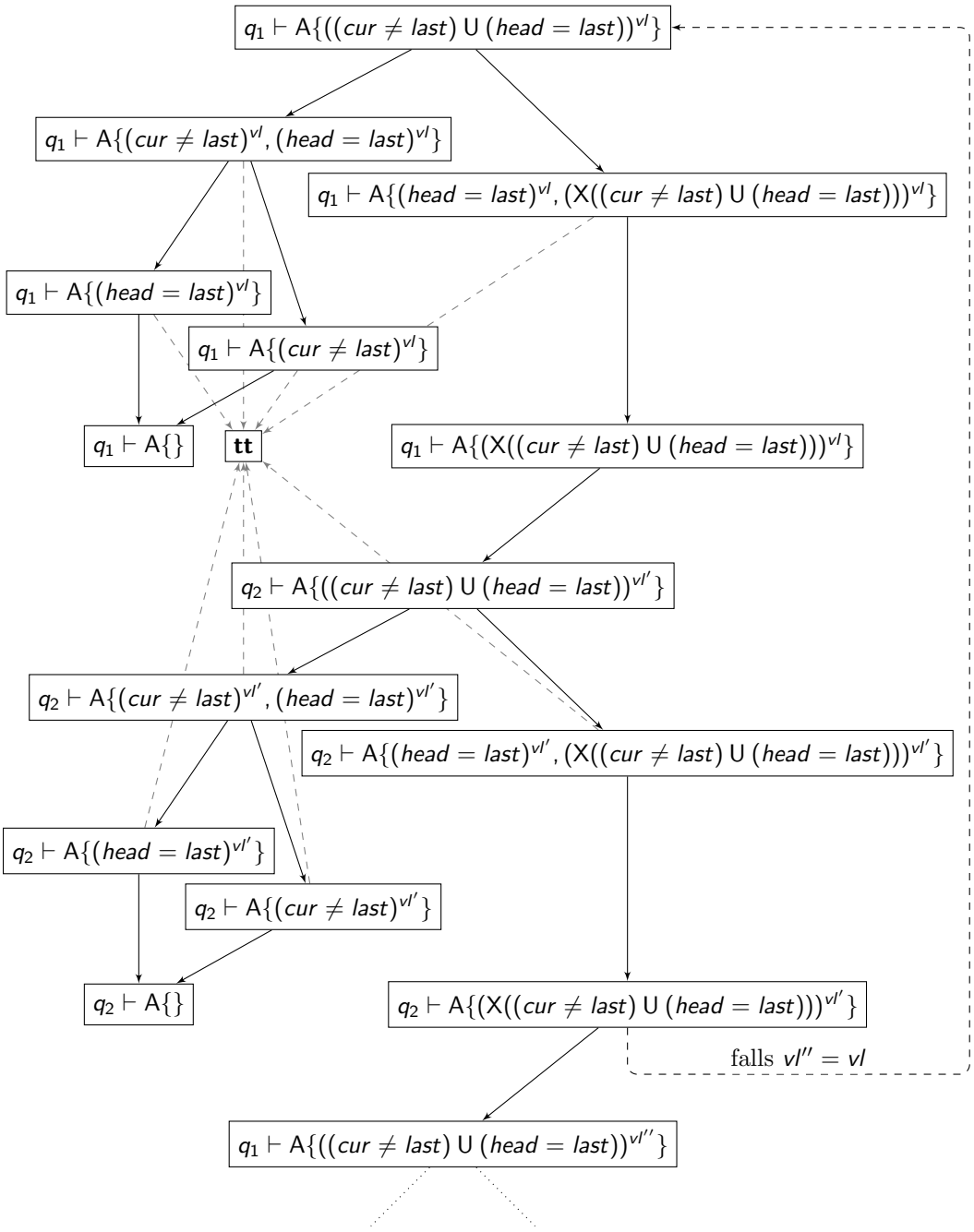


Abbildung 3.2: Beweisstruktur für Beispiel 11.

statt $R^{\models}$ angewendet werden müsste.

Abhängig von der Korrespondenzfunktion kommt kann es unterschiedlich lange dauern, bis wieder eine komplette Übereinstimmung der Forderung erreicht wird (links gestrichelt angedeutet), also bis der Zustand (q_1 oder q_2), die Formel und die Variablenbelegung wieder exakt gleich sind. Da wir aber beim Modelchecking nur endliche Zustandsräume mit nur endlich vielen Objekten betrachten, gibt es nur endlich viele Variablenbelegungen; daher entsteht nach

endlich vielen Schritten der gezeigte Zyklus. Wegen U *ist er auch nicht erfolgreich. Wäre aber eine der Propositionen* **tt**, *dann würde dieser Teil der Beweisstruktur gar nicht erreichbar sein, also auch nicht aufgebaut werden.*

Beispiel 12
Weiter betrachten wir in Abbildung 3.3 noch $\varphi = (head = last) \vee \exists x.(head \neq x)$ *für nur einen Zustand* q_1*, der zwei Objekte,* v_1 *und* v_2 *enthält. Dabei sei* $vl_1 = vl[x \mapsto v_1]$ *und* $vl_2 = vl[x \mapsto v_2]$*. Offenbar entsteht so kein unendlicher Pfad.*

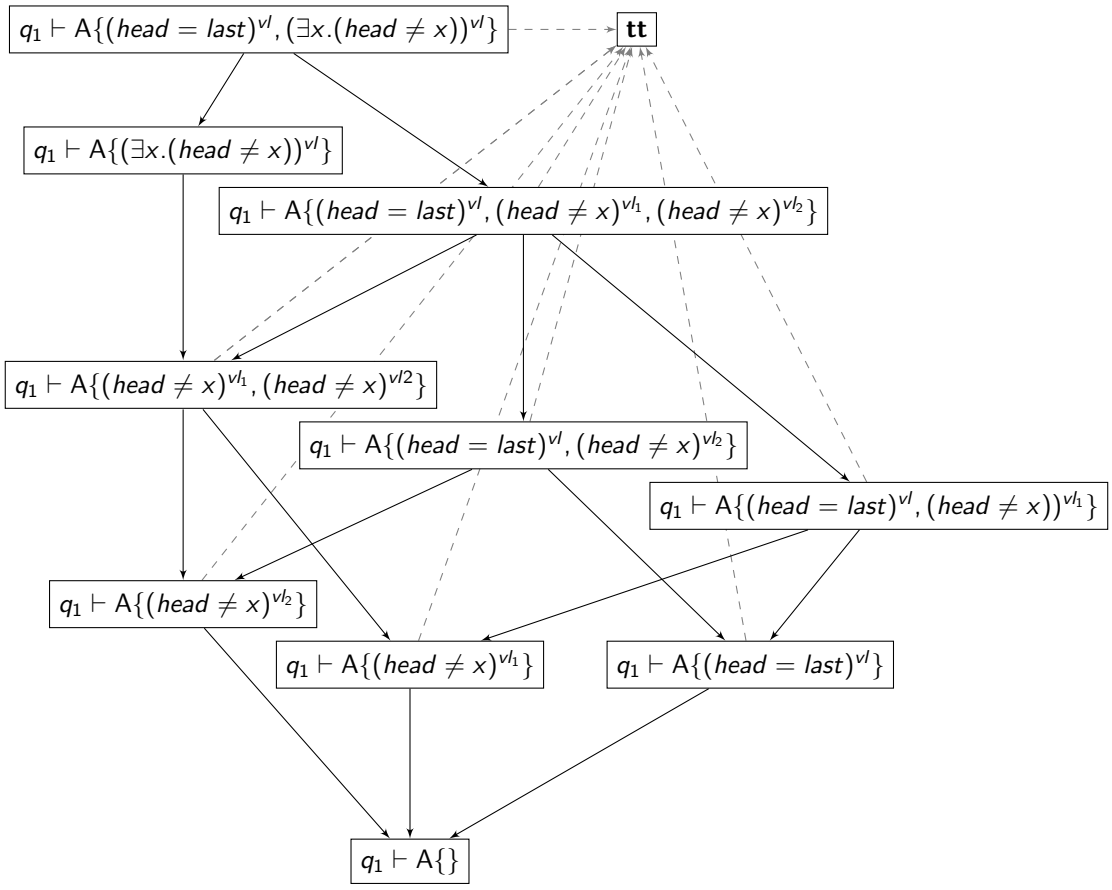


Abbildung 3.3: Beweisstruktur für Beispiel 12.

Ein wesentliche Ziel ist immer noch auszusagen, dass $q \vdash A\varphi^{vl}$ genau dann erfolgreich ist, wenn $q, vl \models A\varphi$.

Satz 57
Gegeben ein Zustandsraum K mit Zuständen Q. Sei $q \in Q$, vl eine Variablenbelegung und φ eine Q*LTL-Formel. Weiter sei (V, E) eine Beweisstruktur für $q \vdash A\{\varphi^{vl}\}$.
Dann gilt $q, vl \models A\varphi$ gdw. (V, E) erfolgreich ist.

Beweis: Um diesen Satz zu beweisen, müssen wir im Anschluss zunächst noch einige Hilfsmittel einführen. Die beiden Richtungen werden dann in Lemma 61 und Lemma 63 getrennt gezeigt werden. □

Zunächst noch eine nützliche Folgerung aus diesem Satz:

Korollar 58
Gibt es für γ eine erfolgreiche Beweisstruktur, dann muss auch jede andere Beweisstruktur für γ erfolgreich sein.

Dieses Resultat ist insofern von Bedeutung, dass bei der Suche nach einer erfolgreichen Beweisstruktur für eine bestimmte Forderung kein Backtracking notwendig ist.

Zum Beweis des Satzes verwenden wir folgende Definitionen:

Definition 59
Sei (V, E) eine Beweisstruktur für $q \vdash \mathsf{A}\Phi$. Außerdem sei $\pi = \alpha_1\alpha_2 \ldots$ ein (endlicher oder unendlicher) Pfad in (V, E).

- Wir sagen: Die Beweisregel R *kommt an Stelle* i *in* π *vor*, wenn durch Anwendung von Regel R die Menge $M = \{\gamma \mid (\pi[i], \gamma) \in E\}$ entsteht und $\pi[i+1] \in M$ ist.

- Die Folge der von π besuchten Zustände $P(\pi) = p_1 p_2 \ldots \in Q^*$ entsteht aus π, indem man zunächst die Folge σ definiert als:
  $$\sigma[i] = \begin{cases} q_{\pi[1]} & \text{für } i = 1, \\ q_{\pi[i]} & \text{wenn } i > 1 \text{ und an Stelle } i-1 \text{ kommt } R^{\mathsf{X}} \text{ vor}, \\ \bullet & \text{sonst.} \end{cases}$$
  Daraus ergibt sich dann $P(\pi)$ durch Entfernen aller Vorkommen von $\bullet$.
  Immer wenn sich der Zustand q geändert hat (was nur durch X, also R^{X} möglich ist) gibt es also einen Eintrag in $P(\pi)$.

Hierfür lassen sich einige Eigenschaften leicht beweisen:

Lemma 60
Sei (V, E) eine Beweisstruktur für $q \vdash \mathsf{A}\Phi$ und π ein Pfad in (V, E). Es gilt:

- $P(\pi)$ ist ein Pfad im Zustandsraum K.

- π ist genau dann unendlich lang, wenn R^{X} an *unendlich* vielen Stellen in π vorkommt.

- $P(\pi)$ ist unendlich gdw. π unendlich ist.

- Sei $1 \leq i \leq j$ und für alle k mit $i \leq k \leq j$ komme R^{X} nicht an Stelle k vor. Dann ist $P(\pi[i \ldots]) = P(\pi[j \ldots])$, d. h. wir befinden uns noch im selben Zustand ($q_{\pi[i]} = q_{\pi[j]}$).

Wir beweisen nun eine Richtung von Satz 57:

Lemma 61
Sei $q \in Q$ und $\Phi = \{\varphi_1^{vl_1}, \ldots, \varphi_n^{vl_n}\}$ eine Menge von Q*LTL-Formeln mit Variablenbelegungen $vl_1, \ldots, vl_n$, so dass $q \models \mathsf{A}\Phi$. Außerdem sei (V, E) eine Beweisstruktur für $\gamma \equiv q \vdash \mathsf{A}\Phi$.
Dann ist (V, E) erfolgreich.

Beweis: (vgl. [2]) Angenommen $G = (V, E)$ ist nicht erfolgreich. Dann ist

- entweder G nicht partiell erfolgreich, worauf aus Lemma 54 unmittelbar folgt, dass $q \not\models \mathsf{A}\Phi$, da es einen verletzenden endlichen Pfad gibt, der in dem Blatt endet,

- oder G ist nicht erfolgreich, womit es einen unendlichen Pfad π in G mit $\pi[1] = \gamma$ gibt, der nicht erfolgreich ist. Wir konstruieren nun einen bei q beginnenden Pfad in Q, der für alle $\varphi^{vl} \in \Phi$ nicht erfüllt ist, womit $q, vl \not\models \varphi$.

Wir beweisen dazu per Induktion über die Formelgröße von φ: Wenn $\varphi^{vl} \in \pi[i]$ für $i \leq 1$ ist, dann gilt: $P(\pi[i\,...]), vl \not\models \varphi$. Das heißt, dass gerade $P(\pi)$ ein nicht erfüllender Pfad ist.

Wir beginnen damit, die Aussage für die kleinsten in G vorkommenden Teilformeln, nämlich $\varphi \in \mathsf{EP}$, zu zeigen:
Induktionsanfang: Gegeben ist i mit $\varphi^{vl} \in \pi[i]$, wobei $\varphi \in \mathsf{EP}$ und $vl \in Vl_{\pi[i]}$.
Zu zeigen ist: $P(\pi[i\,...]), vl \not\models \varphi$.
Da π unendlich lang ist, kommt wegen Lemma 60 Regel R^{X} an einer Stelle $j \geq i$ wieder vor. Weil aber φ nicht von der Form $\mathsf{X}\varphi'$ ist (sondern $\in \mathsf{EP}$), folgt aus dem Aufbau von R^{X} (alle in Formeln müssen mit X beginnen), dass $i \neq j$.
Es ist also (für ein bestimmtes k) $\varphi^{vl} \in \pi[l]$ falls $i \leq l \leq k$ und $\varphi^{vl} \notin \pi[l]$ falls $k < l \leq j$; auch die Variablenbelegung kann sich mangels $\forall$, bzw. $\exists$ in EP erst bei X wieder ändern (durch kf).
Um aber φ entfernen zu können, musste zwischen i und j irgendwann $R^{\models}$ oder $R^{\not\models}$ angewendet worden sein. Weil π unendlich ist, kann es jedoch nicht $R^{\models}$ gewesen sein (das mit **tt** den Pfad terminieren würde). Damit komme $R^{\not\models}$ an Stelle k, $i \leq k < j$, vor, wofür $q_{\pi[k]}, vl \not\models \varphi$ gelten muss.
Wie in Lemma 60 festgestellt, folgt außerdem $q_{\pi[k]} = q_{\pi[i]} = P(\pi[i\,...])$. Daher gilt $P(\pi[i\,...]), vl \not\models \varphi$.
Induktionsschritt: Sei also für die Teilformeln φ' von φ bereits gezeigt: Ist $\varphi'^{vl'} \in \pi[i]$ für $i \leq 1$, dann gilt $P(\pi[i\,...]), vl' \not\models \varphi'$.
Gegeben ist daher i mit $\varphi^{vl} \in \pi[i]$ und $vl \in Vl_{\pi[i]}$, wobei φ die im Folgenden angegebene Form hat. Dabei gehen wir hier nur auf die interessanten Fälle näher ein, da alle anderen analog folgen (vgl. [2]). Zu zeigen ist $P(\pi[i\,...]), vl \not\models \varphi$:

$(\varphi_1 \wedge \varphi_2)^{vl}$**:** Wie schon beim Induktionsanfang folgt die Existenz von $i \leq k < j$, so dass $R^{\wedge}$ an Stelle k vorkommt und R^{X} an Stelle j.

