Separationslogik

Florian Chudigiewitsch, 3216390

BACHELORARBEIT

Informatik B. Sc.

Gottfried Wilhelm Leibniz Universität Hannover

Institut für Theoretische Informatik

Erstprüfer: Prof. Dr. Heribert Vollmer

Zweitprüfer: Dr. rer. nat. Arne Meier

Betreuer: M. Sc. Anselm Haak

im August 2018

Inhaltsverzeichnis

Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Gottfried Wilhelm Leibniz Universität Hannover, am 15. August 2018

Florian Chudigiewitsch

Kurzfassung

Die Separationslogik ist ein Kalkül zur Verifikation eines Algorithmus, welcher in einer Sprache, die Speicher freiadressierbar zur Verfügung stellt, geschrieben wurde. Freiadressierbarer Speicher wird dabei meist über Zeiger angesprochen; die untersuchte Sprache verfügt demnach über eine Form der Heapinteraktion im Sinne der Sprache C.

Im Gegensatz zur Prädikatenlogik wird bei den hinzugefügten Operationen die Disjunktheit von Speicherzellen vorausgesetzt, was die erforderlichen Klauseln deutlich reduziert und auch das Mindset eines Programmierers eher widerspiegelt.

In der vorliegenden Arbeit soll ein Überblick über diese Logik gegeben werden, indem zuerst eine Sprache mit Heapinteraktion definiert wird, anhand derer dann die theoretischen Grundlagen der Separationslogik erklärt werden. Anschließend werden einige Verifikationen mit der Separationslogik durchgeführt, und schließlich ein Ausblick auf die Anwendbarkeit der Logik auf verschiedene praxisnahe Beispiele gegeben sowie mögliche Erweiterungen der Logik diskutiert.

Kapitel 1

Einleitung

Bereits im Jahre 1969 – die Informatik als eigenständige wissenschaftliche Disziplin steckte noch in den Kinderschuhen – veröffentlichte C. A. R. Hoare ein Paper, in welchem er die Axiomatisierung einer einfachen Programmiersprache angab [Hoa69].

Er begründetete die Bedeutung dieser Axiomatisierung damit, dass nur durch eine formale Verifikation eines Programmes dessen richtige Funktionsweise sichergestellt werden kann, und fehlerhafte Programme fatale Folgen haben können. Hoare nennt dazu beispielhaft: „a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war“.

In den darauffolgenden Jahrzehnten sollten sich diese Behauptungen in erstaunlichem Umfang bewahrheiten: 1999 verlor die NASA den 125 Millionen Dollar teuren Mars Orbiter durch einen Programmfehler [Hot99], im Jahre 2005 brach das Oberbecken des Pumpspeicherkraftwerks Taum Sauk [Wik18a], da der Pumpvorgang trotz des Erreichens des maximalen Wasserstands nicht abgebrochen wurde. 2015 stürzte ein Airbus A400M aufgrund eines Softwarefehlers ab [Kel15] und im Jahr 1983, mitten im Kalten Krieg, führte ein Computerfehler fast zum Ausbruch eines 3. Weltkriegs [Tho13].

Ein Problem, welches auch bereits in dem Paper von Hoare genannt wurde, ist die Behandlung von Sprachen, die *freiadressierbaren Speicher* bereitstellen. Hierbei muss in der Prädikatenlogik für *jedes* Paar von Speicheradressen deren Ungleichheit ausgedrückt werden (hierauf wird vor allem in Abschnitt 3.2 eingegangen), was zu einem besonders hohen Anstieg von logischen Aussagen in Abhängigkeit der verwendeten Speicherzellen führt.

Eine Logik, die hierfür Abhilfe schaffen soll, ist die sogennante *Separationslogik*, welche zum ersten Mal im Jahre 2002 von John C. Reynolds vorgestellt wurde und über die in dieser Arbeit ein Überblick gegeben wird. Sie stellt

eine Erweiterung der Hoare-Logik dar, indem sie einige Operationen hinzufügt, welche die Disjunktheit von Speicherzellen voraussetzen.

Als Motivation und Grundlage für die Separationslogik wird zuerst die Programmiersprache $\mathcal{L}$, eine Sprache mit Zeigerarithmetik, vorgestellt. Diese ist äußerst einfach aufgebaut, um die Grundlagen der Separationslogik möglichst verständlich zeigen zu können. Die Erkenntnisse, welche die Betrachtung der Sprache liefert, lassen sich zudem sehr leicht auf bekannte und häufig verwendete Zeigersprachen – allen voran C/C++ – übertragen.

Danach wird die Hoare-Logik, die Grundlage der Separationslogik, vorgestellt und erklärt, um schließlich auf die Separationslogik selbst eingehen zu können.

Wir werden sehen, wie die Separationslogik die Prädikaten- und Hoare-Logik mit neuen Verknüpfungsoperationen erweitert und welche Auswirkungen dies auf die Interaktion mit unserem Speichermodell hat; insbesondere auf die Interaktion mit dem Heap, da dies die Prämisse für die Erstellung der Separationslogik darstellt.

Anschließend werden wir einige einfache Verifikationen von Programmen aus $\mathcal{L}$ durchführen, um ein Verständnis von der Methodik dieser Verifizierungen bekommen zu können.

Schließlich werden einige Anwendungen der Separationslogik oder direkter Derivate aufgeführt, um die praktische Relevanz der Separationslogik zu verdeutlichen, sowie ein Ausblick auf mögliche Erweiterungen und Anpassungen gegeben.

Kapitel 2

Die Programmiersprache $\mathcal{L}$

Da es die Motivation der Separationslogik ist, die Programme einer Programmiersprache mit Zeigerarithmetik zu verifizieren, wollen wir nun eine solche definieren. Diese richtet sich vor allem nach der in [SN17]. Wir werden sie dementsprechend im Folgenden auch $\mathcal{L}$ nennen.

2.1 Speichermodell

Wir verwenden ein Speichermodell, welches dem der Programmiersprache C sehr nahe kommt: Es verfügt über einen Stack s und einen Heap h.

Definition 2.1 (Stack)
Sei $s\colon \mathsf{Var} \rightharpoonup \mathbb{N}_0$ *eine partielle Funktion, wobei* Var *die endliche Menge aller Variablenidentifier ist. Dann nennen wir* s Stack.
Wir schreiben abkürzend

$$s[x_1 : v_1, \ldots, x_n : v_n]$$

für

$$\bigwedge_{1 \leq i \leq n} s(x_i) = v_i,\ n, v_i \in \mathbb{N},\ x_i \in \mathsf{Var}, i \in \mathbb{N}.$$

Definition 2.2 (Heap)
Sei $h\colon \mathsf{Loc} \rightharpoonup \mathbb{N}$ *eine partielle Funktion, wobei* $\mathsf{Loc} \subseteq \mathbb{N}$ *die Menge aller Adressen ist. Dann nennen wir* h Heap.
Wir schreiben abkürzend

$$h[a_1 : v_1, \ldots, a_n : v_n]$$

für

$$\bigwedge_{1 \leq i \leq n} h(a_i) = v_i,\ n, v_i \in \mathbb{N},\ a_i \in \mathsf{Loc}, i \in \mathbb{N}.$$

Haben zwei Heaps h_0 *und* h_1 *disjunkte Urbildmengen, schreiben wir* $h_0 \perp h_1$.

Um eine kompakte Darstellung der Zustände von Stack und Heap zu ermöglichen, definieren wir nun für beide eine alternative Darstellung. Diese ist eine Menge, welche aus Tupeln besteht, die wiederum aus einer Variable bzw. einer Adresse und dem Funktionswert für diese Eingabe bestehen. Falls eine Verwechslungsgefahr besteht schreiben wir s_ξ bzw. h_ξ; wird aus dem Kontext die Darstellung klar, nur s bzw. h.

Definition 2.3 (Mengendarstellung von Stack und Heap)

$$s_\xi := \{(x, n)^* \mid s(x) = n, x \in \mathsf{Var}, n \in \mathbb{N}_0\}$$

$$h_\xi := \{(a, n)^* \mid h(a) = n, a \in \mathsf{Loc}, n \in \mathbb{N}_0\}$$

Im folgenden Beispiel werden die Darstellungen von Stack und Heap aufgezeigt:

Beispiel 2.1 (Darstellung von Stack und Heap)
Mögliche Zustände von s und h sind:

$$s_\xi := \{(x, 5), (y, 23), (z, 0)\}$$

$$h_\xi := \{(20, 5), (21, 23), (22, 0)\}$$

Dann gilt zum Beispiel:

$$s(x) = 5 \wedge s(z) = 0 \text{ } bzw. \text{ } s[x : 5, z : 0]$$

$$h(20) = 5 \wedge h(21) = 23 \text{ } bzw. \text{ } h[20 : 5, 21 : 23]$$

Der Stack bildet den statischen Speicher des Programms, der Heap stellt den dynamischen Speicher bereit. Der aktuelle Zustand des Stacks bildet zusammen mit dem aktuellen Zustand des Heaps den *Berechnungszustand* als Tupel (s, h). Die Menge aller möglichen Zustände nennen wir States.

Definition 2.4 (Definitionsbereich der Speicheradressen)
Wir bezeichnen den Definitionsbereich von h als

$$\text{dom } h.$$

Es gilt

$$\text{dom } h \subseteq \mathsf{Loc}.$$

Wir schreiben $h\rceil A$, wenn $\text{dom } h = A \subseteq \mathsf{Loc}$.

Der Definitionsbereich von h ist für zukünftige Betrachtungen besonders wichtig, da er die Menge der im Heap allokierten Speicherzellen darstellt. Andere Bezeichnungen für dom h sind *allokierter Speicher* und *initialisierter Speicher*. Sind diese in C und C++ noch zwei getrennte Aktionen, fassen wir sie hier zu einer zusammen, entsprechend des vom C++ Erfinder Bjarne Stroustrup empfohlenen RAII-Prinzips („Resource Acquisition Is Initialisation“, vgl. [Sto16]).

