Automatisierte Logik und Programmierung *II*
Graphentheorie in NuPRL

Bernd Österholz und Andre Neumann
Universität Potsdam, Theoretische Informatik – Sommersemester 2004

Zusammenfassung

Dieses Projekt führt eine neue Untertheorie in die NuPRL Objekttheorie ein. Dabei handelt es sich um einfache Graphen, welche sich auf einfache Knoten und Kanten stützen. Dazu wird ein Grundgerüst für die Graphen erstellt, welches einige Hilfsdefinitionen beinhaltet. Zu der neuen Objekttheorie werden auch Theoreme aufgestellt, welche Beweisführungen vereinfachen. Zusätzlich werden einige nützliche Taktiken für ein einfachere Beweisführung eingeführt.

1 Einleitung

Das vorliegende Dokument behandelt die von uns im Rahmen eines Projektes in *NuPRL* eingeführte Graphentheorie.

Im Teil mathematische Definitionen werden die Mathematischen Grundlagen der Graphentheorie besprochen. Der Namenskonventionen Teil beinhaltet die von uns eingeführten Konventionen für Namen und Namensräume. Im Definitionen Teil geht es erst allgemein um unsere Definitionen und dann werden die von uns aufgestellten Definitionen im einzelnen betrachtet. Diese Aufstellung wird Tabellarisch im Anhang wiederholt und dient in beiden Fällen mehr zum Nachschlagen dieser, als als zusammenhängendes Dokumentstück zum lesen. Im Abschnitt Anmerkungen zur Implementation geht es um einige Dinge die bei der Implementation aufgefallen sind, unter anderen auch um Probleme und Anregungen zu Verbesserungen.

2 Mathematische Definitionen

Bei den Mathematischen Definitionen wurde versucht, sich so weit wie möglich an den englischen Definitionen aus [1] zu halten. Da wir nur ein Grundgerüst für die Graphen erstellen wollen, d.h. die Definitionen der Knoten und Kanten, sowie Mengen bzw. Listen auf diesen Objekten, werden nur wichtigere Definitionen von uns realisiert. Es besteht jedoch die Möglichkeit, die Definitionen einfach zu erweitern, hierzu haben wir auch ein paar Anregungen im Ausblick.

Ein Graph ist ein Paar aus einer Menge von Knoten und einer Menge von gerichteten Kanten. Formal gesagt ist der Graph G als das Paar (V, E) definiert, wobei (1) V die Menge $\{x_1, x_2, \ldots, x_n\}$ der Knoten ist, mit x_i als Knoten (bzw. Vertices) und (2) E die Menge $\{u_1, u_2, \ldots, u_m\}$ der Kanten[1] ist, mit u_i als Kanten die dem karthesischen Produkt $V \times V$ entsprechen. Die u_i werden als gerichtete Kanten (bzw. directed Edges oder Arrows) bezeichnet. Ein Paar $(x, y) \in V \times V$ kann mehrfach in der Menge der Kanten E vorkommen.

Die Komponenten einer Kante (x, y) werden als Quelle (*source* oder *initial endpoint*) bzw. Ziel (*destination* oder *terminal endpoint*) bezeichnet. Der Knoten x wird als Nachfolger *successor* von y bezeichnet, wenn es eine Kante $(y, x) \in E$ gibt, der Knoten x wird als Vorgänger *predecessor* von y bezeichnet, wenn es eine Kante $(x, y) \in E$ gibt. Die Knoten x und y werden als Nachbarn (*neighbours*) bezeichnet, wenn x Vorgänger oder Nachfolger von y ist.

Ein Pfad (path) der Länge n ist eine Folge/Liste von Kanten $\mu = (u_1, u_2, \ldots, u_i, \ldots, u_n)$ in welcher der Zielknoten der Kante u_i gleich dem Quellknoten der Kante u_{i+1} ist, für alle $i < n - 1$. Ein Zyklus (circuit) ist ein Pfad, bei dem der Quellknoten der ersten Kante gleich des Zielknotens der letzten Kante ist.

In einem Graphen $G = (V, E)$ ist ein Hamilton-Pfad (hamiltonian path) definiert als ein Pfad, der jeden Punkt des Graphens genau einmal durchläuft. Ganz genauso wird der Hamilton-Zyklus (hamiltonian circuit) definiert, jeder Knoten des Graphen muß in dem Zyklus genau einmal durchlaufen werden, bevor der Zyklus am Startpunkt endet.

Da es sich bei der in *NuPRL* implementierten Typtheorie um eine konstruktive (intuitonistische) Theorie handelt sind nur potentiell unendliche Graphen möglich, da alle Graphen sozusagen konstruierbar bleiben müssen.

[1] Kantenmengen sind sind zwar ungeordnet, können jedoch doppelte Kanten enthalten.

3 Namenskonventionen

In unserer *NuPRL* Graphentheorie Implementierung gibt es drei Teilbereiche den der Definitionen, der Theoreme und der Taktiken. Die zu den Namensräumen gehörenden Objekte liegen jeweils, soweit möglich, in einem eigenen Verzeichnis. Das Verzeichnis für die Definitionen, das heißt ihre Abstraktion, Displayform und ihrem Wohlgeformtheitsziel, ist `definitions`, das für Theoreme `theorem` und das für Taktiken `tactics`. Da aber leider einige Theoreme die sich auf Definitionen beziehen in den Wohlgeformtheitszielen anderen Definitionen benötigt werden, sind solche auch im Definitionsverzeichnis zu finden.

Da wir einige neue Definitionen zu der bereits vorhandenen *NuPRL* Objekttheorie hinzufügen wollen, müssen wir passende Namen für unsere Abstraktionen ausdenken. Es kann leider nie ausgeschlossen werden, das diese Namen schon vorhanden sind. Deshalb haben wir uns entschlossen, alle unsere Definitionen mit einem Präfix zu versehen. Wir haben den Präfix `g_` gewählt, da es sich hierbei um eine einfache Graphentheorie handelt. Zusätzlich haben wir gesehen, das wir verschiedene Arten von Definitionen haben, wie z.B. neue Datentypen, einfache Funktionen und boolsche Funktionen. Die folgende Tabelle zeigt alle verschiedenen Präfixe.

Präfix	Beschreibung
`gt_`	Abstraktion ist ein neuer Datentyp
`gc_`	Abstraktion beschreibt den Konstruktor zu einem Datentyp
`gf_`	Abstraktion beschreibt eine Funktion (unteranderem Prädikat), welche einen beliebigen Rückgabewert haben kann
`gb_`	Abstraktion entspricht einer boolschen Funktion, welche den Rückgabewert des Datentypen $\mathbb{B}$ besitzt

Zu jeder boolschen Funktion gibt es eine Abstraktion, welche den Datentypen $\mathbb{B}$ auf die Wahrheitswerte *True* bzw. *False* abbildet. Diese Funktion ist dann immer entscheidbar, da klassischen boolsche Funktionen immer entscheidbar sind.

Theoreme und Taktiken werden durchgängig klein geschrieben, wobei Teile zusammengesetzter Namen mit Unterstrichen verbunden werden.

4 Nuprl Implementation

4.1 Definitionen

Die Grunddefinitionen sind die Definitionen von Knoten (*Vertices*), Kanten (*Edges*), Knotenlisten (*VertexLists*), Kantenlisten (*EdgeLists*) und für Graphen (Graphs). Da diese Typen von uns definiert wurden, sind noch einige Hilfsdefinitionen wie ein Gleichheitstest oder Mengebeziehungen wichtig, um z.B. die Nachfolger eines bestimmten Knotens zu bestimmen. Auf diese Definitionen werden dann weitere graphenspezifische Definitionen aufgebaut.

Es wurde versucht bei den Definitionen die grundlegende Graphendefinitionen zusammenzufassen, ohne die entstehende Theorie unnötig groß zu machen. Auch wurde versucht die Definitionen möglichst einfach zu halten, um nicht den Umgang mit ihnen unnötig zu erschweren. Im folgenden werden alle Definitionen einzeln erläutert und die jeweilige Displayform und Abstraktion vorgestellt. Die Abstraktion benutzt die Displayform, wie in NuPRL, was zu gleich aussehenden Termen führen kann. Genauere Abstraktionen können in der NuPRL Implementation gefunden werden. Eine tabellarische Zusammenfassung aller Definitionen ist im Anhang zu finden.

4.1.1 Definitionen für Knoten

4.1.1.1 Menge aller Knoten (*gt_Vertices*)

Der Basisdatentyp der Graphentheorie ist der Datentyp Knoten, welcher durch die Abstraktion *gt_Vertices* realisiert wird. Dies entspricht der Menge aller Knoten, wobei diese Knotenmenge auf die Menge der natürlichen Zahlen ($\mathbb{N}$) abgebildet wird. Damit sind die Knoten in gewisser Weise benannt. Dies ist nötig, da Knoten sonst ununterscheidbar wären und damit nicht vernünftig verwendbar.

Displayform: *Vertices*
Abstraktion: $Vertices == \mathbb{N}$

4.1.1.2 Konstruieren von Knoten (*gc_Vertex*)

Nachdem wir den Datentyp definiert haben, muß ein Konstruktor definiert werden, mit dem man bestimmte Knoten erstellen kann. Diese Aufgabe erfüllt die Abstraktion *gc_Vertex*, welche eine natürliche Zahl als Knoten repräsentiert.

Displayform: $v < number >$
Abstraktion: $vx == x$

4.1.1.3 Vergleich auf Knoten (*g{b/f}_EqualVertex*)

Die Funktionen *g{b/f}_EqualVertex* sind nötig, um zwei Knoten zu vergleichen. Dies hat den Vorteil, dass die Repräsentation unter dem Knoten (in diesem Fall die natürlichen Zahlen) gewechselt werden kann. Das Ergebnis liefert *btrue* bzw. *True*, wenn die beiden übergebenen Knoten gleich sind. Diese Gleichheit wird über die Gleichheit der natürlichen Zahlen ($\mathbb{N}$) abgebildet. Dieser Vergleich ist immer entscheidbar.