Also muss (nach der Anwendung von $R^{\wedge}$) $\pi[k+1] \equiv q \vdash \mathsf{A}\Phi$ sein, wobei entweder $\varphi_1^{vl} \in \Phi$, oder $\varphi_2^{vl} \in \Phi$, also auf dem Pfad entweder das linke oder das rechte Teilziel von $R^{\wedge}$ auftritt, und wieder nur $q = q_{\pi[k+1]} = q_{\pi[i]} = P(\pi[i\,...])$ möglich ist. Sei also o. B. d. A. $\varphi_1^{vl} \in \Phi$.

Mit der Induktionsannahme $P(\pi[i\,...]), vl \not\models \varphi_1$ folgt wegen der Semantik von $\wedge$ sofort $P(\pi[i\,...]), vl \not\models \varphi_1 \wedge \varphi_2$, d. h. $P(\pi[i\,...]), vl \not\models \varphi$.

$(\varphi_1 \,\mathsf{U}\, \varphi_2)^{vl}$**:** Hier gibt es zwei Fälle zu betrachten: Entweder nutzt der unendliche Pfad jedesmal die rechte Seite von R^{U}, so dass es kein k gibt, so dass $P(\pi[i\,...])[k\,...], (kf_{\pi[i\,...\,k]} \circ vl) \models \varphi_1 \,\mathsf{U}\, \varphi_2$. Mit der Semantik von U hat das zur Folge, dass $P(\pi[i\,...]), vl \not\models \varphi_1 \,\mathsf{U}\, \varphi_2$.

Oder aber es gibt ein kleinstes $j \geq i$, so dass $\{\varphi_1^{(}vl'), \varphi_2^{(}vl')\} \subseteq \pi[j]$. Aus der Induktionsannahme wissen wir, dass sowohl $P(\pi[j\,...]), vl' \not\models \varphi_1$, als auch $P(\pi[j\,...]), vl' \not\models \varphi_2$, wobei $vl' = kf_{\pi[i\,...\,j]} \circ vl$.

Dann betrachte die Positionen, an denen von R^{X} vorkommt:
$i = k_1 < k_2 < ... < k_m < k_{m+1} = j$.

Für jedes k_n, und damit $vl'' = kf_{\pi[i...k_n]}$ (das sich bis einschließlich k_{n+1} nicht ändern kann), muss zwischen k_n und k_{n+1} irgendwann R^{U} auftreten, etwa bei l_n, wodurch $\{\varphi_2^{vl''}, \mathsf{X}(\varphi_1 \,\mathsf{U}\, \varphi_2)^{vl''}\} \subseteq \pi[l_n + 1]$, da $\{\varphi_1^{vl''}, \varphi_2^{vl''}\} \not\subseteq \pi[l_n + 1]$ ($\subseteq$ erst bei k_{m+1}) ist, also die rechte Seite von R^{U} hier auf π verwendet wird. Mit der Induktionsannahme gilt dann: $P(\pi[l_n + 1]), vl'' \not\models \varphi_2$.

Für die Folge $P(\pi[i\,...])$ gibt es dann ein r mit $P(\pi[i\,...])[r\,...] = P(\pi[j\,...])$, so dass $P(\pi[i\,...])[r\,...], (kf_{\pi[i...j]} \circ vl) \not\models \varphi_2$ und für alle $1 \leq q \leq r$ gilt $P(\pi[i\,...])[q\,...], (kf_{...} \circ vl) \not\models \varphi_1$.

Aus der Semantik von U folgt dann die Behauptung.

$(\varphi_1 \,\mathsf{R}\, \varphi_2)^{vl}$**:** Da π nicht erfolgreich ist, jedoch unendliche Pfade über R erfolgreich wären, muss in R^{R} irgendwann der linke Zweig gewählt worden sein, also ein $j \geq i$ existieren mit $\varphi_2 \in \pi[j]$.

Entweder ist bereits $q_{\pi[j]} = q_{\pi[i]}$ und aus der Induktionsannahme $P(\pi[i\,...]), vl \not\models \varphi_2$ folgt aus der Semantik von R auch $\not\models \varphi_1 \,\mathsf{R}\, \varphi_2$, wobei sich vl auch hier vor j nicht geändert haben kann, oder wir müssen dazu für alle $i \leq k < j$ zeigen, dass R nicht durch φ_1 „beendet“ wurde: $P(\pi[k\,...]), (kf_{\pi[i...k]} \circ vl) \not\models \varphi_1$.

Wie schon bei $\wedge$ und U lässt sich das, für alle solche k auf die Induktionsannahme $P(\pi[k\,...])$ und R^{R} zurückführen, da sich der Zustand nur bei R^{X} ändern kann (und die Variablenbelegung nur bei X, $\forall$ oder $\exists$).

$(\forall x.\varphi_1)^{vl}$ für $x \in LV$**:** Es gibt also $i \leq k < j$, so dass $R^{\forall x}$ an Stelle k in π vorkommt, R^{X} an Stelle j und vl sich nicht geändert hat (d. h. wir betrachten das erste Vorkommen von $R^{\forall x}$).

Also muss $\pi[k + 1] \equiv q \vdash \mathsf{A}\Phi$ sein und wegen dem Aufbau von $R^{\forall x}$ ein $\varphi_1^{vl'} \in \Phi$ vorkommen, also mit unverändertem φ_1, aber einer geänderten Variablenbelegung vl', die sich aber aus vl auf die selbe Weise ergibt, wie in der Semantik von Q*CTL*.

Wieder ist $q = q_{\pi[k+1]} = q_{\pi[i]} = P(\pi[i\,...])$ und damit folgt wegen der Induktionsannahme $q, vl' \not\models \varphi_1$ für $vl_1 = vl'$ und der Semantik von $\forall$ sofort $q, vl \not\models \forall x.\varphi_1$, da φ_1 unter vl' nicht erfüllt ist, wenn also insbesondere x mit $vl'(x)$ belegt wird.

$(\exists x.\varphi_1)^{vl}$ für $x \in LV$**:** Es gibt also $i \leq k < j$, so dass $R^{\exists x}$ an Stelle k in π vorkommt und vl sich nicht geändert hat.

Somit ist $\pi[k + 1] \equiv q \vdash \mathsf{A}(\Phi, \varphi_1^{vl_1}, ... \varphi_1^{vl_m})$ und wieder $q = q_{\pi[k+1]} = q_{\pi[i]} = P(\pi[i\,...])$.

Wegen der Induktionsannahme wissen wir, dass $q, vl' \not\models \varphi_1$ für alle $vl' \in \{vl_1, \dots, vl_m\}$. Da es sich aber um die selben $vl_1, \dots, vl_m$ handelt, wie sie auch in der Semantik von $\exists$ verwendet werden, um ausgehend von vl der Logikvariable x alle möglichen Belegungen zuzuweisen, folgt $q, vl \not\models \exists x.\varphi_1$ und damit $P[i\,...], vl \not\models \exists x.\varphi_1$. □

Im Beweis der Gegenrichtung von Satz 57 verwenden wir den Begriff des gescheiterten Pfades als Gegenstück zum erfolgreichen Pfad:

Definition 62 (Gescheiterter Pfad)
Ein Pfad π in einer gegebenen Beweisstruktur $G = (V, E)$ heißt *gescheitert*, wenn π entweder endlich ist und dabei die letzte Forderung in π ein nicht erfolgreiches Blatt in G ist, oder aber unendlich ist und nicht erfolgreich.

Eine Beweisstruktur ist also genau dann nicht erfolgreich, wenn sie einen gescheiterten Pfad enthält.

Lemma 63
Sei $q \in Q, \Phi = \{\varphi_1^{vl_1}, \ldots, \varphi_n^{vl_n}\}$ eine Menge von Q*LTL-Formeln mit Variablenbelegungen $vl_1, \ldots, vl_n$ und (V, E) eine erfolgreiche Beweisstruktur für $\gamma \equiv q \vdash \mathsf{A}\Phi$.
Dann gilt: $q \models \mathsf{A}\Phi$.

Beweis: (vgl. [2], hier nur die Idee) Angenommen $q \not\models \mathsf{A}\Phi$. Dann gibt es einen Berechnungspfad τ in K, so dass $q, vl \not\models \mathsf{A}\varphi$ für jedes $\varphi_i^{vl_i} \in \Phi$. Außerdem kennen wir für jeden Besuch eines $\forall$ die nötige Variablenumbelegung, mit der φ auf τ nicht erfüllt wird.

Rekursiv lässt sich aus diesem Berechnungspfad τ ein (endlicher oder unendlicher) Pfad π in (V, E) konstruieren, der zeigt, dass (V, E) nicht erfolgreich sein kann, da π scheitert. Dazu dient wieder $P(\pi)$, das entweder (für endliches π) ein Präfix von τ sein wird, oder aber ganz $P(\pi) = \tau$. Die Details sind in [2] nachzulesen. Eine Erweiterung um vl stellt an keiner Stelle ein Problem dar.

Neu hinzu kommen die Fälle $\forall x$ und $\exists x$ für $x \in LV$, wobei sich bei $\forall$ jeweils die gegebenen Variablenumbelegungen nutzen lassen, um den passenden Nachfolgeknoten in (V, E) auszuwählen (vgl. $R^{\forall}$).

Bei $\exists$ besteht dieser Bedarf, wie aus $R^{\exists}$ ersichtlich, nicht. Wegen $q, vl \not\models \mathsf{A}\varphi$ kann aber auch keine Umbelegung erfolgreich sein.

An welchen Stellen umbelegt werden muss, ist durch die Formel bereits festgelegt. Wegen der Korrespondenz zwischen dem Aufbau der Beweisregeln und der Semantik von Q*CTL* muss diese Position mit dem Vorkommen von insb. $R^{\forall}$ zusammenfallen. Somit lässt sich formal in der Semantik bei Besuchen von $\forall x$ eine Folge $\nu_1 \nu_2 \ldots \in Vl^*$ konstruieren, die auch π scheitern lässt. □

Damit wäre Satz 57 vollständig bewiesen.

Ein weiterer Schritt in Richtung Algorithmus ist die Feststellung, dass zum Entscheiden, ob eine Beweisstruktur erfolgreich ist (also insb. ob *jeder* unendliche Pfad erfolgreich ist), unser Blick auf die starken Zusammenhangskomponenten fallen sollte:

Definition 64 (Nichttriviale starke Zusammenhangskomponente)
Sei $G = (V, E)$ ein Graph. Eine *starke Zusammenhangskomponente (SCC)* ist eine maximale Teilmenge $S \subseteq V$, so dass es für alle $v, w \in S$ in G einen Pfad von v nach w gibt (auch v ist ein solcher Pfad für $S = \{v\}$!).
Sie heißt *nichttrivial*, wenn sie mindestens eine Kante enthält, d. h. es gibt (nicht notwendigerweise disjunkte) $v, w \in S$ mit $(v, w) \in E$.

Für den nächsten Satz, der den Erfolg einer Beweisstruktur an den SCCs fest macht, definieren wir uns eine Menge, die für starke Zusammenhangskomponenten der Beweisstruktur die dort erfolgreichen Formeln liefert:

Definition 65
Sei (V, E) eine Beweisstruktur. Für $S \subseteq V$ definieren ist die Menge:

$$\begin{aligned} Success(S) = \{(\varphi \,\mathsf{R}\, \theta)^{vl} \mid \exists \gamma \in S \text{ mit } (\varphi \,\mathsf{R}\, \theta)^{vl} \in \gamma \text{ und} \\ \text{für alle bei } \gamma \text{ startenden Pfade } \pi \text{ in } S \text{ ist } \theta^{kf_{\pi[1\ldots i]} \circ vl} \notin \pi[i]\} \end{aligned}$$

Wir nennen S *erfolgreich*, wenn $Success(S) \neq \emptyset$.

In [2] fällt diese Definition etwas einfacher aus; müssten wir kf nicht berücksichtigen, so wäre auf allen Pfaden zu fordern, dass $\theta \notin \pi$. Dann muss aber schon gar nicht mehr auf Pfade zurückgegriffen werden: Es genügt zu überprüfen, ob $\theta \notin \gamma$ für alle $\gamma \in S$.

Der Beweis des folgenden Satzes liefert wichtige Erkenntnisse für den Aufbau des Algorithmus.

Satz 66
Eine partiell erfolgreiche Beweisstruktur (V, E) ist genau dann erfolgreich, wenn jede nichttriviale starke Zusammenhangskomponente $S \subseteq V$ erfolgreich ist, d. h. $\mathit{Success}(S) \neq \emptyset$.

Beweis: (vgl. [2]) Zunächst zeigen wir, dass aus einer erfolgreichen Beweisstruktur folgt, dass $\mathit{Success}(S) \neq \emptyset$ ist.
Nehmen wir also an, dass $\mathit{Success}(S) = \emptyset$. Wir konstruieren nun einen (unendlichen) nicht erfolgreichen Pfad in S und beginnen bei einem beliebigen $\gamma \in S$. Weil $\mathit{Success}(S) = \emptyset$ ist, gibt es entweder kein φ, θ und vl, so dass $(\varphi \,\mathsf{R}\, \theta)^{vl} \in \gamma$ (aber ohne R kann kein unendlicher Pfad erfolgreich sein), oder es gibt ein $\gamma' \in S$, das von γ über einen Pfad π erreichbar ist, so dass $(\theta)^{kf_{\pi[1\ldots|\pi|]} \circ vl} \in \gamma'$ ist. Dann erweitern wir unseren Pfad um π, also den Weg von γ nach γ'. Jeder Nachfolgezustand von γ' muss aber, um erfolgreich zu werden, wieder ein R treffen. Allerdings lässt sich auch von dort wieder ein γ' erreichen, so dass wir auf dem Pfad wieder ein $\theta^{(vl')}$ einfügen, der das getroffene R „entwertet“. Also ist der konstruierte Pfad nicht erfolgreich.
Für die Gegenrichtung zeigen wir, dass aus $\mathit{Success}(S) \neq \emptyset$ folgt, dass (V, E) erfolgreich ist.
Wir betrachten einen beliebigen unendlichen Pfad π in S. Da $\mathit{Success}(S) \neq \emptyset$ ist, gibt es $\gamma \in S$, sowie φ, θ und vl, so dass $(\varphi \,\mathsf{R}\, \theta)^{vl} \in \gamma$ ist und für jedes von dort erreichbare γ' ist $\theta^{vl'} \notin \gamma'$, wobei vl' die über den jeweils zum Erreichen verwendeten Pfad (durch kf) entstandene Variablenbelegung ist.
Es bleibt zu zeigen, dass π auch „dieses“ R trifft. Dazu stellen wir fest, dass es (weil es eine starke Zusammenhangskomponente ist) einen Pfad von γ zu einem $\pi[i]$ gibt, für ein $i \geq 1$. Weil $\theta^{vl'}$ auf diesm Pfad nicht auftreten kann, muss wegen dem Aufbau der Regel R^{R} auf diesem Pfad an jeder Stelle entweder $(\mathsf{X}(\varphi \,\mathsf{R}\, \theta))^{vl'}$ (mit entsprechendem vl') oder $(\varphi \,\mathsf{R}\, \theta)^{vl'}$ vorkommen; insbesondere ist also $(\varphi \,\mathsf{R}\, \theta)^{vl'} \in \pi[i]$ oder $(\mathsf{X}(\varphi \,\mathsf{R}\, \theta))^{vl'} \in \pi[i]$.
Im ersten Fall sind wir fertig, da auch auf dem weiteren Pfad $\pi[i...]$ kein $\theta^{vl''}$ bezüglich der bei $\pi[i]$ geltenden Variablenbelegung vl' von $\varphi \mathsf{R} \theta$ auftreten kann, weil vl' von der bei γ geltenden und als gut befundenen Belegung vl herkommt.
Im zweiten Fall wissen wir, da es sich um einen unendliche Pfad handelt, wegen Lemma 60, dass R^{X} an einer Stelle $j \geq i$ vorkommt. Durch die Anwendung von R^{X} ist dann aber $(\varphi \,\mathsf{R}\, \theta)^{vl''} \in \pi[j+1]$. Wieder kann kein schädliches $\theta^{vl'''}$ mehr später auftreten, da die Variablenbelegung sich nur durch kf geändert hat, aber bereits für jeden Pfad von γ aus – insbesondere also den, der zu $\pi[i]$ geht und dann mit $\pi[i...]$ identisch ist – $\theta^{vl'''}$ für genau diese über kf angepasste Variablenbelegung vl''' nicht enthalten ist. □

3.2 Algorithmus

3.2.1 Idee

Der wesentliche Teil des Modelcheckingalgorithmus besteht daraus, starke Zusammenhangskomponenten zu finden und nachzuweisen, dass $\mathit{Success}(S) \neq \emptyset$ ist. Natürlich werden endliche Pfade, die also in **tt** oder $\mathsf{A}\{\}$ enden, auch behandelt; diese Tests sind aber geradezu trivial.