2.2 Befehle

Wir haben nun ein Speichermodell für unsere Sprache und können damit verschiedene Zustände darstellen. Es fehlt uns noch die Möglichkeit, Zustände zu wechseln. Dies wird durch Befehle der Programmiersprache erreicht. Die Auswirkungen eines Befehls auf den Berechnungszustand werden durch folgende Übergangsrelation formalisiert:

Definition 2.5 (Übergangsrelation β)
Seien s und s' Stacks, h und h' Heaps und c ein Befehl. Dann ist ein Schritt in der Übergangsrelation definiert als:

$$\beta\colon \langle c,\ (s,h)\rangle \rightarrowtail (s',h').$$

Ist der Zustand (s',h') in endlich vielen Relationsschritten von $\langle c,\ (s,h)\rangle$ aus erreichbar, schreiben wir:

$$\hat{\beta}\colon \langle c,\ (s,h)\rangle \overset{*}{\rightarrowtail} (s',h').$$

Für den Spezialfall des Abbruchs eines Befehls schreiben wir:

$$\beta\colon \langle c,\ (s,h)\rangle \rightarrowtail \textbf{abort}.$$

Dieser erfolgt gdw. ein Parameter von s oder h nicht im Definitionsbereich liegt.

Definition 2.6 (Ausdruck)
Ein Ausdruck e ist ein Konstrukt der Programmiersprache, welches nur im Kontext des Zustands des Stacks s zu einem Wert v evaluiert wird. Formal:

$$s \models e \Downarrow v$$

Wir unterscheiden zwischen Booleschen sowie arithmetischen Ausdrücken. Für Boolesche Ausdrücke gilt $v \in \{\textbf{true}, \textbf{false}\}$, für arithmetische gilt $v \in \mathbb{N}$. Die Grammatik für arithmetische und Boolesche Ausdrücke wird in Definition 2.7 aufgeführt.

Unsere Sprache mit Heapinteraktion soll nun erst intuitiv, dann formal definiert werden. Tabelle 2.1 gibt einen Überblick über alle Befehle der Programmiersprache zusammen mit der intuitiven Beschreibung ihrer Semantik. Diese wird dann in Abschnitt 2.3 und 2.4 formalisiert. Es notieren die Symbole x eine Variable, e und e' einen Ausdruck, b einen Booleschen Ausdruck und c einen Befehl.

Skip `skip`
Lässt den Berechnungszustand unverändert.

Assignment x `:=` e
x wird der Wert zu dem e evaluiert zugewiesen.

Komposition c_1`;...;`c_n
Die Befehle c_1 bis c_n werden sequentiell ausgeführt.

Kondition `if` b `{` c_1 `}` `else {` c_2 `}`
Wenn im aktuellen Berechnungszustand b zu **true** evaluiert, so wird c_1 ausgeführt, ansonsten c_2.

Schleife `while` b `{` c `}`
c wird ausgeführt, bis b zu **false** evaluiert.

Allokation x `:= cons(`$e_1, \ldots, e_n$`)`
Es werden n aufeinander folgende Speicherzellen im Heap reserviert und mit den Ausdrücken e_1 bis e_n initialisiert. Die Adresse der ersten reservierten Speicherzelle wird dann in x gespeichert. Sind nicht genügend Speicherzellen vorhanden, bricht das Programm ab.

Lookup x `:= [`e`]`
Speichert den Wert, welcher im Heap an der Adresse e gespeichert ist, in x. Liegt die Adresse außerhalb des vom Heap adressierbaren Speichers oder ist die adressierte Speicherzelle nicht initialisiert, so bricht das Programm ab.

Mutation `[`e`]` `:=` e'
Speichert den Wert, zu dem e' evaluiert, in die Speicherzelle mit Adresse e. Liegt diese nicht im vom Heap adressierbarem Speicherbereich, so bricht das Programm ab.

Deallokation `free(`e`)`
Gibt die Speicherzelle mit der Adresse e frei. Ist diese nicht adressierbar, so bricht das Programm ab.

Tabelle 2.1: Befehle der Programmiersprache

Da die definierte Sprache eine echte Obermenge der in [Vol17a] definierten WHILE-Sprache darstellt und alle Erweiterungen lediglich die Speicheradressierung betreffen, ist leicht zu sehen, dass die $\mathcal{L}$-Berechenbarkeit äquivalent zur Turing-Berechenbarkeit und damit zur C-Berechenbarkeit ist.

Die Syntax der Sprache kann durch die folgende kontextfreie Grammatik formal definiert werden:

Definition 2.7 (Grammatik von $\mathcal{L}$)

$$G = (\{\langle cmd\rangle, \langle aexp\rangle, \langle bexp\rangle, \langle var\rangle\}, T, P, \langle cmd\rangle)$$

T, das Teminalalphabet, entspricht dem ANSI-Alphabet und P ist definiert als:

$$\begin{aligned}
\langle cmd\rangle \rightarrow\ & \text{“}\texttt{skip}\text{”} \\
& \mid \langle cmd\rangle \text{ “;” } \langle cmd\rangle \\
& \mid \text{“}\texttt{if}\text{” } \langle bexp\rangle \text{ “\{” } \langle cmd\rangle \text{ “}\}\texttt{else}\{\text{” } \langle cmd\rangle \text{ “}\}\text{”} \\
& \mid \text{“}\texttt{while}\text{” } \langle bexp\rangle \text{ “\{” } \langle cmd\rangle \text{ “}\}\text{”} \\
& \mid \langle var\rangle \text{ “ := } \texttt{cons}\text{(” } \langle aexp\rangle\text{“, ”} \cdots\text{“, ”} \langle aexp\rangle \text{ “)”} \\
& \mid \langle var\rangle \text{ “ := [” } \langle aexp\rangle \text{ “]”} \\
& \mid \text{“[” } \langle aexp\rangle \text{ “] := ” } \langle aexp\rangle \\
& \mid \text{“}\texttt{free}\text{(” } \langle aexp\rangle \text{ “)”} \\
\langle aexp\rangle \rightarrow\ & \texttt{int} \\
& \mid \langle var\rangle \\
& \mid \langle aexp\rangle \text{ “ + ” } \langle aexp\rangle \\
& \mid \langle aexp\rangle \text{ “ − ” } \langle aexp\rangle \\
& \mid \langle aexp\rangle \text{ “ ∗ ” } \langle aexp\rangle \\
& \mid \langle aexp\rangle \text{ “/” } \langle aexp\rangle \\
& \mid \langle aexp\rangle \text{ “\%” } \langle aexp\rangle \\
\langle bexp\rangle \rightarrow\ & \texttt{true} \mid \texttt{false} \\
& \mid \langle aexp\rangle \text{ “ = ” } \langle aexp\rangle \\
& \mid \text{“!” } \langle bexp\rangle \\
& \mid \langle bexp\rangle \text{ “\&” } \langle bexp\rangle \\
& \mid \langle bexp\rangle \text{ “|” } \langle bexp\rangle \\
& \mid \langle bexp\rangle \text{ “−> ” } \langle bexp\rangle
\end{aligned}$$

Die originale Grammatik von $\mathcal{L}$ nach [SN17] wurde kosmetisch etwas angepasst. Durch die geschweiften Klammern bei den „while“– und „if“–Regeln ist es für einen Parser einfacher zu unterscheiden, ob der Befehl schon ein neuer ist oder noch zum entsprechenden Block gehört. Damit sie einfacher am Computer eingegeben werden können, wurden die logischen Operatoren der C-Syntax nachempfunden, „–>“ stellt jedoch statt eines Elementzugriffes eine logische Implikation dar. $\langle var\rangle$ ist ein Variablenidentifier, `int` eine natürliche Zahl.

2.3 Zeigerunabhängige Semantik

Wir wollen nun zuerst die Semantik jener Sprachelemente formalisieren, die nicht von Zeigern abhängen. Diese Vorgehensweise wird im weiteren Verlauf der Arbeit angewandt, da für diesen Teil der Sprache lediglich die Hoare-Logik und noch nicht die Separationslogik benötigt wird. Definition 2.8 zeigt die Semantik zu den entsprechenden Befehlen der Sprache, die sich direkt aus der intuitiven Semantik ableitet. Für `if` und `while` werden zwei Regeln benötigt, je nachdem, ob die Kondition zu wahr oder falsch evaluiert.

Definition 2.8 (Semantik I)

$$\frac{}{\langle \textbf{skip},\ (s,h)\rangle \rightarrowtail (s,h)}\ \textit{Skip}$$

$$\frac{s \models e \Downarrow v}{\langle \textbf{x} := \textbf{e},\ (s,h)\rangle \rightarrowtail (s[x:v],h)}\ \textit{Assign}$$

$$\frac{\langle c_1,(s,h)\rangle \rightarrowtail (s',h')}{\langle c_1; c_2,(s,h)\rangle \rightarrowtail \langle c_2,\ (s',h')\rangle}\ \textit{Seq}$$

$$\frac{s \models e \Downarrow \textbf{true}\quad \langle c_1,(s,h)\rangle \rightarrowtail (s',h')}{\langle \textbf{if}\ e\ \{\ c_1\ \}\ \textbf{else}\ \{\ c_2\ \},\ (s,h)\rangle \rightarrowtail (s',h')}\ \textit{If-T}$$

$$\frac{s \models e \Downarrow \textbf{false}\quad \langle c_2,(s,h)\rangle \rightarrowtail (s',h')}{\langle \textbf{if}\ e\ \{\ c_1\ \}\ \textbf{else}\ \{\ c_2\ \},\ (s,h)\rangle \rightarrowtail (s',h')}\ \textit{If-F}$$

$$\frac{s \models e \Downarrow \textbf{true}\quad \langle c,(s,h)\rangle \rightarrowtail (s',h')\quad \langle \textbf{while}\ \ e\ \{\ c\ \},\ (s',h')\rangle \rightarrowtail (s'',h'')}{\langle \textbf{while}\ \ e\ \{\ c\ \},\ (s,h)\rangle \rightarrowtail (s'',h'')}\ \textit{W-T}$$

$$\frac{s \models e \Downarrow \textbf{false}}{\langle \textbf{while}\ \ e\ \{\ c\ \},\ (s,h)\rangle \rightarrowtail (s,h)}\ \textit{W-F}$$

2.4 Zeigerabhängige Semantik

Nun folgen die Sprachelemente, die Zeiger nutzen. Hier machen wir auch Gebrauch von der **abort**-Regel, die laut Definition zum Abbruch des Befehls führt, wenn auf nicht allokierten Speicher zugegriffen wird. Man beachte vor allem die Definition von zwei Inferenzregeln pro Sprachelement, je nachdem, ob die Operation ausgeführt werden kann, oder abgebrochen werden muss, da die angegebene Adresse im Heap nicht definiert ist.

