TABLEAUX-BEWEISER für polykontexturale Logik mit Transjunktionen

Steven Bashford, Johannes 5 Joemann, Rudolf Kaehr

Dortmund, 24. März 1993

Forschungsprojekt "Theorie komplexer biologischer Systeme"

Gefördert von der Volkswagn-Stiftung

Inhaltsverzeichnis

1 Entwicklung eines Tableaubeweismechanismus für aussagenlogische Formeln der PKL 1

1.1 Tableau-Beweisbäume . . . 1
1.2 Allgemeingültigkeit und Widersprüche . . . 4
1.3 Repräsentation eines Beweisbaumes . . . 5
1.4 Die Tableauregeln . . . 8
1.4.1 Übergang zur PKL . . . 9
1.4.2 Repräsentation der T-Anteile . . . 11
1.4.3 T-Anteil freie Terme . . . 18
1.4.4 Geschachtelte /// - Terme . . . 19
1.4.5 Zusammenfassung der Regeln . . . 21
1.5 Der Widerspruchbegriff . . . 23
1.6 Die Darstellung der DNF . . . 25
1.6.1 Modifikation der Regeln . . . 25
1.6.2 Die konj. verknüpfung von Dnf's | −− | . . . 25
1.6.3 Die disj. Verknüpfung |++| . . . 27
1.6.4 Die beiden Operatoren &&&' und |||' . . . 28

2 Entwicklung eines Tableaubeweismechanismus für aussagenlogische Formeln der PKL 28

2.1 Tableau-Beweisbäume . . . 29
2.2 Allgemeingültigkeit und Widersprüche . . . 32
2.3 Repräsentation eines Beweisbaumes . . . 33
2.4 Die Tableauregeln . . . 36
2.4.1 Übergang zur PKL . . . 37
2.4.2 Repräsentation der T-Anteile . . . 39
2.4.3 T-Anteil freie Terme . . . 46
2.4.4 Geschachtelte /// - Terme . . . 47
2.4.5 Zusammenfassung der Regeln . . . 49
2.5 Der Widerspruchbegriff . . . 51
2.6 Die Darstellung der DNF . . . 53
2.6.1 Modifikation der Regeln . . . 53
2.6.2 Die konj. verknüpfung von Dnf's | −− | . . . 53
2.6.3 Die disj. Verknüpfung |++| . . . 55
2.6.4 Die beiden Operatoren &&&' und |||' . . . 56

3 Die Möglichkeiten des PKLBEWEISERS Version 1.0 57

3.1 Definition von Tableauregeln und Definition von neuen Regeln basierend auf schon existierenden . . . 57
3.2 Beweisen von Aussagenlogischen Formeln der Pkl . . . 59
3.3 Baum-Darstellung sowie Term-Darstellung von Beweisbäumen . . . 60
3.4 Durchwandern der Beweisbäume mit der Möglichkeit zum partiellen Ausführen von Beweisschritten . . . 61
3.5 Lokale Variablendarstellung . . . 61
3.6 BEMERKUNGEN . . . 6

4 Die Möglichkeiten des PKLBEWEISERS Version 1.0 62

4.1 Definition von Tableauregeln und Definition von neuen Regeln basierend auf schon existierenden . . . 63
4.2 Beweisen von Aussagenlogischen Formeln der Pkl . . . 65
4.3 Baum-Darstellung sowie Term-Darstellung von Beweisbäumen . . . 66

4.4 Durchwandern der Beweisbäume mit der Möglichkeit zum partiellen Ausführen von Beweisschritten . . . 67
4.5 Lokale Variablendarstellung . . . 67
4.6 BEMERKUNGEN . . . 67

5 KONZEPTE UND STRUKTUR DER IMPLEMENTIERUNG 69
5.1 Das Konzept . . . 69
5.1.1 Operationale Kern . . . 69
5.1.2 Benutzerschnittstelle . . . 69
5.2 Hilfsmittel (Benutzte Tools) . . . 69
5.3 Modularer Aufbau und Abhängigkeit der Sourcen . . . 70
5.4 Ein kleiner Wegweiser durch die Folderstruktur . . . 71

6 KONZEPTE UND STRUKTUR DER IMPLEMENTIERUNG 71
6.1 Das Konzept . . . 71
6.1.1 Operationale Kern . . . 72
6.1.2 Benutzerschnittstelle . . . 72
6.2 Hilfsmittel (Benutzte Tools) . . . 72
6.3 Modularer Aufbau und Abhängigkeit der Sourcen . . . 72
6.4 Ein kleiner Wegweiser durch die Folderstruktur . . . 74