Für starke Zusammenhangskomponenten gibt es die als Algorithmus von Tarjan[12] bekannte Modifikation der Tiefensuche. Diese wird hier so erweitert, dass $\mathit{Success}(S)$ schon während der Tiefensuche erstellt wird. Die Beweisstruktur wird dabei erst durch die Tiefensuche aufgebaut; dadurch muss aber auch der Zustandsraum erst bei Bedarf aufgebaut werden, weil jeder Forderung ein Zustand zugrundeliegt.

Eine Tiefensuche wird nun an irgendeiner Stelle einen „ersten“ Knoten einer starken Zusammenhangskomponente erreichen. Im Beweis des letzten Satzes sehen wir, dass an dieser Stelle entweder $(\mathsf{X}(\varphi \mathsf{R} \theta))^{vI}$ oder $(\varphi \mathsf{R} \theta)^{vI}$ sein kann – und für eine erfolgreiche SCC sogar muss! Daher können wir an dieser Stelle bereits auswerten, welche „$(\varphi \mathsf{R} \theta)^{vI}$“ (egal ob mit oder ohne X) *noch* in dieser SCC erfolgreich sind. Diese Information speichern wir an diesem Knoten als sogenannte `valid`-Menge. Natürlich stellen wir erst später fest, dass es sich um eine SCC handelt, nämlich wenn wir den ersten Zykel finden, sprich: wir wieder auf eine „frühere“, aber noch aktive (d. h. noch im Stack des SCC-Algorithmus befindliche), Forderung treffen. Während wir also den Graph erkunden, führen wir außerdem die Menge der noch erfolgreichen R-Formel mit, werfen aber all die Formeln heraus, für deren mit kf auf den aktuellen Zustand angepasste Variablenbelegung vI' auch $(\theta)^{vI}$ in der Forderung (d. h. im Knoten) enthalten ist.

Treffen wir dann schließlich wieder auf einen Knoten, den wir schon gesehen haben und der noch aktiv ist, dann wissen wir bereits, dass die Variablenbelegung von $(\varphi \mathsf{R} \theta)^{vI}$ bzw. $(\mathsf{X}(\varphi \mathsf{R} \theta))^{vI}$ übereinstimmen muss, sonst wären die Forderungen nicht gleich. Da wir beim ersten Besuch des Knotens aber für den Pfad bis zum Zykelschluss schon sichergestellt haben, dass $(\theta)^{vI}$ für *genau diese* Variablenbelegung nicht vorkommt, genügt es, hier den Schnitt aus der beim ersten Besuch des Knotens gespeicherten `valid`-Menge und der nun akkumulierten zu bilden. Der Schnitt lässt sich aber mit einem Kniff sehr effizient bestimmen: Wir speichern zusätzlich zu den Formeln in der akkumulierten `valid`-Menge die Tiefensuchnummer des Knotens, bei dem wir die Formel das erste Mal gesehen haben („Startpunkt“). Im Schnitt können dann keine Formeln sein, die eine größere Tiefensuchnummer erhalten haben. Außerdem können Formeln mit einer geringeren Tiefensuchnummer nur dadurch in die akkumulierte Menge gekommen sein, dass sie bereits beim ersten Besuch des Knotens darin enthalten waren. Es genügt also, alle Formeln mit zu großer Tiefensuchnummer zu entfernen. Bleiben bei diesem Schnitt keine Formeln mehr übrig, dann kann die SCC nicht erfolgreich sein. Aber auch wenn die Menge nicht leer ist, brauchen wir hier mit der Tiefensuche nicht weiter zu erforschen.

Eine starke Zusammenhangskomponente kann natürlich aus mehr als einem Zykel bestehen. Während wir in der Tiefensuchen zunächst wieder auf dem Rückweg sind, kommen wir zu Knoten, bei denen wir noch nicht alle Möglichkeiten erforscht haben. Aus einer dieser Möglichkeiten kann ein weiterer Zykel als Teil unserer SCC entstehen. Das bereitet aber keine wesentlichen Probleme. Wichtig ist, dass wir auf dem Rückweg eine akkumulierte und dann eingeschränkte `valid`-Menge für die Zwischenknoten (die Knoten also, die, bezogen auf die Tiefensuchnummer, zwischen dem ersten und dem letzten Knoten des Zykelschlusses liegen) übernehmen. Das entspricht gerade dem, was passieren würde, wenn nach dem Zykelschluss weitergelaufen werden würde, bis der Zwischenknoten ein weiteres Mal erreicht wird. Zum Übernehmen müsste dann aber die Korrespondenzfunktion rückwärts ausgewertet werden. Das lässt sich vermeiden, indem von der an diesem Knoten gespeicherten `valid`-Menge ausgegegangen wird und aus dieser die Formeln entfernt werden, die nach Anwendung der Korrespondenzfunktion nicht mehr in der `valid`-Menge des komplett erforschten Nachfolgerknotens vorkommen. Durch den gespeicherten Startpunkt ist es möglich, diese Mengendifferenz relativ effizient zu implementieren.

Im allerersten Schritt Algorithmus von Tarjan zur Bestimmung der SCCs werden die For-

derungen auf einem Stack gelegt, der zusätzlich zum Aufrufstack der Tiefensuche erzeugt wird. Diese Forderungen nehmen wir im letzten Schritt aber erst dann wieder vom Stack, wenn eine SCC komplett erforscht ist. Dazu wird an jedem Knoten zusätzlich die niedrigste in der SCC auftretende Tiefensuchnummer gespeichert, sowie beim Erforschen mitgeführt und entsprechend aktualisiert. Erst wenn wir auf dem Rückweg bei genau diesem Knoten wieder angekommen sind, kann die SCC vom Stack genommen werden. Gibt es keine Zyklen, dann entsprich die niedrigste gesehene Tiefensuchnummer aber gerade unserer eigenen Knotennummer und der Knoten wird nach der vollständigen Erforschung korrekterweise direkt entfernt.

Ob erforschte Teilbäume erfolgreich sind (weil entweder `valid` in einer SCC nicht leer ist, oder **tt** gesehen wurde), oder nicht erfolgreich (weil keine Teilforderungen mehr übrig sind oder `valid` in einer SCC leer war) wird jeweils als Rückgabewert zurückgegeben. Zusätzlich werden aber in der globalen Menge `F` die endgültig gescheiterten Forderungen notiert, um in solchen Fällen schnell abbrechen zu können. Alle Knoten der inkrementiell entstehenden Beweisstruktur (V, E) werden dagegen in der globalen Menge `V` gespeichert. E ist dagegen nur implizit durch die Beweisregeln gegeben; auf eine Ausformulierung der Implementierung dieser in Pseudocode haben wir verzichtet: Die Funktion `Assertion::subgoals()` muss dazu die aus dieser Forderung resultierenden Teilziele bestimmen. Ohne bereits auf die Einzelheiten der Implementierung eingegangen zu sein, soll dennoch gesagt sein: E ist gerade die Menge, für die (`asold`,`asnew`) $\in E$ gdw. `asnew` $\in$ `asold.subgoals()`.

3.2.2 Datenstrukturen

Um schließlich den Algorithmus anzugeben, verwenden wir an C++ angelehnten Pseudocode. Zunächst geben wir die notwendigen Datenstrukturen an; im nächsten Schritt kommen wir dann zur Implementierung der darauf operierenden Methoden. Insbesondere `Set` und `Stack` setzen wir aber als bekannt voraus. Ebenso verzichten wir auf Details, wie etwa die interne Repräsentation von Formeln.

```
1  struct Formula {
2    // Wenn die Formel die Form form hat, z.B. ßub1 U sub2", dann gib true zurück,
3    // sowie die in form benannten Teilformeln über sub1,...
4    bool is(String form,Formula &sub1,...);
5    // Hat die Formel die Form ∀x.φ (bzw. ∃x.φ),
6    // dann gib den Namen der Logikvariable (x) und die Teilformel zurück:
7    bool is_forall(String &name,Formula &sub);
8    bool is_exists(String &name,Formula &sub);
9    // Ist diese Formel bereits ∈ EP?
10   bool is_atomic();
11   // Bestimmt die zu ¬φ äquivalente Formel, bei der
12   // keine Negationen außerhalb von EP mehr auftreten
13   Formula negation();
14
15   // ... interne Repräsentation
16 };

18 struct Object { ... }; // Interne Repräsentation der Objekte
19 struct LVState { // Logikvariablenbelegung
20   Object get(String name);
21   void set(String name,Object obj);
22
23   // ... interne Repräsentation
```

```
24 };

26 struct QFormula { // Formel mit Variablenbelegung
27   Formula phi;
28   LVState vl;      // Belegung der Logikvariablen
29 };

31 struct SCCInfo {
32   QFormula formula;   // Formel der Form (phi1 R phi2)^vl
33   int sp;             // Startpunkt, d.h. Tiefensuchnummer des Knotens,
34                       // an dem die Formel das erste Mal auftritt
35 };

37 struct State {
38   // Die Nachfolger dieses Zustands im Zustandsraum:
39   Set<State> successors();
40   // Gibt den Wahrheitswert von atomaren Teilformeln (-> is_atomic()) zurück).
41   bool models(Formula phi,LVState vl);
42   // Die im aktuellen Zustand enthaltenen Objekte:
43   Set<Object> objects();
44
45   // Um die Korrespondenzfunktion für den Übergang von hier nach to auszuwerten,
46   // gibt Variablenbelegung für Zustand to zurück:
47   LVState update(State to, LVState current);
48
49   // ... interne Repräsentation
50 };

52 struct Assertion { // Eine Forderung
53   State state;
54   Set<QFormula> formulas;
55
56   // Gib die Teilziele zurück, die zu beweisen sind, um diese Forderung zu
       erfüllen.
57   // Dazu werden die Beweisregeln angewendet, wobei die Informationen über
58   // die Nachfolgezustände von state eingehen;
59   // ebenso wird die jeweilige Belegung der Logikvariablen wird hier verwendet,
60   // um zwischen R^⊨ und R^⊭ auszuwählen.
61   // Bei X wird außerdem die Korrespondenzfunktion benötigt.
62   Set<Assertion> subgoals();
63   bool is_true(); // Test, ob diese Forderung zu tt geworden ist.
64
65   // Zusätzliche Felder für den Algorithmus (werden auch dort initialisiert):
66   int dfsn;
67   int low;
68   Set<SCCInfo> valid;
69 };

71 Set<Assertion> V={}; // gesehene Forderungen, am Anfang leer
72 Set<Assertion> F={}; // gescheiterte Forderungen

74 class ModelcheckQsLTL {
75   int dfn = 0; // Tiefensuchnummer, anfangs 0
76
77   // Stack zur Konstruktion der starken Zusammenhangskomponente (SCC):
78   Stack<Assertion> stack={}; // leer
79
80   // Initialisiere die für den Algorithmus verwendeten Felder in Assertion as
```

```
81   private void init(Assertion &as,Set<SCCInfo> valid);
82   private bool dfs(Assertion &as,Set<SCCInfo> valid);
83
84   bool check(Assertion as);
85 };
```

3.2.3 Implementierung

Kommen wir nun zur Implementierung der wesentlichen Methoden:

Wesentlich für den Benutzer ist die Funktion `ModelcheckQsLTL::check`. Diese startet die Tiefensuche für die übergebene Forderung. Da es keinen früheren Knoten gibt, wird als `valid`-Menge die leere Menge übergeben. Rückgabewert ist, ob für die Forderung `as` eine erfolgreiche Beweisstruktur gefunden wurde.

```
1 bool ModelcheckQsLTL::check(Assertion as) {
2   return dfs(as,{});
3 }
```

Die Funktion `ModelcheckQsLTL::init` initialisiert die für den Algorithmus zusätzlich notwendigen Felder. Insbesondere wird hier die `valid`-Menge an diesem Knoten bestimmt. Zunächst werden dazu die an diesem Knoten gültigen $\phi_1 \,\mathsf{R}\, \phi_2$ und $\mathsf{X}(\phi_1 \,\mathsf{R}\, \phi_2)$ bestimmt. Die Variablenbelegung des Vorgängerknotens, die bereits über die Korrespondenzfunktion auf den aktuellen Knoten umgerechnet wurde, wird als Argument übergeben und muss dann nur noch für die Startpunkte herangezogen werden. Wir verwenden eine Hilfsfunktion `add_to_valid`, die direkt im Anschluss angegeben ist.

```
1 void ModelcheckQsLTL::init(Assertion &as,Set<SCCInfo> valid) {
2   this.dfn = this.dfn+1;
3   as.dfsn = this.dfn;
4   as.low = this.dfn;
5
6   // In der Formelmenge unserer Forderung suchen wir
7   // (phi1 R phi2) oder X (phi1 R phi2)
8   foreach (formula in as.formulas) {
9     Formula phi1,phi2;
10    if (formula.phi.is("phi1 R phi2",&phi1,&phi2)) {
11      add_to_valid(as,valid,this.dfn,phi1,phi2,formula.vl);
12    } else if (formula.phi.is("X phi1 R phi2",&phi1,&phi2)) {
13      add_to_valid(as,valid,this.dfn,phi1,phi2,formula.vl);
14    }
15  }
16 }

18 // Hilfsfunktion um die entsprechenden (phi1 R phi2)^vl zu as.valid hinzuzufügen
19 // valid ist die übergebene Menge
20 // dfn unsere eigene Tiefensuchnummer
21 void add_to_valid(Assertion &as,Set<SCCInfo> valid,
22                   int dfn,Formula phi1,Formula phi2,LVState vl) {
23   // Wenn phi2 vorkommt, dann wird (phi R phi2) nicht Teil der Menge
24   if (as.formulas.contains(QFormula(phi2,vl))) {
25     return;
26   }
27
28   // Wir erstellen dann den SCCInfo-Eintrag; ein eventuelles X beachten wir nicht
29   SCCInfo sinfo;
30   sinfo.formula=QFormula(Formula("phi1 R phi2",phi1,phi2),vl);
```

```
31
32   // Den Startpunkt erhalten wir aus der übergebenen valid-Menge
33   // andernfalls muss unsere eigene Tiefensuchnummer verwendet werden
34   SCCInfo result=valid.find_by_formula(sinfo.formula);
35   if (result) {
36     sinfo.sp=result.sp;
37   } else {
38     sinfo.sp=dfn;
39   }
40   as.valid.insert(sinfo);
41 }
```

Das Herz des Algorithmus ist schließlich die Funktion `ModelcheckQsLTL::dfs`. Sie lässt sich in drei Teile gliedern:

```
1. Initialisierung der Mengen
2. Bestimmen der Nachfolgeknoten und Erforschen dieser
3. Entfernen von vollständigen SCCs
```

Aufgerufen wird `dfs` mit der `valid`-Menge des Vorgängerknotens, die aber schon durch die Korrespondenzfunktion auf den in unserer Forderung enthaltenen Zustand angepasst wurde.