Definition 2.9 (Semantik II)

$$\frac{s \models e_1 \Downarrow v_1, \ldots, s \models e_n \Downarrow v_n \qquad l, \ldots, l+n-1 \in \mathsf{Loc} - \mathsf{dom}\ h}{\langle \mathbf{x} := \mathbf{cons}(e_1, \ldots, e_n),\ (s,h)\rangle \rightarrowtail (s[x:l], h[l:v_1, \ldots, l+n-1:v_n])}\ \mathit{Alloc}$$

$$\frac{s \models e_1 \Downarrow v_1, \ldots, s \models e_n \Downarrow v_n \qquad l, \ldots, l+n-1 \notin \mathsf{Loc} - \mathsf{dom}\ h}{\langle \mathbf{x} := \mathbf{cons}(e_1, \ldots, e_n),\ (s,h)\rangle \rightarrowtail \mathbf{abort}}\ \mathit{AllocFail}$$

$$\frac{s \models e \Downarrow v \qquad v \in \mathsf{dom}\ h}{\langle \mathbf{x} := [\mathbf{e}],\ (s,h)\rangle \rightarrowtail (s[x:h(v)], h)}\ \mathit{Look}$$

$$\frac{s \models e \Downarrow v \qquad v \notin \mathsf{dom}\ h}{\langle \mathbf{x} := [\mathbf{e}],\ (s,h)\rangle \rightarrowtail \mathbf{abort}}\ \mathit{LookFail}$$

$$\frac{s \models e \Downarrow v \qquad s \models e' \Downarrow v' \qquad v \in \mathsf{dom}\ h}{\langle [\mathbf{e}] := \mathbf{e}',\ (s,h)\rangle \rightarrowtail (s, h[v:v'])}\ \mathit{Mut}$$

$$\frac{s \models e \Downarrow v \qquad v \notin \mathsf{dom}\ h}{\langle [\mathbf{e}] := \mathbf{e}',\ (s,h)\rangle \rightarrowtail \mathbf{abort}}\ \mathit{MutFail}$$

$$\frac{s \models e \Downarrow v \qquad v \in \mathsf{dom}\ h}{\langle \mathbf{free}(e),\ (s,h)\rangle \rightarrowtail (s, h\rceil(\mathsf{dom}\ h\ -\ \{v\}))}\ \mathit{Free}$$

$$\frac{s \models e \Downarrow v \qquad v \notin \mathsf{dom}\ h}{\langle \mathbf{free}(e),\ (s,h)\rangle \rightarrowtail \mathbf{abort}}\ \mathit{FreeFail}$$

Nun haben wir eine komplette syntaktisch sowie semantisch formale Sprachdefinition, für die wir eine Logik kreieren können, die uns erlaubt, eine Verifikation durchzuführen.

Kapitel 3

Hoare-Logik

Wir wollen nun als erstes die Axiomatisierung und Erstellung der Inferenzregeln derjenigen Sprachelemente vornehmen, die sich nicht der Zeiger bedienen und verwenden dafür die Hoare-Logik [Hoa69], die im Jahre 1969 von C. A. R. Hoare den Grundstein für die Axiomatisierung von Computerprogrammen legte. Da wir im weiteren Verlauf immer wieder Aussagen über den Berechnungszustand machen werden, müssen diese ebenfalls formalisiert werden. Eine Assertion ist wie folgt definiert:

Definition 3.1 (Assertion)
Eine Assertion P *ist eine prädikatenlogische Aussage über den Berechnungszustand eines Programms. Wir schreiben* $[\![P]\!]\ s\ h$ *um den Wert, zu dem P in Abhängigkeit von einem gegebenen Berechungszustand* (s, h) *evaluiert, zu repräsentieren.*

3.1 Hoare Tripel

Die Hoare-Logik setzt sogenannte *Hoare Tripel* ein. Diese sind wie folgt aufgebaut:

$$\{P\}Q\{R\}.$$

Sie sind folgendermaßen definiert:

Definition 3.2 (Hoare Tripel)
Seien P und R Assertions und Q ein Programm. Es gilt {P}Q{R}, gdw.

$$\forall (s,h) \in \mathsf{States}, [\![P]\!]\ s\ h \implies \overbrace{\neg(\langle Q,\ (s,h)\rangle \overset{*}{\rightarrowtail} \mathbf{abort})}^{(1)}$$
$$\wedge\ \underbrace{(\forall (s',h') \in \mathsf{States}, (\langle Q,\ (s,h)\rangle \overset{*}{\rightarrowtail} (s',h')) \Rightarrow [\![R]\!]\ s'\ h')}_{(2)}$$

Teil (1) der Definition sichert ab, dass das Programm korrekt ausgeführt wird. Teil (2) sagt aus, dass R in allen Berechnungszuständen, die durch die Ausführung von Q herbeigeführt werden, gilt.

Wir nennen im Folgenden P *Präkondition* und R *Postkondition*, das Hoare Tripel wird auch *Spezifikation für* Q genannt. Sollten keine Präkonditionen existieren, schreiben wir

$$\{\mathbf{true}\}Q\{R\}.$$

Formeln, auf die das Programm Q keinen Einfluss nimmt, nennen wir *Invarianten*.

Ursprünglich setzt Hoare das Programm in Klammern, wir werden jedoch die Prä- und Postkonditionen in Klammern setzen, da sich diese Schreibweise in späterer Literatur durchgesetzt hat.

Wir wollen nun die Axiome, die wir im weiteren Verlauf der Arbeit verwenden werden und für deren Definition die Hoare-Logik ausreicht, angeben.

Im Folgenden bezeichnet x einen Variablenidentifier, e einen Ausdruck, $Q_i, i \in \mathbb{N}$ Programme von $\mathcal{L}$, und R, P und $P[e/x]$ Assertions, wobei $P[e/x]$ aus P durch die Ersetzung von allen Vorkommen von x in P durch e gewonnen wird.

Definition 3.3 (Axiome für zeigerunabhängige Sprachelemente)

$$\frac{}{\{P\}\ skip\ \{P\}}\ skip$$

$$\frac{}{\{P[e/x]\}\ x := e\ \{P\}}\ assign$$

$$\frac{\{P\}Q_1\{R\} \quad \{R\}Q_2\{S\}}{\{P\}Q_1; Q_2\{S\}}\ seq$$

$$\frac{\{P \wedge e\}Q_1\{R\} \quad \{P \wedge \neg e\}Q_2\{R\}}{\{P\}\mathbf{if}\ e\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2\{R\}}\ if$$

$$\frac{\{P \wedge e\}Q\{P\}}{\{P\}\mathbf{while}\ e\ \mathbf{do}\ Q\{P \wedge \neg e\}}\ while$$

3.2 Inferenzregeln der Hoare-Logik

Nachdem wir nun die Axiome der Hoare-Logik definiert haben, brauchen wir noch eine Menge an Inferenzregeln.
Die in der vorliegenden Arbeit verwendeten Inferenzregeln legen die Inferenzregeln der Prädikatenlogik aus [Vol17b] zugrunde und erweitern diese.

Diese Erweiterungen sollen hier kurz formal notiert und – soweit erforderlich – umgangssprachlich erklärt werden:

Aus der Definition 3.2 lassen sich direkt die Inferenzregeln für den *Strengthening Precedent* und den *Weakening Consequent* herleiten.

Definition 3.4 (Strengthening Precedent)

$$\frac{P \to R \quad \{R\}Q\{S\}}{\{P\}Q\{S\}}\, SP$$

Definition 3.5 (Weakening Consequent)

$$\frac{\{P\}Q\{R\} \quad R \to S}{\{P\}Q\{S\}}\, WC$$

Eine weitere nützliche Regel ist die *Existential Quantification*:

Definition 3.6 (Existential Quantification)

$$\frac{\{P\}Q\{R\}}{\{\exists v.P\}Q\{\exists v.R\}}\, EQ$$

Wobei $v \notin \text{frei}(Q)$.

frei(Q) ist hier die Menge der *freien Variablen in* Q (vgl. [Vol17b]).

Führt die Ausführung von Q zu zwei unterschiedlichen Postkonditionen, kann man diese mit einer Konjunktion verbinden; es ergibt sich die *Conjunction*-Regel:

Definition 3.7 (Conjunction)

$$\frac{\{P\}Q\{R_1\} \quad \{P\}Q\{R_2\}}{\{P\}Q\{R_1 \wedge R_2\}}\, CJ$$

Wir können auch bestimmte Variablen substituieren:

Definition 3.8 (Substitution)
Sei

$$\delta \colon v_1 \to e_1, \ldots, v_n \to e_n,$$

mit $v_i \in \text{Mod}(Q) \Rightarrow \forall i \neq j, e_i \notin \text{frei}(e_j)$ *eine Substitution, so gilt:*

$$\frac{\{P\}Q\{R\}}{\{P/\delta\}Q/\delta\{R/\delta\}}\, S$$

Mod(Q) ist dabei die Menge der Variablen, welche durch die Ausführung des Programms Q *modifiziert* werden.