Displayform: $< a >\equiv_b< b >$
Abstraktion: $a \equiv_b b == (a =_Z b)$
Displayform: $< a >\equiv< b >$
Abstraktion: $a \equiv b == (\uparrow (a \equiv_b b))$

4.1.2 Definitionen für Kanten

4.1.2.1 Menge aller Kanten (*gt_Edges*)

Die Menge aller Kanten wird durch die Abstraktion *gt_Edges* definiert, welche als Produkt von zwei Knoten definiert ist. Diese Kante entspricht einer gerichteten Kante. Dabei ist der erste Knoten im Produkt der Quellknoten/Ursprungsknoten/Anfangsknoten (source) und der zweite Knoten der Zielknoten (destination).

Displayform: *Edges*
Abstraktion: $Edges == Vertices \times Vertices$

4.1.2.2 Konstruktion von Kanten (*gc_Edge*)

Um eine bestimmte Kante zu erstellen, muß ein Konstruktor definiert werden. Dieser Konstruktor verbindet zwei bestimmte Knoten zu einer Kante.

Displayform: $< a >\rightarrow< b >$
Abstraktion: $a \rightarrow b ==< a, b >$

4.1.2.3 Ursprungsknoten einer Kante (*gf_Source*)

Dieses Funktion liefert den ersten Knoten, d.h. den Quellknoten (source) einer übergebenen Kante zurück.

Displayform: $< a > .s$
Abstraktion: $a.s == a.1$

4.1.2.4 Zielknoten einer Kante (*gf_Destination*)

Dieses Funktion gibt den zweiten Knoten, d.h. den Zielknoten (destination) der übergebenen Kante zurück.

Displayform: $< a > .d$
Abstraktion: $a.d = a.2$

4.1.2.5 Vergleich auf Kanten (*g{b/f}_EqualEdge*)

Mithilfe dieser Funktion können zwei Kanten verglichen werden, wobei *btrue* bzw. *True* bei Gleichheit zurückgeliefert wird. Zwei Kanten sind gleich, wenn sie die gleichen Quellknoten und die gleichen Zielknoten haben. Dieser Vergleich ist immer entscheidbar, da wir nur jeweils zwei Knoten miteinander vergleichen.

Displayform: $< a >\equiv_b < b >$
Abstraktion: $a \equiv_b b == (a.s \equiv_b b.s \wedge_b a.s \equiv_b b.s)$
Displayform: $< a >\equiv < b >$
Abstraktion: $a \equiv b == (\uparrow (a \equiv_b b))$

4.1.3 Definitionen für Knotenmengen

4.1.3.1 Testen ob ein Knoten in der Liste ist (*g{b/f}_IsVertexInList*)

Diese Funktionen sind wichtig um den Datentyp Knotenmenge zu erstellen. Da wir Knotenmengen betrachten wollen, müssen wir testen, ob es sich bei einer Liste um eine Menge handelt. Dazu haben wir als ersten Schritt diese Funktion erstellt, welche testet ob ein Knoten in einer Liste von Knoten ist oder nicht. Somit dient als Eingabe ein Knoten und eine Liste von Knoten.

Displayform: $< v >\in_b < l >$
Abstraktion: $v \in_b l == (rec - case(l)\ of\ [] \Rightarrow ff | h :: T \Rightarrow rec.v \equiv_b h \wedge_b rec)$
Displayform: $< v >\in < l >$
Abstraktion: $v \in l == (\uparrow (v \in_b l))$

4.1.3.2 Testen ob eine Knotenliste eine Knotenmenge ist (*g{b/f}_IsVertexSet*)

Diese Funktionen dienen zum testen, ob es sich bei der Eingabeliste um eine Menge handelt, d.h. ob jeder Knoten der Liste von Knoten genau einmal vorkommt. Wenn dies der Fall ist wird ein *btrue* bzw. *True* zurückgegeben. Diese Funktionen werden für den Teilmengentyp *gt_VertexLists* benutzt.

Displayform: $IsVertexSet(< l >)$
Abstraktion: $IsVertexSet(l) == (rec - case(l)\ of\ [] \Rightarrow tt | h :: t \Rightarrow rec.(\neg_b h \in_b t) \vee rec))$
Displayform: $IsVertexSet(< l >)$
Abstraktion: $IsVertexSet(l) == (\uparrow IsVertexSet(l))$

4.1.3.3 Menge aller Knotenmengen (*gt_VertexLists*)

Dieser Datentyp wird auf den Teilmengentypen abgebildet, wobei alle Knotenlisten enthalten sind, in denen jeder Knoten höchstes einmal vorkommt. Obwohl wir Listen aus der *NuPRL* Objekttheorie benutzen, können wir diese, mithilfe noch folgender Funktionen, als Menge annehmen. Wir haben diesen Schritt gewählt, da dies die einfachste Möglichkeit war, eine Menge von Knoten und später auch Kanten abzubilden. Im weiteren werden wir von Knotenmengen sprechen, obwohl es eigentlich Knotenlisten sind.

Displayform: $VertexLists$
Abstraktion: $VertexLists = \{vl : Vertices\ list | \uparrow IsVertexSet(vl)\}$

4.1.3.4 Testen ob ein Knoten in der Menge ist (*g{b/f}_IsVertexIn*)

Diese Funktionen prüfen, ob der gegebene Knoten in einer bestimmten Knotenmenge ist. Dies ist entscheidbar, da die Menge endlich ist und der Vergleich der Knoten entscheidbar ist.

Displayform: $< v >\in_b < l >$
Abstraktion: $v \in_b l == (v \in_b l)$
Displayform: $< v >\in < l >$
Abstraktion: $v \in l == (\uparrow v \in_b l)$

4.1.3.5 Erweitern der Knotenmenge (*gf_AddVertex*)

Diese Funktion fügt zu einer gegebenen Knotenmenge einen neuen Knoten hinzu, wenn er noch nicht in der Knotenmenge vorhanden ist. Das Ergebnis ist die neue Knotenmenge.

Displayform: $< v > . < l >$
Abstraktion: $v.l == (if\ v \in_b l\ then\ l\ else\ [v/l]\ fi)$

4.1.3.6 Kardinalität einer Knotenmenge (*gf_VertexCount*)

Diese Funktion gibt die Anzahl der Knoten in einer Knotenmenge zurück.

Displayform: $\| < vl > \|$
Abstraktion: $\|vl\| == (\|vl\|)$

4.1.3.7 Vereinigung von Knotenmengen (*gf_UnionVertexLists*)

Dieses Funktion vereinigt zwei Knotenmengen zu einer neuen Knotenmenge, wobei alle Knoten der beiden Knotenmengen nur einmal in der resultierenden Knotenmenge vorkommen.

Displayform: $< l1 > \cup < l2 >$
Abstraktion: $l1 \cup l2 == (rec - case(l1)\ of\ [] \Rightarrow l2 | h :: t \Rightarrow rec.h.rec)$

4.1.3.8 Durchschnitt von Knotenmengen (*gf_IntersectionVertexLists*)

Diese Funktion berechnet den Durchschnitt zweier Knotenlisten, d.h. die resultierende Knotenmenge enthält nur die Knoten, die in beiden Knotenmengen vorhanden waren.

Displayform: $< l1 > \cap < l2 >$
Abstraktion: $l1 \cap l2 == (rec - case(l1)\ of\ [] \Rightarrow [] | h :: t \Rightarrow rec.if\ h \in_b l2\ then\ h.rec\ else\ rec)$

4.1.3.9 Teilmengenbeziehung (*g{b/f}_IsSubsetOfVertexList*)

Diese Funktionen prüfen, ob die erste gegebene Knotenmenge eine Teilmenge der zweiten Knotenmenge ist. Dies ist entscheidbar, da die Mengen endlich sind und der Vergleich zweier Knoten entscheidbar ist.

Displayform: $< l1 > \subseteq_b < l2 >$
Abstraktion: $l1 \subseteq_b l2 == (rec - case(l1)\ of\ [] \Rightarrow tt | v :: rl \Rightarrow rec.v \in_b l2 \wedge_b rec)$
Displayform: $< l1 > \subseteq < l2 >$
Abstraktion: $l1 \subseteq l2 == (\uparrow l1 \subseteq_b l2)$

4.1.3.10 Vergleich von Knotenmengen (*g{b/f}_EqualVertexLists*)

Diese Funktionen vergleichen zwei Knotenmengen. Zwei Knotenmengen sind demnach gleich, wenn die eine Knotenmenge jeweils eine Teilmenge der anderen Knotenmenge ist.

Displayform: $< l1 > \equiv_b < l2 >$
Abstraktion: $l1 \equiv_b l2 == (l1 \subseteq_b l2 \wedge_b l2 \subseteq_b l1)$
Displayform: $< l1 > \equiv < l2 >$
Abstraktion: $l1 \equiv l2 == (\uparrow l1 \equiv_b l2)$

4.1.3.11 Startknoten der Knotenliste (*gf_FirstVertex*)

Diese Funktion liefert den ersten Knoten der Knotenliste (in dem Fall sprechen wir wirklich von Listen) zurück. Da wir uns für die Liste entschlossen haben, können wir auch den Startknoten bestimmen. Wenn die Knotenmenge leer ist, wird ein Null-Knoten $v0$ zugegeben. Ein Fehlerfall würde Beweise unnötig erschweren, der Fall der leeren Menge kann einfach überprüft werden.

Displayform: $< vl > .fst$
Abstraktion: $vl.fst == (case\ vl\ of\ [] \Rightarrow v0 | h :: t \Rightarrow h\ esac)$

4.1.4 Definitionen für Kantenlisten

4.1.4.1 Kantenlisten (*gt_EdgeLists*)

Dieser Datentyp stellt die Menge der Kantenlisten dar. Dabei sind Kantenlisten immer geordnet und es können doppelte Kanten enthalten sein. Für Graphen wären eventuell ungeordnete Kantenlisten besser, wobei dann auch auch eine weitere Definition von geordnete Kantenlisten mit all den dazugehörigen Definitionen nötig wäre, was die Theorie wahrscheinlich aber nur unnötig größer machen würde. Doppelte Kanten sind sowohl für die Definition von Graphen als auch für die Definition von Wegen und ähnlichem erwünscht.