7 DOKUMENTATION DER WICHTIGSTEN SCHNITTSTELLEN DER MODULE 75
7.1 Das Modul Der AUSSAGENLOGISCHEN Formeln . . . 75
7.1.1 Spezifikation der elementaren Datentypen . . . 75
7.1.2 Der Aufbau der Formeln . . . 75
7.1.3 Operationen auf den atomaren Daten von Formeln . . . 75
7.1.4 Weitere Datengeneratoren . . . 76
7.1.5 Ordnung auf den Wahrheitswerten und Operatoren . . . 76
7.2 Grundlegende Module zur Spezifikation des Beweisers . . . 77
7.2.1 DIE TABLEAUS . . . 77
7.2.2 TABLEAU-BEWEISER . . . 79
7.2.3 Zusammenfügen von Dnf's . . . 79
7.3 Modul TABDEF zur Generierung von Tableauregeln und Formel . . . 80
7.3.1 Formeln . . . 81
7.3.2 Tableaus . . . 81
7.4 Modul zum Durchwandern der Bäume . . . 82
7.5 Modul zur Umwandlung von Bäumen und Formeln in Ascii Repräsentation . . . 84
7.5.1 Die Operationen . . . 84

8 Die Signaturen 86

Zusammenfassung

Der folgende Bericht umfasst meine Arbeit im Rahmen der Grundlagenforschung zur polykontexturalen Aussagenlogik. Ein grundlegendes Verständnis dieser Materie ist eine unbedingte Voraussetzung für das Verstehen des folgenden Textes.
Meine Arbeit umfasst den Entwurf eines Tableaubeweismechanismus sowie dessen Implementierung. Bei dem Entwurf geht es nicht darum einen schon vollstaendigen und korrekten Beweismechanismus zu finden. Ziel ist es eine möglichst gute Annäherung zu finden, die ein besseres Verständnis von Zusammenhängen liefern soll. Weiterhin soll ein mathematisches Grundgerüst aufgebaut werden, das den Zugang zur Materie anschaulicher gestalten soll.

Die Implementierung hat die folgenden Ziele:

- Durch die mannigfaltigen manuellen Vorarbeiten von Rolf Kaehr stehen schon viele PKL-Tautologien zum Testen des BWM zur Verfügung. Durch das Arbeiten mit dem Beweissystem kann festgestellt werden an welchen stellen der Mechanismus hinreichend ist, und an welchen nicht.
- Viele Zusammenhänge sind beim Entwurf des BWM noch nicht vollständig klar, sodaß viele Annahmen gemacht werden müssen. Das Beweissystem stellt ein Tool dar, mit dem diese Zusammenhaenge erforscht werden sollen, und das die Entwicklung neuer bzw modifizierter Regeln ermöglicht.
- Die Art der Darstellung von Beweisen soll ebenfalls den Zugang zur Materie erleichtern.

Der Bericht gliedert sich in folgende Teile.

1. Der erste Teil beschreibt den Entwurf des Beweismechanismus. Beweise werden mit Hilfe von Tableaubeweisbäumen gefuehrt. Unter diesem Kontext wird zunächst eine geeignete Darstellung von Bäumen und Regeln die den Beweisablauf steuern entwickelt.
2. Im zweiten Teil gehe ich näher auf die Eigenschaften und Moeglichkeiten des Beweistools ein. Dieses Kapitel darf nicht als Benutzerhandbuch mißverstanden werden.
3. Der letzte Teil beinhaltet die Implementierung des Beweistools. Hierzu gehören die Beschreibung des Konzeptes der Implementierung und eine abstrakte Dokumentation der Sourcen, die die Einarbeitung in das Programmpaket erleichtern soll.

1 Entwicklung eines Tableaubeweismechanismus für aussagenlogische Formeln der PKL

Im folgenden beschreibe ich die Entwicklung eines Beweismechanismus für aussagenlogische Formeln der PKL.
Dies umfasst den Entwurf

- einer Darstellung von Tableaubeweisbäumen ,
- einer Menge von Reduktionsregeln ,
- einer Menge von Expansionsregeln.

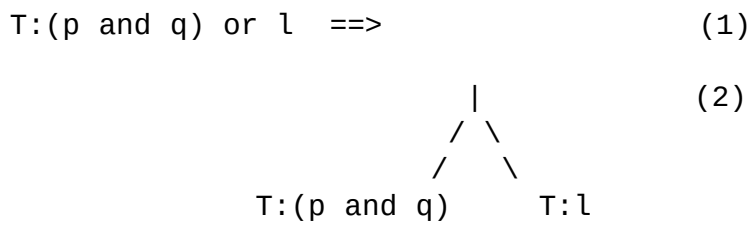
Die Reduktionsregeln überführen einen Beweisbaum in eine Normalform, die logische Interpretationen über die ursprüngliche Formel erlauben (z.B , daß es sich bei der Formel um eine Tautologie oder eine Kontradiktion handelt). Bei den Expansionsregeln geht es um die Darstellung von Tableauregeln. Hier müssen Formeln des Beweisbaumes in die entsprechende Baumdarstellung transformiert werden.
Dieses Kapitel gibt zunächst eine kleine Einführung in Tableaus und Tableaubeweisbäume im klassischen Sinne. Danach wird schrittweise eine Regelsatz für Reduktionen und Expansionen entwickelt. Diese werden danach unter dem Kontext der PKL erweitert.
Der hier beschriebene Regelsatz ist ein erster Ansatz und ist noch nicht vollständig und korrekt . Er dient aber dazu ein erstes Verständnis für einen Reduktionsmechnismus zu gewinnen. Diese Erfahrungen bilden die Grundlagen für weitere Modifikationen und Verfeinerungen des Regelsatzes.