Die oben genannten Regeln können ohne Änderungen auch in der Separationslogik verwendet werden. Allerdings gilt dies nicht für die in der Hoare-Logik verwendete *rule of constancy*: Gilt $\text{Mod}(Q) \cap \text{frei}(S) = \emptyset$ – das Programm Q verändert also keine der freien Variablen in einer Assertion S – so können wir die rule of constancy anwenden. Gilt also vor der Ausführung des Programms $P \wedge S$, so muss danach $R \wedge S$ gelten. Die daraus resultierende Inferenzregel lautet:

Definition 3.9 (Constancy in der Hoare-Logik)

$$\frac{\{P\}Q\{R\}}{\{P \wedge S\}Q\{R \wedge S\}}\, C$$

Verwendet man jedoch Sprachen mit freiadressierbarem Speicher, kann diese Regel zu ungültigen Programmspezifikationen führen. Betrachten wir die Anwendung der Rule of Constancy auf folgendes Hoare Tripel unter Verwendung des Heaps:

Beispiel 3.1

$$\frac{\{h[(a+i):4]\}\ [a+i] := 8\ \{h[(a+i):8]\}}{\{h[(a+i):4] \wedge h[(a+j):4]\}\ [a+i] := 8\ \{h[(a+i):8] \wedge h[(a+j):4]\}}\, C$$

Die aus der Inferenzregel resultierende Spezifikation ist ungültig, da nicht explizit $i \neq j$ angegeben wurde. Will man in der Prädikatenlogik also mit dem Heap arbeiten, muss die Disjunktheit aller Speicherzellen *explizit* angeben werden, um undefinierte Seiteneffekte auszuschließen. Da dies paarweise für alle Adressen geschehen muss, würde die Anzahl der benötigten Aussagen quadratisch im Verhältnis zur Anzahl der Speicherzellen wachsen. Dieses Problem wird durch die Separationslogik gelöst.

Kapitel 4

Separationslogik

4.1 Neue Assertions

Die Separationslogik wird verwendet, um die Interaktionen mit dem Heap beschreiben zu können, ohne dass dadurch ungültige Spezifikationen entstehen können. Dies wird durch eine Erweiterung der prädikatenlogischen Assertions um eine neue Konstante sowie neuer Verknüpfungen erreicht.

Definition 4.1 (Leerer Heap)
Es gilt **emp** *gdw.* $\text{dom}\ h = \emptyset$.

Definition 4.2 (Zeiger)
Seien e_1 *und* e_2 *Ausdrücke. Es gilt*

$$[\![e_1 \mapsto e_2]\!]\ s\ h,$$

gdw. $(s \models e_1 \Downarrow v_1) \wedge (s \models e_2 \Downarrow v_2) \wedge (h_\xi = \{(v_1, v_2)\})$.

Definition 4.3 (Separationskonjunktion)
Seien P *und* R *Assertions. Es gilt*

$$[\![P * R]\!]\ s\ h,$$

gdw. $h = h_1 \cup h_2$ *mit* $h_1 \perp h_2$, *sodass* $[\![P]\!]\ s\ h_1$ *und* $[\![R]\!]\ s\ h_2$ *gilt.*

Definition 4.4 (Separationsimplikation)
Seien P *und* R *Assertions. Es gilt*

$$P\ {-\!\!*} R,$$

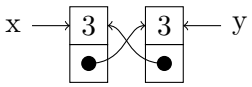
gdw. $\forall h'.(h' \perp h \wedge [\![P]\!]\ s\ h') \implies [\![R]\!]\ s\ (h \cup h')$.

Wir definieren uns nun einige Abkürzungen. Diese bieten keine semantische Erweiterung zu der Logik, sondern sind lediglich dafür da, später kürzere Verifikationen durchführen zu können. Diese Definitionen werden besonders nützlich, wenn man Sequenzen im Heap untersucht.

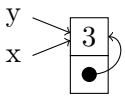
$$
\begin{aligned}
e \mapsto - &\triangleq \exists x. e \mapsto x \ \textit{wobei}\ x \notin \mathit{frei}(e) \\
e \hookrightarrow e' &\triangleq e \mapsto e' * \mathbf{true} \\
e \mapsto e_1, \ldots, e_n &\triangleq e \mapsto e_1 * \ldots * e + n - 1 \mapsto e_n \\
e \hookrightarrow e_1, \ldots, e_n &\triangleq e \hookrightarrow e_1 * \ldots * e + n - 1 \hookrightarrow e_n \\
&\quad \textit{gdw.}\ e \mapsto e_1, \ldots, e_n * \mathbf{true}
\end{aligned}
$$

Beispiel 4.1 (Struktur des Heaps bei verschiedenen Assertions)
Es folgt eine visuelle Darstellung von Heaps, die entsprechende Assertions erfüllen. Ein Kasten ist eine Speicherzelle im Heap und ein Pfeil auf eine Zelle stellt die Adresse dar. Hier wird auch noch einmal der Unterschied zwischen der Konjunktion und Separationskonjunktion deutlich:

(a) $x \mapsto 3, y * y \mapsto 3, x$



(b) $x \mapsto 3, y \wedge y \mapsto 3, x$



Im Folgenden werden die Axiome für die Separationskonjunktion dargelegt. Wir sehen, dass **emp** das neutrale Element bezüglich der Separationskonjunktion darstellt. Weiterhin gilt das Kommutativgesetz (2), das Assoziativgesetz (3) sowie einige Distributiv- und Semidistributivgesetze (4) - (7).

Definition 4.5 (Axiome für die Separationskonjunktion)

$$
\begin{aligned}
P * \mathbf{emp} &\iff P && (1)\\
P_1 * P_2 &\iff P_2 * P_1 && (2)\\
(P_1 * P_2) * P_3 &\iff P_1 * (P_2 * P_3) && (3)\\
(P_1 \vee P_2) * Q &\iff (P_1 * Q) \vee (P_2 * Q) && (4)\\
(P_1 \wedge P_2) * Q &\implies (P_1 * Q) \wedge (P_2 * Q) && (5)\\
(\exists x. P) * Q &\iff \exists x. (P * Q)\ \textit{wobei}\ x \notin \mathrm{frei}(Q) && (6)\\
(\forall x. P) * Q &\implies \forall x. (P * Q)\ \textit{wobei}\ x \notin \mathrm{frei}(Q) && (7)
\end{aligned}
$$

Man kann anhand von Beispielen leicht sehen, warum die Rückrichtung der Semidistributivgesetze (5) und (7) nicht gilt: Die für (5) in Frage kommende Implikation

$$(P_1 * Q) \wedge (P_2 * Q) \Longrightarrow (P_1 \wedge P_2) * Q$$

gilt zum Beipiel nicht für $P_2 := \neg P_1, Q := \mathbf{true}$ für ein beliebiges P_1. Das Beispiel für (7) wird analog dazu konstruiert.

Um aufzuzeigen, welche interessanten Eigenschaften die Separationskonjunktion mit sich bringt, seien hier einige Beispiele für Assertions aufgelistet:

Beispiel 4.2
Gegeben sei ein Stack s mit $s(x) \neq s(y)$ sowie zwei Heaps $h_1 = \{(s(x), 1)\}$, $h_2 = \{(s(y), 2)\}$.

Zuerst ein einfaches Beispiel für die Separationkonjunktion:

$$[\![x \mapsto 1 * y \mapsto 2]\!]\ s\ h \Longleftrightarrow h = h_1 \cup h_2$$

In den folgenden zwei Beispielen wird noch einmal der Unterschied zur Konjunktion deutlich:

$$[\![x \mapsto 1 \wedge x \mapsto 1]\!]\ s\ h \Longleftrightarrow h = h_1$$
$$[\![x \mapsto 1 * x \mapsto 1]\!]\ s\ h \Longleftrightarrow \mathbf{false}$$

In diesen Beispielen kann eine Separation des Heaps gewählt werden, welche die Assertion erfüllt:

$$[\![x \mapsto 1 * (x \mapsto 1 \vee y \mapsto 2)]\!]\ s\ h \Longleftrightarrow h = h_1 \cup h_2$$
$$[\![(x \mapsto 1 \vee y \mapsto 2) * (x \mapsto 1 \vee y \mapsto 2)]\!]\ s\ h \Longleftrightarrow h = h_1 \cup h_2$$

Ersetzen wir das erste „oder“ im vorigen Beispiel jedoch durch eine Separationskonjunktion, ist dies nicht mehr möglich:

$$[\![(x \mapsto 1 * y \mapsto 2) * (x \mapsto 1 \vee y \mapsto 2)]\!]\ s\ h \Longleftrightarrow \mathbf{false}$$

Das letzte Beispiel veranschaulicht die Funktionsweise der Separationsimplikation:

$$[\![(x \mapsto 1) -\!\!* (x \mapsto 1 * y \mapsto 2)]\!]\ s\ h \Longleftrightarrow h = h_2$$

Das letzte Beispiel kann noch etwas verallgemeinert werden: Sei P eine Assertion, die für einen Stack gilt, welcher die Variable x auf die Adresse α abbildet, und einen Heap, der α zu 16 abbildet. Dann würde die Assertion

$$(x \mapsto 16) -\!\!* p$$

für den gleichen Stack und den Heap $h' = h\rceil(\text{dom } h - \{\alpha\})$ gelten.

4.2 Axiome für zeigerabhängige Sprachelemente

Die Erweiterung der Hoare-Logik um die oben gennanten Elemente ermöglicht es uns nun schließlich, die Sprachelemente von $\mathcal{L}$ mit korrekten Axiomen zu betrachten.