Displayform: $EdgeLists$
Abstraktion: $EdgeLists = (Edges\ list)$

4.1.4.2 Erweitern der Kantenliste (*gf_AddEdge*)

Diese Funktion fügt eine Kante zu einer gegebenen Kantenliste hinzu, wobei diese an den Anfang der Liste gesetzt wird.

Displayform: $(< e >).\ < l >$
Abstraktion: $(e).l == ([e/l])$

4.1.4.3 Testen ob eine Kante in der Liste ist (*g{b/f}_IsEdgeIn*)

Diese Funktionen prüfen, ob die gegebene Kante in einer bestimmten Kantenliste existiert. Das Ergebnis ist *btrue* bzw. *True* oder *bfalse* bzw. *False*, falls die Kante nicht in der Liste zufinden ist. Dies ist entscheidbar, da die Mengen endlich sind und der Vergleich zweier Kanten entscheidbar ist.

Displayform: $< e >\in_b < l >$
Abstraktion: $e \in_b l == (rec - case(l)\ of\ [] \Rightarrow ff | h :: t \Rightarrow rec.e \equiv_b h \vee_b rec)$
Displayform: $< e >\in < l >$
Abstraktion: $e \in l == (\uparrow e \in_b l)$

4.1.4.4 Kardinalität der Kantenliste (*gf_EdgeCount*)

Diese Funktion gibt die Anzahl der Kanten in einer Kantenliste zurück.

Displayform: $\| < el > \|$
Abstraktion: $\|el\| == (\|el\|)$

4.1.4.5 Vereinigung von Kantenlisten (*gf_UnionEdgeLists*)

Diese Funktion vereinigt zwei gegebene Kantenlisten zu einer neuen Liste. Beide Listen werden aneinander gehängt und mehrfache Vorkommen bleiben erhalten, wobei die Symmetrie auf Listengleichheit nicht gewährleistet ist, aber auf die Mengengleichheit schon.

Displayform: $< l1 > \cup < l2 >$
Abstraktion: $l1 \cup l2 == (rec - case(l1)\ of\ [] \Rightarrow l2 | h :: t \Rightarrow rec.h.rec)$

4.1.4.6 Durchschnitt von Kantenlisten (*gf_IntersectionEdgeLists*)

Dies Funktion berechnet eine Art Listendurchschnitt zweier Kantenlisten, wobei dieser Durchschnitt bzgl. der Mengen- und Listengleichheit nicht symmetrisch ist. Das folgende Beispiel zeigt das Problem. Wenn von zwei Mengen $\{a\}$ und $\{a, a\}$ der Durchschnitt berechnet wird, ist das Ergebniss dieser Abstraktion, je nachdem in welche Reihenfolge gewählt wurde, entweder $\{a\}$ bzw. $\{a, a\}$, diese Listen sind ungleich bzgl. beider Gleichheiten.

Displayform: $< l1 > \cap < l2 >$
Abstraktion: $l1 \cap l2 == (rec - case(l1)\ of\ [] \Rightarrow []|h :: t \Rightarrow rec.if\ h \in_b l2\ then\ h.rec\ else\ rec)$

4.1.4.7 Strikter Durchschnitt von Kantenlisten (*gf_IntersectionStrictEdgeLists*)

Auf Grund der ersten Abstraktion zur Durchschnittsberechnung haben wir eine zweite, symmetrische Durchschnittsbildung erstellt. Diese Funktion berechnet den Listendurchschnitt zweier Kantenlisten, welcher bzgl. der Mengengleichheit symmetrisch ist. Wenn das gleiche Beispiel aus der obigen Abstraktion genommen wird, ist das resultierende Ergebniss $\{a\}$, egal in welcher Reihenfolge die Mengen der Abstraktion übergeben werden.

Displayform: $< l1 > \cap! < l2 >$
Abstraktion: $l1 \cap! l2 == (l2 \cap (l1 \cap l2))$

4.1.4.8 Teilmengenbeziehung von Kantenlisten (*g{b/f}_IsSubsetOfEdgeList*)

Diese Funktionen prüfen, ob die erste gegebene Kantenliste eine Teilmenge der zweiten Kantenliste ist, dabei wird auch die Anzahl der vorhandenen Kanten, deren Quell- und Zielknoten gleich sind, berücksichtigt. Dies ist entscheidbar, da die Listen endlich sind und der Vergleich zweier Knoten entscheidbar ist.

Displayform: $< l1 > \subseteq_b < l2 >$
Abstraktion: $l1 \subseteq_b l2 == (||l1|| =_z ||l1 \cap! l2||)$
Displayform: $< l1 > \subseteq < l2 >$
Abstraktion: $l1 \subseteq l2 == (\uparrow l1 \subseteq_b l2)$

4.1.4.9 Vergleich von Kantenlisten (*g{b/f}_EqualEdgeLists*)

Diese Funktionen vergleichen zwei Kantenlisten. Zwei Kantenlisten sind demnach gleich, wenn die eine Kantenliste jeweils eine Teilmenge der anderen Kantenliste ist.

Displayform: $< l1 > \equiv_b < l2 >$
Abstraktion: $l1 \equiv_b l2 == (l1 \subseteq_b l2 \wedge_b l2 \subseteq_b l1)$
Displayform: $< l1 > \equiv < l2 >$
Abstraktion: $l1 \equiv l2 == (\uparrow l1 \equiv_b l2)$

4.1.4.10 Ermitteln aller Knoten in einer Kantenliste (*gf_GetVerticesOfEdges*)

Diese Funktionen liefert alle Knoten, die in den Kanten der übergebenen Kantenliste auftauchen. Das Ergebnis ist eine Knotenmenge, wobei natürlich alle Knoten nur einmal in der Menge enthalten sind.

Displayform: $< l > .vl$
Abstraktion: $l.vl == (rec - case(l)\ of\ [] \Rightarrow []|h :: t \Rightarrow rec.(h).s.(h).d.rec)$

4.1.4.11 Startkante eine Kantenliste (*gf_FirstEdge*)

Diese Funktionen liefert die erste Kante der gegebenen Kantenliste zurück. Ist die übergebene Kantenliste leer wird eine Null Kante zurückgegeben, die von einem Knoten v0 zu einem Knoten v0 geht. Eine Fehlermeldung in diesem Fall würde eine spätere Benutzung dieser Funktion sicher erschweren, da der Fall einer leeren Liste relativ leicht anders abgefangen werden kann (z.B. mit $gf\_EdgeCount = 0$).

Displayform: $< l > .fst$
Abstraktion: $l.fst == (case\ l\ of\ [] \Rightarrow v0 \rightarrow v0|h :: t \Rightarrow h\ esac)$

4.1.4.12 Zielkante einer Kantenliste (*gf_LastEdge*)

Mithilfe diese Funktionen wird die letzte Kante der übergebenen Kantenliste ermittelt. Wie auch *gf_FirstEdge*, gibt diese Funktion eine Null Kante zurück, wenn die gegebene Kantenliste leer ist.

Displayform: $< l > .lst$
Abstraktion: $l.lst == (rec - case(l)\ of\ [] \Rightarrow v0 \rightarrow v0|h1 :: t \Rightarrow rec.case\ t\ of [] \Rightarrow h1|h2 :: t2 \Rightarrow rec\ esac)$

4.1.4.13 Liefere die Kanten mit einem bestimmten Quellknoten (*gf_GetEdgesBySource*)

Diese Funktion liefert alle Kanten, der übergebenen Kantenliste, zurück, die den übergebenen Knoten als Quellknoten haben. Das Ergebnis ist eine Kantenliste.

Displayform: $GetEdgesBySource(< v >; < el >)$
Abstraktion: $GetEByS(v; el) == (rec - case(el)\ of\ [] \Rightarrow []|h :: tl \Rightarrow rec.if\ v \equiv_b (h).s\ then\ (h).rec\ else\ rec\ fi)$

4.1.4.14 Liefere die Kanten mit einem bestimmten Zielknoten (*gf_GetEdgesByDestination*)

Diese Funktion liefert alle Kanten, der übergebenen Kantenliste, die den übergebenen Knoten als Zielknoten haben, in einer Kantenliste zurück.

Displayform: $GetEdgesByDestination(< v >; < el >)$
Abstraktion: $GetEByD(v; el) == (rec - case(el)\ of\ [] \Rightarrow []|h :: tl \Rightarrow rec.if\ v \equiv_b (h).d\ then\ (h).rec\ else\ rec\ fi)$

4.1.4.15 Liefert alle Nachfolger eines Knotens zu einer Kantenliste (*gf_SuccessorsForEdgeList*)

Diese Funktion liefert alle Nachfolgerknoten eines Knotens bezüglich der übergebenen Kantenliste zurück. Also alle Knoten die als Zielknoten der Kanten aus der Kantenliste vorkommen, welche den übergebenen Knoten als Quellknoten besitzen.

Displayform: $SuccessorsForEdgeList(< v >; < el >)$
Abstraktion: $SuccessorForEdgeList(v; el) == (rec-case(el)\ of\ [] \Rightarrow []|h :: tl \Rightarrow rec.if\ (h).s \equiv_b v\ then\ (h).d.rec\ else\ rec\ fi)$

4.1.4.16 Nachfolger einer Knotenmenge zu einer Kantenliste (*gf_SuccessorsOfVerticesForEdgeList*)

Diese Funktion liefert alle Nachfolgerknoten der Knoten aus der übergebenen Knotenliste bezüglich der übergebenen Kantenliste zurück. Also alle Knoten die als Zielknoten der Kanten aus der Kantenliste vorkommen, welche einen Knoten aus der übergebenen Knotenliste als Quellknoten besitzen.

Displayform: $SuccessorsForEdgeList(< vl >; < el >)$
Abstraktion: $SuccessorsForEdgeList(vl; el) == (rec - case(vl)\ of\ [] \Rightarrow []|h :: t \Rightarrow rec.$
$SuccessorsForEdgeList(h; el) \cup rec)$

4.1.4.17 Pfadcheck (*g{b/f}_IsPath*)

Diese Funktionen prüfen, ob die gegebene Kantenliste ein Pfad ist. Dies ist der Fall, wenn in der Kantenliste bei allen aufeinanderfolgenden Kanten, der Zielknoten der ersten Kante gleich dem Ursprungsknoten der zweiten Kante ist. Nach dieser Definition sind leere Kantenfolgen auch Pfade. Dies ist entscheidbar, da wir von endlichen Menge ausgehen und der Vergleich zweier Knoten entscheidbar ist.