Im zweiten Teil wird der Fall einer leeren Teilzielmenge sowie von **tt** gesondert betrachtet. Ansonsten werden nun alle Teilziele der Reihe nach erforscht, wobei aber zuerst überprüft wird, ob dieses Teilziel schon einmal untersucht wurde: Entweder, weil es gescheitert ist, oder weil es noch im SCC-Stack liegt, oder aber weil es erfolgreich war. Der allgemeine Fall (weder leer, noch **tt**) ist dabei:

```
Für alle Nachfolgeknoten {
  2.1. Gesehen und verworfen, oder
  2.2. Gesehen und Zyklus, oder
  (2.3. Gesehen und erfolgreich), oder
  2.4. Nicht gesehen: rekursiver Aufruf von dfs
}
```

Die Fällen 2.2. und 2.4. sind etwas umfangreicher, ebenso 3.; bei 2.4. muss zunächst die akkumulierte `valid`-Menge angepasst werden, wofür wir eine Hilfsfunktion, `update_valid`, verwenden. Nach dem Aufruf von `dfs`, also auf dem Rückweg, muss bei gefundenem Zykel u. a. die Menge im aktuellen Zustand eingeschränkt werden. Dazu verwenden wir die Hilfsfunktion `remove_valid`. Diese Hilfsfunktionen sind wieder direkt angefügt.

```
1 bool ModelcheckQsLTL::dfs(Assertion &as,Set<SCCInfo> valid) {
2   bool ret=true;
3
4   init(as,valid);
5   this.stack.push(as);
6   V.insert(as);
7
8   Set<Assertion> subgoals = as.subgoals();
9   if (subgoals.empty()) {
10    ret=false;
11  } else if ( (subgoals.size()==1)&&(subgoals[0].is_true()) ) {
12    ret=true;
13  } else {
14    Assertion subgoal;
```

```
15      foreach (subgoal in subgoals) {
16        if (V.contains(subgoal)) { // schon gesehen
17          if (F.contains(subgoal)) { // und verworfen
18            ret=false;
19            break;
20          } else if (this.stack.contains(subgoal)) { // Zyklus gefunden!
21            as.low=min(as.low,subgoal.low); // frühesten Startzustand ermitteln
22
23            // Alle scinfos aus as.valid wegwerfen, die noch dazugekommen sind.
24            SCCInfo scinfo;
25            foreach (scinfo in as.valid) {
26              if (scinfo.sp > subgoal.dfsn) {
27                as.valid.remove(scinfo);
28              }
29            }
30
31            if (as.valid.empty()) { // keine potentielle SCC mehr übrig
32              ret=false;
33              break;
34            }
35          } // andernfalls war es erfolgreich, ok.
36
37        } else { // subgoal noch nicht gesehen
38          // valid muss auf den neuen Zustand umgerechnet werden und
39          // Elemente müssen bei Vorkommen von phi2 entfernt werden
40          Set<SCCInfo> newvalid=update_valid(as,subgoal);
41
42          ret=dfs(subgoal,newvalid);
43
44          if (subgoal.low <= as.dfsn) { // Zyklus gefunden und wir sind Teil davon.
45            as.low=min(as.low,subgoal.low); // frühesten Startzustand updaten
46            // Menge der noch möglichen SCCs übernehmen,
47            // bzw. unsere Menge entsprechend reduzieren
48            as.valid=remove_valid(as,subgoal);
49          }
50        }
51      }
52    }
53
54    if (as.dfsn == as.low) {
55      // Es gibt keinen Zyklus, der „vor uns" endet - und sich ums aufräumen kümmert
56      // Wir sind daher (höchstens) der Start einer SCC (die nun bewiesen ist).
57      while (as != this.stack.top()) {
58        if (!ret) {
59          F.insert(this.stack.top());
60        }
61        this.stack.pop();
62      }
63    }
64    return ret;
65  }

67  // Verwendet die Korrespondenzfunktion um as.valid auf next umzurechnen
68  // entfernt außerdem Formeln (phi1 R phi2)^vl aus der valid-Menge,
69  // für die phi2^vl in next.formulas enthalten ist.
70  Set<SCCInfo> update_valid(Assertion as,Assertion next) {
71    Set<SCCInfo> ret;
72    SCCInfo scinfo;
```

```
73   foreach (scinfo in as.valid) {
74     // Korrespondenzfunktion auf vl anwenden
75     LVState vl=as.state.update(next.state,scinfo.formula.vl);
76     // phi2 extrahieren
77     Formula phi1,phi2;
78     scinfo.formula.phi.is("phi1 R phi2",&phi1,&phi2);
79     if (!next.formulas.contains(QFormula(phi2,vl))) {
80       // phi2 nicht enthalten:
81       // -> Mit neuer Variablenbelgung zur neuen valid-Menge hinzufügen.
82       scinfo.formula.vl=vl;
83       ret.insert(scinfo);
84     }
85   }
86   return ret;
87 }

89 // Gibt die Menge zurück, die entsteht, wenn aus as.valid die Elemente
90 // entfernt werden, die in next.valid - nach Anwendung der
91 // Korrespondenzfunktion - nicht mehr enthalten sind.
92 // Diese Funktion kann ggf. auch effizienter implementiert werden und
93 // ähnelt im Moment update_valid().
94 Set<SCCInfo> remove_valid(Assertion as,Assertion next) {
95   Set<SCCInfo> ret;
96   SCCInfo scinfo;
97   foreach (scinfo in as.valid) {
98     LVState vl=as.state.update(next.state,scinfo.formula.vl);
99     SCCInfo newsc=scinfo;
100    newsc.formula.vl=vl; // das hatte update_valid() daraus gemacht
101    if (next.valid.contains(newsc)) {
102      // noch vorhanden, gut!
103      ret.insert(scinfo);
104    }
105  }
106  return ret;
107 }
```

3.2.4 Korrektheit

Lemma 67 (Invariante)
Der über `ModelcheckQsLTL::dfs` aufrufbare Algorithmus erhält die Invariante, dass, für eine Momentaufnahme $G = (V, E)$ der inkrementiell erstellten Beweisstruktur und jede in diesem G enthaltene starke Zusammenhangskomponente S, immer $\mathit{valid}(\mathit{max}(S)) = \mathit{Success}(G)$ ist, wobei $\mathit{max}(S)$ die Forderung `as` aus S mit der höchsten Tiefensuchnummer (`as.dfsn`) zurückgibt und $\mathit{valid}(\texttt{as}) = \{\varphi^{vl} \mid \texttt{sinfo} \in \texttt{as.valid}, \varphi = \texttt{sinfo.formula.phi}, vl = \texttt{sinfo.formula.vl}\}$ die in `as.valid` enthaltenen Formeln mit der dem Zustand `as.state` zugeordneten Variablenbelegung.

Beweis: Siehe [2]. □

Dieses Lemma ergibt zusammen mit Satz 66 den folgenden Satz:

Satz 68 (Korrektheit des Algorithmus)
Wenn `ModelcheckQsLTL::check` terminiert, dann gilt für jedes $\texttt{as} \in \texttt{V}$ der Form $q \vdash \mathsf{A}\Phi$, dass $q \models \mathsf{A}\Phi$ gdw. $\texttt{as} \notin \texttt{F}$.

Korollar 69
`ModelcheckQsLTL::check(`$q \vdash A\Phi$`)` gibt genau dann `true` zurück, wenn $q \models A\Phi$.

3.2.5 Erweiterung auf Q*CTL*

Um Q*CTL* zu testen und unseren Modelchecking-Algorithmus, wie bereits versprochen, darauf zu erweitern, lassen wir $R^{\models}$ bzw. $R^{\not\models}$ jetzt auch für $q \vdash A(\Phi, \psi^{vl})$ mit $\psi \in \mathsf{SF}_{\mathrm{Q^*CTL^*}}$ zu, da $\models$ für Stateformeln und einen gegebenen Zustand (mit Variablenbelegung) bereits komplett bestimmt ist. Dabei stoßen wir wieder auf die Mehrdeutigkeit zwischen $\forall_P$ und $\forall_S$ (bzw. $\exists$) in Pfadformeln, also die Frage, wann vom Pfadmodus („LTL“) auf den Zustandsmodus („CTL“) gewechselt wird, pragmatisch so beantworten, dass wir in $R^{\models}$ (bzw. $R^{\not\models}$) $\forall_S$ und $\exists_S$ nicht berücksichtigen. Das entspricht aber fast genau der in Lemma 46 ($\forall_\mathcal{P}$ und $\exists_\mathcal{P}$ sind ausreichend für Q*CTL*) bereits als semantisch bedeutlungslos gezeigten Variation, außer dass wir bei Stateformeln, die mit $\forall$ (bzw. $\exists$) beginnen, kein „Hilfs-A“ einfügen, sondern zusätzlich $\forall_\mathcal{S}$ und $\exists_\mathcal{S}$ implementieren, worauf aber erst zurückgegriffen wird, wenn $\forall_\mathcal{P}$ und $\exists_\mathcal{P}$ nicht anwendbar sind.

Dazu wird in `Assertion::subgoals()` ab sofort `checkQsCTLs(...)` für die entsprechenden Regeln aufgerufen. Außerdem muss, um eine beliebige Q*CTL*-Formel zu testen, ebenfalls nur diese eine Funktion aufgerufen werden:

```
1  // Top-level-Funktion zum Testen von q, vl ⊨ ψ für ψ ∈ SF_{Q*CTL*}:
2  // Initial sollte vl alle Logikvariablen auf null abbilden (leer sein).
3  bool checkQsCTLs(State q,LVState vl,Formula psi) {
4    if (V.contains(as)) {
5      if (F.contains(as)) {
6        return false;
7      }
8      return true;
9    }
10   V.insert(as);
11   Formula psi1,psi2,phi;
12   String name;
13   if (psi.is_atomic()) {
14     if (q.models(psi,vl)) {
15       return true;
16     }
17   } else if (psi.is("psi1 ∧ psi2",&psi1,&psi2)) {
18     if (checkQsCTLs(q,vl,psi1) && checkQsCTLs(q,vl,psi2)) {
19       return true;
20     }
21   } else if (psi.is("psi1 ∨ psi2",&psi1,&psi2)) {
22     if (checkQsCTLs(q,vl,psi1) || checkQsCTLs(q,vl,psi2)) {
23       return true;
24     }
25   } else if (psi.is("A phi",&phi)) {
26     ModelcheckQsLTL mc;
27     Set<QFormula> formulas;
28     formulas.insert(QFormula(phi,vl));
29     if (mc.check(Assertion(q,formulas))) {
30       return true;
31     }
32   } else if (psi.is("E phi",&phi)) {
33     ModelcheckQsLTL mc;
34     Set<QFormula> formulas;
35     phi=phi.negation();
```

```
36      formulas.insert(QFormula(phi,vl));
37      if (!mc.check(Assertion(q,formulas))) {
38        return true;
39      }
40    } else if (psi.is_forall(&name,&psi1)) { // ∀
41      Object obj;
42      foreach (obj in q.objects()) {
43        vl.set(name,obj);
44        if (!checkQsCTLs(q,vl,psi1)) {
45          F.insert(as);
46          return false;
47        }
48      }
49      return true;
50    } else if (psi.is_exists(&name,&psi1)) { // ∃
51      Object obj;
52      foreach (obj in q.objects()) {
53        vl.set(name,obj);
54        if (checkQsCTLs(q,vl,psi1)) {
55          return true;
56        }
57      }
58    }
59    F.insert(as);
60    return false;
61  }
```

3.2.6 Komplexität

Zur Komplexität des Algorithmus sei gesagt, dass der in [2] angegebene Algorithmus in $\mathcal{O}(|\psi| * 2^{\mathcal{O}(|\psi|)} * |K|)$ liegt, da die Beweisstruktur das Kreuzprodukt von Teilformeln mit Zuständen des Zustandsraumes ist und jeder Knoten der Beweisstruktur höchstens einmal besucht wird, wobei $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$, $|K| = |Q|$ und $|\psi|$ die Länge der Formel ψ ist.

In unserem Fall kommen dann noch all die möglichen Variablenbelegungen hinzu; für jeden Zustand $q \in Q$ gibt es, null als mögliche Belegung eingeschlossen, $(|obj(q)|+1)^{LV}$ verschiedene Belegungen. Es ist $(|obj(q)|+1)^0 = 1$ und $|K| = \sum_{q \in Q} 1$. Insgesamt erhalten wir: `checkQsCTLs` ist in $\mathcal{O}(|\psi| * 2^{\mathcal{O}(|\psi|)} * \sum_{q \in Q}(|obj| + 1)^{LV})$.

4 Markierungen

Wir machen alle in einem Zustand möglichen Variablenbelegung explizit, indem wir für eine gegebene Anzahl Logikvariablen, etwa $|LV|$, in der Heapkonfiguration zusätzliche Hyperkanten f mit entsprechendem Label „$lab(f) = x$“ für $x \in LV$ hinzufügen (wenn $vl(x) \neq \mathsf{null}$). Für jedes $v \in V_H$ entsteht so eine markierte Kopie der Heapkonfiguration (und damit des Zustands) mit jeweils $att(f) = v$.

Wenn $K = (Q, Q_0, \triangleright, obj, kf, var, sel)$ endlich ist, also $|Q| < \infty$, $\sum_{q \in Q} |obj(q)| < \infty$, dann wächst der Zustandsraum dadurch zwar auf $\sum_{q \in Q}(|obj(q)| + 1)^{LV}$ Zustände an (wir dürfen null nicht vergessen), bleibt aber endlich.

Technisch erweitern wir unser Alphabet um Markierungen, die die Logikvariablen im Stile der Programmvariablen modellieren:

Definition 70 (Markierte Heapkonfiguration)
Eine *markierte Heapkonfiguration* basiert auf einer Heapkonfiguration, bei der zusätzlich *Markierungen* hinzukommen; die Menge aller Markierungen sei Mar_Σ.
Dann muss neben den in Definition 9 (Heapkonfiguration) geforderten Eigenschaften gelten:

- $\Sigma = Var_\Sigma \dot\cup Mar_\Sigma \dot\cup Sel_\Sigma$ (d. h. wir benutzen ein erweitertes Alphabet),
- $\forall x \in Mar_\Sigma : rk(x) = 1$,
- $\forall x \in Mar_\Sigma : |\{e \in E \mid lab(e) = x\}| \leq 1$ (Markierung höchstens einmal vorhanden),

Außerdem sei MHC_Σ die Menge aller markierten Heapkonfigurationen über Σ.
Damit wir Markierungen für Logikvariablen verwenden können sollte $Mar_\Sigma \subseteq LV$ sein (oder eine entsprechende Korrespondenz existieren).
Im Folgenden setzen wir o. B. d. A. $Mar_\Sigma = LV$.

Definition 71 (Isomorphie markierter Heapkonfigurationen)
`rep` (und damit $\cong$) ist auf markierte Heapkonfigurationen übertragbar, ohne dass `enum` geändert werden muss (Markierungen tragen dazu nichts bei); [†] um die Ordnung der Kanten herzustellen, sortieren wir (bevor wir die lexikalische Ordnung befragen) alle Markierungskanten nach ganz *hinten*. Da wir auch Mar_Σ ordnen können, lassen sich diese Kanten dann geordnet an die auf bereits vorgestellte Weise sortierten Variablen- und Selektorkanten anfügen.
Als Pendant zu $\overline{\mathsf{HC}_\Sigma}$ sei daher $\overline{\mathsf{MHC}_\Sigma} = \mathsf{MHC}_\Sigma/\cong$.

[†] Falls dabei ein markierter Knoten wegen Nichterreichbarkeit (von einer Variablen aus) entfernt wird, so wird natürlich auch att und lab entsprechend angepasst.

Um die bereits eingeführten Konzepte und Logiken auf markierte Heapkonfigurationen zu übertragen, stellen wir eine unmarkierte Heapkonfiguration als Äquivalenzklasse von markierte Heapkonfiguration dar:

Definition 72 (Äquivalente markierte Heapkonfigurationen)
Wir nennen zwei markierte Heapkonfigurationen $H, H' \in \mathsf{MHC}_\Sigma$ *äquivalent*, $H \approxeq H'$, wenn sie nur durch Ummarkierung auseinander hervorgehen:

- Mit $H = (V, E, att, lab, ext)$ sei $H^- = (V, E \backslash E_{mark}, att\restriction_{E\backslash E_{mark}}, lab\restriction_{E\backslash E_{mark}}, ext)$ die Heapkonfiguration $H^- \in \mathsf{HC}_\Sigma$, die entsteht, wenn in H alle Markierungen entfernt werden. Dazu ist $E_{mark} = \{e \in E \mid lab(e) \in Mar_\Sigma\}$.
- Es sei $H \approxeq H'$ gdw. $H^- = H'^-$.
- Für $\overline{H} \in \overline{\mathsf{MHC}_\Sigma}$ sei $\overline{H}^- = \{H^- \mid H \in \overline{H}\}$.