Definition 4.6 (Axiome für zeigerabhängige Sprachelemente)

$$\frac{}{\{x = X \wedge \mathbf{emp}\}\ \mathtt{x} := \mathtt{cons}(e_1, \ldots, e_k)\ \{x \mapsto e_1[X/x], \ldots, e_k[X/x]\}}\ \mathit{alloc}$$

$$\frac{}{\{e \mapsto v \wedge x = X\}\ \mathtt{x} := [\mathtt{e}]\ \{x = v \wedge e[X/x] \mapsto v\}}\ \mathit{lookup}$$

$$\frac{}{\{e \mapsto -\}\ \mathtt{e} := [\mathtt{e}']\ \{e \mapsto e'\}}\ \mathit{mut}$$

$$\frac{}{\{e \mapsto -\}\ \mathtt{free}(e)\ \{\mathbf{emp}\}}\ \mathit{free}$$

Da wir nun unsere axiomatische Basis zur Verifikation von $\mathcal{L}$-Progammen auf die Sprachelemente mit Heapinteraktion ausgeweitet haben, fehlen noch die Inferenzregeln unter Berücksichtigung dieser Axiome. Als erstes wollen wir die rule of constancy durch eine für $\mathcal{L}$ geeignete Inferenzregel ersetzen. Diese ist die sogennante *frame rule*: Sie ersetzt die Konjunktion der rule of constancy durch die Separationskonjunktion, um die Disjunktheit der Speicherbereiche zu garantieren:

Definition 4.7 (Frame Rule)

$$\frac{\{P\}Q\{R\}}{\{P * S\}Q\{R * S\}}\ FR$$

Da alle anderen Inferenzregeln der Separationslogik *lokal* sind, sich also nur auf den Speicherbereich, den der entsprechende Befehl benötigt, beziehen, wird die frame rule benutzt, um das lokale Schließen auf den globalen Kontext hin auszudehnen.

Wenn man nun den Strengthening Precendent sowie den Weakening Consequent zu dem *Consequent* zusammenfasst, ergeben sich zusammengefasst folgende grundlegende Inferenzregeln:

Definition 4.8 (Inferenzregeln)

$$\frac{P \to R \quad \{R\}Q\{S\} \quad S \to T}{\{P\}Q\{T\}}\, CON$$

$$\frac{\{P\}Q\{R\}}{\{\exists v.P\}Q\{\exists v.R\}}\, EQ$$

Wobei $v \notin \text{frei}(Q)$.

$$\frac{\{P\}Q\{R\}}{(\{P\}Q\{R\})[e_1/x_1, \ldots, e_k/x_k]}\, S$$

$$\frac{\{P\}Q\{R\}}{\{P * S\}Q\{R * S\}}\, FR$$

Die Korrektheit der Axiome und Inferenzregeln in Bezug auf die Semantik $\mathcal{L}$ ist leicht zu sehen. Die Vollständigkeit der Logik, also die Möglichkeit, sämtliche validen Hoare-Tripel ableiten zu können, wurde in [Yan01] gezeigt.

4.3 Abgeleitete Axiome

Die gegebenen Axiome sind zwar bereits vollständig, um jedoch eine etwas übersichtlichere Beweisführung zu ermöglichen, fügen wir noch ein paar Regeln hinzu, welche ebenfalls aus der natürlichen Semantik von $\mathcal{L}$ abgeleitet werden können. Diese werden an dieser Stelle nur kurz erklärt, eine genauere Betrachtung der Axiome wird in [Rey08] vollzogen.

4.3.1 Assignment

Das erste abgeleitete Axiom ist das des *forward reasoning*. Hier findet die Substitution eines Wertes im Gegensatz zum „normalen“ Axiom der Zuweisung nicht in der Prä- sondern in der Postkondition statt.

Definition 4.9 (Forward Reasoning)

$$\frac{}{\{x = X\}\mathbf{x} := \mathbf{e}\{x = e[X/x]\}}\, fr$$

Ein weiteres die Regeln für das Assignment erweiterndes Axiom ist das sogenannte *Floyd's forward running axiom*.

Definition 4.10 (Floyd's forward running axiom)

$$\frac{}{\{P\}\mathbf{x} := \mathbf{e}\{\exists x'.x = e[x'/x] \wedge P[x'/x]\}}\ floyd$$

Die folgenden Axiome entstehen alle nach den gleichen zwei Schemata und werden deswegen zusammengefasst: Das erste Schema ist das Anwenden der frame rule, um das lokale Schließen global zu machen, das zweite ist das *backwards reasoning*, also das Gegenteil des forward reasoning.

4.3.2 Weitere Axiome

Definition 4.11 (Global Reasoning)
Mutation:

$$\frac{}{\{(e \mapsto -) * r\}[\mathbf{e}] := \mathbf{e}'\{(e \mapsto e') * r\}}\ mgr$$

Deallokation:

$$\frac{}{\{(e \mapsto -) * r\}\mathbf{free}(\mathbf{e})\{r\}}\ dgr$$

Allokation:

$$\frac{}{\{r\}\mathbf{x} := \mathbf{cons}(\mathbf{e_1}, \ldots, \mathbf{e_n})\{\exists v'.(v \mapsto e_1, \ldots, e_n) * r'\}}\ agr$$

Lookup:

$$\frac{}{\{\exists v''.(e' \mapsto v'') * (r/v' \mapsto v)\}\mathbf{x} := [\mathbf{e}]\{\exists v'.(e' \mapsto v) * (r/v'' \to v)\}}\ lgr$$

Ersetzt man nun zum Beispiel beim Mutation Global Reasoning r durch $(e \mapsto e') -\!\!* p$ und macht Gebrauch von der Implikation $q * (q -\!\!* p) \to p$, so erhält man die Regel des *Mutation Backwards Reasoning*. Dies ist bei den anderen Axiomen analog anzuwenden:

Definition 4.12 (Backwards Reasoning)
Mutation:

$$\frac{}{\{(e \mapsto -) * ((e \mapsto e') -\!\!* p)\}[\mathbf{e}] := \mathbf{e}'\{p\}}\ mbr$$

Allokation:

$$\frac{}{\{\forall v'.(v' \mapsto e_1, \ldots, e_n) -\!\!* p'\}\mathbf{x} := \mathbf{cons}(\mathbf{e_1}, \ldots, \mathbf{e_n})\{p\}}\ abr$$

Lookup:

$$\frac{}{\{\exists v'.(e \mapsto v') * ((e \mapsto v') -\!\!* p'\}\mathbf{x} := [\mathbf{e}]\{p\}}\ lbr$$

Nun haben wir eine vollständige und gut verwendbare Axiommenge, mit der wir Verifikationen von Algorithmen in $\mathcal{L}$ durchführen können.

Kapitel 5

Verifikationen mit Separationslogik

Wir wollen nun einige Verifikationen mit der Separationslogik durchführen. Dafür führen wir zunächst einige Datenstrukturen ein, die uns bei der Verifikation helfen werden.

5.1 Listen

5.1.1 Einfach verkettete Listen

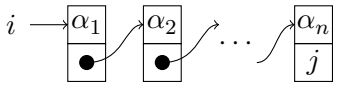
Wir leiten unsere Listen von Sequenzen aus natürlichen Zahlen her:

Definition 5.1 (Notation für Sequenzen)
Seien α und β Sequenzen. Dann schreiben wir:

- *ε für die leere Sequenz.*
- *$[x]$ für die ein-elementige Sequenz, die x enthält.*
- *$\alpha \cdot \beta$ für die Komposition von α und β.*
- *$\alpha^\dagger$ für die Umkehrung von α.*
- *$|\alpha|$ für die Anzahl der Elemente in α.*
- *α_i für das i-te Element in α.*

Die einfachste Darstellung von Sequenzen als Datenstruktur im Heap sind *einfach verkettete Listen*. Wir schreiben `list` α (i,j), für eine Liste von i nach j mit α als Inhalt. Eine visuelle Darstellung, wie diese Liste in einem Heap gespeichert werden würde, ist unter Abbildung 5.1 zu finden.

Abbildung 5.1: Einfach verkettete Liste im Heap



Wir können aus der Abbildung leicht eine induktive Definition von Listen ableiten:

Definition 5.2 (Einfach verkettete Liste)

$$\texttt{list}\ \varepsilon\ (i,j) \triangleq \mathbf{emp} \wedge i = j$$
$$\texttt{list}\ a \cdot \alpha\ (i,k) \triangleq \exists j.\ i \mapsto a, j * \texttt{list}\ \alpha\ (j,k)$$

Weiterhin lassen sich einige Äquivalenzen erkennen:

$$\texttt{list}\ a\ (i,j) \iff i \mapsto a, j$$
$$\texttt{list}\ \alpha \cdot \beta\ (i,k) \iff \exists j.\ \texttt{list}\ \alpha\ (i,j) * \texttt{list}\ \beta\ (j,k)$$
$$\texttt{list}\ \alpha \cdot b\ (i,k) \iff \exists j.\ \texttt{list}\ \alpha\ (i,j) * j \mapsto b, k$$

Die folgenden Verifikationen folgen alle einem bestimmten Schema: Zuerst werden die Prä- und Postkonditionen festgelegt. Dann werden wir, von der Präkondition ausgehend, die Semantik eines jeden Befehls auf die Assertion anwenden. Wenn wir zwischen zwei Befehlen unsere Assertion mithilfe unserer Inferenzregeln anpassen, stehen die alte und die neue Assertion untereinander. Die Rechtfertigung für die Umformung wird jeweils am rechten Rand aufgeführt. Die zum Algorithmus gehörigen Befehle sind grau hinterlegt. Können wir am Ende des Programms unsere Postkondition herleiten, gilt das Programm als verifiziert.