1.1 Tableau-Beweisbäume

Ein Tableaubeweisbaum im klassischen Sinne repräsentiert die disjunktiven und konjunktiven Verzweigungen eines Tableaubeweises. Zum besseren Verständnis gebe ich eine kurze Einführung in die Technik von Tableaubeweisen.
Beispiel: (Für einen klassischen Tableaubeweis)

```
T : (p and q) or l
```

Wir fragen hier für welche Belegungen der Variablen p und q die Gesammtformel den Wert T (true) annehmen kann. Der Beweis besteht aus der Zerlegung der Formel in seine Bestandteile oder besser Teilformeln.



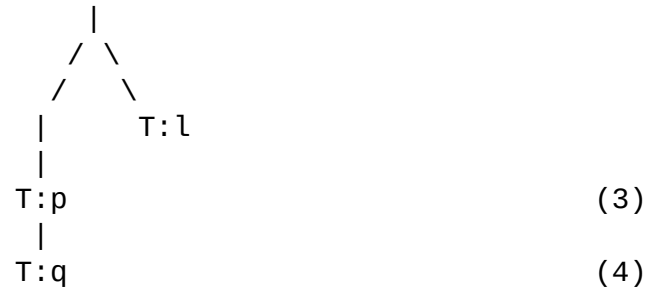
Die Verzweigung des Baumes stellt zwei mögliche Alternativen von Belegungen dar,d.h. die Formel kann den Wert true annehmen, wenn entweder die Teilformel (p and q) den Wert true annimmt oder die Teilformel l den Wert true annimmt. Die Formel (1) wird nicht mehr benoetigt da sie jetzt durch den entstandenen Baum repräsentiert wird.
Der Rechte Ast des Baumes zeigt eine atomare signierte Formel, die als Belegung der Variablen l interpretiert werden kann.

Definition 1: Unter einer signierten Formel versteht man einen Ausdruck von der Form

```
tv: fml
```

tv ist ein Element aus einer Wahrheitsmenge und fml ist Element einer Formelmenge.

Eine signierte Formel wie `"T:(p and q)"`, ist also eine Formel die mit einer Frage nach einem Wahrheitswert versehen ist.
Zurück zu unserem Beispiel. Der linke Ast des Baumes kann noch weiter abgebaut werden:

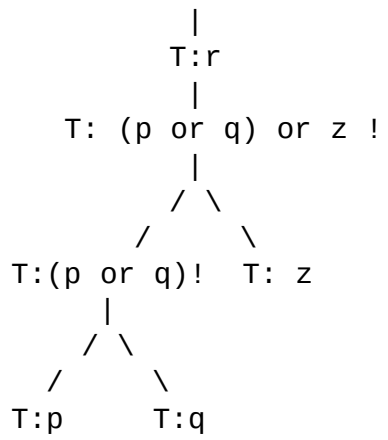


Der Abbau der Teilformel (2) führt zu einem konjunktiven Ast, d.h. die Bedingungen (3) und (4) müssen beide erfüllt werden, was leicht zu erkennen ist.
Die obige Formel nimmt den Wert T an wenn entweder p und q den Wert T annehmen, oder l gleich T, man erhält also zwei alternative Wertbelegungen der Variablen.
Allgemein ist jetzt zu sagen, das man alle Alternativen (disj.) Belegungen erhält , indem man die atomaren Formeln jedes Weges von der Wurzel des Baumes zu einem Blatt zusammenfast. Jede dieser Mengen bildet eine konjunktion von Belegungen.
Für das obige Beispiel erhält man die beiden Mengen:

```
M1 = {p=T,q=T}
M2 = {l=T}
```

Ein etwas größeres Beispiel:

```
T: r and ((p or q) or z)  ==>
```



Der Übersicht halber sind hier die Teilformeln der Formel an den Wurzeln der Teilbäume beibehalten worden (mit ! markiert). Hier erhält man die folgenden Mengen von alternativen Belegungen:

```
M1 = { r=T , p=T }
M2 = { r=T , z=T }
M3 = { r=T , q=T }
```

"r" ist in jeder Menge enthalten, da es auf jedem Weg von der Wurzel zu einem Blatt liegt.

1.2 Allgemeingültigkeit und Widersprüche

Ein wichtiges Ziel von Tableaubeweisen ist die Allgemeingültigkeit von Formeln zu zeigen. Dies wird im klassischen Fall dadurch bewältigt, indem man zeigt, daß die Frage nach dem Wert false nur Mengen von Belegungen liefert, die pro Menge mindestens einen Widerspruch aufweisen. D.h. das es ein Belegungspaar in jeder Menge gibt von der Form

```
{ ... , p=T , .... , p=F }
```

Eine Variable die beide Wahrheitswerte annehmen soll führt im klassischen Sinn zum Widerspruch. Beispiel:

```
F:p or (not p) !              (1)
      |
     F:p                      (2)
      |
  F:(not p) !                 (3)
      |
     T:p                      (4)
```

Dieser Baum enthält nur einen Ast. Die Menge von Belegungen die man erhält, enthält nur das folgende Element

```
M = {F:p,T:p}
```

Diese enthält einen Widerspruch, also ist unsere Formel eine Tautologie.
Ich fasse noch einmal kurz zusammen. Ein Beweis führt entweder zum Widerspruch, indem alle seine disjunktiven Aeste einen Widerspruch enthalten, oder er liefert eine Menge von alternativen Belegungen der Variablen. Die Evaluierung der Formel unter diesen Belegungen liefert dann jeweils den erfragten Wahrheitswert des Beweises.