Dass sich `rep` mit Markierungen gut verträgt, und damit $\approxeq$ auch für $\overline{\mathsf{MHC}_\Sigma}$ wohldefiniert ist, besagt das folgende Lemma:

Lemma 73
- Aus $H \cong H'$ folgt $H^- \cong H'^-$. (Die Umkehrung gilt i. A. nicht!)
- Mit $H \in \mathsf{MHC}_\Sigma$ ist $\mathtt{rep}(H^-) = \mathtt{rep}(H)^-$.
- Für $\overline{H} \in \overline{\mathsf{MHC}_\Sigma}$ folgt dann $\mathtt{rep}(\overline{H}^-) = \mathtt{rep}(\overline{H})^-$.

Beweis: Für Teil 1 stellen wir fest, dass H^- aus H durch Einschränken der Kantenmenge hervorgeht. Sind dann wegen $H \cong H'$ Isomorphiefunktionen iso_V und iso_E gegeben, dann ist auch schon $H^- \cong H'^-$ vermöge der selben Knotenbijektion iso_V und der auf die Kantenmenge von H^- eingeschränkten Knotenbijektion iso_E. Außerdem bleiben Markierungen unter Isomorphie erhalten.
Beim zweiten Teil wissen wir aus der Definition vom `rep`, dass $H \cong \mathtt{rep}(H)$, sowie $H^- \cong \mathtt{rep}(H^-)$. Durch Anwenden des ersten Teils des Lemmas auf $H \cong \mathtt{rep}(H)$, so erhalten wir $H^- \cong \mathtt{rep}(H)^-$. Zusammen ergibt das wegen der Transitivität von $\cong$, dass $\mathtt{rep}(H^-) \cong \mathtt{rep}(H)^-$. Da nun H^- sich von H nur in der Kantenmenge unterscheidet und `enum` Markierungen nicht verwendet, muss schon $iso_V = \mathsf{id}$ sein. Weiter ist durch die Erweiterung von `rep` an Markierungskanten sichergestellt, dass diese ganz hinten einsortiert sind. Ein Weglassen (durch Einschränken der Kantenmenge) verändert also nicht die Variablen- und Selektorkanten. Daher ist auch $iso_E = \mathsf{id}$ bezüglich $\mathtt{rep}(H^-)$ und $\mathtt{rep}(H)^-$. Damit gilt die Behauptung.
Zum Beweis der dritten Aussage wenden wir die Definition auf $\mathtt{rep}(\overline{H}^-)$ an und erhalten $M = \{\mathtt{rep}(H') \mid H' \in \overline{H}^-\} = \{\mathtt{rep}(H^-) \mid H \in \overline{H}\} = \{\mathtt{rep}(H)^- \mid H \in \overline{H}\}$. Weil aber bekannt ist, dass $|\{\mathtt{rep}(H) \mid H \in \overline{H}\}| = 1$, muss auch $|M| = 1$. Daher ist $\mathtt{rep}(\overline{H}^-) = H'$ für das $H' \in M$, nämlich $H' = \mathtt{rep}(H)^- = \mathtt{rep}(\overline{H})^-$. □

Eine wichtige Folge dieses Lemmas ist, dass das durch $\overline{H}^-$ bereits angedeutete $\overline{H} \approxeq \overline{H}'$ auch wohldefiniert ist. Insbesondere ist $\overline{\mathsf{MHC}_\Sigma}/_{\approxeq} = \mathsf{MHC}_\Sigma/_{\cong}/_{\approxeq}$ und $\mathsf{MHC}_\Sigma/_{\approxeq}/_{\cong}$ in gewissem Sinne vertauschbar.

4.1 Erweiterte Zustandsräume

Wir möchten nun erweiterte Zustandsräume herstellen, die später dazu verwendet werden können, die Logikvariablen über Markierungen darzustellen. Wir geben zunächst die formale Definition an und gehen im Anschluss auf die vorgenommenen Änderungen ein:

Definition 74 (Erweiterter objektorientierter Zustandsraum)
Ein *erweiterter objektorientierter Zustandsraum* $K = (Q, Q_0, \widetilde{\triangleright}, var, sel, \cong)$ besteht aus

- einer Menge Q von „markierten“ Zuständen,
- den Startzuständen $Q_0 \in Q$,
- der *erweiterten Zustandsübergansrelation* $\widetilde{\triangleright} \subseteq Q \times (\{\varepsilon\} \dot\cup Mar_\Sigma) \times Q$, wobei wir
  - $\widetilde{\triangleright}$ schreiben, wenn keine Variablenumbelegung vorgenommen wird – oder aber $\widetilde{\triangleright}_\varepsilon$, wo Verwechslungen auftreten könnten. Das korrespondiert mit dem bisherigen Verständnis von $\triangleright$:
    „$q \widetilde{\triangleright} q'$ gdw. $q \triangleright q'$“, und
  - $\widetilde{\triangleright}_x$ schreiben, um auszudrücken, dass von einem (markierten) Zustand q zu einem alternativ markierten Zustand q' übergegangen wird. Intuitiv gesprochen gilt für q, vl mit „$q \in Q^-$“, $vl \in Vl_q$ und $x \in LV$:
    „$(q, vl) \widetilde{\triangleright}_x (q, vl')$ gdw. $vl' = vl[x \mapsto v]$ mit $v \in V_q$“, wobei (q, vl) gerade ein markierter Zustand $\in Q$ ist.
    Es handelt sich um zusätzliche Kanten, die wir zur einfacheren Unterscheidung *blaue Kanten* nennen wollen; die bereits von $\triangleright$ bekannten Kanten dagegen *rote Kanten*.

  Wie schon bei $\triangleright$ fordern wir, dass $\widetilde{\triangleright}_\varepsilon$ ebenfalls total ist.
  Weiter besteht K aus:

- var zum Auswerten von Variablen *und Markierungen*, wobei der Bildbereich V_q nun nicht mehr „bekannt“ ist, da wir obj nicht mehr explizit mitgeben, und
- sel zum Auswerten von Selektoren, wie schon vom objektorientierten Zustandsraum bekannt, sowie
- einer Relation $\cong \subseteq Q \times Q$, die angibt, welche Zustände sich nur in ihrer Markierung unterscheiden.

Um die Korrektheit einer Logik, die Markierungen zum Verwalten der Logikvariablen nutzt, garantieren zu können, müssen wir außerdem eine gewisse „Gutmütigkeit“ von $\widetilde{\triangleright}$ (zusammen mit $\cong$) fordern. Zunächst formulieren wir die Bedingungen nur informell, wir werden sie später in Definition 78 präzisieren:

1. $\widetilde{\triangleright}_x$ führt nicht aus einer Äquivalenzklasse heraus.
2. Äquivalente Zustände führen unter $\widetilde{\triangleright}$ in die selben Äquivalenzklassen.
3. Alle von einem Zustand aus über $\widetilde{\triangleright}$ erreichbaren Zielzustände sind nicht untereinander äquivalent.

4. Ein markierter Zustand ist eindeutig, d. h. es gibt keinen dazu äquivalenten, der sich überhaupt nicht unterscheidet.

5. Jede Ummarkierung erreicht alle Objekte.

6. $\widetilde{\triangleright}$ erhält Markierungen.

7. Die Variablen und Selektoren unterscheiden sich für zwei äquivalente Zustände nicht.

Es fällt auf, dass *obj* und *kf* weggefallen sind. Diese sollen jetzt durch Markierungen dargestellt werden; *kf* ist dann nicht mehr nötig, da durch die für Markierungen verwendeten Hyperkanten und Labels auch beim Zusammenfassen isomorpher Heapkonfigurationen Objekte sicher identifiziert werden.

Auf die erweiterte Zustandsübergansrelation $\widetilde{\triangleright}$ sind wir bereits eingegangen.

Dann gibt es noch die Relation $\cong$ in der Signatur von $\widetilde{K}$. Dieses dient dazu, für zwei Zustände q und q' einfach entscheiden zu können, ob sie durch Umbelegung auseinander hergegangen sind. Wird etwa $Q = \mathsf{MHC}_\Sigma \times (Stm \dot\cup \{\varepsilon\})$ gewählt, dann lässt sich das für $(H, stm), (H', stm') \in Q$ direkt auf $H \cong H'$ zurückführen. Technisch könnte $q \cong q'$ auch aus $\widetilde{\triangleright}$ extrahiert werden, indem man die von q aus nur über blaue Kanten erreichbaren q' sammelt. Es wird also die transitive Hülle bzgl. der blauen Kanten gebildet, wobei $|LV|$ Schritte ausreichend sind. Leider reicht das aber noch nicht ganz aus: null muss zusätzlich berücksichtigt werden, da wir keine blauen Kanten in Richtung null haben wollen, jedoch blaue Kanten von null weg immer existieren. Wir geben daher $\cong$ der Einfachheit halber hier zusätzlich an.

Wir wollen nun zwei erweiterte Zustandsräume angeben:

Definition 75 (Erweiterter Zustandsraum über markierten Heapkonfigurationen)
Für ein Programm $pgm \in Stm$ und Startheapkonfigurationen $HC_0 \subseteq \mathsf{HC}_\Sigma$ über dem Alphabet Σ ist der erweiterte Zustandsraum $K^{M1}_{pgm,HC_0} = (Q, Q_0, \widetilde{\triangleright}, var, sel, \widetilde{\cong})$ gegeben durch:

- $Q = \mathsf{MHC}_\Sigma \times (Stm \dot\cup \{\varepsilon\})$, wobei $q = (H_q, pgm_q) \in Q$, $H_q = (V_{H_q}, E_{H_q}, att_{H_q}, lab_{H_q}, ext_{H_q})$,
- $Q_0 = \{(H, pgm) \mid H \in HC_0\}$, wobei zwar $HC_0 \subseteq \mathsf{HC}_\Sigma$ ist, wir diese H aber als markierte Heapkonfigurationen $\in \mathsf{MHC}_\Sigma$ auffassen können, bei der *keine* Markierungen gesetzt sind.
- Weiter ist $\widetilde{\triangleright}$ gegeben durch:
  - $q \,\widetilde{\triangleright}\, q'$ gilt entsprechend der Definition 19 (Semantik von Statements), die ohne Änderung auf MHC_Σ übertragbar ist. Dabei entstehen auch die Übergänge zwischen entsprechenden zu q und q' äquivalenten Zuständen, wo sie den in der Semantik angegebenen Regeln genügen! †
  - $q \,\widetilde{\triangleright}_x\, q'$ wird für alle $x \in Mar_\Sigma$ wie folgt gebildet:

$$\frac{v \in V_{H_q}}{H, \langle S\rangle \;\widetilde{\triangleright}_x\; H[x \hookrightarrow v], \langle S\rangle},$$

    wobei wir $H[x \hookrightarrow v]$, das in Definition 18 für $x \in Var_\Sigma$ eingeführt wurde, auf Markierungen erweitern.

†Wenn Objekte verschwinden (ignoriert werden), weil sie *nur* noch von Markierungen aus erreichbar wären, dann verbindet $\widetilde{\triangleright}$ einen solchen Zustand q mit dem entsprechenden Folgezustand q', bei dem dann die Markierung fehlt (gleich null ist).

Schließlich ist:

- $var(q)(x) = \begin{cases} att_{H_q}(e)[1] & \text{wenn } e = edge_{H_q}(x) \neq \mathsf{null} \\ \mathsf{null} & \text{sonst} \end{cases}$
  für jedes $q \in Q$, $x \in Var_\Sigma \cup Mar_\Sigma$ und
- $sel(q)(v,s) = \begin{cases} att_{H_q}(e)[2] & \text{wenn } e = edge_{H_q}(v,s) \neq \mathsf{null} \\ \mathsf{null} & \text{sonst} \end{cases}$
  für jedes $q \in Q$, $v \in V_{H_q}$ und $s \in Sel_\Sigma$, sowie
- $q \approxeq q'$ gdw. sowohl $pgm_q = pgm_{q'}$, also auch $H_q \approxeq H_{q'}$.

Dem letzten Schritt der eben eingeführten Definition, nämlich $\approxeq$ von MHC_Σ auf Zustände auszuweiten, wollen wir eine eigene Definition widmen:

Definition 76 (Äquivalente Zustände)
Gegeben $Q = \mathsf{MHC}_\Sigma \times (Stm \dot\cup \{\varepsilon\})$. Wir erweitern $\approxeq$ auf Q; für $q, q' \in Q$ als $q = (H, pgm)$ und $q' = (H', pgm')$ ist: $q \approxeq q'$ genau dann, wenn $pgm = pgm'$ und $H \approxeq H'$.
Entsprechend verwenden wir $[q]_\approxeq$, etc.; analog gehen wir bei $\overline{Q} = \overline{\mathsf{MHC}_\Sigma} \times (Stm \dot\cup \{\varepsilon\})$ vor.

Definition 77 (Erweiterter Zustandsraum über isomorphen markierten Heapkonfigurationen)
Für ein Programm $pgm \in Stm$ und Startheapkonfigurationen $HC_0 \subseteq \mathsf{HC}_\Sigma$ über dem Alphabet Σ wird der erweiterte Zustandsraum $K^{M2}_{pgm,HC_0} = (Q, Q_0, \widetilde{\triangleright}, var, sel, \approxeq)$ aufbauend auf K^{M1}_{pgm,HC_0} analog zu K^{2}_{pgm,HC_0} (der auf K^{1}_{pgm,HC_0} aufbaut) gebildet:

- $Q = \overline{\mathsf{MHC}_\Sigma} \times (Stm \dot\cup \{\varepsilon\})$, wobei $\overline{q} = (\overline{H}_{\overline{q}}, pgm_{\overline{q}}) \in Q$,
- $Q_0 = \{([H]_{\cong,\mathsf{MHC}_\Sigma}, pgm) \mid H \in HC_0\}$,
- $\widetilde{\triangleright}$ entsteht wie folgt:
  - Ausgehend von $q \mathrel{\widetilde{\triangleright}} q'$, mit $q, q' \in \mathsf{MHC}_\Sigma$ und K^{M1} verwenden wir auch hier (vgl. Definition 29):
    $$\frac{\mathtt{rep}(\overline{H}), \langle S\rangle \mathrel{\widetilde{\triangleright}} H', \langle S'\rangle}{\overline{H}, \langle S\rangle \mathrel{\widetilde{\triangleright}} [H']_{\cong,\mathsf{MHC}_\Sigma}, \langle S'\rangle}$$
    Die Korrespondenzfunktion interessiert uns dabei nicht mehr.
  - $\overline{q} \mathrel{\widetilde{\triangleright}_x} \overline{q}'$ wird für alle $x \in Mar_\Sigma$ wie folgt gebildet:
    $$\frac{v \in V_{H_q}}{\overline{H}, \langle S\rangle \mathrel{\widetilde{\triangleright}_x} [\mathtt{rep}(\overline{H})[x \hookrightarrow v]]_{\cong,\mathsf{MHC}_\Sigma}, \langle S\rangle},$$
    wobei wir $H[x \hookrightarrow v]$, das in Definition 18 für $x \in Var_\Sigma$ eingeführt wurde, auf Markierungen erweitern.
- Mit dem Repräsentanten $H_q = \mathtt{rep}(\overline{H}_q)$ verwenden wir wieder die Funktionen $var(\overline{q})(x)$ und $sel(\overline{q})(v,s)$ aus K^{M1}, und
- $\overline{q} \approxeq \overline{q}'$ nutzen wir, wie es in Definition 76 angegeben ist.