Als Beispiel für die Verifikation eines Algorithmus mit einfach verketteten Listen wollen wir uns uns die Löschung des ersten Elements einer Liste ansehen:

Verifikation 5.1 (Löschen aus einfach verketteter Liste)

Präkondition: $\{\texttt{list}\ a \cdot \alpha\ (i,k)\}$ **Postkondition:** $\{\texttt{list}\ \alpha\ (i,k)\}$

$\{\texttt{list}\ a \cdot \alpha\ (i,k)\}$
$\{\exists j.\ i \mapsto a, j * \texttt{list}\ \alpha\ (j,k)\}$ (5.2)
$\{i \mapsto a * \exists j.\ i+1 \mapsto j * \texttt{list}\ \alpha\ (j,k)\}$ (Abkürzungen)

```
j := [i + 1];
```

$\{i \mapsto a * i+1 \mapsto j * \texttt{list}\ \alpha\ (j,k)\}$

```
free(i);
```

$\{i + 1 \mapsto j * \texttt{list}\ \alpha\ (j, k)\}$

```
free(i + 1);
```

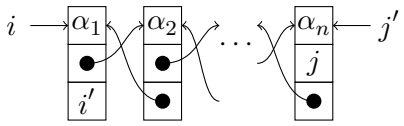
$\{\texttt{list}\ \alpha\ (j, k)\}$

```
i := j
```

$\{\texttt{list}\ \alpha\ (i, k)\}$

◁

Abbildung 5.2: Doppelt verkettete Liste im Heap



5.1.2 Doppelt verkettete Listen

Eine weitere Möglichkeit, Listen zu implementieren, sind die *doppelt verketteten Listen*. Hier enthält jedes Element der Liste zusätzlich zu einem Zeiger auf das nachfolgende Element noch einen Zeiger auf den Vorgänger. Die entsprechende Darstellung ist in Abbildung 5.2 zu finden.

Die induktive Definition der doppelt verketteten Liste erfolgt analog zur einfach verketteten:

Definition 5.3

$$\texttt{dlist}\ \varepsilon\ (i, i', j, j') \triangleq \mathbf{emp} \wedge i = j \wedge i' = j'$$
$$\texttt{dlist}\ a \cdot \alpha\ (i, i', k, k') \triangleq \exists j.\ i \mapsto a, j, i' * \texttt{dlist}\ \alpha\ (j, i, k, k')$$

Es ergeben sich auch ähnliche Eigenschaften:

$$\texttt{dlist}\ a\ (i, i' j, j') \iff i \mapsto a, j, i' \wedge i = j'$$
$$\texttt{dlist}\ \alpha \cdot \beta\ (i, i', k, k') \iff \exists j, j'.\ \texttt{dlist}\ \alpha\ (i, i', j, j') * \texttt{dlist}\ \beta\ (j, j', k, k')$$
$$\texttt{dlist}\ \alpha \cdot b\ (i, i' k, k') \iff \exists j'.\ \texttt{dlist}\ \alpha\ (i, i', k', j') * k' \mapsto b, k, j'$$

Somit ergibt sich eine ebenfalls leicht verständliche Verifikation.

Verifikation 5.2 (Löschen aus doppelt verketteter Liste)

Präkondition: $\{\texttt{dlist}\ a \cdot \alpha\ (i, i', k, k')\}$
Postkondition: $\{\texttt{dlist}\ \alpha\ (i, i', k, k')\}$

$\{\texttt{dlist}\ a \cdot \alpha\ (i, i', k, k')\}$
$\quad \{\exists j.\ i \mapsto a, j, i' * \texttt{dlist}\ \alpha\ (j, i, k, k')\}$ (5.3)
$\quad \{i \mapsto a * \exists j.\ i + 1 \mapsto j, i * \texttt{dlist}\ \alpha\ (j, i, k, k')\}$ (Abkürzungen)

```
j := [i + 1];
```

$\quad \{i \mapsto a * i + 1 \mapsto j, i' * \texttt{dlist}\ \alpha\ (j, i, k, k')\}$

```
free(i);
```

$\quad \{i + 1 \mapsto j, i' * \texttt{dlist}\ \alpha\ (j, i, k, k')\}$

```
free(i + 1);
```

$\quad \{i + 2 \mapsto i' * \texttt{dlist}\ \alpha\ (j, i, k, k')\}$

```
k := [i + 2];
```

$\quad \{i + 2 \mapsto i' * \texttt{dlist}\ \alpha\ (j, i', k, k')\}$

```
free(i + 2);
```

$\quad \{\texttt{dlist}\ \alpha\ (j, i', k, k')\}$

```
i := j
```

$\{\texttt{dlist}\ \alpha\ (i, i', k, k')\}$

◁

Weitere Arten von Listenimplementierungen und deren Verifikationsmethoden können in [Rey08] nachgeschlagen werden.

5.2 Bäume und gerichtete zyklenfreie Graphen

5.2.1 Speichern von Bäumen und gerichteten zyklenfreien Graphen

Weitere zentrale Datentypen sind *Bäume* und *gerichtete zyklenfreie Graphen* (im Folgenden abgekürzt als DAG für *directed acyclic graph*). Somit wollen wir auch diese in unserer Programmiersprache formalisieren, um Algorithmen, die diese Elemente verwenden, verifizieren zu können. Da Algorithmen, die mit Bäumen und Graphen arbeiten, oft Rekursionen beinhalten, werden wir zusätzlich sehen, wie man diese mittels der Separationslogik überprüfen kann.

Die intuitive Idee der rekursiven Definition eines Baumes ist es, zwischen Blättern, die das Rekursionsende darstellen, und Teilbäumen zu unterscheiden. In unserem Fall beschränken wir uns auf binäre Bäume, das Prinzip lässt sich jedoch auch einfach auf Bäume mit beliebig vielen Kinderknoten erweitern. Formal verwenden wir eine minimale Algebra, welche sich an die aus LISP bekannten *S-Expressions* [Wik18b] anlehnt: Demgemäß besteht eine S-Expression entweder aus einem atomaren Element – einem Blatt des Baumes – oder aus der Komposition zweier S-Expressions.

Definition 5.4 *S-Expressions*

$$\tau \in \mathcal{S} \text{ gdw. } \tau \in \mathsf{Atoms} \text{ oder } \tau = (\tau_1 \cdot \tau_2) \text{ mit } \tau_1, \tau_2 \in \mathcal{S}$$

Auf unser Speichermodell angepasst gilt insbesondere $\mathsf{Atoms} = \mathbb{N}$. Um nun unterscheiden zu können, ob ein Blatt oder eine Komposition vorliegt, definieren wir uns ein neues Prädikat:

Definition 5.5 *Atomprädikat*

$$\texttt{isatom}(\tau) \text{ gdw. } \tau \in \mathsf{Atoms}$$

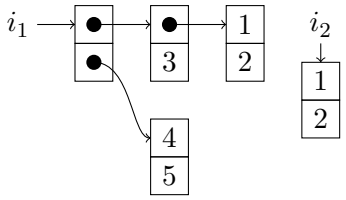
Somit ist leicht zu sehen, dass die Algebra dem semantischen Aufbau eines Baumes entspricht. Gleiches gilt auch für DAGs. Wir unterscheiden die beiden Datentypen dahingehend, dass sich zwei DAGs im Gegensatz zu Bäumen Speicherbereiche teilen können. Dies wird auch bei der induktiven Definition 5.6 der Prädikate **tree** τ (i) und **dag** τ (i) deutlich. Diese geben an, ob i die Adresse der Wurzel eines Baumes oder DAGs ist, die den Ausdruck τ repräsentiert. Dadurch, dass beim DAG nicht die Separationskonjunktion verwendet wird, ist das Teilen von Speicherbereichen erlaubt.

Definition 5.6 *Wurzelprädikat*

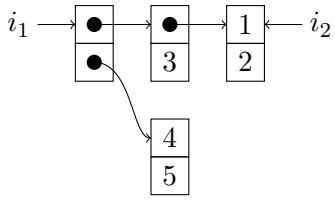
$$\begin{aligned}
\texttt{tree}\ a\ (i) &\text{ gdw. } \mathbf{emp} \wedge i = a \\
\texttt{tree}\ (\tau_1 \cdot \tau_2)\ (i) &\text{ gdw. } \exists i_1, i_2 . i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2) \\
\texttt{dag}\ a\ (i) &\text{ gdw. } i = a \\
\texttt{dag}\ (\tau_1 \cdot \tau_2)\ (i) &\text{ gdw. } \exists i_1, i_2 . i \mapsto i_1, i_2 * (\texttt{dag}\ \tau_1\ (i_1) \wedge \texttt{dag}\ \tau_2\ (i_2))
\end{aligned}$$

Beispiel 5.1 *Bäume und DAGs im Heap*
Gegeben seien $\tau_1 = (((1 \cdot 2) \cdot 3) \cdot (4 \cdot 5)), \tau_2 = (1 \cdot 2)$. *Dann besitzen erfüllende Heaps zu den folgenden Prädikaten die Struktur:*

(a) $\texttt{tree}\ \tau_1\ (i_1) \wedge \texttt{tree}\ \tau_2\ (i_2)$



(b) $\texttt{dag}\ \tau_1\ (i_1) \wedge \texttt{dag}\ \tau_2\ (i_2)$



5.2.2 Verifikation mit Rekursion

Um Rekursion zu ermöglichen, müssen wir unsere Sprache um Prozeduren erweitern. Eine Prozedur h hat dabei die Form $h(x_1, \ldots, x_m; y_1, \ldots, y_n)\{c\}$ für einen beliebigen Befehl c. $y_1, \ldots, y_n$ sind dabei freie Variablen, die von c modifiziert werden, $x_1, \ldots, x_m$ sind die restlichen freien Variablen in c.

Betrachten wir nun ein Programm, welches einen Baum rekursiv von der Wurzeladresse i zur Adresse j kopiert:

```
copytree(i;j) {
    if isatom(i) {
        j := i
    } else {
        i1 := [i];
        i2 := [i + 1];
        copytree(i1;j1);
        copytree(i2;j2);
        j := cons(j1, j2)
    }
}
```

Wir fokussieren uns zunächst darauf, was beim Aufruf der Rekursion passiert und gehen dabei von der intuitiven Spezifikation von `copytree(i;j)` aus: $\{\texttt{tree}\ \tau\ (i)\}$ **copytree**$(i; j)$ $\{\texttt{tree}\ \tau\ (i) * \texttt{tree}\ \tau\ (i)\}$. Um nun eine rekursive Prozedur zu verifizieren, muss vor dem Rekursionsaufruf die Präkondition der Prozedur stehen, die zur Postkondition führt. Verifiziert man dann die Korrektheit der Prozedur mit den gleichen Prä- und Postkonditionen, so wurde die Korrektheit rekursiv gezeigt. Die Herleitung der Verifikationsschritte bei den Rekursionsaufrufen ist im Anhang unter 8.1 zu finden.