Definition 2: Ein Tableaubeweisbaum heißt widersprüchlich im klassischen Sinne, wenn ALLE Mengen von alternativen Belegungen mindesten einen Belegungspaar der Form

```
p=T und p=F
```

für beliebige Variablen p enthält. Man sagt hier auch, der Beweisbaum schliesst.

1.3 Repräsentation eines Beweisbaumes

Zur Darstellung eines Baumes werde ich eine Notation wählen die die verschiedenen Verzweigungen eines Baumes und dessen Bestandteile kennzeichnet. Nennen wir den Typ eines Beweisbaumes tableau_tree.

- Man benoetigt die Moeglichkeit den leeren Baum darzustellen. Wir bezeichnen ihn mit

```
=> Empty
```

- Die elementaren Daten des Baumes sind signierte Formeln. Im Baum wird dieser durch den Typkonstruktor SF gekennzeichnet.

```
=> SF (truethvalue,formula)
```

  Man koennte auch sagen SF konstruiert mit seinen Bestandteilen eine signierte Formel.

- Als nächstes benoetigen wir die Darstellungen der konj. sowie disj. Verzweigungen. Wir gehen von unmittelbaren Darstellung durch den oben illustrierten Baum weg, und ziehen die Termdarstellung (oder Konstruktordarstellung) vor. Nachteil dieser Darstellung ist, daß sie nicht so anschaulich ist wie die gezeigten Bäume. Sie erlaubt aber eine kompaktere Handhabung und unter Zunahme der später entwickelten Regeln lässen sich mit dieser Darstellung algebraische Termumformungen durchführen.

  – Die konj. Verzweigung nennen wir Alpha und bezeichnen sie mit dem Konstruktor && . Die Komponennten sind natürlich vom Typ tableau_tree. Binäre Konstruktoren werden infix notiert.

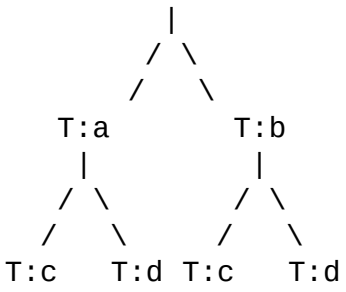
```
=>  tableau_tree && tableau_tree
```

  – die disj. Verzweigung nennen wir Beta und bezeichnen sie mit dem Konstruktor || .

```
=>  tableau_tree || tableau_tree
```

Ein Beispiel: (zunächst für die Baumdarstellung)

```
T: (a or b) and (c or d)  ==>
```



Die Expansion unserer Termdarstellung sieht folgendermaßen aus: (wir starten mit der signierten Formel)

```
SF(T , (a or b) and (c or d))    (1)

==> SF(T,(a or b))  &&  SF(T,(c or d))

==> (SF(T,a) || SF(T,b))  &&  (SF(T,c) || SF(T,d))
```

Offen ist jetzt wie man Belegungsmengen gelangt. Ein erster Anstatz ist, zunächst alle ||'s(Betas) auf oberste Ebene zu ziehen. Das heißt, daß wir einen Term der Form

```
A1 || A2 || ... || An
```

erhalten, wobei es kein Ai (i aus 1 . . . n) gibt das einen Teilterm der Form (t1 || t2) enthält. t1 und t2 sind vom Typ tableau_tree. Dies ist Vergleichbar mit der Umformung eines Ausdruckes in disjunktive Normalform. Man kann nun durch reine Termtransformationen zu alternativen Belegungen gelangen. Jetzt zunächst die Regeln, wie man die 'DNF' herstellt. Zur Abkürzung werden Signierte Formeln der Form SF(tf,fml) durch (tf:fml) dargestellt.

REGEL 0 :

```
1)  a1 && (a2 || a3)  ->  (a1 && a2) || (a1 && a3)
2)  (a1 || a2) && a3  ->  (a1 && a3) || (a2 && a3)
```

Anwendung auf das letzte Beispiel:

```
     ( T:a || T:b ) && ( T:c || T:d )

==>  (T:a && ( T:c || T:d ))  ||          (Regel 2)
     (T:b && ( T:c || T:d ))
```

```
==>  (T:a && T:c) ||                    (Regel 1)
     (T:a && T:d) ||
     (T:b && T:c) ||
     (T:b && T:d)
```

Betaklammerungen sind assoziativ und kommutativ und daher stehen die Alpha Terme alle in einer disjunktiven Beziehung. Die Belegungen sind also einfach nur die Argumente der Betas.
In diesem Fall:

```
M1 = { a=T , c=T }
M2 = { a=T , d=T }
M3 = { b=T , c=T }
M4 = { b=T , d=T }
```

