Graphersetzungsregeln

(Theorie und Verwendung)

Motivation:

- intuitive Verständlichkeit
- Ausdrucksmächtigkeit
- problemnahe Modellierung
- formal definierte Semantik

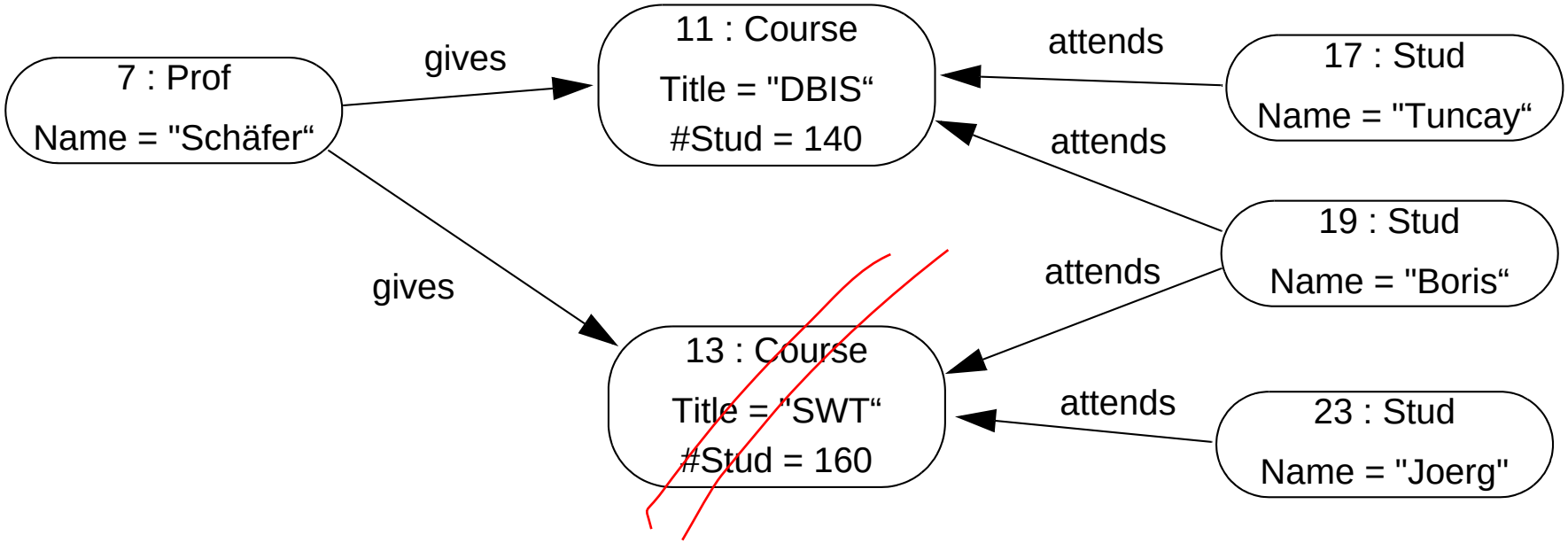
Theorie:

Def. 1: gerichtete, attributierte, knoten- und kantenmarkierte (gakk) Graphen

G := (NL, EL, A, N, E, l, av) mit

NL	endliche Menge von Knotenmarkierungen / -typen / -labeln
EL	endliche Menge von Kantenmarkierungen / -typen / -labeln
A	endliche Menge von Attribut(nam)en
N	endliche Menge von Knoten(bezeichnern)
E	$\subseteq N \times EL \times N$
l	$N \rightarrow NL$
av	$N \times A \rightarrow \{true, false\} \cup N \cup CHAR^* \cup ... \cup P(N) \cup P(CHAR^*) \cup P(...)$

Bsp. 1: gakk Graph:



NL = {Prof, Course, Stud} EL = {gives, attends} A = {Name, Title, #Stud}

N = {7, 11, 13, 17, 19, 23}

E = { (7, gives, 11), (7, gives, 13), (17, attends, 11), (19, attends, 11), (19, attends, 13),
(23, attends, 13) }

l_G = { 7 -> Prof, 11 -> Course, 13 -> Course, 17 -> Stud, 19 -> Stud, 23 -> Stud }

av = { (7, Name) -> "Schäfer", (11, Title) -> "DBIS", (11, #Stud) -> 140, (13, Title) -> "SWT",
(13, #Stud) -> 160, (17, Name) -> "Tuncay", (19, Name) -> "Boris", (23, Name) -> "Joerg" }

Def. 4: Abkürzungen

- Mit GraphClass (NL, EL, A) bezeichnen wir die Menge aller Graphen über NL, EL, A
- Im folgenden seien NL, EL, A gegeben und GC := GraphClass(NL, EL, A)
- Ein Graph $G \in GC$ wird vereinfachend nur mit (N, E, l, av) angegeben
- Sei $G=(N,E,l,av) \in GC$, dann bezeichnet
  - N_G := N die Knotenmenge von G
  - E_G := E die Kantenmenge von G
  - l_G := l die Knotenmarkierungsfunktion von G
  - av_G := die Attributierungsfunktion von G

Def. 5: Basisgraphersetzungsregeln

Eine Graphersetzungsregel ist ein Paar von Graphen (LG, RG) mit LG, RG $\in$ GC und LG und RG sind konsistent markiert,
d.h. $l_{LG}|_{N_{LG} \cap N_{RG}} = l_{RG}|_{N_{LG} \cap N_{RG}}$

Sei grr := (LG, RG) gegeben, dann bezeichnen wir mit:

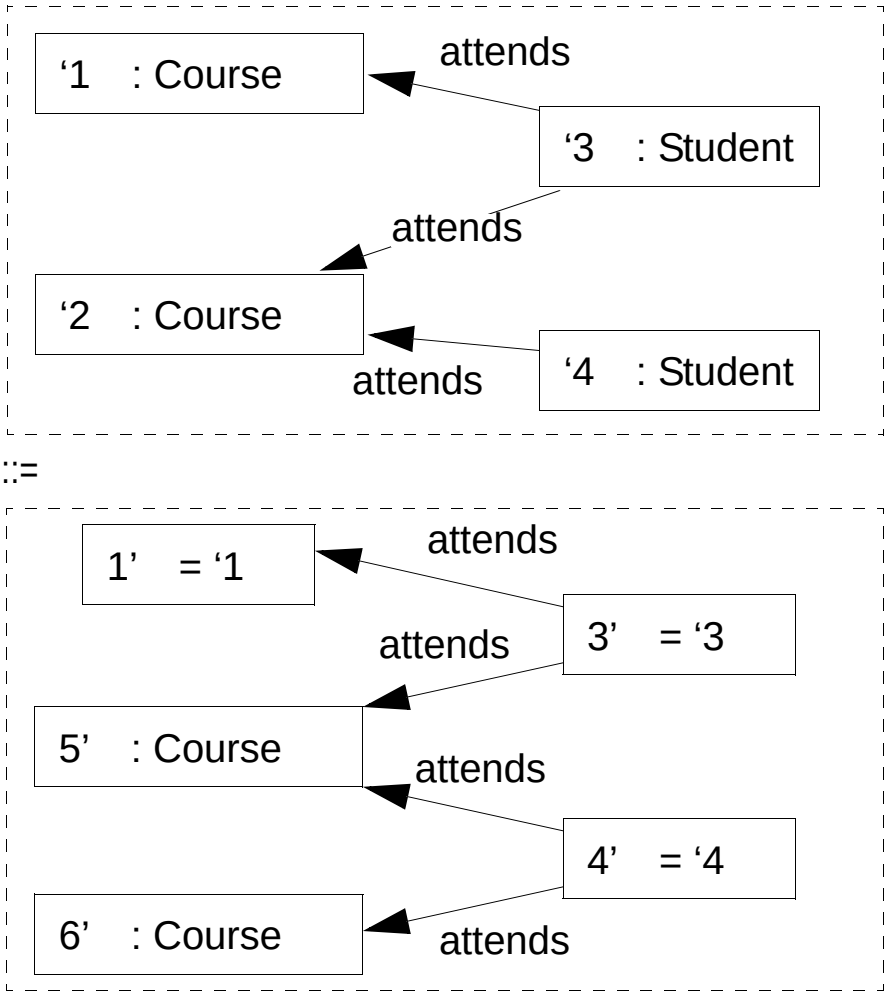
- $DelN_{grr} := N_{LG} - N_{RG}$ die Menge der von grr zu löschenden Knoten
- $DelE_{grr} := E_{LG} - E_{RG}$ die Menge der von grr zu löschenden Kanten