Verifikation 5.3 (Kopieren eines Baumes)

Präkondition: $\{\texttt{tree}\ \tau\ (i)\}$ **Postkondition:** $\{\texttt{tree}\ \tau\ (i) * \texttt{tree}\ \tau\ (i)\}$

$\{\texttt{tree}\ \tau\ (i)\}$

```
if isatom(i) {
```

$\{\texttt{isatom}(\tau) \wedge \mathbf{emp} \wedge i = \tau\}$
$\{\texttt{isatom}(\tau) \wedge ((\mathbf{emp} \wedge i = \tau) * (\mathbf{emp} \wedge i = \tau))\}$

```
j := i
```

$\{\texttt{isatom}(\tau) \wedge ((\mathbf{emp} \wedge i = \tau) * (\mathbf{emp} \wedge j = \tau))\}$

```
} else {
```

$\{\exists \tau_1, \tau_2.\tau = (\tau_1 \cdot \tau_2) \wedge \texttt{tree}\ (\tau_1 \cdot \tau_2)(i)\}$

```
i1 := [i];
i2 := [i + 1];
```

$\{\exists \tau_1, \tau_2.\tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ (\tau_1)(i_1) * \texttt{tree}\ (\tau_2)(i_2)\}$

```
copytree(i1;j1);
```

$\{\exists \tau_1, \tau_2.\tau = (\tau_1 \cdot \tau_2)$
$\wedge\ (i \mapsto i_1, i_2 * \texttt{tree}\ (\tau_1)(i_1) * \texttt{tree}\ (\tau_2)(i_2) * \texttt{tree}\ (\tau_1)(j_1))\}$

```
copytree(i2;j2);
```

$\{\exists \tau_1, \tau_2.\tau = (\tau_1 \cdot \tau_2)$
$\wedge\ (i \mapsto i_1, i_2 * \texttt{tree}\ (\tau_1)(i_1) * \texttt{tree}\ (\tau_2)(i_2)$
$* \texttt{tree}\ (\tau_1)(j_1) * \texttt{tree}\ (\tau_2)(j_2))\}$

```
j := cons(j1, j2)
```

$\{\exists \tau_1, \tau_2.\tau = (\tau_1 \cdot \tau_2)$ (5.6)
$\wedge\ (i \mapsto i_1, i_2 * \texttt{tree}\ (\tau_1)(i_1) * \texttt{tree}\ (\tau_2)(i_2)$
$* j \mapsto j_1, j_2 * \texttt{tree}\ (\tau_1)(j_1) * \texttt{tree}\ (\tau_2)(j_2))\}$
$\{\exists \tau_1, \tau_2.\tau = (\tau_1 \cdot \tau_2) \wedge (\texttt{tree}\ (\tau_1 \cdot \tau_2)(i) * \texttt{tree}\ (\tau_1 \cdot \tau_2)(j))\}$
$\{\texttt{tree}\ \tau\ (i) * \texttt{tree}\ \tau\ (i)\}$

◁

Durch diese Verifikation haben wir uns einen Überblick über die Anwendung der Separationslogik als Kalkül zum Schließen über $\mathcal{L}$-Implementationen einfacher Algorithmen verschafft.

Kapitel 6

Weitere Anwendungsgebiete

In diesem Kapitel wird ein Überblick über die aktuellen Anwendungsgebiete gegeben, sowie mögliche zukünftige Erweiterungen und Anpassungen der Separationslogik vorgestellt und erklärt.

6.1 Andere Ansätze

Die Separationslogik eignet sich vor allem zur Verifikation von terminierenden Programmen, die Interaktionen mit freiadressierbarem Speicher beinhalten. Neben der hier betrachteten und geläufigsten Version der Separationslogik gibt es noch andere Ansätze, die ähnliche Ziele verfolgen: so wurde in einer älteren Version der Separationslogik noch keine freie Speicheradressierung zugelassen, stattdessen bildete der Heap Adressen auf sogenannte *Verbunde* ab, die aus mehreren Speicherzellen zusammengesetzt wurden. Zu einer weiteren Parallelentwicklung gehört die Crash-Hoare-Logik [Che+15]. Werden in der Separationslogik nur Verifikationen unter dem Vorbehalt geführt, dass Befehle korrekt ausgeführt werden, wird in der Crash-Hoare-Logik davon ausgegangen, dass das Programm abstürzen („crashen“) kann. Dies ist nur eine kleine Auswahl von möglichen Abwandlungen, die sich je nach Anwendungsgebiet besser oder schlechter für eine Algorithmenverifikation eignen.

6.2 Schorr-Waite-Algorithmus

Die in der vorliegenden Arbeit durchgeführten Verifikationen sind eher einfacher Natur, um das Grundverständnis der Vorgehensweise beim Verifizieren mit der Seperationslogik in den Fokus stellen zu können. Die möglichen Verifikationen mit der Separationslogik gehen in ihrer Komplexität natürlich weit über die Vorgestellten hinaus. Als Beispiel sei hier die bisher umfangreichste Verifikation genannt, die in der Arbeit von Yang [Yan01]

durchgeführt wurde. In dieser wird jedoch noch die ältere Form der Separationslogik benutzt, welche noch keine freie Adressarithmetik erlaubt. Der Schorr-Waite Algorithmus, der verifiziert wurde, wird benutzt, um sharing und Zyklen in Datenstrukturen aufzuzeigen. Die Verbunde, die für die Verifikation des Algorithmus benutzt wurden, bestanden aus zwei Adressfeldern und zwei Booleschen Feldern. Weiterhin werden noch zahlreiche neue Prädikate definiert und erweiterte Schlussregeln eingeführt, um die Verifikation kompakter zu halten. Dies zeigt deutlich, welches Ausmaß die Verifikationen erreichen können und welche Anpassungen an beispielweise der Sprache oder dem Speichermodell vorgenommen werden können, um die „klassische“ Separationslogik auf bestimmte Anwendungsgebiete abzustimmen.

6.3 Adressarithmetik

Durch die freie Adressarithmetik, die in neueren Versionen der Separationslogik zu finden ist, werden viele Verifikationen deutlich simplifiziert. Noch dazu bringt es die Sprache $\mathcal{L}$ deutlich näher an das in der Praxis verbreitete C heran. Jedoch ergeben sich durch diese freie Adressarithmetik auch Probleme, die in der Logik behandelt werden müssen. Will man zum Beispiel die Idee der Verbunde in neueren Versionen der Separationslogik einführen, so können durch eben diese freie Adressarithemtik auch wahre Assertions über Verbunde getroffen werden, die sich überlappen. Da zum Beispiel die Verifikation des Schorr-Waite-Algorithmus noch ohne freie Adressarithmetik durchgeführt wurde, müsste man beim „Übersetzen“ der Verifikation in eine neuere Version der Separationslogik das Überlappen der Datenverbunde manuell untersagen, sowie die Struktur eines jeden Verbundes prädikatenlogisch beschreiben. Dies wird umso schwieriger, wenn man mehrere Arten von Verbunden mit unterschiedlicher Länge betrachtet.

6.4 Concurrency

Ein weiteres mögliches Anwendungsgebiet für die Separationslogik ist es, Algorithmen zu verifizieren, die Gebrauch von Nebenläufigkeit, im Englischen *Concurrency* genannt, machen. Um beispielsweise *data races* zu verhindern, schlug Hoare in den Siebzigern vor, die Interferenz von nebenläufigen Prozessen syntaktisch auf bestimmte *kritische Regionen* zu beschränken. Jedoch ist eine syntaktische Erkennung von Interferenz zweier Prozesse unter Einfluss von geteilten und dynamisch veränderbaren Datenstrukturen viel zu schwierig, um sie generisch und vollständig durchsetzen zu können. Die Seperationslogik kann hier helfen, da sie genau darauf ausgelegt ist, solche Interaktionen zu behandeln. Als triviales Beispiel kann die Inferenzregel angegeben werden, die angibt, dass zwei nebenläufige Prozesse, notiert als $c_1||c_2$, keine geteilten Variablen mutieren.

Die sich daraus ableitende Inferenzregel hat dann die folgende Form:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \quad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 * P_2\}\ c_1 || c_2\ \{Q_1 * Q_2\}}$$

Es gibt auch bereits Arbeiten, die versuchen, die Separationslogik auf nebenläufige Programme anzupassen [OHe01; OHe02]. Jedoch konnte bis zum jetzigen Zeitpunkt die Korrektheit der Inferenzregeln noch nicht gezeigt werden.

Eine andere weit verbreitete Praxis bei der Implementierung nebenläufiger Programme ist es, *read-only* Zugriffe auf Daten auch von mehreren Prozessen zuzulassen, solange diese die Daten nicht mutieren. Auch dies kann in der Separationslogik in zukünftigen Versionen umgesetzt werden.

Nebenläufige Programme sind zudem seltener darauf ausgelegt, zu terminieren und eher darauf, virtuell unbegrenzt zu laufen. Für das Schließen mit nicht terminierenden Programmen ist die Hoare-Logik allerdings nur begrenzt brauchbar, hier sollte dann überlegt werden, die Separationskonjunkton eher auf der Grundlage temporaler Logiken aufzubauen.