1.4 Die Tableauregeln

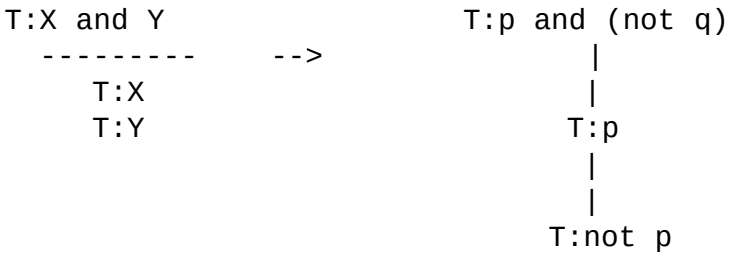
Bei den bisherigen Erläuterungen haben wir uns auf rein klassischem Boden bewegt. Um nun auf neuen Bestandteile hinzugelangen, wird zunächst ein Thema einfügt, das bei der Behandlung der Tableau Methode noch nicht erwähnt wurde. Hierbei handelt es sich um die Tableauregeln. Sie geben an welche Art der Expansion einer signierten Formel auszuführen ist. Im klassischen Fall gibt es für jeden Junktor zwei Regeln, d.h. für jeden Wahrheitswert eine.
Wir betrachten die Regeln für `'and'` und `'or'` und die Negation`'not'` :

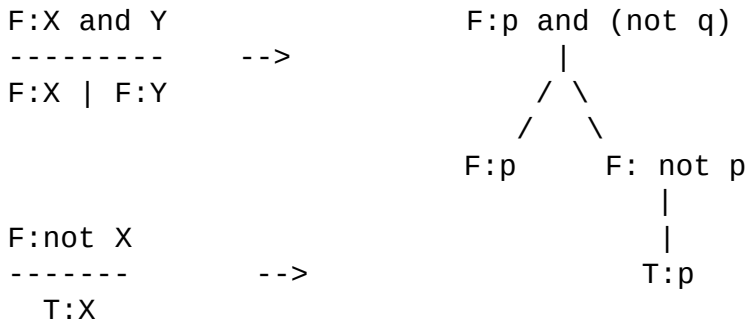
```
1.              T:X and Y         F:X and Y            (1)
                ---------         ---------
                   T:X           F:X | F:Y
                   T:Y

2.              T:X or Y          F:X or Y
                ---------         --------
                T:X | T:Y            F:X
                                     F:Y

3.              T:not X           F:not X
                --------          --------
                  F:X               T:X
```

Man liest die obigen Tableaus für T folgendermassen. Die Formel "X and Ynimmt genau dann den Wert T an, wenn sowohl die Teilformel X als auch die Teilformel Y den Wert T annehmen. Das Untereinanderschreiben hat bezeichnet das konjunktive Verhältnis zweier Teilformeln X und Y zueinander, Nebeneinanderschreiben das disjunktive Verhältnis. Nebeneinander und Untereinanderschreiben korrespondieren also zu den || und && Konstruktoren.
Die Regeln finden nun die folgenden Anwendung innerhalb eines Tableaubeweises:





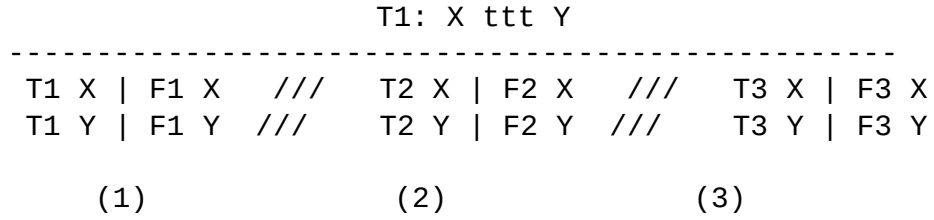
Dies ist im wesentlichen alles was für die weitere Entwicklung der Regeln unter dem Kontext der PKL benoetigt wird.

1.4.1 Übergang zur PKL

Die Tableaus der PKL sehen komplexer aus, und enthalten neben Alpha und Beta Beziehungen, die hier auch gemischt innerhalb der Regeln vorkommen, auch transjunktive Beziehungen, d.h. das es für eine Anfrage für einen Wert aus einem System es auch Belegungsmoeglichkeiten aus anderen Systemen geben kann.
Bei einem n-stelligen System kann es hier natürlich bis zu n-1 transjunktive Alternativen geben.

Definition 3: Als transjunktiven Anteile werden alle Wertangebote bezeichnet, die nicht aus dem gleichen System stammen wie der erfragte Wert.

Die transjunktiven Anteile werden im Tableau durch einen Dreifachstrich getrennt.In folgenden werde ich transj. Anteile auch häufiger mit T-Anteile bezeichnen. Hier nun ein Beispiel für die Transjunktion ttt mit dem Tableau für T1.