Displayform: $IsPath(< el >)$
Abstraktion: $IsPath(el) == (rec - case(el)\ of\ [] \Rightarrow tt|h :: t \Rightarrow rec.case\ t\ of\ [] \Rightarrow tt|h1 :: t1 \Rightarrow (h).d \equiv_b (h1).s \wedge_b rec\ esac)$
Displayform: $IsPath(< el >)$
Abstraktion: $IsPath(el) == (\uparrow IsPath(el))$

4.1.4.18 Zyklencheck (*g{b/f}_IsCycle*)

Diese Funktionen prüfen, ob die gegebene Kantenliste ein Zyklus ist. Dies ist der Fall, wenn in der Kantenliste bei allen aufeinanderfolgenden Kanten, der Zielknoten der ersten Kante gleich dem Ursprungsknoten der zweiten

Kante ist und der Zielknoten der letzten Kante gleich dem Ursprungsknoten der ersten Kante ist. Nach dieser Definition sind leere Kantenfolgen auch Zyklen. Dies ist entscheidbar, da wir von endlichen Kantenmenge ausgehen und der Gleichheitstest zweier Knoten entscheidbar ist.

Displayform: $IsCycle(< el >)$
Abstraktion: $IsCycle(el) == ((el.fst).s \equiv_b (el.lst).d \wedge_b IsPath(el))$
Displayform: $IsCycle(< el >)$
Abstraktion: $IsCycle(el) == (\uparrow IsCycle(el))$

4.1.5 Definitionen für Graphen

4.1.5.1 Menge aller Graphen (*gt_Graphs*)

Die Definition *gt_Graphs* stellt den Datentypen dar, welcher die Menge aller Graphen enthält. Graphen sind Produkte aus Knotenmengen und Kantenlisten, wobei alle Knoten der Kantenliste auch in der Knotenmenge des Graphens enthalten sein müssen. Da der Graph auf den oben genannten Definitionen aufbaut, erbt er natürlich auch ihre Eigenschaften. Deshalb wird keine echte Knotenmenge verwendet. Bei der Kantenliste sind doppelte Kanten erwünscht, allerdings ist die Eigenschaft von der Kantenliste, geordnet zu sein, nicht erwünscht, wie auch bei den Knotenmengen. Eine weitere Definition für die Kantenliste würde allerdings die Theorie unnötig größer machen.

Displayform: $Graphs$
Abstraktion: $Graphs = \{gr : VertexLists \times EdgeLists | gr.2.vl \subseteq gr.1\}$

4.1.5.2 Knotenmenge eines Graphen (*gf_GetVertices*)

Diese Funktion gibt die Knotenmenge des gegebenen Graphen zurück.

Displayform: $< g > .vl$
Abstraktion: $g.vl == (g.1)$

4.1.5.3 Kantenliste eines Graphen (*gf_GetEdges*)

Diese Funktion gibt die Kantenliste des gegebenen Graphen zurück.

Displayform: $< g > .el$
Abstraktion: $g.el == (g.2)$

4.1.5.4 Erweitern des Graphen (*gf_AddVertexToGraph*)

Diese Funktion fügt den übergebenen Knoten zum Graphen hinzu, wenn er noch nicht enthalten ist, und gibt den entstandenen Graphen zurück.

Displayform: $AddVertexToGraph(< v >; < g >)$
Abstraktion: $AddVertexToGraph(v; g) == (< v.g.1, g.2 >)$

4.1.5.5 Test ob ein Knoten im Graphen ist (*g{b/f}_IsVertexInGraph*)

Diese Funktionen prüfen, ob der gegebene Knoten im Graphen zu finden ist oder nicht. Dies ist entscheidbar, da dieser Test nur eine Abstraktion des Tests auf Enthaltenseins in einer Knotenmenge ist.

Displayform: $< v > \in_b < g >)$
Abstraktion: $v \in_b g == (v \in_b g.1)$
Displayform: $< v > \in < g >)$
Abstraktion: $v \in g == (\uparrow v \in_b g)$

4.1.5.6 Erweitern des Graphen (*gf_AddEdgeToGraph*)

Diese Funktion fügt die übergebene Kanten zum Graphen hinzu und gibt den entstandenen Graphen zurück.

Displayform: $AddEdgeToGraph(< e >; < g >)$
Abstraktion: $AddEdgeToGraph(e; g) == (if\ (e).s \in_b g \wedge_b (e).d \in_b g\ then\ < g.vl, (e).g.el >\ else\ g\ fi)$

4.1.5.7 Test ob eine Kante im Graphen ist (*g{b/f}_IsEdgeInGraph*)

Diese Funktionen prüfen, ob die gegebene Kante im Graphen ist. Dies ist entscheidbar, da dies nur eine Abstraktion auf eine zuvor bei Kantenlisten definierte Funktion ist, welche selbst entscheidbar ist.

Displayform: $< e >\in_b < g >$)
Abstraktion: $e \in_b g == (e \in_b g.el)$
Displayform: $< e >\in < g >$)
Abstraktion: $e \in g == (\uparrow e \in_b g)$

4.1.5.8 Konstruktion eines Graphen (*gc_Graph*)

Diese Funktion erzeugt einen Graphen aus der übergebenen Knotenmenge und Kantenliste. Dabei werden Knoten die in der Kantenliste vorkommen, aber nicht in der Knotenmenge sind, zu der Knotenmenge des Graphen hinzugefügt.

Displayform: $G(< vl >; < el >)$
Abstraktion: $G(vl; el) == (< vl \cup el.vl, el >)$

4.1.5.9 Leerer Graph (*g{b/f}_EmptyGraph*)

Diese Funktionen prüfen, ob der gegebene Graph leer ist, d.h. ob er sowohl keine Kanten als auch keine Knoten enthält. Dies ist entscheidbar, da man nur zeigen muß, dass die beiden Listen leer sind.

Displayform: $Empty(< g >)$
Abstraktion: $Empty(g) == (0 =_z ||g.el|| \wedge_b 0 =_z ||g.vl||)$
Displayform: $Empty(< g >)$
Abstraktion: $Empty(g) == (\uparrow Empty(g))$

4.1.5.10 Gleichheit auf Graphen (*g{b/f}_EqualGraphs*)

Diese Funktion liefert *btrue* bzw. *True* zurück, wenn die beiden gegebenen Graphen gleich sind. Zwei Graphen sind gleich, wenn sie die gleichen Knotenmengen und Kantenliste besitzen. Dieser Vergleich ist immer entscheidbar.

Displayform: $< g1 >\equiv_b < g2 >$)
Abstraktion: $g1 \equiv_b g2 == (g1.vl \subseteq_b g2.vl \wedge_b g2.vl \subseteq_b g1.vl \wedge_b g1.el \subseteq_b g2.el \wedge_b g2.el \subseteq_b g1.el)$
Displayform: $< g1 >\equiv < g2 >$)
Abstraktion: $g1 \equiv g2 == (\uparrow g1 \equiv_b g2)$

4.1.5.11 Sub-Graph (*g{b/f}_SubGraph*)

Diese Funktionen liefern *btrue* bzw. *True* zurück, wenn der erste übergebene Graph ein Teilgraph des zweiten übergebenen Graphen ist. Der erste Graph ist ein Teilgraph des Zweiten, wenn seine Knotenmenge bzw. Kantenliste eine Teilmenge der Knotenmenge bzw. Kantenliste des zweiten Graphen ist. Dieser Vergleich ist immer entscheidbar.

Displayform: $< g1 >\subseteq_b < g2 >$)
Abstraktion: $g1 \subseteq_b g2 == (g1.vl \subseteq_b g2.vl \wedge_b g1.el \subseteq_b g2.el)$
Displayform: $< g1 >\subseteq < g2 >$)
Abstraktion: $g1 \subseteq g2 == (\uparrow g1 \subseteq_b g2)$

4.1.5.12 Vereinigung zweier Graphen (*gf_UnionGraphs*)

Diese Funktion vereinigt die beiden übergebenen Graphen zu einem neuen Graphen. Die Vereinigung zweier Graphen, ist der Graph, in dem ihre Knotenmengen bzw. Kantenlisten vereinigt sind.

Displayform: $< g1 > \cup < g2 >$
Abstraktion: $g1 \cup g2 == G(g1.vl \cup g2.vl; g1.el \cup g2.el)$

4.1.5.13 Durchschnitt von Graphen (*gf_IntersectionGraphs*)

Diese Funktion schneidet die beiden übergebenen Graphen zu einem neuen Graphen. Der Durchschnitt zweier Graphen, ist der Graph, in dem ihre Knotenmengen bzw. Kantenlisten geschnitten sind.

Displayform: $< g1 > \cap < g2 >$
Abstraktion: $g1 \cap g2 == G(g1.vl \cap g2.vl; g1.el \cap !g2.el)$

4.1.5.14 Nachbarschaftstest (*g{b/f}_Neighbour*)

Diese Funktion testet, ob zwei Knoten in einem Graphen benachbart, also adjazent, sind. Das Ergebnis ist entweder *btrue* bzw. *True* oder *bfalse* bzw. *False*.

Displayform: $v1 \leftrightarrow_b v2\ in\ g?$
Abstraktion: $v1 \leftrightarrow_b v2\ in\ g? == (rec\text{-}case(g.el)\ of\ [] \Rightarrow f\ f|h :: t \Rightarrow rec.(((h).s \equiv_b v1 \wedge_b (h).d \equiv_b v2) \vee_b ((h).s \equiv_b v2 \wedge_b (h).d \equiv_b v1)) \vee_b rec)$
Displayform: $v1 \leftrightarrow v2\ in\ g?$
Abstraktion: $v1 \leftrightarrow v2\ in\ g? == (\uparrow v1 \leftrightarrow_b v2\ in\ g?)$

4.1.5.15 Unabhängige Knoten eines Graphen (*g{b/f}_IndependentVertices*)

Diese Funktion testet, ob zwei Knoten in einem Graphen nicht benachbart, also nicht adjazent, sind. Das Ergebnis ist dann *btrue* bzw. *True*, sonst *bfalse* bzw. *False*.