6.5 Garbage Collection

Die freie Adressarithmetik verhindert eine weitere, bei modernen Programmiersprachen weit verbreitete Technologie: die sogenannte *Garbage Collection*, bei der nicht mehr benötigte Speicherbereiche durch einen Garbage Collector automatisch freigegeben werden. Betrachtet man zum Beispiel den folgenden Programmteil:

Verifikation 6.1

$\{\mathbf{true}\}$

```
x := cons(1,2);
```

$\{\exists x.x \hookrightarrow 1,2\}$

```
x := 0
```

$\{\exists x.x \hookrightarrow 1,2\}$

◁

So sieht man leicht, dass ein automatischer Garbage Collector den allokierten Speicherbereich löschen könnte und damit die Verifikation zerstören würde.

Eine Garbage Collection ist mit der Version der Separationslogik, die noch keine freie Speicheradressierung bietet, eher vorstellbar.

6.6 Anwendungen

Es gibt bereits einige konkrete Anwendungen der Separationslogik oder von abgeleiteten Logiken. So wurde eine Logik erstellt, die für Verifikationen einer Sprache, die eine Teilmenge der Programmiersprache JavaScript darstellt, benutzt werden kann [GMS12]. Weiterhin können alle Bibliotheken, die in der Teilmenge geschrieben und verifiziert wurden, mit beliebigen JavaScript-Code aufgerufen werden. Ein weiteres Beispiel ist die Verifikation der Implementation von HMAC, einer Methode zur Nachrichtenauthentifizierung im Toolkit OpenSSL [Ber+15]. Hierfür wurde auf der Grundlage der Separationslogik basierende *Verifiable C*-Logik verwendet. Diese ist ein Derivativ der Separationslogik, die gezielt zur Verifikation von C-Code entwickelt wurde. Gerade in diesem Beispiel, einer kryptografischen Anwendung, tritt die Bedeutung der Programmverifikation hervor, da sie sicher stellt, dass ein Programm gemäß seiner Spezifikation arbeitet.

Kapitel 7

Zusammenfassung und Ausblick

Wie wir gesehen haben, ist die Separationslogik eine gut ausgebaute Logik, die sich besonders für die Verifikationen von Programmen mit geteilten Ressourcen eignet. Gerade bei diesen Programmen ist eine Verifikation sehr erwünscht, da die Fehleranfälligkeit von Sprachen, die die freie Verwendung von Zeigern zulassen, als besonders hoch angesehen wird.
Sie hilft, das natürliche Denken eines Programmierers, der mit Zeigersprachen arbeitet, einfacher in Verifikationen abzubilden.

Wir haben zudem gesehen, dass die Separationslogik mittlerweile schon konkrete praktische Anwendungen findet, zum Beispiel in kryptografischen Anwendungen, bei der die Korrektheit der Algorithmen in Bezug auf deren Spezifikation essentiell ist.

Insbesondere die Arbeit von Yang hat jedoch gezeigt, dass die Separationslogik auch noch ausbaufähig ist und neue Inferenzregeln oder abgeleitete Axiome hinzugefügt werden können, um effizientere Verifikationen durchführen zu können.

Auch der Vergleich von $\mathcal{L}$ zu C zeigt, dass die Lücke zwischen der Theorie der Verifikation und der praktischen Anwendung dieser noch groß ist. Wir haben in der Arbeit zwar schon viele wichtige Sprachelemente wie einfache Datenstrukturen oder Rekursion betrachtet, jedoch geht die Anzahl der Sprachelemente selbst einer eher kompakten Sprache wie C noch weit darüber hinaus: es gibt beispielsweise schon einige frühe Ideen, die Separationslogik mit einem primitiven Typensystem zu verbinden. Diese ähneln dem Typsystem von C jedoch noch nicht. Auch gibt es zwar verschiedene Versionen der Separationslogik, die mit Verbunden – in C `struct` genannt – sowie freier Adressierung arbeiten, jedoch noch kein Hybridsystem. Auch für

die Behandlungen von Prozeduren als *first class citizens*, bzw. das Benutzen von Code Pointern gibt es bisher nur grobe Konzepte.

Eine weitere Schwierigkeit, die sich bei der Formalisierung praktisch relevanter Sprachen ergibt, ist, dass sich diese auch im ständigen Wandel befinden. Neue Sprachstandards wie zum Beispiel im Falle von C++ verändern die Syntax und Semantik der Sprache, sodass bei jeder Veränderung auch eine Anpassung der Logik von Nöten ist. Dieses Problem wird aber auch unter anderem dadurch mitigiert, dass die Standardkomitees, die neue Sprachstandards festlegen, zumeist sehr darauf bedacht sind, diese abwärtskompatibel zu halten.

Abschließend kann man also sagen, dass es bei der Separationslogik und ihrer Derivate – obwohl sie schon sehr umfassend sind – noch viele Möglichkeiten der Weiterentwicklung gibt und diese auch in absehbarer Zeit nicht erschöpft werden.

Kapitel 8

Anhang

8.1 Herleitung copytree

Hinweis: Um die Darstellung der Herleitung leserlich zu halten, wurden in den letzten drei Schritten nach der Präkondition und dem Programm Zeilenumbrüche eingefügt.

$$\dfrac{\dfrac{\dfrac{\{\texttt{tree}\ \tau\ (i)\}\ \mathbf{copytree(i;j)}\ \{\texttt{tree}\ \tau\ (i) * \texttt{tree}\ \tau\ (i)\}}{\{\texttt{tree}\ \tau_1\ (i_1)\}\ \mathbf{copytree(i_1;j_1)}\ \{\texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_1\ (i_1)\}}\,\text{S}}{\{(\tau = (\tau_1 \cdot \tau_2) \wedge i \mapsto i_1, i_2) * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2)\}\ \mathbf{copytree(i_1;j_1)}\ \{(\tau = (\tau_1 \cdot \tau_2) \wedge i \mapsto i_1, i_2) * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2) * \texttt{tree}\ \tau_1\ (i_1)\}}\,\text{FR}$$

$$\dfrac{\{(\tau = (\tau_1 \cdot \tau_2) \wedge i \mapsto i_1, i_2) * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2) * \texttt{tree}\ \tau_1\ (i_1)\}\ \mathbf{copytree(i_1;j_1)}\ \{\tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2))\}}{\{\tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2))\}}\,\text{CON}$$

$$\dfrac{\{\tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2) * \texttt{tree}\ \tau_1\ (i_1))\}\ \mathbf{copytree(i_1;j_1)}\ \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2))\}}{\{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2))\}}\,\text{EQ}$$

$$\{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2) * \texttt{tree}\ \tau_1\ (i_1))\}\ \mathbf{copytree(i_1;j_1)}\ \{\exists \tau_1, \tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \texttt{tree}\ \tau_1\ (i_1) * \texttt{tree}\ \tau_2\ (i_2) * \texttt{tree}\ \tau_1\ (i_1))\}$$

Quellenverzeichnis

Literatur

[Ber+15] Lennart Beringer u. a. „Verified correctness and security of OpenSSL HMAC“. *24th Usenix Security Symposium* (2015). URL: https://www.cs.princeton.edu/~eberinge/verified-hmac.pdf.

[Che+15] Haogang Chen u. a. „Using Crash Hoare Logic for Certifying the FSCQ File System“. *MIT CSAIL* (2015).

[GMS12] Philippa Gardner, Sergio Maffeis und Gareth Smith. „Towards a Program Logic for JavaScript“. *Imperial College London* (2012). URL: https://www.doc.ic.ac.uk/~maffeis/papers/popl12.pdf.

[Hoa69] C. A. R. Hoare. „An Axiomatic Basis for Computer Programming“. *Communications of the ACM* 10 (Oktober 1969), S. 576–583.

[Kel15] Leo Kelion. *Airbus A400M plane crash linked to software fault.* 2015. URL: http://www.bbc.com/news/technology-32810273 (besucht am 22. 07. 2018).

[OHe01] P. W. O'Hearn. „Notes on conditional critical regions in spatial pointer logic“. *Unpublished* (2001).

[OHe02] P. W. O'Hearn. „Notes on separation logic for shared-variable concurrency“. *Unpublished* (2002).

[Rey08] John C. Reynolds. „An Introduction to Separation Logic (Preliminary Draft)“. *ITU University, Copenhagen* (2008). URL: http://www.cs.cmu.edu/~jcr/copenhagen08.pdf.

[SN17] Abhishek Kr Singh und Raja Natrajan. „An Outline of Separation Logic“. *Tata Institute of Fundamental Research, Mumbai* (2017).

[Sto16] Bjarne Stoustrup. „RAII“. In: *The C++ Language.* Addison Wesley, 2016. Kap. 5, S. 112.

[Vol17a] Heribert Vollmer. *Folien zur Vorlesung Grundlagen der Theoretischen Informatik.* Leibniz Universität Hannover. 2017.

[Vol17b] Heribert Vollmer. *Folien zur Vorlesung Logik und formale Systeme*. Leibniz Universität Hannover. 2017.

[Yan01] Hongseok Yang. „Local Reasoning for Stateful Programs“. Ph. D. Urbana- Champaign, Illinois: University of Illinois, 2001. URL: www.cs.ox.ac.uk/people/hongseok.yang/paper/thesis.ps.

Online-Quellen

[Hot99] Robert Lee Hotz. *Mars Probe Lost Due to Simple Math Error*. 1999. URL: http://articles.latimes.com/1999/oct/01/news/mn-17288 (besucht am 22. 07. 2018).

[Tho13] Iain Thomsom. *30 years on: The day a computer glitch nearly caused World War III*. 2013. URL: https://www.theregister.co.uk/2013/09/27/30_years_on_the_day_a_computer_glitch_nearly_caused_world_war_iii/ (besucht am 22. 07. 2018).

[Wik18a] Wikipedia. *Pumpspeicherkraftwerk Taum Sauk*. 2018. URL: https://de.wikipedia.org/wiki/Pumpspeicherkraftwerk_Taum_Sauk#Ungl%C3%BCck (besucht am 22. 07. 2018).

[Wik18b] Wikipedia. *S-expression*. 2018. URL: https://en.wikipedia.org/wiki/S-expression (besucht am 23. 07. 2018).