Die erste Alpha-Beta Komponente (1) bezeichne ich den 'lokalen Anteil', da er im selben System bleibt. (2) und (3) sind die transjunktiven Anteile.
Lange Zeit hat uns die Frage beschäftigt, wie die Transjunktion im Tableau darzustellen ist. So stand zur Frage ob sie als herkoemmliche Disjunktion, also als Beta aufzufassen sei. Wir sind nun zu dem ersten Entschluss gekommen, das die Transjunktiven Anteile zunächst unabhängige Alternativen zum lokalen Anteil bilden, und zu den Belegungen des resultierenden Beweises des lokalen Anteils keine Beziehung haben.
Allerdings sind sie auch keine richtigen Betas , da sie ansonsten mit Teilen des Baumes eine Beziehung hätten die es eigentlich nicht gibt.
Wir führen hier eine explizite Darstellung für T-Anteile in Beweisbäumen, sowie einen erweiterten Regelsatz ein.
Beispiel für einen Ausdruck mit Transjunktion

```
H = (p ttt q) aaa (p ooo q)

    ( aaa ist hier die dreistellige Konjunktion und
      ooo die dreistellige Disjunktion )

                  t1:H
                    |
               t1:(p ttt q)
                    |
               t1:(p ooo q)
```

Teilformeln werden von nun in der Art expandiert indem sie an Ort und Stelle durch ihre regelbestimmte Alpha-Beta Struktur ersetzt werden.
Wir erhalten zunächst

```
          T1:(p ttt q)
                |
           T1:(p ooo q)          (*)

==>
          T1:(p ttt q)
                |  (!)
               / \
              /   \
          T1:p    T1:q            (DTB)
```

`(!)` ist die Anwendung der Tableauregel für (*).
Durch die verschachtelte Alpha-Beta Struktur müssen für ttt mehrere Expansionsschritte gleichzeitig vorgenommen werden. Der Übersichtlichkeit halber wird jedes weitere Vorkommen des Disjunktiven Teilbaumes mit `(!)` abgekürzt. Die transjunktiven Anteile vermerke ich an dem Knoten der Regelanwendung. Der Grund hierfür wird später hoffentlich klar.

```
 ==>
                L(1)                  T(2)                  T(3)
                  ____________________________________________
                 /\                    /\                    /\
                /  \                  /  \                  /  \
           T1:p      F1:p        T2:p      F2:p      T3:p      F3:p
             |         |           |         |         |         |
           T1:q      F1:q        T2:q      F2:q      T3:q      F3:q
             |         |
            (!)       (!)
            / \       / \
       T1:p T1:q T1:p T1:q
```

Hier wäre der Beweis abgeschlossen, und neben den Belegungen des eigenen Systems sind hier noch zusätzliche Belegungen aus den anderen Systemen. Der Widerspruchbegriff muss hier natürlich daraufhin erweitert werden das auch alle alternativen transj. Äste schließen müssen.
Ich werde jetzt zunächst auf unsere Darstellung der transj. Anteile in unserer Baum(Term)darstellung eingehen, und dann letzteres Beispiel nochmal unter leicht modifiziertem Gesichtspunkt aufgreifen.

1.4.2 Repräsentation der T-Anteile

Wir benoetigen eine Struktur, die die T-Anteile an gegebener Stelle festhällt bzw kennzeichnet. Sie enthält zum einen den weiteren lokalen Teilbaum sowie eine Liste der T-Anteile, die ja selbst auch wieder Bäume sind. Wir nennen den Konstruktor Trans und bezeichnen ihn mit `///` .

```
=>  tableau_tree /// tableau_tree list
```

Ich greife das letzte Beispiel nocheinmal auf , und bearbeite es unter dieser Darstellung. Hierzu gebe ich das Tableau für die Transjunktion nochmal in leicht modifizierter Darstellung an.
Die Liste t1,t2,. . .,tn von T-Anteilen wird durch <t1,. . .,tn> abgekürzt.
Beispiel:

```
        H = (p ttt q) aaa (p ooo q)

        T1: H

==>     T1: (p ttt q) && T1:(p ooo q)

==>     T1: (p ttt q) && (T1:p || T1:q)

==>     (1 /// <2,3>) && (T1:p || T1:q)
```

1,2 und 3 bezeichnen hier die drei Anteile in der Tableauregel für ttt. Die Reduktion gerät hier ins stoppen, da wir hier keine Tableau Regel mehr anwenden koennen, aber auch keine Regel haben, die Betas auf oberste Ebene transportieren kann.
Wir hatten nun angenommen, dass die T-Anteile unabhängig von Rest des Baumes seien.Gesucht sind nun Regeln, die den Baum in die folgende Form bringen:

```
dnf /// <t1..tn>
```

`"dnf"` ist der lokale Baum in disjunktiver Normalform.Er enthält selbst keine Äste die T-Anteile enthalten.
Eine mögliche Regel für das obige Beispiel wäre :

```
(a /// list) && b   ==>   (a && b ) /// list
```

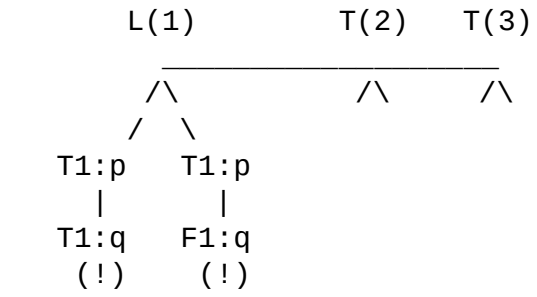
Es ist nun Aufgabe Regeln zu finden, die auch die Beziehungen zwischen T-Anteilen aus verschiedenen Aesten mit einbeziehen. Hierzu betrachten wir ein geeignetes Beispiel.

```
H = (p ttt q) aaa (p ttt r)
```