- $CoreN_{grr} := N_{LG} \cap N_{RG}$ die Kern- oder Kontextknoten von grr

- $AddN_{grr} := N_{RG} - N_{LG}$ die Menge der von grr neu zu erzeugenden Knoten
- $AddToAddE_{grr} := E_{RG} \cap (AddN_{grr} \times EL \times AddN_{grr})$
- $AddToCoreE_{grr} := E_{RG} \cap (AddN_{grr} \times EL \times CoreN_{grr})$
- $CoreToAddE_{grr} := E_{RG} \cap (CoreN_{grr} \times EL \times AddN_{grr})$
- $CoreToCoreE_{grr} := E_{RG} \cap (CoreN_{grr} \times EL \times CoreN_{grr})$

Bsp. 5: Basisgraphersetzungsregel:

production grr =



condition ‘1.Title = "DBIS"; ‘2.Title = "SWT";
transfer 5’.Title := "GT"; 6’.Title := "PSP";
end;

N_{LG} = { 1, 2, 3, 4 }

E_{LG} = { (3, attends, 1), (3, attends, 2),
(4, attends, 2) }

l_{LG} = { 1 -> Course, 2 -> Course, 3 -> Student,
4 -> Student }

av_{LG} = { (1, Title) -> "DBIS",
(2, Title) -> "SWT" }

N_{RG} = { 1, 3, 4, 5, 6 }

E_{RG} = { (3, attends, 1), (3, attends, 5),
(4, attends, 5), (4, attends, 6) }

l_{RG} = { 1 -> Course, 3 -> Student, 4 -> Student,
5 -> Course, 6 -> Course }

av_{RG} = { (1, Title) -> "DBIS", (5, Title) -> "GT",
(6, Title) -> "PSP" }

$DelN_{grr}$ = { 2 }
$DelE_{grr}$ = { (3, attends, 2), (4, attends, 2) }

$CoreN_{grr}$ = { 1, 3, 4 }

$AddN_{grr}$ = { 5, 6 }

$AddToAddE_{grr}$ = { }

$AddToCoreE_{grr}$ = { }

$CoreToAddE_{grr}$ = { (3, attends, 5), (4, attends, 5), (4, attends, 6) }

$CoreToCoreE_{grr}$ = { (3, attends, 1) }

Ausführung von Graphersetzungsregeln

1. Anwendungsstelle suchen
2. Löschungen
3. Neu erzeugen

1. Suchen:

Suche Teilgraph der zu linker Regelseite passt

Def. 6: Teilgraph

Seien TG, G $\in$ GC gegeben.

TG ist Teilgraph von G (in Zeichen TG <= G)

:<=>

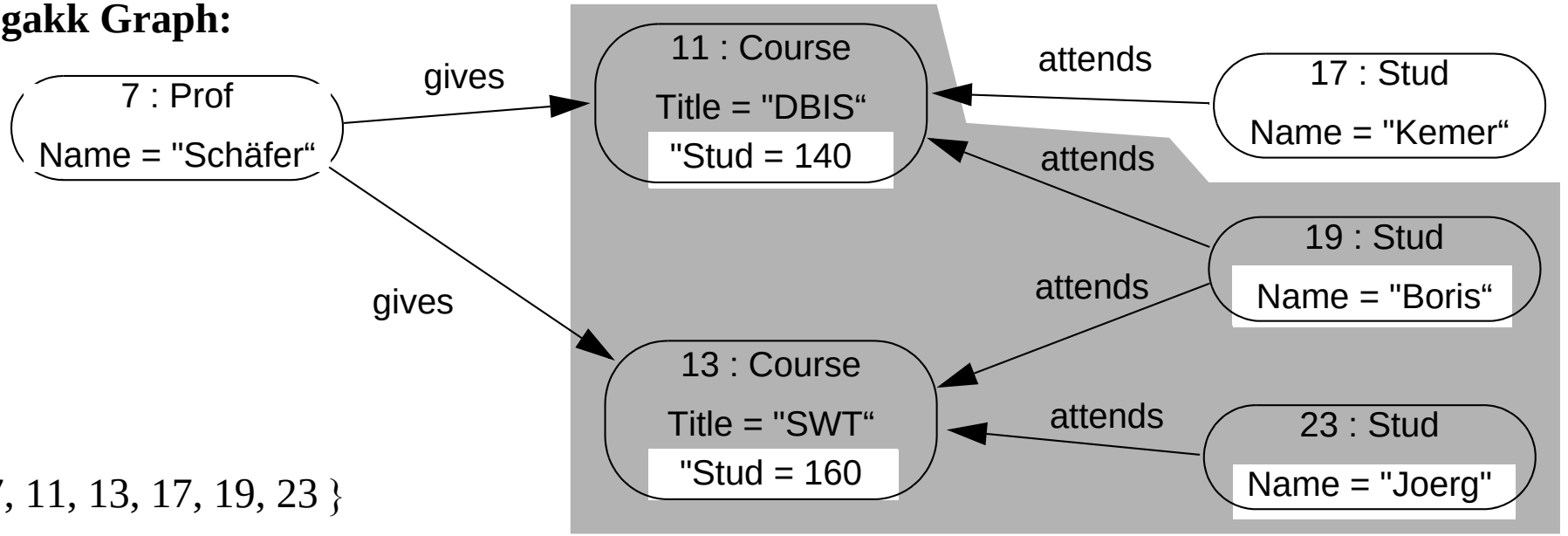
$N_{TG} \subseteq N_G$

$E_{TG} \subseteq (E_G \cap (N_{TG} \times EL \times N_{TG}))$ Teilmenge der Kanten von G, die in TG liegt

$l_{TG} = l_G|_{N_{TG}}$ Knotenmarkierungen stimmen überein

$\forall ((n, a) \rightarrow v) \in av_{TG}$ gilt: $n \in N_{TG}$, $av_G(n, a) = v$

Bsp. 1:gakk Graph:



N_G = { 7, 11, 13, 17, 19, 23 }

E_G = { (7, gives, 11), (7, gives, 13), (17, attends, 11), (19, attends, 11), (19, attends, 13), (23, attends, 13) }

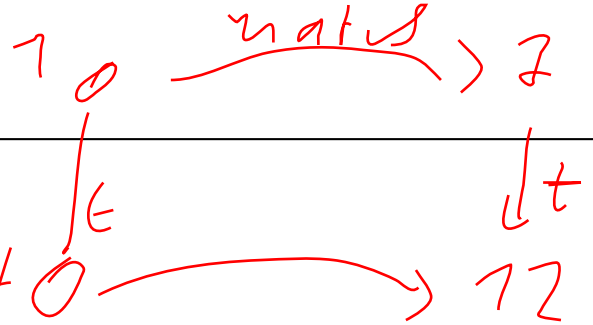
l_G = { 7 -> Prof, 11 -> Course, 13 -> Course, 17 -> Stud, 19 -> Stud, 23 -> Stud }

av_G = { (7, Name) -> "Schäfer", (11, Title) -> "DBIS", (11, #Stud) -> 140, (13, Title) -> "SWT",
(13, #Stud) -> 160, (17, Name) -> "Tuncay", (19, Name) -> "Boris", (23, Name) -> "Joerg" }

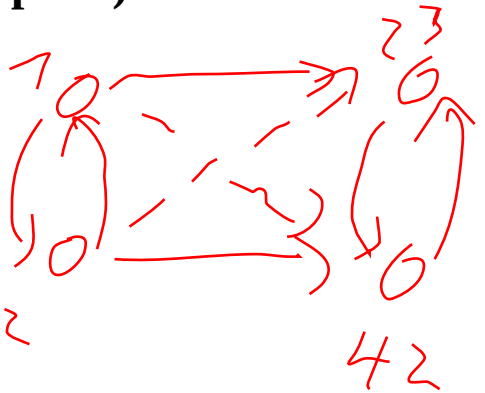
N_{TG} = { 11, 13, 19, 23 } E_{TG} = { (19, attends, 11), (19, attends, 13), (23, attends, 13) }

l_{TG} = { 11 -> Course, 13 -> Course, 19 -> Stud, 23 -> Stud }

av_{TG} = { (11, Title) -> "DBIS", (13, Title) -> "SWT" }



Def. 7 Graphisomorphismen (Abbildung zwischen Strukturgleichen Graphen):



Seien G1, G2 $\in$ GC gegeben.