Displayform: $v1\neg \leftrightarrow_b v2\ in\ g?$
Abstraktion: $v1\neg \leftrightarrow_b v2\ in\ g? == \neg(v1 \leftrightarrow_b v2\ in\ g?)$
Displayform: $v1\neg \leftrightarrow v2ing?$
Abstraktion: $v1\neg \leftrightarrow v2\ in\ g? == (\uparrow v1\neg \leftrightarrow v2\ in\ g?)$

4.1.5.16 Abhänigige Kanten eines Graphen (*g{b/f}_DependentEdges*)

Diese Funktion testet, ob zwei Kanten in einem Graphen benachbart, also adjazent, sind. Das Ergebnis ist dann *btrue* bzw. *True*, sonst *bfalse* bzw. *False*. Zwei Kanten sind benachbart, wenn sie einen gemeinsamen Knoten besitzen.

Displayform: $e1 \leftrightarrow_b e2$
Abstraktion: $e1 \leftrightarrow_b e2 == (e1.s \equiv_b e2.s \vee_b e1.d \equiv_b e2.s \vee_b e1.s \equiv_b e2.d \vee_b e1.d \equiv_b e2.d)$
Displayform: $e1 \leftrightarrow e2$
Abstraktion: $e1 \leftrightarrow e2 == (\uparrow e1 \leftrightarrow_b e2)$

4.1.5.17 Unabhänigige Kanten eines Graphen (*g{b/f}_IndependentEdges*)

Diese Funktion testet, ob zwei Kanten in einem Graphen benachbart, also adjazent, sind. Das Ergebnis ist dann *bfalse* bzw. *False*, sonst *btrue* bzw. *True*. Zwei Kanten sind benachbart, wenn sie einen gemeinsamen Knoten besitzen.

Displayform: $e1\neg \leftrightarrow_b e2$
Abstraktion: $e1\neg \leftrightarrow_b e2 == (\neg(e1 \leftrightarrow_b e2))$
Displayform: $e1\neg \leftrightarrow e2$
Abstraktion: $e1\neg \leftrightarrow e2 == (\uparrow e1\neg \leftrightarrow_b e2)$

4.1.5.18 Nachfolger eines Knotens bzgl. eines Graphens (*gf_Successors*)

Diese Funktion liefert alle Nachfolgerknoten eines Knotens des übergebenen Graphen. Also alle Knoten die als Zielknoten von Kanten des Graphen vorkommen, welche den übergebenen Knoten als Quellknoten besitzen.

Displayform: $Successors(< v >; < g >)$
Abstraktion: $Successors(v; g) == (SuccessorsForEdgeList(v; g.el)$

4.1.5.19 Nachfolger einer Knotenmenge bzgl. eines Graphens (*gf_SuccessorsOfVertices*)

Diese Funktion liefert alle Nachfolgerknoten der Knoten aus der übergebenen Knotenliste bezüglich des übergebenen Graphen zurück. Also alle Knoten die als Zielknoten der Kanten des Graphen vorkommen, welche einen Knoten aus der übergebenen Knotenliste als Quellknoten besitzen.

Displayform: $Successors(< vl >; < g >)$
Abstraktion: $Successors(vl; g) == (SuccessorsForEdgeList(vl; g.el))$

4.1.5.20 Vorgänger eines Knotens bzgl. eines Graphens (*gf_Predecessors*)

Diese Funktion liefert alle Vorgängerknoten eines Knotens des übergebenen Graphen. Also alle Knoten die als Quellknoten von Kanten des Graphen vorkommen, welche den übergebenen Knoten als Zielknoten besitzen.

Displayform: $Predecessors(< v >; < g >)$
Abstraktion: $Predecessors(v; g) == (rec{-}case(g.el)\ of\ [] \Rightarrow []|h :: t \Rightarrow rec.if\ (h).d \equiv_b v\ then\ (h).s.rec\ else\ rec)$

4.1.5.21 Vorgänger einer Knotenmenge bzgl. eines Graphens (*gf_PredecessorsOfVertices*)

Diese Funktion liefert alle Vorgängerknoten der Knoten aus der übergebenen Knotenliste bezüglich des übergebenen Graphen zurück. Also alle Knoten die als Quellknoten der Kanten des Graphen vorkommen, welche einen Knoten aus der übergebenen Knotenliste als Zielknoten besitzen.

Displayform: $Predecessors(< vl >; < g >)$
Abstraktion: $Predecessors(vl; g) == (rec - case(vl)\ of\ [] \Rightarrow []|h :: t \Rightarrow rec.Predecessors(h; g) \cup rec)$

4.1.5.22 Verbundene Knoten (*gf_ConnectedVertices*)

Diese Funktion prüft, ob zwei Knoten des Graphen verbunden sind. Zwei Knoten des Graphen sind verbunden, wenn es einen Pfad im Graphen von einem zum anderen Knoten gibt. Trotz der Entscheidbarkeit dieses Problems, wird leider aufgrund der Definition keine boolsche Funktion bereitgestellt. Andere in Erwägung gezogene Definitionen gestalteten sich leider als zu schwierig.

Displayform: $ConnectVertices(< v1 >, < v2 >, < g >)$
Abstraktion: $ConnectVertices(v1, v2, g) = (\exists el : EdgeLists.\ \uparrow (IsPath(el) \wedge_b el \subseteq g.el \wedge_b ((v1 \equiv_b (el.fst).s \wedge_b v2 \equiv_b (el.lst).d) \vee_b (v1 \equiv_b (el.lst).d \wedge_b v2 \equiv_b (el.fst).s))))$

4.1.5.23 Test auf Hamiltonpfad (*g{b/f}_IsHamiltonPath*)

Diese Funktion überprüft, ob der übergebene Kantenliste für den übergebenen Graphen ein hamiltonischer Pfad ist, d.h. sie ist ein Pfad bestehend nur aus Kanten des Graphen und durchläuft alle Knoten des Graphen genau einmal, wobei Start- und Endknoten des Pfades als einmal durchlaufen gelten, sie sind also demnach im Hamiltonpfad unterschiedlich. Das Ergebnis ist *btrue* bzw. *True* wenn die Kantenliste ein hamiltonischer Pfad ist, sonst *bfalse* bzw. *False*.

Displayform: $IsHamiltonPath(< l >; < g >)$
Abstraktion: $IsHamiltonPath(l; g) == (IsPath(l) \wedge_b (||l|| =_z ||g.vl|| - 1) \wedge_b l.vl \equiv_b g.vl)$
Displayform: $IsHamiltonPath(< l >; < g >)$
Abstraktion: $IsHamiltonPath(l; g) == (\uparrow IsHamiltonPath(l; g))$

4.1.5.24 Test auf Hamiltonzyklus (*g{b/f}_IsHamiltonCylcle*)

Diese Funktion überprüft, ob die übergebene Kantenlist für den übergebenen Graphen ein hamiltonischer Zyklus ist, d. h. sie ist ein Zyklus bestehend nur aus Kanten des Graphen und durchläuft alle Knoten des Graphen bis auf den Startknoten genau einmal, wobei natürlich der Startknoten des Zyklus gleich seinem Endkonten ist. Das Ergebnis ist *btrue* bzw. *True* wenn die Kantenliste ein hamiltonischer Zyklus ist, sonst *bfalse* bzw. *False*.

Displayform: $IsHamiltonCycle(< l >; < g >)$
Abstraktion: $IsHamiltonCycle(l; g) == (IsCycle(l) \wedge_b (||l|| =_z ||g.vl||) \wedge_b l.vl \equiv_b g.vl)$
Displayform: $IsHamiltonCycle(< l >; < g >)$
Abstraktion: $IsHamiltonCycle(l; g) == (\uparrow IsHamiltonCycle(l; g))$

4.2 Theoreme

Mithilfe einer kleinen Liste von Theoremen sind die Beweise von Wohlgeformtheitszielen oder anderen Theoremen bedeutend vereinfacht worden. Die folgenden Theoreme sind nicht immer in einer mathematischen Theorie sinnvoll, aber für die hier definierten Graphen sehr hilfreich. In der mathematischer Sicht kann von gewissen Voraussetzungen ausgegangen werden, wie zum Beispiel der Menge als Basisdatentyp.

4.2.1 `sublist_of_set_is_set`

`sublist_of_set_is_set` ist ein einfaches Theorem, mit dem gezeigt wird, dass wenn von einer Listen, die das Mengenkriterium erfüllt das jedes Element in ihr nur eimal vorkommt, das erste Element entfernt wird, sie dann immernoch das Kriterium erfüllt. Dieses Theorem ist wichtig, da wir Listen als Mengen nutzen und somit die Eigenschaft, das jedes Element nur einmal enthalten, nicht sichergestellt werden kann.

Syntax: $\forall v : Vertices.\forall vl : Vertices\ list.(IsVertexSet([v/vl]) \Rightarrow IsVertexSet(vl))$

4.2.2 `vertexset_is_vertexlist`

`vertexset_is_vertexlist` sagt aus, dass jede Knotenmenge einer *NuPRL* Knotenliste entspricht. Dies ist nur in unseren Abstraktionen möglich und nötig, da wir die *NuPRL* Listen als Mengen nutzen und dadurch einige Beweise durch dieses Theorem vereinfacht werden.

Syntax: $\forall l : VertexLists.(l \in Vertices\ list)$

4.2.3 `addvertex_set_consistence`

`addvertex_set_consistence` sagt aus, wenn ein Knoten mittels `gf_AddVertex` zu einer Knotenmenge hinzugefügt wird, sie weiterhin eine Menge bleibt.