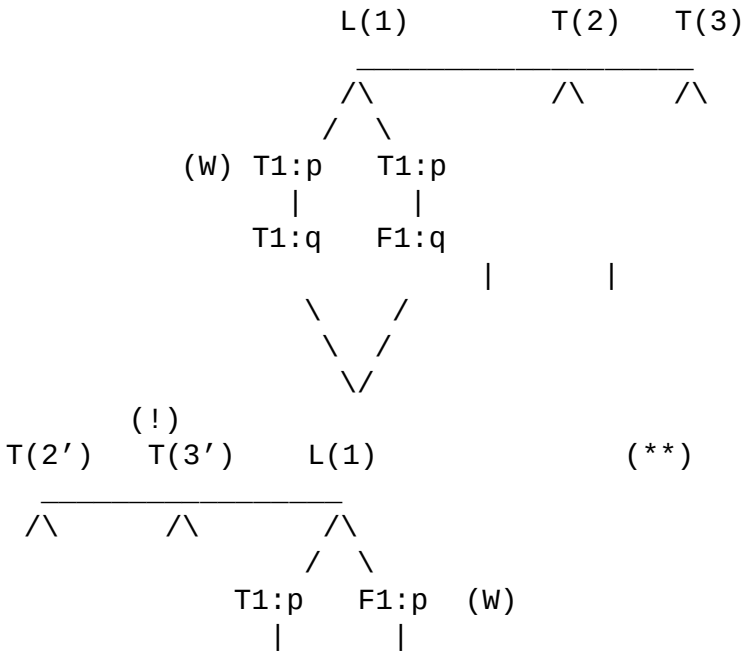
Wir betrachten die ersten Expansionsschritte:

```
T1:(p ttt q)       (1)
     |
T1:(p ttt r)       (2)
```

Expandiert wird nun zunächst Teilbaum (1). Vorkommen von Teilbaum (2) stelle ich wie schon vorhin abkekürzt durch (!) dar. Auch die T-Anteile kennzeichne ich nur noch durch abstrakte Bezeichner wie z.B. T(2). Es ergibt sich der folgende Baum:



Durch Expandierung von Teilbaum (2) kommen wir zu der folgenden Darstellung:



T1:r F1:r

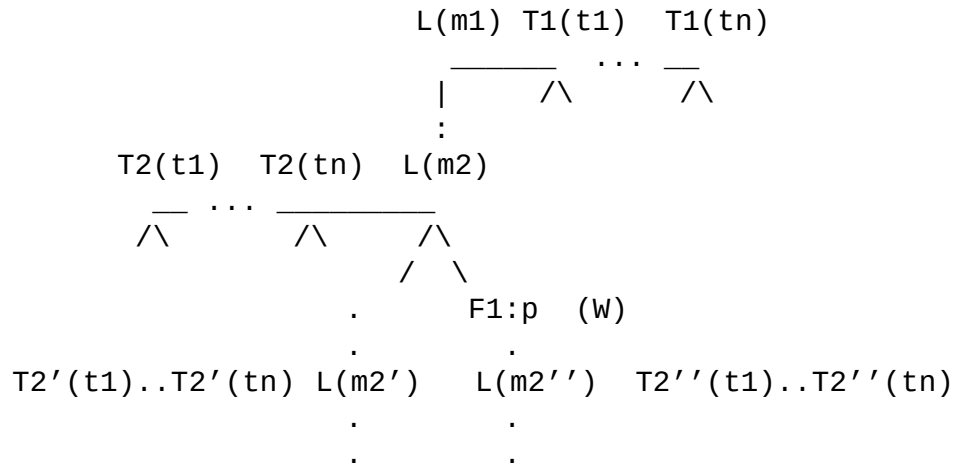
Nach unseren bisherigen Annahmen sind die T-Anteile T(2),T(2'), T(3) und T(3') völlig unabhängig voneinander. Für die T-Anteile T(2) und T(3) sollte dies auch der Fall sein, da wir hier gesagt hatten, das dieses Alternativen aus verschiedenen Systemen sind. Ebenso bei T(2') und T(3').
Betrachtet man aber die Beziehung zwischen T(2) und T(2') sowie zwischen T(3) und T(3') so stellen wir fest, das sie im Baum in einer Art Alpha Beziehung stehen. Dies ist auch nicht verwunderlich, da diese Beziehung ja aus den Termen (1) und (2) entstanden ist die ja auch in dieser Beziehung zueinander standen.
Es liegt nicht fern, daß es berechtigt ist anzunehmen, daß T-Anteile die im Baum einer konjunktiven Beziehung unterliegen diese auch in den einzelnen Subsystemen aufweisen. Hierbei ist darauf zu achten das dies nur für gleiche Subsysteme gilt. In Bezug auf das obige Beispiel heißt das, daß T(2) und T(2') in einer konjunktiven Beziehung stehen. Unter verschiedenen Subsystemen besteht nach wie vor keine Beziehung.

BEMERKUNG: Es ist wichtig an dieser stelle zu sagen, daß die letzten Annahmen auf reiner Intuition beruhen, und an dieser Stelle noch durch keine analytischen Beobachtungen begründet sind !!!