Eine **bijektive** Funktion match: G1-> G2 heißt Graphisomorphismus

:<=>

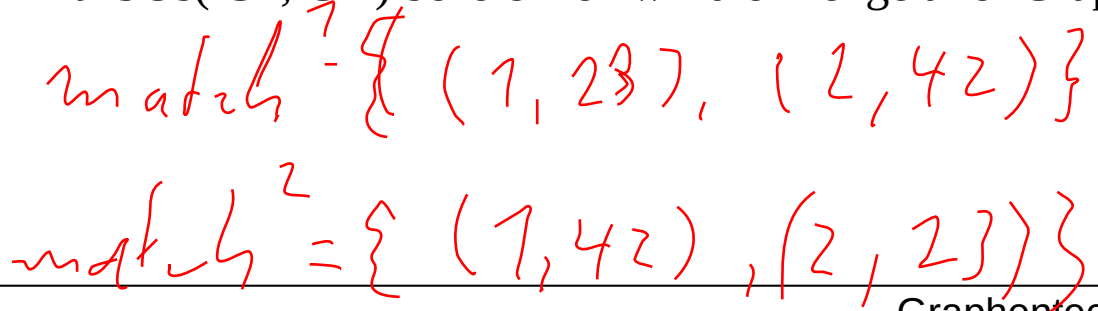
match (N_{G1}) = N_{G2} wobei match(N) := { match(n) | n $\in$ N }

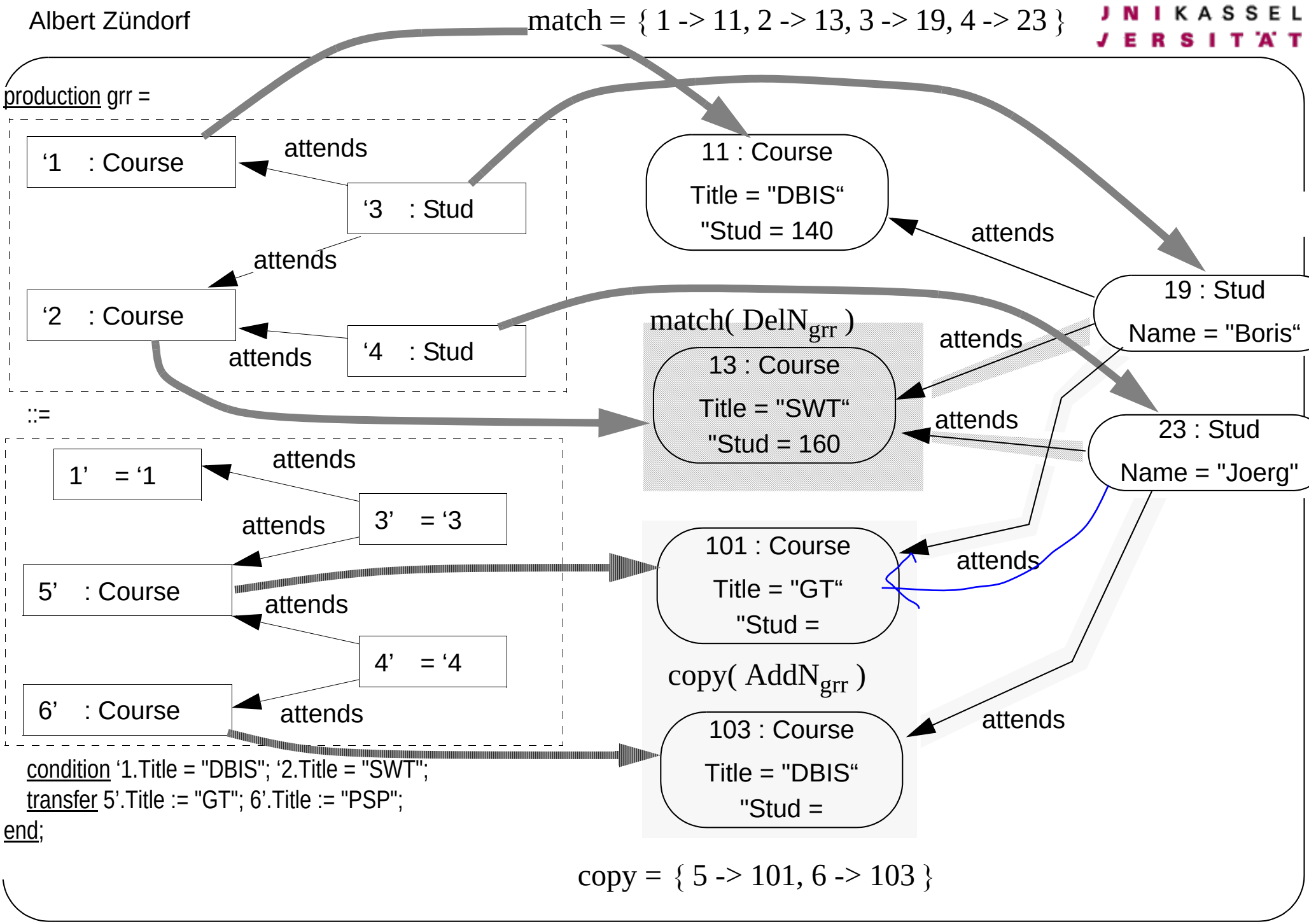
match (E_{G1}) = E_{G2} wobei match(E) := { (match(sn), el, match(tn)) | (sn, el, tn) $\in$ E }

$\forall$ n $\in$ N_{G1} gilt l_{G1}(n) = l_{G2}(match (n)) die Knotenmarkierungen bleiben erhalten

$\forall$ n $\in$ N_{G1}, a $\in$ A mit av_{G1}(n, a) = X gilt av_{G2}(match (n), a) = X

Mit ISOs(G1, G2) bezeichnen wir die Menge aller Graphisomorphismen von N_{G1} zu N_{G2}




match = { 1 -> 11, 2 -> 13, 3 -> 19, 4 -> 23 }
production grr =
‘1 : Course
attends
‘3 : Stud
attends
‘2 : Course
attends
‘4 : Stud
::=
1’ = ‘1
attends
3’ = ‘3
attends
5’ : Course
attends
4’ = ‘4
6’ : Course
attends
condition ‘1.Title = "DBIS"; ‘2.Title = "SWT";
transfer 5’.Title := "GT"; 6’.Title := "PSP";
end;
11 : Course
Title = "DBIS“
"Stud = 140
attends
19 : Stud
Name = "Boris“
match(DelN grr)
attends
13 : Course
Title = "SWT“
"Stud = 160
attends
23 : Stud
Name = "Joerg"
101 : Course
Title = "GT“
"Stud =
attends
copy(AddN grr)
103 : Course
Title = "DBIS“
"Stud =
attends
copy = { 5 -> 101, 6 -> 103 }

Def. 8 Semantik von Graphersetzungsregeln:

Sem(grr) $\subseteq$ GC $\times$ GC oder auch Sem[grr] : GC -> P(GC)

mit (G1, G2) $\in$ Sem(grr) oder auch G2 $\in$ Sem[grr] (G1)

:<=>

1. (Suchen)

   $\exists$ TG $\in$ GC mit TG <= G1 und $\exists$ match $\in$ ISOs(LR, TG) (TG heißt Anwendungsstelle)

2. (Löschen)

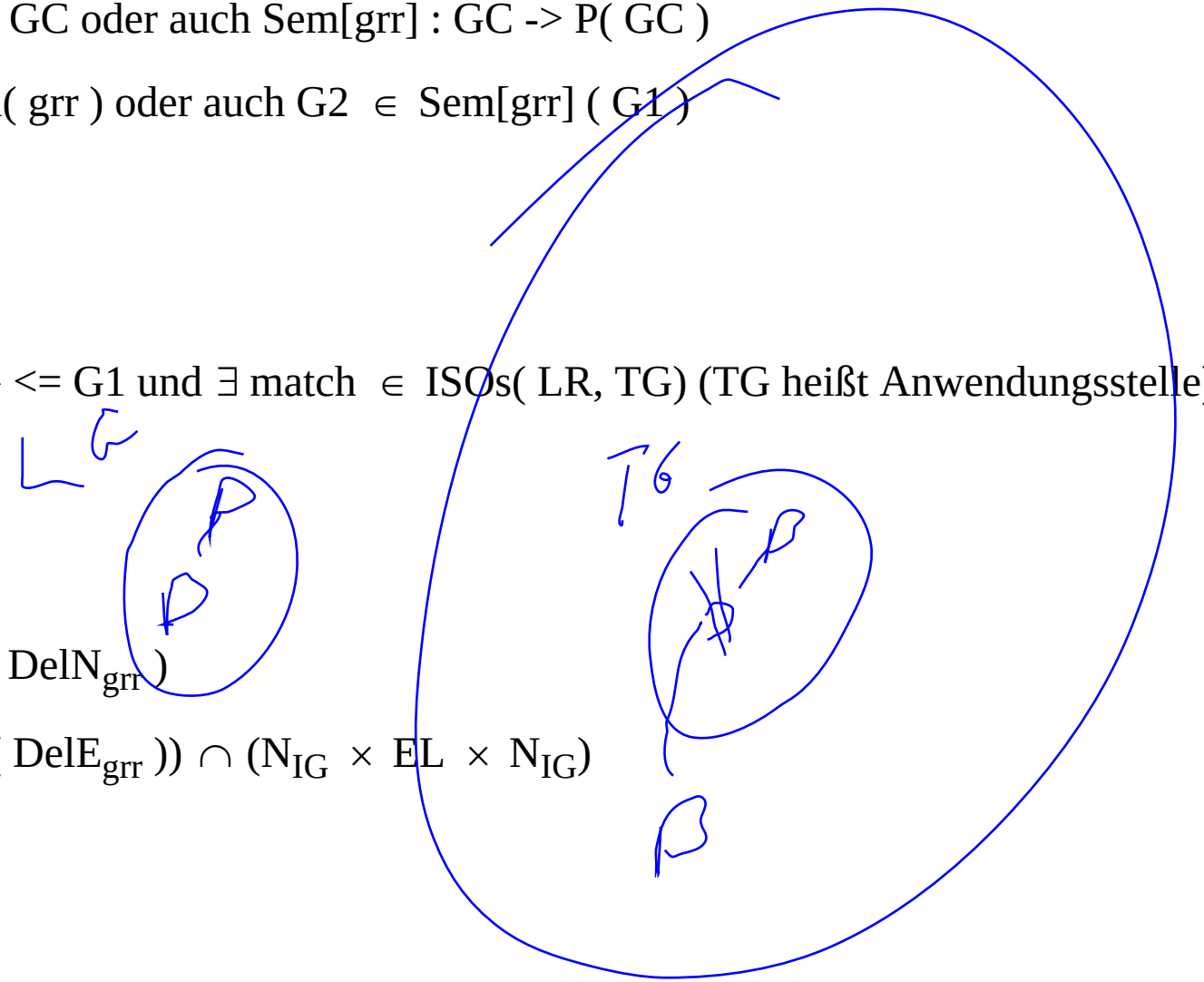
   $\exists$ IG <= G1 mit

   $N_{IG} = N_{G1}$ - match ($DelN_{grr}$)

   $E_{IG} = (E_{G1}$ - match ($DelE_{grr}$)) $\cap$ ($N_{IG} \times EL \times N_{IG}$)

   $l_{IG} = l_{G1}|_{N_{IG}}$

   $av_{IG} = av_{G1}|_{N_{IG} \times A}$



3. (Erzeugen)

(neue Knoten)

$\exists\, N_{new}$ mit $N_{new} \cap N_{G1} = \varnothing$ und $\exists$ bijektive Funktion copy mit copy ($AddN_{grr}$) = N_{new}

wobei copy durch den match eindeutig festgelegt seien soll, d.h.

match = match' => copy = copy'

(sonst Menge von Ergebnisgraphen nur mit unterschiedlichen Nummern für N_{new})

und

$N_{G2} = N_{IG} \cup N_{new}$

$E_{G2} = E_{IG} \cup copy\,(\, AddToAddE_{grr}\,)$

$\cup\ \{\ (copy(\, sn\,), el, match(\, tn\,)) \mid (sn, el, tn) \in AddToCoreE_{grr}\ \}$

$\cup\ \{\ (match(\, sn\,), el, copy(\, tn\,)) \mid (sn, el, tn) \in CoreToAddE_{grr}\ \}$

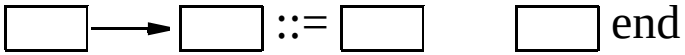
$\cup\ match\,(\, CoreToCoreE_{grr}\,)$

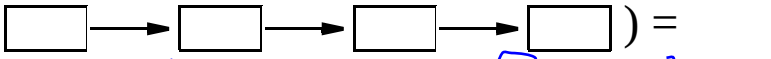
$l_{G2}(\, n\,) = l_{GR}(\, nr\,)$ falls n = copy(nr) und = $l_{IG}(\, n\,)$ sonst

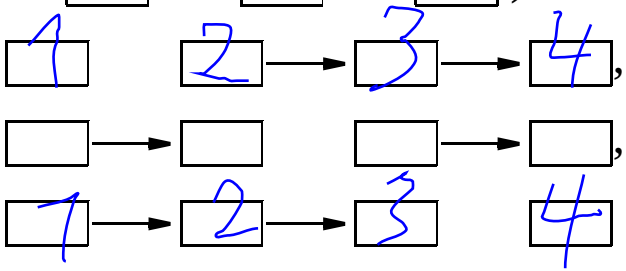
$av_{G2}(\, n, a\,) = av_{GR}(\, nr, a\,)$ falls n = copy(nr) oder n = match(nr) und = $av_{IG}(\, n, a\,)$ sonst

Warum Sem(grr) $\subseteq$ GC $\times$ GC ?

Bei Sem[grr] : GC -> P(GC):

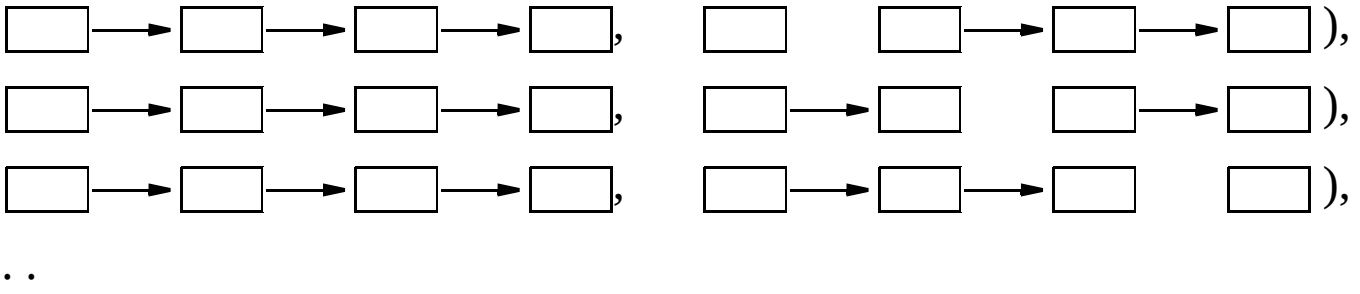
production grr =  end

Sem [grr] () =

{ 

Sem (grr & grr) = ??