Syntax: $\forall v : Vertices.\forall vl : VertexLists.(IsVertexSet(vl) \Leftrightarrow IsVertexSet(v.vl))$

4.2.4 `triple_band`

`triple_band` ist eine triviales Lemma welches zeigt, das *assert* auf drei bool-Ausdrücke, genauso funktioniert, wie auf zwei bool-Ausdrücke. Dieses Lemma ist analog zum Lemma des $\mathbb{B}$ Datentyps `assert_of_band`, es vereinfacht lediglich unsere Beweise.

Syntax: $\forall a, b, c : \mathbb{B}.(\uparrow (a \wedge_b b \wedge_b c) \Leftrightarrow \uparrow a \wedge \uparrow b \wedge \uparrow c))$

4.2.5 `symetric_equal_vertex`

`symetric_equal_vertex` sagt aus, dass der Gleichheitstest symmetrisch ist für Knoten.

Syntax: $\forall v, v2 : Vertices.(v \equiv v2 \Leftrightarrow v2 \equiv e)$

4.2.6 symetric_equal_edge

symetric_equal_edge sagt aus, dass der Gleichheitstest symmetrisch ist für Kanten.

Syntax: $\forall e, e2 : Edges.(e \equiv e2 \Leftrightarrow e2 \equiv e)$

4.2.7 reorder_first_and_second_vertex

reorder_first_and_second_vertex sagt aus, dass bei einer Menge von Knoten die ersten beiden Knoten vertauscht werden können, ohne die Mengeneigenschaft zu verletzen. Dieses Theorem ist sinnvoll, da wir nicht von reinen Mengen ausgehen, sondern Listen betrachten, diese sich jedoch auch wie Mengen verhalten, wenn wir keine neuen Elemente hinzufügen und die Ordnung vernachlässigen.

Syntax: $\forall v, v2 : Vertices.\forall vl : Vertices\ list.(IsVertexSet([v; v2/vl]) \Leftrightarrow IsVertexSet([v2; v/vl]))$

4.2.8 union_of_vertex_list_is_list_of_vertices

union_of_vertex_list_is_list_of_vertices sagt aus, dass das Vereinigen von zwei Knotenmengen immer eine Liste von Knoten ergibt. Dieses Theorem ist wichtig für das Wohlgeformtheitsziel der Abstraktion *gf_UnionVertexLists*.

Syntax: $\forall l, l2 : VertexLists.(l \cup l2 \in Vertices\ list)$

4.2.9 intersection_of_vertex_list_is_list_of_vertices

intersection_of_vertex_list_is_list_of_vertices sagt aus, dass der Durchschnitt von zwei Knotenmengen immer eine Liste von Knoten ergibt. Dieses Theorem ist wichtig für das Wohlgeformtheitsziel der Abstraktion *gf_IntersectionVertexLists*.

Syntax: $\forall l, l2 : VertexLists.(l \cap l2 \in Vertices\ list)$

4.2.10 subset_growth

subset_growth sagt aus, dass die Teilmengenbeziehung erhalten bleibt, wenn die Obermenge erweitert wird, d.h. um einen Knoten vergrößert.

Syntax: $\forall k : Vertices.\forall vl1, vl2 : VertexLists.(vl1 \subseteq vl2 \Rightarrow vl1 \subseteq k.vl2)$

4.2.11 subset_equality

subset_equality beschreibt die Eigenschaft der Reflexivität der Teilmengenrelation auf Knotenmenge. Eine Knotenmenge ist gleich zu sich selbst, damit ist die Teilmengebeziehung auch erfüllt.

Syntax: $\forall s : VertexLists.(s \subseteq s)$

4.2.12 subset_reduce

subset_reduce sagt aus, dass wenn einer Menge mit *gf_Addvertex* ein Element hinzugefügt wird und sie dann Teilmenge einer anderen Menge ist, auch die ursprünglich Menge eine Teilmenge der anderen Menge ist.

Syntax: $\forall k : Vertices.\forall vl1, vl2 : VertexLists.(k.vl1 \subseteq vl2 \Rightarrow vl1 \subseteq vl2)$

4.2.13 add_vertex_same_successors

add_vertex_same_successors sagt aus, dass die Nachfolger eines Knotens gleich bleiben, nachdem ein neuer Knoten zum Graphen hinzugefügt wurde. Dies ist möglich, da wir nur einen neuen Knoten hinzufügen und keine neue Kante, somit sind die Nachfolger gleich.

Syntax: $\forall v, v2 : Vertices.\forall g : Graphs.(Successors(v; g) = Successors(v; AddVertexToGraph(v2; g)) \in VertexLists)$

4.2.14 AddVertex_IsIn

AddVertex_IsIn sagt aus, dass der Knoten nach hinzufügen zu einer Menge auf jeden Fall enthalten sein muß.

Syntax: $\forall v : Vertices.\forall vl : VertexLists.(v \in v.vl)$

4.2.15 member_list_growth

member_list_growth beschreibt eine wichtige Eigenschaft einer Knotenmenge. Diese Eigenschaft einer Menge/ Liste sagt aus, dass wenn ein Knoten in einer Knotenmenge ist, auch er sich nach Erweiterung (hinzufügen von neuen Knoten) weiterhin in dieser Knotenmenge befindet.

Syntax: $\forall k, k1 : Vertices.\forall vl : VertexLists.(k \in vl \Rightarrow k \in k1.vl)$

4.2.16 equal_int_to_nat

equal_int_to_nat sagt aus, dass die Gleichheit von zwei ganzen Zahlen auch in den natürlichen Zahlen gilt, wenn die beiden Zahlen aus den natürlichen Zahlen stammen.

Syntax: $\forall i, k : \mathbb{N}.(i = k \in \mathbb{Z} \Rightarrow i = k \in \mathbb{N})$

4.2.17 element_in_vertex_subset_consistence

element_in_vertex_subset_consistence sagt aus, dass ein Knoten, der in einer Teilmenge ($vl1$) einer Menge ($vl2$) ist, auch in der Menge ($vl2$) ist.

Syntax: $\forall v : Vertices.\forall vl1, vl2 : VertexLists.((v \in vl1 \wedge vl1 \subseteq vl2) \Rightarrow v \in vl2)$

4.2.18 inc_length_with_addvertex

inc_length_with_addvertex besagt, dass wenn ein Knoten, der nicht in der Menge ist, zu dieser hinzugefügt wird, die Menge danach um ein Element größer ist.

Syntax: $\forall l : VertexLists.\forall v : Vertices.(\neg k \in l \Leftrightarrow ||k.l|| = 1 + ||l||)$

4.2.19 same_length_with_addvertex

same_length_with_addvertex besagt, dass wenn ein Knoten, der in der Menge ist, zu dieser, mit Hilfe von gf_AddVertex, hinzugefügt wird, die Menge danach genauso groß ist.

Syntax: $\forall l : VertexLists.\forall v : Vertices.(k \in l \Leftrightarrow ||k.l|| = ||l||)$

4.3 Tactics

In diesem Abschnitt werden einige neue Taktiken vorgestellt, die uns das Beweisen von Teilzielen erleichtert haben. Die Taktiken sind nicht komplex, ermöglichen aber Beweise übersichtlicher und schneller zu lösen. Es wird jeweils angegeben, wie das jeweilige *tactical* abgebildet wird.

4.3.1 ListIndC c

ListIndC c ist eine Vereinfachung der Induktion über Listen.

Code:
```
let ListIndC c = ListInd c THEN Reduce 0 THEN Auto;;
```

4.3.2 Fold2 f c d

Fold2 f c d ist eine Vereinfachung des Zusammenfaltens von dem selben Ausdruck in zwei verschiedenen Slots (Hypothesen bzw. Konklusion).

Code:
```
let Fold2 f c d = Fold f c THEN Fold f d;;
```

4.3.3 `Unfold2 f c d`

`Unfold2 f c d` arbeitet genauso wie `Fold2`, allerdings wird hier der jeweilige Slot aufgefaltet.

Code: `let Unfold2 f c d = Unfold f c THEN Unfold f d;;`

4.3.4 `ReduceF f c`

`ReduceF f c` faltet eine Abstraktion *f* im Slot *c* auf, reduziert selbigen und faltet die Abstraktion wieder zusammen. Da viel mit Listen gerarbeitet wird, kann es oft vorkommen, das Konstrukte mit *cons* entstehen, welche reduziert werden können, wozu die Abstraktion aber erst aufgefaltet werden muß.

Code: `let ReduceF f c = Unfold f c THEN Reduce c THEN Fold f c;;`

5 Anmerkungen zur Implementation

Konzeptionell wäre es im nachhinein besser gewesen, erst einmal eine eigene Mengentheorie aufzubauen und darauf dann die Graphentheorie aufzusetzen, da in der Mathematik die Graphen auf den Mengen aufbauen. In der jetzigen Implementierung fehlen viele nützliche Mengenkonzepte, einige eingebrachte Mengenkonzepte liegen nur in relativ unhandlicher Form vor oder/und sind doppelt Implementiert (bei Knotenmengen und Kantenlisten). Ein Beispiel dafür ist die Vereinigungsfunktion die sowohl für Knotenmengen als auch für Kantenlisten vorliegt, für diese gibt es auch noch keine Theoreme für die Symmetrie, Reflexivität und Transitivität. Es ist zudem relativ umständlich diese Theoreme jedes mal neu zu Beweisen. Viele der in der Graphentheorie vorliegenden Theoreme sind auch eher Theoreme für Mengen als für Graphen.

Die vorliegende Dokumentation war oft nicht sehr hilfreich. Im vorliegenden *NuPRL* Handbuch wäre ein Index sicherlich hilfreich. Die Aufnahme der Basistheorien der *NuPRL* Bibliothek ist vielleicht nicht so sehr anzustreben, da die Bibliotek ständig erweitert und verändert wird. Hilfreich wäre hier eine Suchfunktion die nicht nur die Namen der Objekte in der Bibliothek durchsucht, sondern auch nach Konzepten suchen kann, z.B. eine Suche nach allen Theoremen die etwas mit dem Datentyp **bool** und dem Boolschen **bor** zu tun haben. Eine Möglichkeit die Ansichtart im *NuPRL* Navigator zu ändern, so dass z. B. nur die Theoreme oder Definitionen in einem Verzeichnis angezeigt werden, würde die Suche eventuell auch vereinfachen.