Im nächsten Schritt werden die in Definition 74 bereits beschriebenen Bedingungen an die erweiterte Zustandsübergangsrelation wichtig, die wir daher jetzt formalisieren:

Definition 78 (Bedingungen für erweiterte Zustandsräume)
Wir beginnen damit, etwas Notation einzuführen:

- $\{q \widetilde{\triangleright}\} = \{q' \mid q \widetilde{\triangleright} q'\}$, die Menge der von q aus über $\widetilde{\triangleright}$ direkt erreichbaren Zustände und
- für alle $x \in Mar_\Sigma$ die Menge $\{q \widetilde{\triangleright}_x\} = \{q' \mid q \widetilde{\triangleright}_x q'\}$, die alle durch Ummarkierung von x direkt erreichbaren Zustände angibt.

Dann fordern wir für erweiterte Zustandsräume:

1. $\widetilde{\triangleright}_x$ führt nicht aus einer Äquivalenzklasse heraus:

$$\forall q, q' \in Q \text{ mit } q \cong q' \text{ gilt } \forall x \in Mar_\Sigma : [\{q \widetilde{\triangleright}_x\}]_\cong = [\{q' \widetilde{\triangleright}_x\}]_\cong \qquad \text{(Mengen!)}$$

2. Äquivalente Zustände führen unter $\widetilde{\triangleright}$ in die selben Äquivalenzklassen:

$$\forall q, q' \in Q \text{ mit } q \cong q' \text{ gilt: } [\{q \widetilde{\triangleright}\}]_\cong = [\{q' \widetilde{\triangleright}\}]_\cong$$

   Wir können daher $\widetilde{\triangleright}$ direkt auf Äquivalenzklassen anwenden (und nennen es dann $\triangleright$): Mit $\widetilde{q}, \widetilde{q}' \in Q/_\cong$ schreiben wir $\widetilde{q} \triangleright \widetilde{q}'$:

$$\frac{q \in \widetilde{q} \quad \wedge \quad q \ \widetilde{\triangleright} \ q'}{\widetilde{q} \ \triangleright \ [q']_\cong}$$

   Das heißt, dass gerade $\widetilde{q}' = [q']_\cong$ ist.

3. Alle von einem Zustand aus über $\widetilde{\triangleright}$ erreichbaren Zielzustände sind nicht untereinander äquivalent (sondern fallen höchstens komplett zusammen):

$$\forall q \in Q \ \forall q_1, q_2 \in \{q \widetilde{\triangleright}\} : [q_1]_\cong \neq [q_2]_\cong, \text{ wenn } q_1 \neq q_2.$$

4. Ein markierter Zustand ist eindeutig, d. h. es gibt keinen dazu äquivalenten, der sich überhaupt nicht unterscheidet:

$$\forall q, q' \in Q \text{ mit } q \cong q' : \exists x \in Mar_\Sigma : var(q)(x) \neq var(q')(x).$$

5. Jede Ummarkierung erreicht alle Objekte:

   $\forall q \in Q \ \forall x, y \in Mar_\Sigma : var(\{q \widetilde{\triangleright}_x\})(x) = var(\{q \widetilde{\triangleright}_y\})(y), \qquad$ (Mengen!)
   $\forall q \in Q \ \forall x \in Var_\Sigma : var(q)(x) \in var(\{q \widetilde{\triangleright}_m\})(m)$, mit $m \in Mar_\Sigma$ beliebig,
   $\forall q \in Q \ \forall x \in Mar_\Sigma, s \in Sel_\Sigma : sel(q)(var(q)(x), s) \in var(\{q \widetilde{\triangleright}_m\})(m)$, mit $m \in Mar_\Sigma$ beliebig.

   Wegen der ersten Bedingung genügt es in den beiden anderen nur ein beliebiges $m \in Mar_\Sigma$ zu fordern. Sie sagen aus, dass Objekte, die über Variablen und Selektoren erreicht werden können auch markiert werden.
   Außerdem wird hier erzwungen, dass allen äquivalenten Zuständen in einer Äquivalenzklasse eine gemeinsame Objektmenge zugrunde liegt.

6. $\widetilde{\triangleright}$ erhält Markierungen:

   $\forall q \in Q, q' \in \{q \widetilde{\triangleright}\} : \forall x \in Mar_\Sigma : var(q)(x) = var(q')(x) \vee var(q')(x) = \mathsf{null}$,
   $\forall q \in Q, q' \in \{q \widetilde{\triangleright}\} : \forall x, y \in Mar_\Sigma \text{ mit } var(q)(x) = var(q)(y) : var(q')(x) = var(q')(y)$.

7. Die Variablen und Selektoren unterscheiden sich für zwei äquivalente Zustände nicht:

$$\forall q, q' \in Q \text{ mit } q \approxeq q' \text{ gilt } \forall x \in Var_\Sigma : var(q)(x) = var(q')(x),$$
$$\forall q, q' \in Q \text{ mit } q \approxeq q' \text{ und } \forall m \in Mar_\Sigma \text{ gilt mit } v = var(q)(m) :$$
$$\forall s \in Sel_\Sigma : sel(q)(v, s) = sel(q')(v, s).$$

Lemma 79 (o. Beweis)
K^{M1} und K^{M2} erfüllen diese Bedingungen.

4.2 Äquivalenzklassen erweiterter Zustände

Um nun Q*CTL* auf erweiterte Zustandsräume zu übertragen, werden wir uns den erforderlichen Änderungen zunächst so nähern, dass wir einen Schritt „zurück“ gehen und die markierten Zustände wieder zusammenfassen.

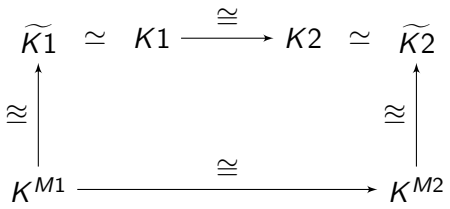
Definition 80 (Zustandsraum über markierten Heapkonfigurationen)
Sei der (nicht-erweiterte) Zustandsraum $\widetilde{K}^1_{pgm,HC_0} = (\widetilde{Q}, \widetilde{Q_0}, \triangleright, obj, kf, \widetilde{var}, \widetilde{sel})$ ausgehend vom erweiterten Zustandsraum $K^{M1}_{pgm,HC_0} = (Q, Q_0, \widetilde{\triangleright}, var, sel, \approxeq)$, wobei $q = (H_q, pgm_q) \in Q$, gebildet, indem:

- $\widetilde{Q} = Q/_{\approxeq}$,
- $\widetilde{Q_0} = \{[q]_{\approxeq} \mid q \in Q_0\}$,
- $\tilde{q} \triangleright \tilde{q}'$ wie in Definition 78 beschrieben.
- $obj(\tilde{q}) = var(\{q \widetilde{\triangleright}_m\})(m)$ für ein $m \in Mar_\Sigma$. Das genügt, da wir gefordert haben, dass von jeder Markierung alle Objekte getroffen werden.
- $kf_{\tilde{q} \triangleright \tilde{q}'}(v) = v'$ für $v \in V_q$ wird gebildet, indem ein beliebiges $m \in Mar_\Sigma$ gewählt wird, sowie $q \in \{q \in \tilde{q} \mid var(q)(m) = v\}$ (beliebig), und dann $V' = \{var(q')(m) \in V_{q'} \mid q' \in \{q \widetilde{\triangleright}\} \cap \tilde{q}'\}$ gebildet wird, wobei $|V'| = 1$ (auch $var(q')(m) = \mathsf{null}$ kommt explizit vor). Setze dann $v' \in V'$.
  Dass $|V'| = 1$ sein muss folgt aus den Bedingungen an erweiterte Zustandsräume; aufgrund der selben Bedingungen ist auch die Nutzung von beliebigen m und q zulässig.
- $\widetilde{var}(\tilde{q})(x) = v$ mit $v \in V_x(\tilde{q}) = \{var(q)(x) \in V_q \mid q \in \tilde{q}\}$, wobei wieder $|V_x(\tilde{q})| = 1$ sein muss, ebenso
- $\widetilde{sel}(\tilde{q})(v, s) = v$ mit $v \in V_{v,s}(\tilde{q}) = \{sel(q)(v, s) \in V_q \mid q \in \tilde{q}\}$.

Definition 81 (Zustandsraum über isomorphen markierten Heapkonfigurationen)
Analog lässt sich der Zustandsraum $\widetilde{K}^2_{pgm,HC_0} = (\widetilde{Q}, \widetilde{Q_0}, \triangleright, obj, kf, \widetilde{var}, \widetilde{sel})$ ausgehend vom erweiterten Zustandsraum $K^{M2}_{pgm,HC_0} = (\overline{Q}, \overline{Q}_0, \widetilde{\triangleright}, var, sel, \approxeq)$ bilden, wobei $\widetilde{Q} = \overline{Q}/_{\approxeq}$, und so weiter, ist.

Lemma 82 (o. Beweis)
Es gilt: $K^1 \simeq \widetilde{K}^1$ und $K^2 \simeq \widetilde{K}^2$.
Insbesondere sind sie dann mit Q*CTL* auch nicht unterscheidbar.

Es ergibt sich also folgender Zusammenhang:



4.3 Modifizierter Zustandsraum und Logik

Wir wissen nun, wie wir mit markierten Zuständen umgehen können, ohne dass unsere Logik Schaden nimmt. Allerdings werden die Markierungen dabei noch gar nicht verwendet. Um sie als Ersatz für die Logikvariablen zu verwenden, würden wir gerne statt auf Äquivalenzklassen auf den markierten Zuständen an sich arbeiten. Bei Stateformeln bereitet das auch kaum Schwierigkeiten; Pfadformeln sind allerdings insofern ein Problem, dass die Variablenbelegung bei einem Pfad nur für den ersten Zustand gilt, für weiteren Zustände auf dem Pfad aber noch nicht endgültig festgelegt sind.

Zum Beispiel sagt A etwas über alle Pfade aus, wobei Pfad sich hier *auf Klassen äquivalenter Zustände* bezieht – wie auch an $\widetilde{K}^1$ zu sehen ist. Es wäre falsch, nun einfach beliebige markierte Zustände auf Pfaden zu verwenden, weil dann mehr Möglichkeiten entstehen würden. Innerhalb einer Äquivalenzklasse darf, bzw. muss, die Markierung dagegen genau bei $\exists$ und $\forall$ gewechselt werden.

Weil es also im Bezug auf die Logik wenig Probleme bereitet, äquivalente Zustände zu Klassen zusammenzufassen, bilden wir nun zunächst einen Hybrid-Zustandsraum aus K^{M1} und $\widetilde{K}^1$ (bzw. K^{M2} und $\widetilde{K}^2$), der Variablenbelegungen durch markierte Zustände ersetzt, d. h. statt $\widetilde{q}, vl$ und $\widetilde{\pi}, vl$ soll in der Logik nun $\widetilde{q}, q$ und $\widetilde{\pi}, q$ verwendet werden, und eine entsprechende Markierungsfunktion einführt, jedoch als Zustandsmenge weiterhin Äquivalenzklassen nutzt.

Für diesen Zustandsraum passen wir dann die Semantik der Logik an. In einem weiteren Schritt werden wir dann mit den so gewonnenen Erkenntnissen schließlich auch die Semantik auf erweiterten Zustandsräumen angeben können.

Wir ersetzen nun $obj(\widetilde{q})$ des Zustandsraums $\widetilde{K}^1$ bzw. $\widetilde{K}^2$ durch eine Markierungsfunktion $mark(q, m)$, wobei $q \in \widetilde{q}$ und $m \in Mar_\Sigma$. Sie gibt die Teilmenge $Q' \subseteq \widetilde{q}$ zurück, deren Elemente sich gerade nur bezüglich der Markierung m unterscheiden.

Um initial nicht markierte $q_0 \in \widetilde{Q}_0$ zu bekommen, übergeben wir jetzt direkt Q_0;

Markierungen werten wir mit $var(q)(x)$ aus. Wir verwenden diese Funktion dann natürlich auch für Variablen (statt $\widetilde{var}(\widetilde{q})(x)$).

Schließlich vereinfachen wir die Korrespondenzfunktion, indem wir direkt $q \in \widetilde{q}$ übergeben und $q' \in \widetilde{q'}$ zurückerhalten.

Beispiel 13

In Abbildung 4.1 ist dargestellt, wie der Hybrid-Zustandsraum aussehen kann. Der Schwerpunkt der Darstellung liegt nicht in den Details – es sind etwa nicht alle der formal geforderten Kanten eingezeichnet – sondern darin, die Zusammenhänge wiederzugeben.

Die Grafik zeigt, wie $\triangleright$ die roten Kanten, d. h. $\widetilde{\triangleright}$, bündelt. Die blauen Kanten, $\widetilde{\triangleright}_x$, können durch $mark(q, x)$ abgefragt werden. Sie liefern indirekt die in q enthaltenen Objekte, da alle

Objekte durch eine Ummarkierungen erreicht werden. $\widetilde{kf}_{\widetilde{q}\triangleright\widetilde{q}'}$ *rekonstruiert die in* $\widetilde{\triangleright}$ *enthaltenen Informationen für gegebenes* $\widetilde{q}$*,* $\widetilde{q}'$ *und* q*.*

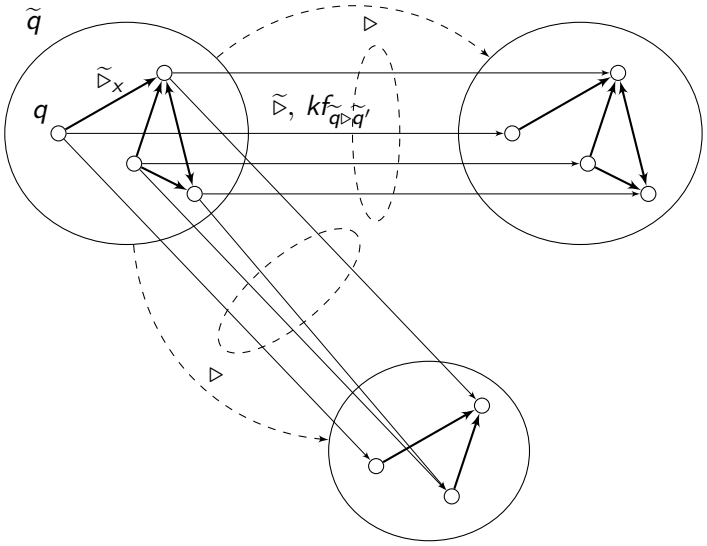


Abbildung 4.1: Beispiel 13 eines modifizierten Zustandsraumes.

Definition 83 (Modifizierter Zustandsraum)
Wir bilden den *modifizierten Zustandsraum* $K = (\widetilde{Q}, \widehat{Q_0}, \triangleright, mark, \widetilde{kf}, var, \widetilde{sel})$ ausgehend von $\widetilde{K}^1$ und K^{M1}, bzw. $\widetilde{K}^2$ und K^{M2}, indem wir

- $\widetilde{Q}$ von $\widetilde{K}^i$ (d. h. $\widetilde{K}^1$, bzw. $\widetilde{K}^2$) übernehmen,
- $\widehat{Q_0} = \{([q]_{\cong}, q) \mid q \in Q_0\} \subseteq \widetilde{Q} \times Q$,
- $\widetilde{q} \triangleright \widetilde{q}'$ von $\widetilde{K}^i$,
- $mark : Q \times Mar_{\Sigma} \longrightarrow 2^Q$, so dass $mark(q, m) = \{q \widetilde{\triangleright}_m\}$ ist, wofür $\widetilde{\triangleright}$ aus K^i verwendet wird. Das entspricht gerade der Definition von *obj* in K^i für fest gewähltes $m \in Mar_{\Sigma}$. Die Definitionen sind somit austauschbar.
- $\widetilde{kf}_{\widetilde{q}\triangleright\widetilde{q}'}(q) = q'$ für $q \in \widetilde{q}$ entsteht, indem man $Q' = \{q \widetilde{\triangleright}\} \cap \widetilde{q}'$ bildet, auch dafür ist bereits $|Q'| = 1$ (Bedingung 3 an erweiterte Zustandsräume), womit $q' \in Q'$ eindeutig definiert ist.
  Die Definition von kf aus $\widetilde{K}^i$ musste im Gegensatz zu dieser Definition noch ohne spezifische Markierungsinformation auskommen, um die Korrespondenz zu extrahieren.
- $var(q)(x)$ wird aus K^i übernommen, sowie
- $\widetilde{sel}(\widetilde{q})(v, s) = v$ von $\widetilde{K}^i$. Genauso gut könnten wir aber *sel* von K^i verwenden, da die Selektoren nicht von den Markierungen abhängen.