Sem(grr) $\subseteq$ GC $\times$ GC:

Sem(grr) = { 

. . .

}

Semantik unabhängig von Startgraph, einfachere Beschreibung von Kontrollstrukturen

z.B.: Sem (grr_1 or grr_2) := Sem (grr_1) $\cup$ Sem (grr_2)

Einfache Kontrollstrukturen für Graphersetzungsregeln:

Def. 9 Zusammensetzen von Graphersetzungsregeln zu Graphtransformationen

1) Jede Graphersetzungsregel ist eine Graphtransformation

2) Seien gt1 und gt2 Graphtransformationen, dann ist auch

- gt1 & gt2 eine Graphtransformation mit
  Sem(gt1 & gt2) := { (G1, G3) | ∃ G2 mit (G1, G2) ∈ Sem(gt1) und (G2, G3) ∈ Sem(gt2) }

- gt1 or gt2 eine Graphtransformation mit
  Sem(gt1 or gt2) := Sem(gt1) ∪ Sem(gt2)

- def(gt1) eine Graphtransformation mit
  Sem[def(gt1)] := { (G, G) | ∃ G2 mit (G, G2) ∈ Sem(gt1) }

- not def(gt1) eine Graphtransformation mit
  Sem [not def (gt1)] := { (G, G) | ¬∃ G2 mit (G, G2) ∈ Sem(gt1) }

Alle anderen Kontrollstrukturen von Progres aus obigen Grundverknüpfungen ableitbar:

- Sem(choose gt1 else gt2 end) := Sem[gt1 or (not def (gt1) & gt2)]
  sequentielles oder

- Sem(skip) := Sem[def(gt1) or not def (gt1)]
  noop von Progres

- Sem(loop gt1 end) := Sem(choose gt1 & loop gt1 end
  else skip end)
  rekursive Definition von Iteration

- Sem(fail) := Sem[not def (skip)] = {}
  halt von Progres

Semantik von Progres, komplexere Sprachkonstrukte:

(Im folgenden Semantikdefinition mit obigen Basiskonstrukten)

Prozedur- und Sichtbarkeitskonzepte der Sprache Progres:

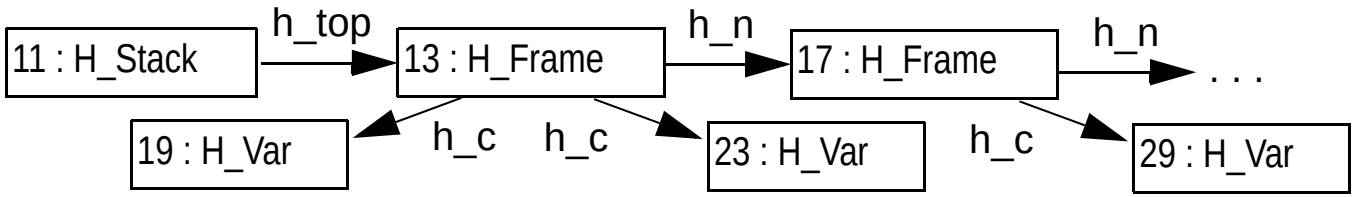
- Benennung von Graphtransformationen durch Transaktionen:
  transaction InstallSWT = CreateProf & CreateCourse & SetProfForCourse end;
- Progres kennt lokale Variablen, In- und Out-Parameter für Operationen
- Progres erlaubt Rekursion

  => wir benötigen Prozedurkeller

- Keine geschachtelten Transaktionsdeklarationen (oder anderer Prozeduren)
- keine globalen Variablen (außer implizitem Arbeits-/Wirts-Graphen)

  => Static Link unnötig

Beispiel:

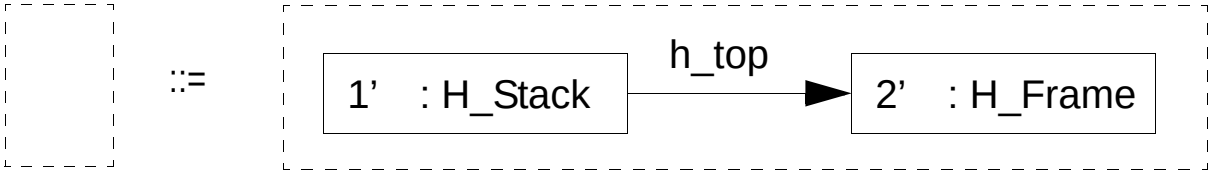


Ausführung einer Progres-Spezifikation (Programmstart):

Sei Spec eine bezügl. statischer Semantik fehlerfreie Progresspezifikation, dann gilt:

- Sem(Spec) := Sem (H_CreateStack & Main)
  mit

production H_CreateStack =



end;

und

- Main ist eine benutzerdefinierte Operation in Spec
- OBDA. kommen H_CreateStack, H_Stack, h_top, H_Frame in Spec nicht vor

(Prozedurkeller wird gleich für Parameter und lokale Variablen benötigt)

Transaktionen (Prozedurkeller**):**

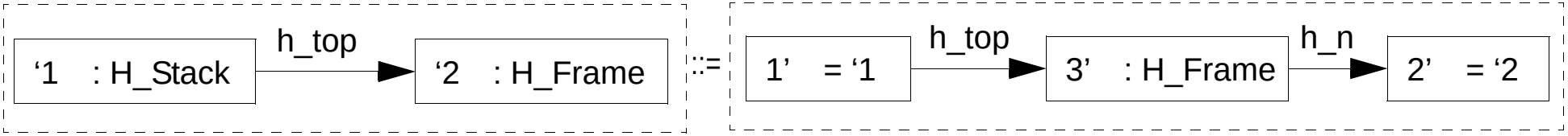
Sei gt eine Graphtransformation und Spec enthalte Deklaration der Form:

transaction T = gt end;

dann gilt:

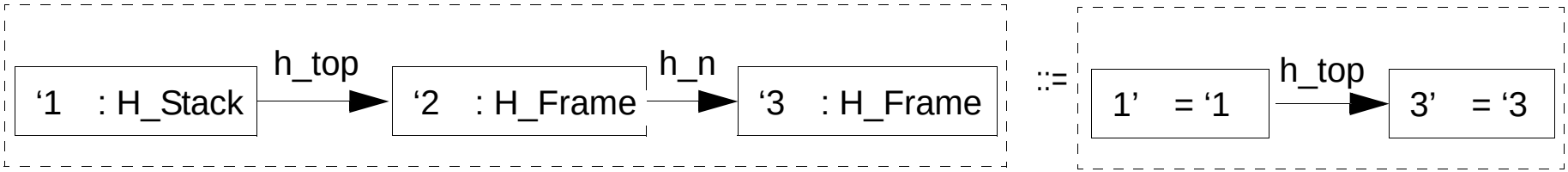
- Sem(T) := Sem(H_PushFrame & gt & H_PopFrame)
  mit:

production H_PushFrame =



end;

production H_PopFrame =



end;

Bemerkung:

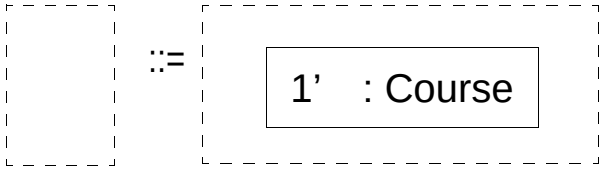
- OBDA gilt im folgenden, daß die zur Simulation eingeführten Knoten- und Kantentypen, Attriubte und Operationsnamen in Spec nicht vorkommen (evtl. geeignet umbenennen)