Technisch sind die ganzen auf- und zusammenfallt Schritte die bei Beweisen immer wieder vorkommen unschön. Dies wird unter anderem dadurch bedingt, dass viele unserer Abstraktionen nur auf einer anderen Definition aufbauen, welche eigentlich nur der Erleichterung von Beweisen dient. Ein Beispiel dafür sind die Definitionen `gb_IsVertexInList`, `gb_IsVertexIn` und `gf_IsVertexIn`, wobei die nachfolgende Definition jeweils auf der vorgehenden Definition aufbaut, oft musste `gf_IsVertexIn` bis zur Definition von `gb_IsVertexInList` aufgefaltet werden und nachher dann wieder zurück.

Weiterhin mussten in Beweisen sehr oft Teilbeweise, die zumindest ähnlich, wenn nicht sogar gleich waren, immer wieder neu oder zumindest ähnlich bewiesen werden. Die Einführung von neuen Taktiken ist oftmals unzweckmäßig, da diese Teilbeweise nur in einigen wenigen Beweisen oder auch nur einem Beweis vorkommen. Hier wäre eine Kopierfunktion angebracht mit der Teilbeweise in verschiedene Speicher kopiert werden können und dann je nach Bedarf wieder eingefügt und verändert werden können. Es wäre auch denkbar dass Beweisintern einfach auf andere Teilbeweise verwiesen wird oder eine Taktik die für Teilbeweise andere ähnliche Teilbeweisziele sucht und dann deren Beweis ausprobiert.

Sehr oft tritt auch die Endung `THEN Auto` in unseren Beweisen auf, eine Abkürzung hierfür wäre sinnvoll gewesen. z.B. `TA`.

Einige mal hat die `Auto` Taktik einen Beweis (entgegen der Erwartung) so fortgeführt das die entstandenen Teilbeweise nicht mehr beweisbar waren, dann konnte mit ihr nicht gearbeitet werden.

Sehr selten gab es auch Probleme mit Basisdefinitionen die nicht weiter aufgefaltet werden konnten, Beweise auf Basis ihrer Definitionen wurden damit unmöglich.

Einmal gab es das Problem das ein Teilbeweisziel mehr als die ganze Editorseite ausfüllte. Nach der Anzeige dieses Teilzieles arbeitete der Beweiseditor nur noch fehlerhaft, die ersten Zeilen wurden nicht mehr angezeigt und eine Möglichkeit zum Scrollen fehlte.

6 Zusammenfassung

Es wurde eine einfache Graphentheorie von gerichteten Graphen aufgestellt.

Sie basiert hauptsächlich auf die in *NuPRL* vorhandene Listentheorie und bool-Theorie. Die Verwendung der Listentheorie ist darauf zurückzuführen, dass keine Mengentheorie zur Zeit in *NuPRL* verfügbar war und das aufstellen einer neuen Mengentheorie, als zu aufwendig angesehen wurde. Die bool-Theorie wurde verwendet, um die Entscheidbarkeit einiger Probleme in die Theorie mit aufzunehmen. Mit der aufgestellten Graphentheorie ist es möglich graphentheoretische Beweise zu führen. Trotz das größtenteils nur grundlegenden Definitionen aufgenommen wurden, ist die aufgestellte Theorie recht umfangreich geworden. Alles in allem bietet sie aber schon eine gute Basis.

7 Ausblick

Die aufgestellte Theorie stellt nur einige Grundlagen der Graphentheorie bereit. Es gibt viele weitere nützliche Definition aus dem Beriech der Graphen, die aufbauend auf der bestehenden Theorie aufgestellt werden können. Beispiele hierfür sind ungerichtete Graphen, Graphen mit Labeln/Benennung von Knoten und/oder Kanten, Klicken und Isomorphiebeziehungen.

Auch können andere Theorien, die auf Graphen aufbauen, auf der aufgestellten Graphentheorie aufgebaut werden, z.B. Automatentheorien oder Ablauftheorien. Es können auch Beweise, die Graphen benutzen, geführt werden, wie zum Beispiel Komplexitätsbeweise.

Unabhängig davon wäre eine separate Mengentheorie nützlich, aufgrund der weiten Verbreitung dieser in der Mathematik. Dies würde sicherlich auch unsere Beweise vereinfachen.

Literatur

[1] Claude Berge. *Graphs*. NORTH-HOLLAND, ELSEVIER SIENCE PUBLISHER B.V., Sara Burgerhartstraat 25, P.O. Box 211, 1000 AE Amsterdam, The Netherlands, 1991. ISBN 0444876030.

[2] Prof. Dr. G. Biess. *Graphentheorie*. BSB B. G. Teubener Verlagsgesellschaft, Leipzig, 1988. ISSN 0138-1318.

[3] Christoph Kreitz. *Automatisierte Logik und Programmierung*. Technische Hochschule Darmstadt, Alexanderstr. 10, 64283 Darmstadt, 1995. Skript zur Vorlesung Automatisierte Logik und Programmierung.

[4] Christoph Kreitz. *The Nuprl Proof Development System, Version 5*. Department of Computer Science, Cornell-University, Ithaca, NY 14853-7501, U.S.A., 2002. Reference Manual and User's Guide.

[5] Christoph Kreitz. *Automatisierte Logik und Programmierung I und II*. Theoretische Informatik, University of Potsdam, 2003. Vorlesungsskript.

[6] Rainer Lang Rainer Bodendiek. *Lehrbuch der Graphentheorie Band 1/2*. Spektrum Akademischer Verlag GmbH, Heidelberg, 1995. ISBN 3-86025-667-x, ISBN 3-86025-668-8.

8 Anhang

Definition	Funktion	Display-Form
gt_Vertices	$\mathbb{N}$	*Vertices*
gc_Vertex(n)	$\mathbb{N} \rightarrow$ *gt_Vertex*	vn
gb_EqualVertex(v_1, v_2)	*gt_Vertices* $\times$ *gt_Vertices* $\rightarrow \mathbb{B}$	$v1 \equiv_b v2$
gf_EqualVertex(v_1, v_2)	*gt_Vertices* $\times$ *gt_Vertices* $\rightarrow \mathbb{P}_1$	$v1 \equiv v2$

Tabelle 1: Zusammenfassung der Definitionen für die Knoten

Definition	Funktion	Display-Form
gt_Edges	*gt_Vertices* $\times$ *gt_Vertices*	*Edges*
gc_Edge(v_1, v_2)	*gt_Vertices* $\times$ *gt_Vertices* $\rightarrow$ *gt_Edges*	$v1 \rightarrow v2$
gf_Source(e)	*gt_Edges* $\rightarrow$ *gt_Vertices*	$e.s$
gf_Destination(e)	*gt_Edges* $\rightarrow$ *gt_Vertices*	$e.d$
gb_EqualEdge(e_1, e_2)	*gt_Edges* $\times$ *gt_Edges* $\rightarrow \mathbb{B}$	$e_1 \equiv_b e_2$
gf_EqualEdge(e_1, e_2)	*gt_Edges* $\times$ *gt_Edges* $\rightarrow \mathbb{P}_1$	$e_1 \equiv e_2$

Tabelle 2: Zusammenfassung der Definitionen für die Kanten

Definition	Funktion	Display-Form
gt_VertexLists	*gt_Vertices list*	*VertexLists*
gb_IsVertexInList(v, $[v_i]$)	*gt_Vertices* $\times$ *gt_Vertices* list $\rightarrow \mathbb{B}$	$v \in_b [v_i]$
gf_IsVertexInList(v, $[v_i]$)	*gt_Vertices* $\times$ *gt_Vertices* list $\rightarrow \mathbb{P}_1$	$v \in [v_i]$
gb_IsVertexSet($[v_i]$)	*gt_Vertices* list $\rightarrow \mathbb{B}$	$IsVertexSet([v_i])$
gf_IsVertexSet($[v_i]$)	*gt_Vertices* list $\rightarrow \mathbb{P}_1$	$IsVertexSet([v_i])$
gb_IsVertexIn(v, $[v_i]$)	*gt_Vertices* $\times$ *gt_VertexLists* $\rightarrow \mathbb{B}$	$v \in_b [v_i]$
gf_IsVertexIn(v, $[v_i]$)	*gt_Vertices* $\times$ *gt_VertexLists* $\rightarrow \mathbb{P}_1$	$v \in [v_i]$
gf_AddVertex(v, $[v_i]$)	*gt_Vertices* $\times$ *gt_VertexLists* $\rightarrow$ *gt_VertexLists*	$v.[v_i]$
gf_VertexCount($[v_i]$)	*gt_VertexLists* $\rightarrow \mathbb{Z}$	$\|\|[v_i]\|\|$
gf_UnionVertexLists($[v_i]$, $[v_j]$)	*gt_VertexLists* $\times$ *gt_VertexLists* $\rightarrow$ *gt_VertexLists*	$[v_i] \cup [v_j]$
gf_IntersectionVertexLists($[v_j]$, $[v_i]$)	*gt_VertexLists* $\times$ *gt_VertexLists* $\rightarrow$ *gt_VertexLists*	$[v_i] \cap [v_j]$
gb_IsSubsetOfVertexList($[v_j]$, $[v_i]$)	*gt_VertexLists* $\times$ *gt_VertexLists* $\rightarrow \mathbb{B}$	$[v_i] \subseteq_b [v_j]$
gf_IsSubsetOfVertexList($[v_j]$, $[v_i]$)	*gt_VertexLists* $\times$ *gt_VertexLists* $\rightarrow \mathbb{P}_1$	$[v_i] \subseteq [v_j]$
gb_EqualVertexLists($[v_j]$, $[v_i]$)	*gt_VertexLists* $\times$ *gt_VertexLists* $\rightarrow \mathbb{B}$	$[v_i] \equiv_b [v_j]$
gf_EqualVertexLists($[v_j]$, $[v_i]$)	*gt_VertexLists* $\times$ *gt_VertexLists* $\rightarrow \mathbb{P}_1$	$[v_i] \equiv [v_j]$
gf_FirstVertex($[v_i]$)	*gt_VertexLists* $\rightarrow$ *gt_Vertices*	$[v_i].fst$