Die wesentliche Einsicht, die diese Definition, zusammen mit der dafür nötigen Vorarbeit, vermitteln will, ist: Der modifizierte Zustandsraum erhält immer noch alle Eigenschaften von

$\widetilde{K}^i$, da er nur die durch K^i gegebenen Grundbausteine verwendet und ein bestimmtes K^i die Struktur des Zustandsraums bereits in den für die Logik relevanten Grenzen festlegt.

Wir verzichten darauf zu zeigen, dass $kf_{\widetilde{q}\triangleright\widetilde{q}'}$ auch aus $\widetilde{kf}_{\widetilde{q}\triangleright\widetilde{q}'}$ wieder rekonstruierbar ist, also auch hier keine Informationen verloren gehen. Ein entsprechender Nachweis dürfte dem kritischen Leser aber nicht schwer fallen.

Kommen wir also zur Logik; hier können wir also zunächst auf vl noch nicht ganz verzichten, da wir zusätzlich zu $\widetilde{q}$ noch Informationen speichern müssen. Wir können dafür aber ein einziges (markiertes) $q \in \widetilde{q}$ verwenden und schreiben statt vl jetzt q, so dass aus $\widetilde{q}, vl \models \varphi$ und $\widetilde{\pi}, vl \models \varphi$ nun $\widetilde{q}, q \models \varphi$ und $\widetilde{\pi}, q \models \varphi$ wird.

Wir modifizieren zuerst die Auswertefunktion $[\![\cdot]\!]_q$ aus Definition 42:

Definition 84 (Semantik von AP)
Sei $K = (\widetilde{Q}, Q_0, \triangleright, mark, \widetilde{kf}, var, \widetilde{sel})$, $\widetilde{q} \in \widetilde{Q}$ und $q \in \widetilde{q}$. Dann ist:

$$[\![\mathsf{null}]\!]_{\widetilde{q},q} = \mathsf{null}$$
$$[\![x]\!]_{\widetilde{q},q} = var(q)(x)$$
$$[\![x.s]\!]_{\widetilde{q},q} = \begin{cases} sel(\widetilde{q})(v) & \text{wenn } v = [\![x]\!]_{\widetilde{q},q} \neq \mathsf{null} \\ \bot & \text{sonst} \end{cases}$$

Hier wird nun ein bestimmtes $q \in \widetilde{q}$ für Variablen x bei $var(q)(x)$ verwendet. Dass das so möglich ist, liegt an den Bedingungen an erweiterte Zustandsräume; Entsprechendes haben wir aber bereits in der Definition von $\widetilde{var}$ ausgenutzt und jetzt auf $[\![\cdot]\!]$ übertragen.

Bei der Semantik von Q*CTL* wird natürlich q, vl durch $\widetilde{q}, q$ ersetzt; wir geben nun nur noch die Regeln an, bei denen die Änderungen weniger trivial sind:

Definition 85 (Semantik von Q*CTL*)
Sei $K = (\widetilde{Q}, Q_0, \triangleright, mark, \widetilde{kf}, var, \widetilde{sel})$, $\widetilde{q} \in \widetilde{Q}$ und $q \in \widetilde{q}$.

Ausgehend von Definition 43 ist dann:

$\widetilde{q}, q \models \mu = \nu$, wenn $[\![\mu]\!]_{\widetilde{q},q} = [\![\nu]\!]_{\widetilde{q},q} \neq \bot$.

$\widetilde{q}, q \models \mathsf{A}\varphi$, wenn für jedes $\widetilde{\pi} \in Paths(\widetilde{q})$ gilt, dass $\widetilde{\pi}, q \models \varphi$.

$\widetilde{q}, q \models \mathsf{E}\varphi$, wenn es ein $\widetilde{\pi} \in Paths(\widetilde{q})$ gibt, so dass $\widetilde{\pi}, q \models \varphi$.

$\widetilde{q}, q \models \forall_{\mathcal{S}} x.\psi$, wenn für jedes $q' \in mark(q, x)$ gilt, dass $\widetilde{q}, q' \models \psi$

$\widetilde{q}, q \models \exists_{\mathcal{S}} x.\psi$, wenn es ein $q' \in mark(q, x)$ gibt, so dass $\widetilde{q}, q' \models \psi$.

$\widetilde{\pi}, q \models \mathsf{X}\varphi$, wenn $\widetilde{\pi}[2\ldots], \widetilde{kf}_{\widetilde{\pi}[1]\triangleright\widetilde{\pi}[2]}(q) \models \varphi$.

$\widetilde{\pi}, q \models \varphi\,\mathsf{U}\,\theta$, wenn es ein $i \geq 1$ gibt, so dass $\widetilde{\pi}[i\ldots], \widetilde{kf}_{\widetilde{\pi}[1\ldots i]}(q) \models \theta$ und
für alle $1 \leq j < i$ gilt $\widetilde{\pi}[j\ldots], \widetilde{kf}_{\widetilde{\pi}[1\ldots j]}(q) \models \varphi$.

$\widetilde{\pi}, q \models \varphi\,\mathsf{R}\,\theta$, wenn es entweder ein $i \geq 1$ gibt, so dass $\widetilde{\pi}[i\ldots], \widetilde{kf}_{\widetilde{\pi}[1\ldots i]}(q) \models \varphi$ und
für alle $1 \leq j \leq i : \widetilde{\pi}[j\ldots], \widetilde{kf}_{\widetilde{\pi}[1\ldots j]}(q) \models \theta$,
oder für alle $i \geq 1$ gilt $\widetilde{\pi}[i\ldots], \widetilde{kf}_{\widetilde{\pi}[1\ldots i]}(q) \models \theta$.

$\widetilde{\pi}, q \models \forall_{\mathcal{P}} x.\varphi$, wenn für jedes $q' \in mark(q, x)$ gilt, dass $\widetilde{\pi}, q' \models \varphi$

$\widetilde{\pi}, q \models \exists_{\mathcal{P}} x.\varphi$, wenn es ein $q' \in mark(q, x)$ gibt, so dass $\widetilde{\pi}, q' \models \varphi$.

Diese Semantik von Q*CTL* entspricht der ursprünglich angegebenen Semantik in dem Maße, in dem die Markierungen auf K gerade den Operationen auf vl entsprechen. Doch gerade das haben wir durch die aufwändige Äquivalenzklassenkonstruktion sichergestellt: Da obj in $\widetilde{K}^1$ und $\widetilde{K}^2$ gerade die erreichbaren Markierungen aufzählt können wir $mark$ verwenden und wissen für jede Markierung x, dass der ursprüngliche Ausdruck $vl[x \mapsto v]$ aus der Semantik zu genau den selben Resultaten (bezüglich var) führt, wie sie $q' \in mark(q'x)$ bereitstellt.

Um diesen Zusammenhang noch einmal zu verdeutlichen, geben wir für $\widetilde{q}, q$ (und dem modifizierten Zustandsraum K) an, wie daraus q, vl (genauer: $\widetilde{q}, vl$ bezüglich $\widetilde{K}^i$) extrahiert werden kann und umgekehrt:

Korollar 86
Da die Zustände $\widetilde{q}$ der selben Menge entstammen, brauchen wir nur vl und q zu betrachten:

- vl ergibt sich aus q durch $vl(x) = var(q)(x)$ für alle $x \in Mar_\Sigma = LV$.
- Gegeben vl. Bilde $Q' = \{q \in \widetilde{q} \mid \forall x \in LV = Mar_\Sigma : var(q)(x) = vl(x)\}$.

Bei Pfaden $\widetilde{\pi}, q$ und π, vl bezieht sich q bzw. vl auf den ersten Zustand, $\widetilde{\pi}[1]$, und $\pi[1]$. Das Ergebnis für $\widetilde{q}, q$ und q, vl überträgt sich somit.

4.4 Q*CTL* über erweiterten Zustandsräumen

Im letzten Schritt wollen wir vom modifizierten Zustandsraum zu allgemeinen erweiterten Zustandsräumen übergehen. Es fällt auf, dass wir im Moment sowohl q als auch $\widetilde{q}$, bzw. $\widetilde{\pi}[1]$ mitführen, obwohl immer $q \in \widetilde{q}$ ist; mehr noch, es gilt: $\widetilde{q} = [q]_{\approxeq}$, d. h. $\widetilde{q}$ ist bereits durch q festgelegt. Daher genügt hier q, für $\widetilde{\pi}$ sind die Änderungen dagegen nicht so einfach.

Eine andere Sicht auf dieses Vorgehen ist, dass wir zusätzliche Information darin speichern, welches Element der Äquivalenzklasse wir verwenden.

Im Detail: Als Grundmenge muss nun Q statt $\widetilde{Q}$ verwendet werden. Damit kann $\widetilde{kf}$ entfallen, da $\widetilde{\triangleright}$ – das wir nun statt $\triangleright$ verwenden –, bereits diese Information enthält, die bei $\triangleright$ des modifizierten Zustandraum artifiziell entfernt wurde. $\widetilde{kf}$ rekonstruierte dann diese Informationen wieder.

Wie schon erwähnt, kann $\widetilde{sel}$ kann ohne weiteres gegen sel ausgetauscht werden. Und $mark$ hängt nur von $\widetilde{\triangleright}$ ab, so dass es nun nicht mehr übergeben werden muss.

Die Logik arbeitete bisher auf den Äquivalenzklassen bezüglich $\approxeq$. Auf diese kann aber, insbesondere wegen der Pfade π, nicht komplett verzichtet werden. Gerade deshalb ist es wichtig, das $\approxeq$ in der Signatur der erweiterten Zustandsräume mitgegeben wird.

Wir geben nun die relevanten Änderungen der Semantik an, beginnend mit AP, das ohne große Änderungen auskommt:

Definition 87 (Semantik von AP über erweiterten Zustandsräumen)
Sei $K = (Q, Q_0, \widetilde{\triangleright}, var, sel, \approxeq)$ ein erweiterter objektorientierter Zustandsraum. Dann ist:

$$
\begin{aligned}
[\![\mathsf{null}]\!]_q &= \mathsf{null} \\
[\![x]\!]_q &= var(q)(x) \\
[\![x.s]\!]_q &= \begin{cases} sel(q)(v) & \text{wenn } v = [\![x]\!]_q \neq \mathsf{null} \\ \bot & \text{sonst} \end{cases}
\end{aligned}
$$

Als nächstes müssen wir eine geeignete Darstellung der in Definition 15 eingeführten Pfade finden:

Konzeptuell muss ein Pfad in K eine maximale Folge $q_1\,\widetilde{q}_2\,\widetilde{q}_3\,...$ von Zuständen sein, so dass $q_1 \in Q$ und $\widetilde{q}_2\,... \in \widetilde{Q}^* = (Q/\cong)^*$.

Dadurch wird für das erste Element eine bestimmte Markierungskonfiguration ausgewählt; bei den weiteren Elementen des Pfads legt man sich dagegen nicht auf nur eine Konfiguration fest.

Dass dann ein Pfad mindestens die Länge 1 haben sollte, ist unerheblich, da wegen der Totalität von $\widetilde{\rhd}$ jeder Pfad in K unendlich lang ist.

Technisch verwenden wir aber, von der konzeptionellen Form abweichend, die bisherige Darstellung als Folge in Q^+, wobei wir $q_2\,...$ nur als Repräsentanten betrachten. Die Menge Paths darf dann aber nicht sowohl (z. B.) q_1q_2, als auch q_1q_2' mit $q_2 \cong q_2'$ enthalten. Zum Glück wird das aber schon durch die für erweiterte Zustandsräume geforderte Eigenschaft, dass alle über $\widetilde{\rhd}_\varepsilon$ erreichbaren Zustände nicht äquivalent sind, sichergestellt.

Außerdem müssen wir, konzeptionell, beim Zustandsübergang von $\pi[1]$ nach $\pi[i]$ für $i > 1$ Änderungen vornehmen. Weil unsere Pfade aber nun bereits bezüglich $\widetilde{\rhd}$ aufgebaut werden, ist etwa $\widetilde{\pi}[2\,...]$, $\widetilde{kf}_{\pi[1]\rhd\pi[2]}(q)$ aus der Semantik für X aber schon genau $\pi[2\,...]$; gleiches gilt bei der Verwendung von $\widetilde{\pi}[i\,...]$, $\widetilde{kf}_{\widetilde{\pi}[1...i]}(q)$ für $\pi[i...]$. Nur bei $\forall$ und $\exists$ müssen wir Änderungen vornehmen und entsprechende Operationen einführen, die einen gegebenen Pfad π mit $q = \pi[1]$ auf ein anderes, gegebenes $q' \cong q$ umschreibt.

Wir geben dem besseren Verständnis wegen die (fast unveränderte) Definition eines – nun erweiterten – Pfades komplett an:

Definition 88 (Erweiterter Pfad)
Sei also $K = (Q, Q_0, \widetilde{\rhd}, \mathit{var}, \mathit{sel}, \cong)$.

- Ein *erweiterter Pfad* in K ist eine maximale Folge $q_1q_2\,... \in Q^+$ von Zuständen mit $q_i \,\widetilde{\rhd}\, q_{i+1}$ für alle $i \geq 1$.
- $\widetilde{\mathit{Paths}_K}$ sei die Menge aller solcher Pfade K; weiter sind
- $\widetilde{\mathit{Paths}_K}(q) = \{\pi \in \widetilde{\mathit{Paths}_K} \mid \pi[1] = q\}$ alle mit q beginnenden Pfade.

Zusätzlich definieren wir aber die „Pathlifting“-Operation:

- $\pi[1 \hookrightarrow q']$ für $q' \cong \pi[1]$, so dass

$$\pi[1 \hookrightarrow q'] \in \Pi' = \{\pi' \in \widetilde{\mathit{Paths}_K}(q') \mid \forall i \geq 1 : \pi[i] \cong \pi'[i]\},$$

  wobei $|\Pi'| = 1$ ist, da gerade die selben Äquivalenzklassen von verschieden markierten, aber äquivalenten Zuständen über $\widetilde{\rhd}$ getroffen werden und dabei nicht mehrmals (Bedingungen 2 und 3 an erweiterte Zustandsräume).

Mit der abschließenden Angabe der Semantik ist Q*CTL* nun auch auf erweiterten Zustandsräumen, worunter auch solche, die auf markierten Heapkonfigurationen basieren, fallen, definiert:

Definition 89 (Semantik von Q*CTL* über erweiterten Zustandsräumen)
Sei $K = (Q, Q_0, \widetilde{\rhd}, \mathit{var}, \mathit{sel}, \cong)$ ein erweiterter objektorientierter Zustandsraum.

Ausgehend von Definition 43 sind dann die wesentlichen Änderungen, die nicht aus einfachem Ersetzen von q, vl mit q und π, vl mit π bestehen:

$q \models \mu = \nu$, wenn $[\![\mu]\!]_q = [\![\nu]\!]_q \neq \bot$.
$q \models \mathsf{A}\varphi$, wenn für jedes $\pi \in \widetilde{\mathit{Paths}}(q)$ gilt, dass $\pi \models \varphi$.
$q \models \mathsf{E}\varphi$, wenn es ein $\pi \in \widetilde{\mathit{Paths}}(q)$ gibt, so dass $\pi \models \varphi$.
$q \models \forall_{\mathcal{S}} x.\psi$, wenn für jedes $q' \in \{q \,\widetilde{\triangleright}_x\}$ gilt, dass $q' \models \psi$
$q \models \exists_{\mathcal{S}} x.\psi$, wenn es ein $q' \in \{q \,\widetilde{\triangleright}_x\}$ gibt, so dass $q' \models \psi$.
$\pi \models \mathsf{X}\varphi$, wenn $\pi[2\,...] \models \varphi$.
$\pi \models \varphi \,\mathsf{U}\, \theta$, wenn es ein $i \geq 1$ gibt, so dass $\pi[i\,...] \models \theta$ und
für alle $1 \leq j < i$ gilt $\pi[j\,...] \models \varphi$.
$\pi \models \varphi \,\mathsf{R}\, \theta$, wenn es entweder ein $i \geq 1$ gibt, so dass $\pi[i\,...] \models \varphi$ und
für alle $1 \leq j \leq i : \pi[j\,...] \models \theta$,
oder für alle $i \geq 1$ gilt $\pi[i\,...] \models \theta$.
$\pi \models \forall_{\mathcal{P}} x.\varphi$, wenn für jedes $q' \in \{q \,\widetilde{\triangleright}_x\}$ gilt, dass $\pi[1 \hookrightarrow q'] \models \varphi$
$\pi \models \exists_{\mathcal{P}} x.\varphi$, wenn es ein $q' \in \{q \,\widetilde{\triangleright}_x\}$ gibt, so dass $\pi[1 \hookrightarrow q'] \models \varphi$.