Operationen mit (In-) Parametern:

(Parameterlose Produktionen benötigen keinen eigenen Frame, Aufruf wie Makro)

Spec enthalte folgende Produktion:

```
production CreateCourse( title : string) =
```



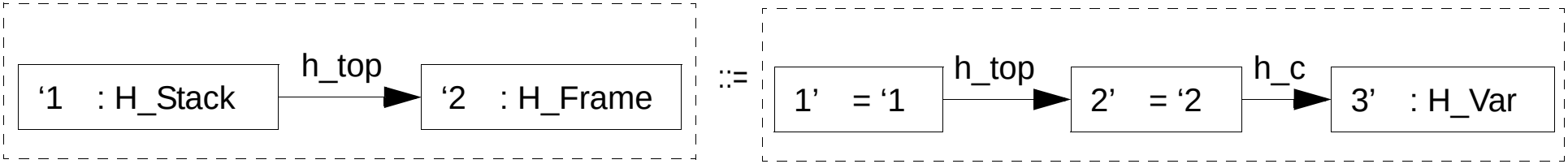

```
    transfer 1’.Title := title; 1’.NoOfStuds := 0;
end;
```

dann gilt:

Sem[CreateCourse("GT")] :=
Sem[H_PushFrame & H_PushStringVar_title_GT & H_CreateCourse_title & H_PopFrame]

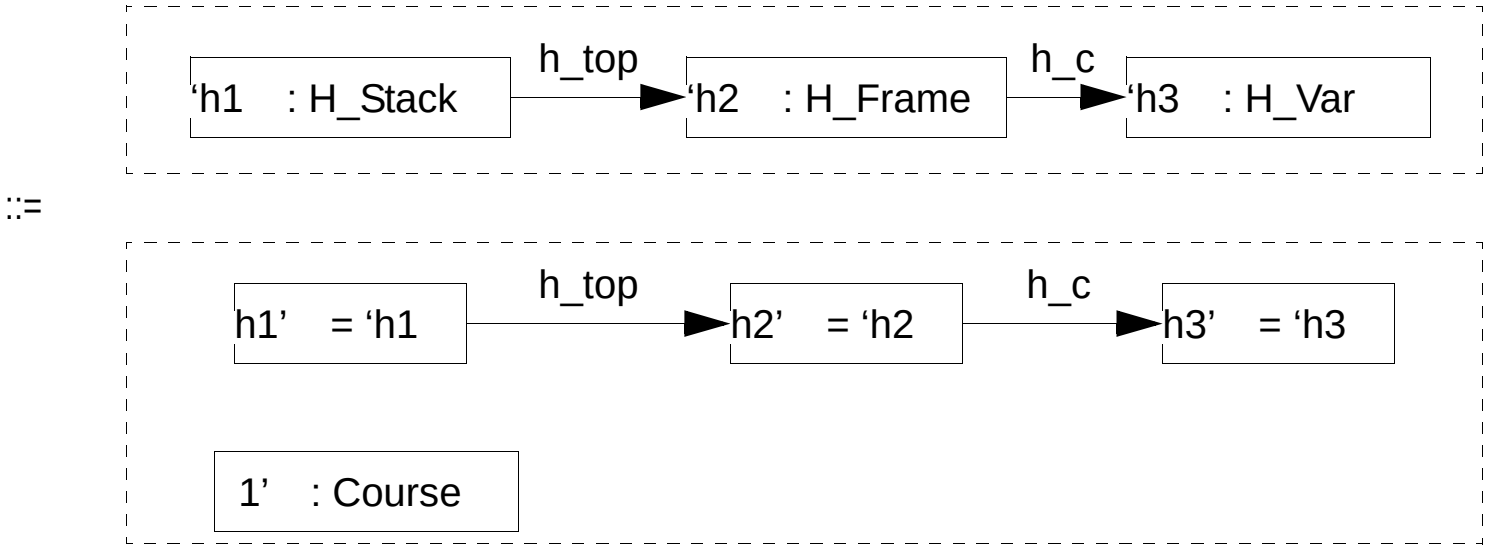
mit:

production H_PushStringVar_title_GT =



transfer 3’.H_VarName := "title"; 3’.H_StringAttr := "GT";
end;

production H_CreateCourse_title =



condition ‘h3.H_VarName = "title"
transfer 1’.Title := ‘h3.H_StringAttr; 1’.NoOfStuds := 0;
end;

Allgemeine Behandlung von Parametern:

- Übergabe von Parametern simuliert durch "Parameter-" Knoten auf Prozedurkeller
- Zugriff auf Parameter simuliert duch Attributzugriff auf Parameterknoten.

Bemerkung:

- Obige Organisation des Prozedurkellers wie allgemein im Compilerbau üblich
- Hier vereinfachend: kein Static-Link notwendig
- Obige Simulation benötigt pro Aufrufstelle spezifische PushTVar_x_val-Operation
- Eine Progres Operation kann natürlich eine beliebige Zahl von In-Parametern unterschiedlicher Typen haben, Simulation analog (selbst :)

ACHTUNG !!!:

Attributnamen und Parameternamen haben gemeinsamen Namensraum, d.h. Parameternamen überdecken gleichnamige Attibute: (Gibt ne komische Fehlermeldung)

~~transfer 1'.Title := Title;~~

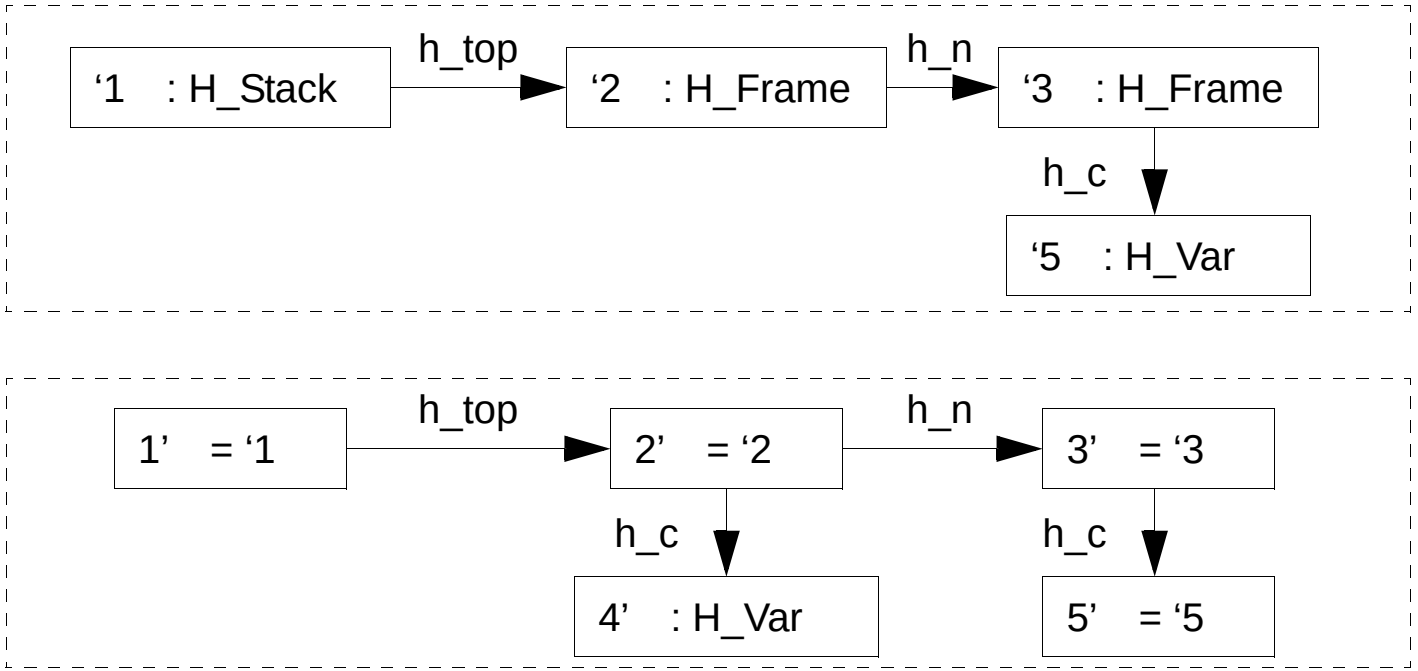
lokale Variablen:

- Sem(use x : T := val do . . . Op(x) . . . end) :=

Sem(H_PushTVar_x_val & . . . (* <-- Behandlung der Variablendeklaration *)
& H_PushFrame & H_PushTVarFromVar("param", "x") (* Variablenzugriff *)
& H_Op_param & H_PopFrame)

mit:

production H_PushTVarFromVar(dest_var, src_var : string) =



::=

condition ‘5.H_VarName = src_var;
transfer 4’.H_VarName := dest_var; 4’.H_TAttr := ‘5.H_TAttr;
end;

Allgemeine Behandlung lokaler Variablen:

- Für jede lokale Variable wird ein Variablen-Knoten im aktuellen Frame angelegt
- Bei Verwendung der Variablen wird auf den zugehörigen Variablen-Knoten zugegriffen

Bemerkung

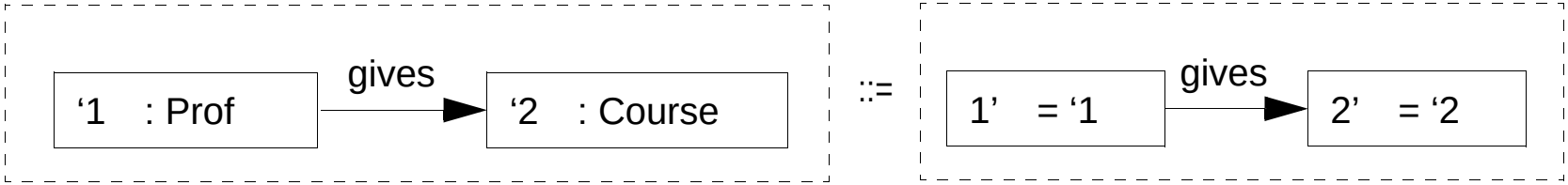
- In einem use-Statemente sind mehrere Variablen zulässig
- der initialisierende Ausdruck ist optional
- Wenn ein initialisierender Ausdruck angegeben ist kann die Typangabe entfallen
- use-Statements können an jeder Stelle in einer Transaktion stehen
- use-Statements können beliebig geschachtelt werden:

```
use     x, y := 0; (* zwei Integer-Variablen x und y *)
        z : string;
do
    . . .
    use i, j : integer := 1 do . . . end
    . . .
end
```

(*eventuelle Namensüberdeckungen werden bei Simulation durch interne Umbenennung behandelt *)

Out-Parameter:

production GetProfOfCourse(title : string ; out name : string) =



condition ‘2.Title = title;
return name := 1’.Name;
end;

Allgemeine Behandlung von Out-Parametern:

- Wie In-Parameter: zusätzlicher Parameterknoten vor Aufruf erzeugen
- Zugriffe werden Attributzugriffe auf Parameterknoten
- Plus: nach Ausführung Ergebnis in Aktualparameter zurückkopiern (Call-By-Value-Return)

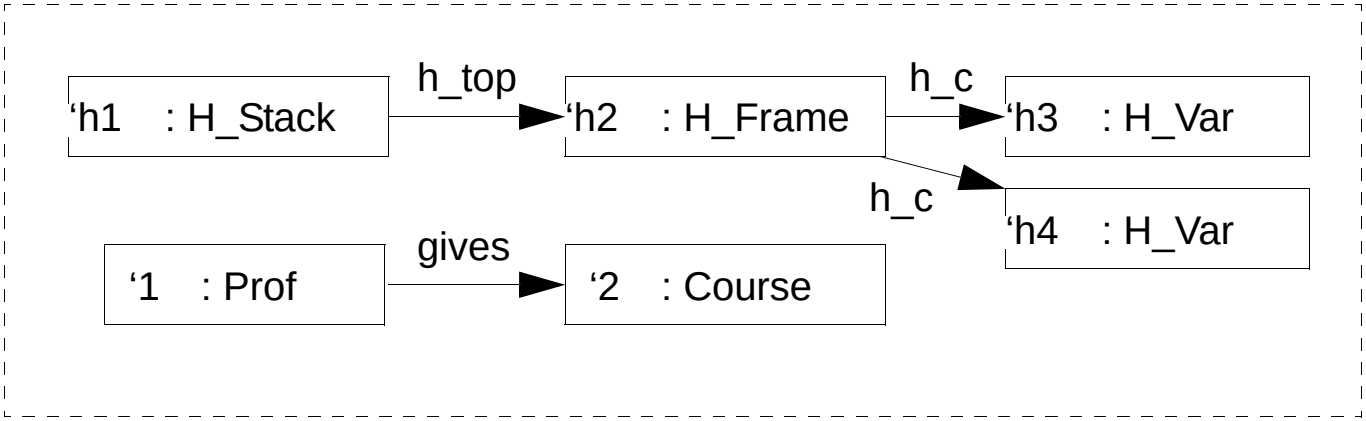
- Sem[use n : string := "" do . . . GetProfOfCourse("SWT", out n) . . . end]

  := Sem[use n: string := "" do

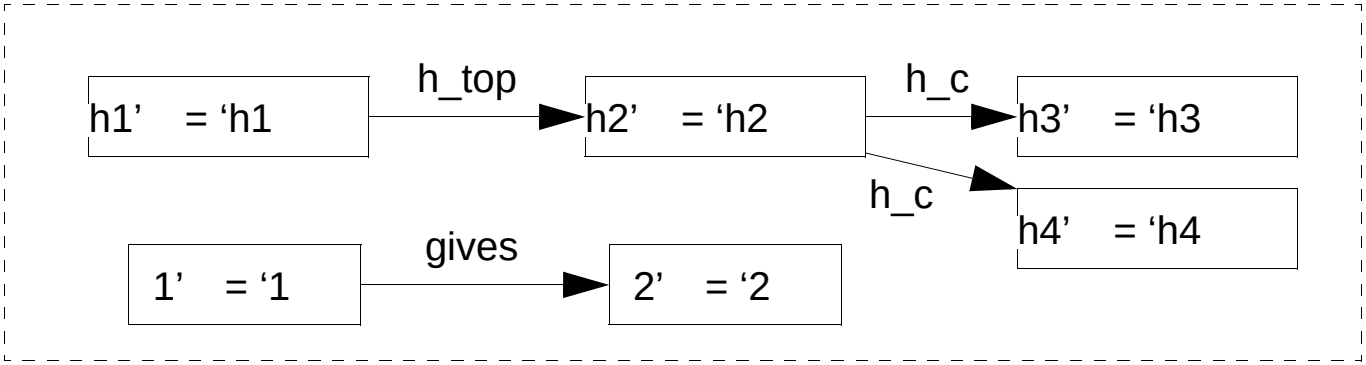
```
        . . .
        H_PushFrame
        & H_PushStringVar_title_SWT (* Behandlung des In-Parameters *)
        & H_PushStringVarFromVar( "name", "n" ) (* Behandlung des Out-Parameters *)
        & H_GetProfOfCourse_title_name
        & H_ReturnStringOutParam( "name", "n" ) (* Uebergeben des Rückgabewertes *)
        & H_PopFrame
        . . .
    end ]
```

  mit:

production H_GetProfOfCourse_title_name =



::=

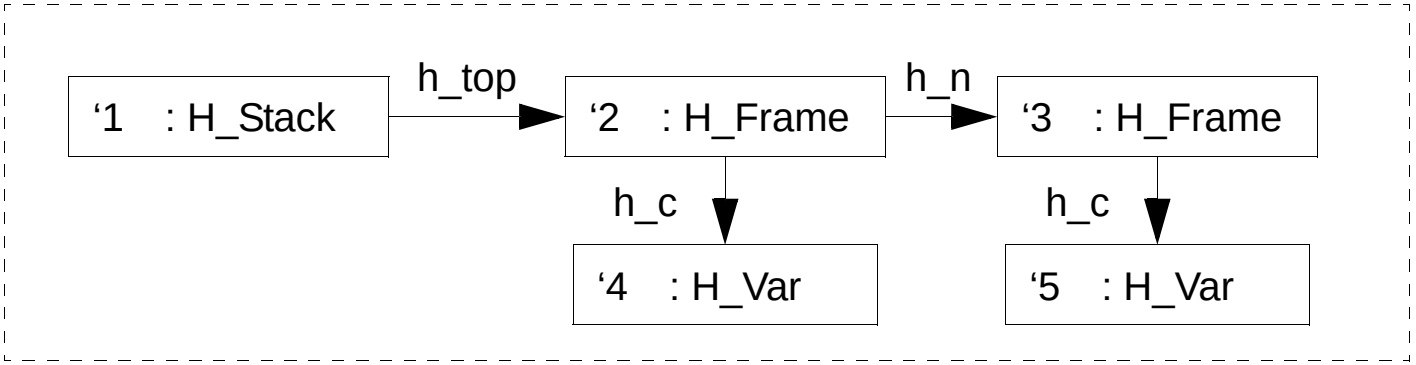


condition ‘2.Title = ‘h3.H_StringAttr;
‘h3.H_VarName = "title";
‘h4.H_VarName = "name";
transfer h4’.H_StringAttr := ‘1.Name;

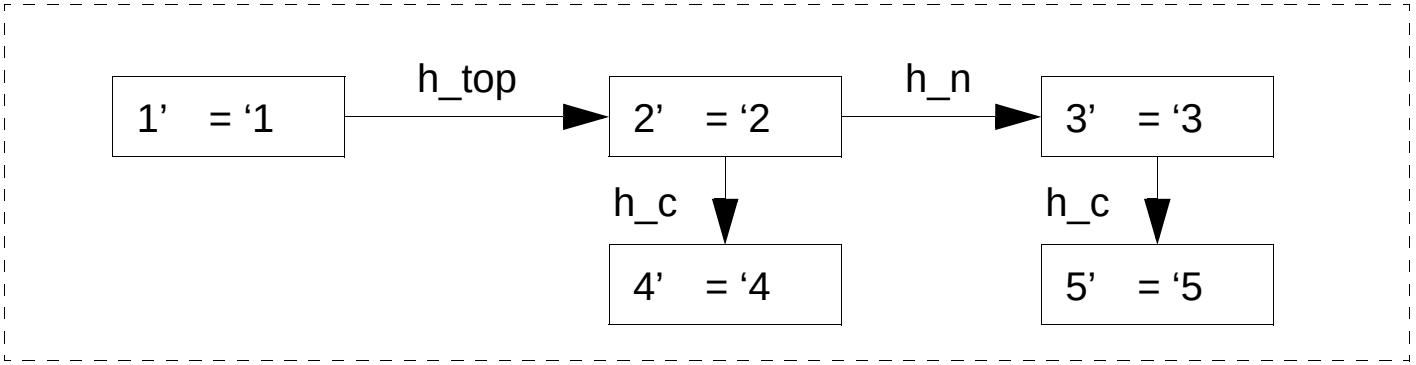
end;

production H_ReturnStringOutParam(src_var, dest_var : string)

=



::=



condition ‘4.H_VarName = src_var;
‘5.H_VarName = dest_var;
transfer 5’.H_StringAttr := ‘4.H_StringAttr;

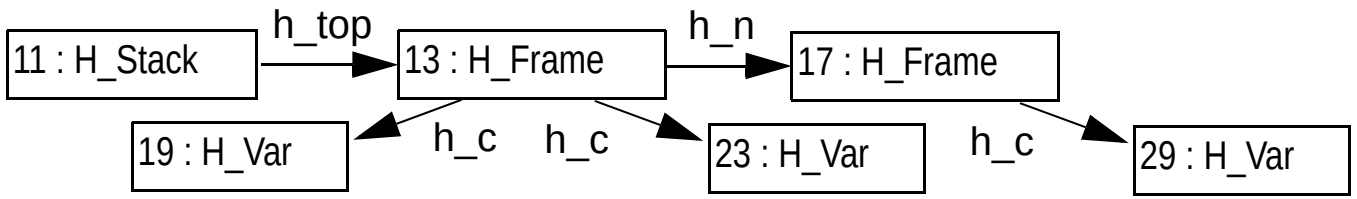
end;

Bemerkungen:

- Hier Call-By-Value-Return (erlaubt uniforme Behandlung von In- und Out-Parametern
- Progres erlaubt beliebig viele out-Parameter (unterschiedlicher Typen)
-

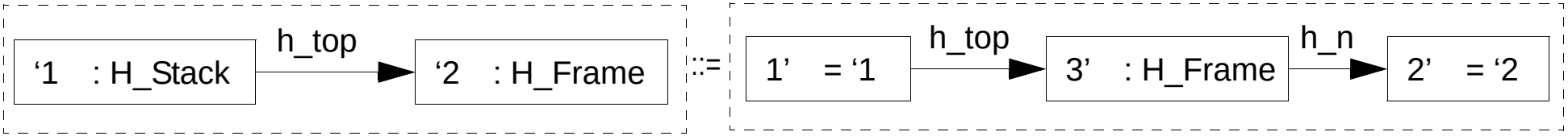
Übungsaufgabe 1:

Graph:



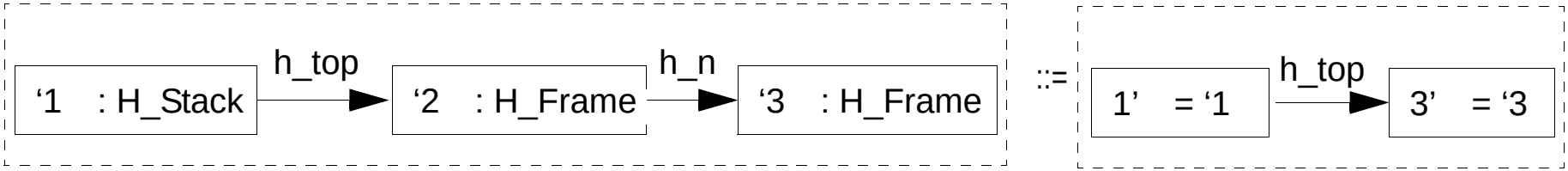
Regeln:

production H_PushFrame =



end;

production H_PopFrame =



end;

Beschreibt formal obigen GAKK-Graphen und obige Regeln, die Anwendung der Regeln und den jeweiligen Intermediate- und Ergebnisgraphen.

Übungsaufgabe 2:

Wir wollen einen Editor und einen Interpreter für (die logischen Datenstrukturen von) Progres mit Fujaba bauen. Überlegt euch Story-Boards für das editieren einer Graphersetzungsregel, d.h. für das

- Anlegen von Knoten- und Kantentypen
- Anlegen von Attributdeklarationen
- Einfügen von Knoten und Kanten in die linke und rechte Regelseite
- Erstellen von Attributbedingungen und -Zuweisungen

Übungsaufgabe 3:

Erstellt Story-Boards für die Ausführung einer Graphersetzungsregel:

- wie soll der Arbeitsgraph modelliert werden
- wie soll die Anwendungsstelle gefunden werden (das machen wir zusammen)
- wie funktioniert die Ausführung

Übungsaufgabe 4:

Erstellt für die Story Boards aus Aufgabe 2 und Aufgabe 3 ein gemeinsames Klassendiagramm mit Hilfe der Fujaba-Umgebung und generiert und compiliert dafür Java Klassen. Programmiert die Regelausführung aus Aufgabe 3 als Story Diagramme und testet sie mit Fujaba