Tabelle 3: Zusammenfassung der Definitionen für die Knotenmengen

Definition	Funktion	Display-Form
gt_EdgeLists	*gt_Edges*list	*EdgeLists*
gf_AddEdge(e, $[e_i]$)	*gt_Edges* × *gt_EdgeLists*→ *gt_EdgeLists*	$(e).[e_i]$
gb_IsEdgeIn(e, $[e_i]$)	*gt_Edges* × *gt_EdgeLists*→ $\mathbb{B}$	$e \in_b [e_i]$
gf_IsEdgeIn(e, $[e_i]$)	*gt_Edges* × *gt_EdgeLists*→ $\mathbb{P}_1$	$e \in [e_i]$
gf_EdgeCount($[e_i]$)	*gt_EdgeLists*→ $\mathbb{Z}$	$\|\|[e_i]\|\|$
gf_UnionEdgeLists($[e_j]$, $[e_i]$)	*gt_EdgeLists* × *gt_EdgeLists*→ *gt_EdgeLists*	$[e_j] \cup [e_i]$
gf_IntersectionEdgeLists($[e_j]$, $[e_i]$)	*gt_EdgeLists* × *gt_EdgeLists*→ *gt_EdgeLists*	$[e_j] \cap [e_i]$
gf_IntersectionStrictEdgeLists($[e_j]$, $[e_i]$)	*gt_EdgeLists* × *gt_EdgeLists*→ *gt_EdgeLists*	$[e_j] \cap ![e_i]$
gb_IsSubsetOfEdgeList($[e_j]$, $[e_i]$)	*gt_EdgeLists* × *gt_EdgeLists*→ $\mathbb{B}$	$[e_j] \subseteq_b [e_i]$
gf_IsSubsetOfEdgeList($[e_j]$, $[e_i]$)	*gt_EdgeLists* × *gt_EdgeLists*→ $\mathbb{P}_1$	$[e_j] \subseteq [e_i]$
gb_EqualEdgeLists($[e_j]$, $[e_i]$)	*gt_EdgeLists* × *gt_EdgeLists*→ $\mathbb{B}$	$[e_j] \equiv_b [e_i]$
gf_EqualEdgeLists($[e_j]$, $[e_i]$)	*gt_EdgeLists* × *gt_EdgeLists*→ $\mathbb{P}_1$	$[e_j] \equiv [e_i]$
gf_GetVerticesOfEdges($[e_i]$)	*gt_EdgeLists*→ *gt_VertexLists*	$[e_i].vl$
gf_FirstEdge($[e_i]$)	*gt_EdgeLists*→ gt_Edge	$[e_i].fst$
gf_LastEdge($[e_i]$)	*gt_EdgeLists*→ gt_Edge	$[e_i].lst$
gf_GetEdgesBySource(v, $[e_i]$)	*gt_Vertices* × *gt_EdgeLists*→ *gt_EdgeLists*	$GetEdgesBySource(v, [e_i])$
gf_GetEdgesByDestination(v, $[e_i]$)	*gt_Vertices* × *gt_EdgeLists*→ *gt_EdgeLists*	$GetEdgesByDestination(v, [e_i])$
gf_SuccessorsForEdgeList(v, $[e_i]$)	*gt_Vertices* × *gt_EdgeLists*→ *gt_VertexLists*	$SuccessorsForEdgeList(v, [e_i])$
gf_SuccessorsOfVerticesForEdgeList($[v_j]$, $[e_i]$)	*gt_VertexLists* × *gt_EdgeLists*→ *gt_VertexLists*	$SuccessorsForEdgeList([v_j], [e_i])$
gb_IsPath($[e_i]$)	*gt_EdgeLists*→ $\mathbb{B}$	$IsPath([e_i])$
gf_IsPath($[e_i]$)	*gt_EdgeLists*→ $\mathbb{P}_1$	$IsPath([e_i])$
gb_IsCycle($[e_i]$)	*gt_EdgeLists*→ $\mathbb{B}$	$IsCicle([e_i])$
gf_IsCycle($[e_i]$)	*gt_EdgeLists*→ $\mathbb{P}_1$	$IsCicle([e_i])$

Tabelle 4: Zusammenfassung der Definitionen für die Kantenlisten

Definition	Funktion	Display-Form
gt_Graphs	*gt_VertexLists* × *gt_EdgeLists*	*Graphs*
gf_GetVertices(g)	*gt_Graphs*→ *gt_VertexLists*	*g.vl*
gf_GetEdges(g)	*gt_Graphs*→ *gt_EdgeLists*	*g.el*
gf_AddVertexToGraph(v, g)	*gt_Vertices* × *gt_Graphs*→ *gt_Graphs*	$AddVertexToGraph(v, g)$
gb_IsVertexInGraph(v, g)	*gt_Vertices* × *gt_Graphs*→ $\mathbb{B}$	$v \in_b g$
gf_IsVertexInGraph(v, g)	*gt_Vertices* × *gt_Graphs*→ $\mathbb{P}_1$	$v \in g$
gf_AddEdgeToGraph(e, g)	*gt_Edges* × *gt_Graphs*→ *gt_Graphs*	$AddEdgeToGraph(e, g)$
gb_IsEdgeInGraph(e, g)	*gt_Edges* × *gt_Graphs*→ $\mathbb{B}$	$e \in_b g$
gf_IsEdgeInGraph(e, g)	*gt_Edges* × *gt_Graphs*→ $\mathbb{P}_1$	$e \in g$
gc_Graph($[v_i]$, $[e_j]$)	*gt_VertexLists*× *gt_EdgeLists*→ *gt_Graphs*	$G([v_i], [e_j])$
gb_EmptyGraph(g)	*gt_Graphs*→ $\mathbb{B}$	$Empty(g)$
gf_EmptyGraph(g)	*gt_Graphs*→ $\mathbb{P}_1$	$Empty(g)$
gb_EqualGraphs(g_1, g_2)	*gt_Graphs* × *gt_Graphs*→ $\mathbb{B}$	$g_1 \equiv_b g_2$
gf_EqualGraphs(g_1, g_2)	*gt_Graphs* × *gt_Graphs*→ $\mathbb{P}_1$	$g_1 \equiv g_2$
gb_SubGraph(g_1, g_2)	*gt_Graphs* × *gt_Graphs*→ $\mathbb{B}$	$g_1 \subseteq_b g_2$
gf_SubGraph(g_1, g_2)	*gt_Graphs* × *gt_Graphs*→ $\mathbb{P}_1$	$g_1 \subseteq g_2$
gf_UnionGraphs(g_1, g_2)	*gt_Graphs* × *gt_Graphs*→ *gt_Graphs*	$g_1 \cup g_2$
gf_IntersectionGraphs(g_1, g_2)	*gt_Graphs* × *gt_Graphs*→ *gt_Graphs*	$g_1 \cap g_2$
gb_Neighbour(v_1, v_2, g)	*gt_Vertices* × *gt_Vertices* × *gt_Graphs*→ $\mathbb{B}$	$v_1 \leftrightarrow_b v_2 ing?$
gf_Neighbour(v_1, v_2, g)	*gt_Vertices* × *gt_Vertices* × *gt_Graphs*→ $\mathbb{P}_1$	$v_1 \leftrightarrow v_2 ing?$
gb_IndependentVertices(v_1, v_2, g)	*gt_Vertices* × *gt_Vertices* × *gt_Graphs*→ $\mathbb{B}$	$v_1 \neg \leftrightarrow_b v_2 ing?$
gf_IndependentVertices(v_1, v_2, g)	*gt_Vertices* × *gt_Vertices* × *gt_Graphs*→ $\mathbb{P}_1$	$v_1 \neg \leftrightarrow v_2 ing?$
gb_DependentEdges(e_1, e_2)	*gt_Edges* × *gt_Edges*→ $\mathbb{B}$	$e_1 \leftrightarrow_b e_2$
gf_DependentEdges(e_1, e_2)	*gt_Edges* × *gt_Edges*→ $\mathbb{P}_1$	$e_1 \leftrightarrow e_2$
gb_IndependentEdges(e_1, e_2)	*gt_Edges* × *gt_Edges*→ $\mathbb{B}$	$e_1 \neg \leftrightarrow_b e_2$
gf_IndependentEdges(e_1, e_2)	*gt_Edges* × *gt_Edges*→ $\mathbb{P}_1$	$e_1 \neg \leftrightarrow e_2$
gf_Successors(v, g)	*gt_Vertices* × *gt_Graphs*→ *gt_VertexLists*	$Successors(v, g)$
gf_SuccessorsOfVertices($[v_i]$, g)	*gt_VertexLists* × *gt_Graphs*→ *gt_VertexLists*	$Successors([v_i], g)$
gf_Predecessors(v, g)	*gt_Vertices* × *gt_Graphs*→ *gt_VertexLists*	$Predecessors(v, g)$
gf_PredecessorsOfVertices($[v_i]$, g)	*gt_VertexLists* × *gt_Graphs*→ *gt_VertexLists*	$Predecessors([v_i], g)$
gf_ConnectedVertices(v_1, v_2, g)	*gt_Vertices* × *gt_Vertices* × *gt_Graphs*→ $\mathbb{P}_1$	$ConnectedVertices(v_1, v_2, g)$
gb_IsHamiltonPath($[e_i]$, g)	*gt_EdgeLists* × *gt_Graphs*→ $\mathbb{B}$	$IsHamiltonPath([e_i], g)$
gf_IsHamiltonPath($[e_i]$, g)	*gt_EdgeLists* × *gt_Graphs*→ $\mathbb{P}_1$	$IsHamiltonPath([e_i], g)$
gb_IsHamiltonCycle($[e_i]$, g)	*gt_EdgeLists* × *gt_Graphs*→ $\mathbb{B}$	$IsHamiltonCycle([e_i], g)$
gf_IsHamiltonCycle($[e_i]$, g)	*gt_EdgeLists* × *gt_Graphs*→ $\mathbb{P}_1$	$IsHamiltonCycle([e_i], g)$

Tabelle 5: Zusammenfassung der Definitionen für die Graphen