Es kann also auf die mühsam erarbeiteten Äquivalenzklassen nicht verzichtet werden, denn in $\pi[1 \hookrightarrow q']$ wird $\cong$ benötigt, um eine sinnvolle und wohldefinierte Logik zu erhalten.

Dadurch, dass sich der (nicht-erweiterte) Zustandsraum und der erweiterte Zustandsraum in ihrer Signatur so deutlich unterscheiden, ist es schwer, einen Äquivalenzbegriff zwischen den Logiken auf diesen beiden zu definieren und formal zu beweisen. Dennoch haben wir durch das schrittweise Vorgehen, in der wir die Logik vom einen in den anderen Zustandsraum überführt haben, versucht überzeugend darzustellen, dass es sich um eine (in gewissem Sinne) äquivalente Logikdefinition von Q*CTL* handelt.

4.5 Anpassung des Modelcheckings

Auch unser bereits für Q*CTL* eingeführtes Modelchecking lässt sich auf erweiterten Zustandsräumen verwenden, indem einige Änderungen vorgenommen werden. Bisher besteht eine Forderung aus nur einem Zustand aber mehreren Variablenbelegungen. Jetzt ist eine Forderung ein Menge von Paaren (q_i, φ_i), wobei alle q_i in einer Forderung äquivalent (bzgl. $\cong$) sein müssen. Dementsprechen müssen natülich die Beweisregeln angepasst werden; tiefergehende Änderungen sind jedoch nicht nötig. $R^{\forall}$ und $R^{\exists}$ nutzen dann ohne Schwierigkeit $\widetilde{\triangleright}_x$ für $x \in \mathit{Mar}_{\Sigma}$.

Das Modelchecking führt Pfadquantisierung auf Zustände und die Verwendung von disjunktiven Formelmengen zurück, was erst dadurch möglich ist, dass in Q*LTL nur A, nicht aber E auftritt. Für Q*CTL* wird im Falle eines E die gesamte Teilformel negiert, wobei die Negation durch Umschreiben der Formel bis zu den EP nach innen propagiert wird. Nach jedem X, das auch bei der Entfaltung von U und R entsteht, müssen so immer alle Nachfolger getestet werden, ohne dass am Anfang alle Pfade aufgezählt werden müssten.

Da die Semantik der Logik aber erst für $\pi[2\,...]$ schwierig wird, also die Pfade umgeschrieben werden müssen, tritt das im Modelchecking nun gar nicht mehr zu Tage. Wie schon bei der Logik festgestellt, garantieren nämlich die von uns geforderten Bedingungen an erweiterte

Zustandsräume, dass durch die schrittweise Anwendung von $\widetilde{\triangleright}_\varepsilon$ keine unerwünschten Pfade entstehen. Ummarkierungen mit $\widetilde{\triangleright}_x$ habe ebenso gerade das gewünschte Verhalten.

Der Modelchecking ist also, zusammenfassend, für erweiterte Zustandsräume sehr elegant, da nur noch $\widetilde{\triangleright}$ benötigt wird. Diese Ergebnis wurde aber erst durch in diesem Kapitel gelegten umfangreicheren theoretischen Grundlagen möglich.

5 Abschließendes

5.1 Stand der Forschung

In [4] wird PLTL (Propositional Linear Time Logic) um $\exists$ erweitert. Weil Negationen an beliebigen Stellen erlaubt sind, ist damit auch $\forall$ abgedeckt. Die Logik bleibt mit Blick auf die atomaren Propositionen aber relativ eingeschränkt: Es kann geprüft werden, ob zwei Logikvariablen gleich sind, ob das durch eine Logikvariable referenzierte Objekt neu entstanden ist, sowie ob das Objekt noch „am Leben“ ist. Wie in unserem Ansatz gibt es auch eine Objektkorrespondenzfunktion zwischen den Zuständen; außerdem wird ein Verfahren zum Modelchecking vorgestellt.

Die Autoren erweitern die Logik dann in [3]: Hinzu kommt eine Pointerauswertung, wobei aber jedes Objekt nur genau ein Nachfolgeobjekt (bzw. null) haben kann. Zusätzlich kann auf Programmvariablen zugegriffen werden und Erreichbarkeit geprüft werden. So können einfache verkettete Listen abstrahiert werden. Wir können in unserer Logik all die genannten Propositionen ohne weiteres hinzufügen und auf Heapkonfigurationen testen. Die Abstraktion von Juggrnaut ist aber bei weitem nicht nur auf verkette Listen beschränkt. In unserer Logik werden daher beliebige Selektoren unterstützt.

In [13] wird PLTL ganz ähnlich erweitert. Statt dem Heap werden aber die Läufe (traces) des Programms schrittweise erweitert und abstrahiert. So kann ein Zustandsraum mit unendlichen Pfaden mit abstrahierten Läufe endlich dargestellt werden; auf den Läufen können dann Formeln verifiziert werden. Um mit bei der Abstraktion verlorengegangene Informationen umzugehen, wird auf dreiwertige Logik zurückgegriffen.

Eine Erweiterung von PCTL auf Q^+CTL (dort nur QCTL genannt) wird in [9] vorgestellt. Dabei wird der Zustandsraum für jede mögliche Logikvariablenbelegung kopiert, was unseren Markierungen sehr ähnlich ist. Modelchecking von QCTL wird realisiert, indem QCTL-Formeln in CTL-Formeln auf diesem erweiterten Zustandsraum überführt werden. Hierfür kann dann ein herkömmlicher CTL-Modelchecking-Algorithmus benutzt werden. Neben dem Quantifizieren über Objekten ist auch das Quantifizieren über Teilmengen von Objekten möglich. Unser Ansatz ließe sich dahingehend erweitern. Für unser Ziel, schließlich mit der in Juggrnaut verwendeten Abstraktion und Überapproximation arbeiten zu wollen, ist der dort gezeigte Ansatz jedoch nicht geeignet; außerdem ist CTL dem von uns erweiterten LTL an Ausdruckskraft unterlegen, da wir im konkreten und ohne Isomorphie nur einen einzigen Pfad haben; in CTL aber wird gerade die gleichzeitige Nutzung verschiedener Temporaloperatoren eingeschränkt. Schließlich gibt es zwar Q*LTL, aber kein Q*CTL; da wir nun aber ganz Q*CTL* verifizieren können, umfasst unser Ansatz sowieso Q^+CTL.

Eine weitere Arbeit des selben Autors, [10], behandelt vor allem Graphabstraktionen über Objekten, es wird aber auch eine auf LTL aufbauende und um $\exists$ (und wegen $\neg$ auch $\forall$) erweiterte Logik eingeführt. Für einen endlichen Zustandsraum, der mit einer gegebenen Graph-

grammatik kompatibel ist, und eine Formel der Logik ist die wesentliche Aussage dann aber nur, dass entscheibar ist, mit exponentieller Worst-Case-Laufzeit, ob die Formel dem Zustandsraum gilt.

5.2 Zusammenfassung

Wir haben uns in dieser Arbeit damit beschäftigt, welche Logiken entstehen können, wenn man zu den bekannten temporalen Logiken CTL, LTL und deren gemeinsamer Obermenge CTL* zusätzlich zu den schon vorhandenen Pfadquantoren auch Objektquantoren einführt. Der von uns erarbeitete Überblick über die Logiken wurde hier vorgestellt.

Weiter waren wir besonders daran interessiert, inwiefern die bei Juggrnaut verwendete Abstraktion auf Logiken Auswirkungen hat. Wünschenswert war, dass die Logik Objekte durch Markierungen identifizieren kann; weiter schien es erstrebenswert, dass solche markierten Zustände, die in gewissem Sinne zu Kopien des Zustandsraumes führen, erst dann erstellt werden, wenn sie durch $\forall$ und $\exists$ nötig werden.

Wir haben dann unter anderem nach On-the-Fly-Modelcheckingalgorithmen gesucht und es ist uns gelungen, den in [2] vorgestellten Algorithmus sowohl im Bezug auf die theoretischen Sätze, die die Korrektheit des Algorithmus sicherstellen, als auch eine praktische Implementierung (in Pseudocode) zu erweitern. So können nun nicht nur Q*LTL-Formeln (oder gar nur QLTL), sondern ganz allgemeine Q*CTL*-Formeln verifiziert werden. Dazu kommt, dass die Erforschung des Zustandsraumes sich bei dem On-the-Fly-Algorithmus alleine auf die für den Beweis der Formel notwendigen Zustände beschränkt. Daher führt der Algorithmus in der *Praxis* zu guten Laufzeiten.

In einem weiteren Schritt in Richtung der in Juggrnaut verwendeten Abstraktion konnten wir schließlich die Logik als auch das Modelchecking auf erweiterte („markierte“) Zustandsräume ausweiten.

Aufgrund der verwendeten Objektkorrespondenzfunktion bestand im Rahmen dieser Arbeit außerdem der Bedarf nach einer genaueren Beschreibung von Zustandsräumen, denen isomorphe Heapkonfigurationen zugrunde liegen. Daher haben wir die in Juggrnaut verwendeten Zustandsräume, denen isomorphe Hypergraphen zugrunde liegen, hier sauber formalisiert; bisher wurde, etwa in [6], nur generell gesagt, dass zwischen isomorphen Hypergraphen nicht unterschieden wird.

5.3 Ausblick

Zum einen sollte es in Zukunft nicht mehr schwer fallen, quantifizierte Logiken auf Abstraktion und insbesondere Überapproximation zu definieren. Unter Überapproximation dürfte die Korrektheit der Teilmenge von Q*CTL*, die auf $\exists$ und E verzichtet, nicht allzu schwer zu zeigen sein. Wir gehen aber für die in Juggrnaut verwendete Abstraktion davon aus, dass die Existenz ($\exists$) eines Objektes in einem Zustand sichergestellt werden kann, indem aus der für die Abstraktion verwendeten Graphgrammatik zusätzliche Bedingungen ableitet werden. $\exists$ muss dabei nicht nur für eine einzige, sondern für eine bestimmte Teilmenge der blauen Kanten die Teilformel verifizierbar sein. Ergibt das Modelchecking $\not\models$, lässt sich daraus nicht viel ableiten: Ein Gegenbeispiel im Abstrakten (in der Überapproximation) muss im Konkreten noch lange nicht existieren. Ist das Ergebnis aber $\models$, dann ist damit bewiesen, dass kein konkreter Zustandraum die Formel verletzten kann.

Zum anderen scheint es erstrebenswert, das durch die Abstraktion verursachtes Unwissen in der Logik explizit zu machen. Dazu wird, Kleene's K3 [8] folgend, ein dritter Logikzustand eingeführt, der etwa **ii** (‚indeterminate', unbestimmt) genannt wird. Im Hinblick auf die Verifikation lässt sich sagen, dass dadurch **ff** aufgesplittet würde: Liefert das Modelchecking **ii**, so kann das weiterhin als **ff** gelesen werden, *wenn* keine weiteren Negationen darauf angewendet werden. Doch gerade das wäre in unserer Logikdefinition sichergestellt, da Negationen nur innerhalb von SF auftreten. Eine entsprechende Semantik von „3-Q*CTL*" lässt sich ohne große Schwierigkeiten angeben. Die Anpassung eines Modelchecking-Algorithmus auf eine solche Semantik könnte Gegenstand einer weiteren Forschungsarbeit sein.

Gerne hätten wir noch gezeigt, dass K^1 und K^2 durch Q*CTL* nicht unterscheidbar sind. Allerdings sind die Anforderungen für Zustandsraumisomorphie zu restriktiv. Wir vermuten, dass Bisimilarität für unsere Zustandsräume sinnvoll definierbar ist und auch ein Zusammenhang zwischen bisimilaren Zusandsräumen und Unterscheidbarkeit durch Q*CTL* besteht, da bei CTL*-Unterscheidbarkeit diese genau mit Bisimilarität zusammenfällt. Wir gehen davon aus, dass dann K^1 und K^2 bisimilar sind, haben es jedoch noch nicht formal gezeigt.

Literaturverzeichnis

[1] Baier, Christel ; Katoen, Joost P.: *Principles of Model Checking.* The MIT Press, 2008. – ISBN 026202649X

[2] Bhat, Girish ; Cleaveland, Rance ; Grumberg, Orna: Efficient On-the-Fly Model Checking for CTL*. In: *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science.* Washington, DC, USA : IEEE Computer Society, 1995. – ISBN 0–8186–7050–6, 388

[3] Distefano, Dino ; Katoen, Joost-Pieter ; Rensink, Arend: Who is Pointing When to Whom? In: *FSTTCS* Bd. 3328, 2004, S. 250–262

[4] Distefano, Dino ; Rensink, Arend ; Katoen, Joost-Pieter: Model Checking Birth and Death. In: *Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science: Foundations of Information Technology in the Era of Networking and Mobile Computing.* Deventer, The Netherlands, The Netherlands : Kluwer, B.V., 2002 (TCS '02). – ISBN 1–4020–7181–7, 435–447

[5] Heinen, Jonathan ; Jansen, Christina: Juggrnaut – An Abstract JVM / RWTH Aachen. 2011 (Aachener Informatik-Berichte AIB-2011-21). – Forschungsbericht. – ISSN 0935–3232

[6] Heinen, Jonathan ; Jansen, Christina ; Katoen, Joost-Pieter ; Noll, Thomas: Juggrnaut: Using Graph Grammars for Abstracting Unbound Heap Structures. In: *Under consideration for Math. Struct. in Comp. Science*, 2012

[7] Heinen, Jonathan ; Noll, Thomas ; Rieger, Stefan: Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. In: *To be published in: Harnessing Theories for Tool Support in Software, TTSS 2009*, Elsevier, 2009 (ENTCS)

[8] Kleene, Stephen C.: *Principles of Model Checking.* Amsterdam : North-Holland Publ. Co., 1962

[9] Rensink, Arend: Model Checking Quantified Computation Tree Logic. In: Baier, Christel (Hrsg.) ; Hermanns, Holger (Hrsg.): *CONCUR 2006 – Concurrency Theory* Bd. 4137. Springer Berlin / Heidelberg, 2006, S. 110–125

[10] Rensink, Arend: Explicit State Model Checking for Graph Grammars. In: Degano, Pierpaolo (Hrsg.) ; De Nicola, Rocco (Hrsg.) ; Meseguer, José (Hrsg.): *Concurrency, Graphs and Models* Bd. 5065. Springer Berlin / Heidelberg, 2008, S. 114–132

[11] Rozier, K.Y.: Linear Temporal Logic Symbolic Model Checking. In: *Computer Science Review* (2010). http://dx.doi.org/doi:10.1016/j.cosrev.2010.06.002. – DOI doi:10.1016/j.cosrev.2010.06.002

[12] Tarjan, Robert: Depth-First Search and Linear Graph Algorithms. In: *SIAM Journal on Computing* 1 (1972), Nr. 2, S. 146–160

[13] Yahav, Eran ; Reps, Thomas ; Sagiv, Mooly ; Wilhelm, Reinhard: Verifying temporal heap properties specified via evolution logic. In: *Proceedings of the 12th European conference on Programming*. Berlin, Heidelberg : Springer-Verlag, 2003 (ESOP'03). – ISBN 3–540–00886–1, 204–222