Wir müssen hier zunächst abwarten ob die weitere Arbeit unter diesen Annahmen zu schlüssigen Ergebnissen gelangt Wir hoffen das aber auch schon der Aufbau des formalen Gerüsts unter diesen Bedingungen weitere Einblicke gewährt.

Unklar sind in diesem Kontext noch die Beziehungen der verschiedenen Systeme eines T-Anteils zueinander, die hier zunächst einmal als unabhängig voneinander angenommen werden.

Der Aufmerksame Leser wird gemerkt haben, das in dem obigen Beispiel die T-Anteile T(1'),T(2') in zwei alternativen Aesten auftauchen, und sich warscheinlich fragen in welcher Beziehung denn diese nun stehen moegen. Hier sei nun zunächst nur gesagt, dass diese scheinbare Beziehung in diesem Fall nur eine Pseudo-Beziehung ist. Sie ergibt sich aus der disjunktiven Aufspaltung, wodurch sich zwei redundante Alternativen der T-Anteile ergeben. Diese redundante Darstellung ist auf die Darstellung durch den Baum zurückzuführen. Wir werden sehen, dass dies in der Termdarstellung nicht mehr der Fall sein wird.
Es gibt aber tatsächlich auch disjunktive Beziehung zwischen den transjunktiven Anteilen. Ich moechte hierfür ein Beispiel konstruieren. Nehmen wir an wir hätten schon folgenden Beweisbaum vorliegen:



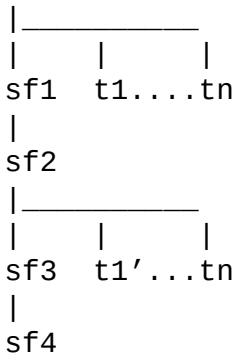
Nehmen wir weiterhin an, das alle Teilbaume unterhalb von L(m2) nur durch Abbau des Teilterms der L(m2) hervorgebracht hat entstanden sind. Dann sollten auch die T-Anteile T2'(t1) und T2"(t1) keine trivialen Redundanzen wie oben aufweisen, was hier heist, das sie also in einer nichttrivialen disjunktiven Beziehung stehen. Es ergibt sich die folgende Struktur der T-Anteile, unter der Annahme das diese die einzigen im obigen Baum sind.
Für alle i aus $\{1 \ldots n\}$ gilt :

```
T1(ti) && (T2(ti) && (T2'(ti) || T2''(ti)) )
                     ========================
```

Man erkennt hier, das die T-Anteile also ihre Struktur im Baum beibehalten, was sie also den gleichen Regeln unterwirft, wie wir sie schon bei den herkoemmlichen Alpha-Beta Strukturen kennengelehrnt haben.
Der Einfachheit halber wollen wir annehmen, das der T-Anteil immer alle Subsysteme umfasst, sodas sich die obigen Beziehungen gleichfoermig für alle T-Anteile konstruieren lassen. Natürlich ist diese Forderung durch die Tableauregeln nicht zu halten. Wir behelfen uns hier dadurch , das alle nicht vorkommenden Subsysteme im T-Anteil durch die leere Alpha-Beta Struktur Empty dargestellt werden. Diese leere Struktur bildet bezüglich dem Alpha und Beta Operatoren (wenn man sie Semantisch interpretiert) ein Neutrales Element, sodass gilt:

```
a && Empty  = a
Empty && a  = a
a || Empty  = a
Empty || a  = a
```

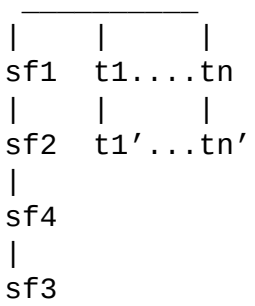
Wir wollen uns nun den Regeln des || Operators zuwenden. Betrachten wir zunächst den folgenden Baum sowie dessen Termrepräsentation:



Die sf's bezeichnen signierte Formeln , die t(a)'s die T-Anteile. In der Termschreibweise sieht der Baum folgendermassen aus:

```
( (sf1 /// <t1..tn>)    &&  sf2 )  &&
( (sf3 /// <t1'..tn'>)  &&  sf4 )
```

Selbst wenn eine der sfn's zu einer beta-Struktur expandiert würde, blieben die Alpha Beziehungen der T-Anteile bestehen.Aus diesen Betrachtungen koennen wir jetzt den Schluss ziehen, dass T-Anteile aus geschachtelten Alpha Termen extrahiert werden koennen. Das Resultat ist dann ein reiner Alpha Term sowie die konjunktiv verknüpften T-Anteile der einzelnen Subsysteme. Für den obigen Baum sieht das folgendermassen aus:



Das Resultat der Termnotation ist

```
(sf1 && sf2 && sf3 && sf4) /// <(t1 && t1')..(tn && tn')>
```

Wir sind nun in der Lage erste Regeln für den && Operator aufzustellen.

REGEL 1 :

Beide Teilterme des && Op's enthalten Transjunktive Anteile

```
(t /// ta) && (t' /// ta')  ==>  (t && t') /// (ta &&&' ta')
```

REGEL 2 :

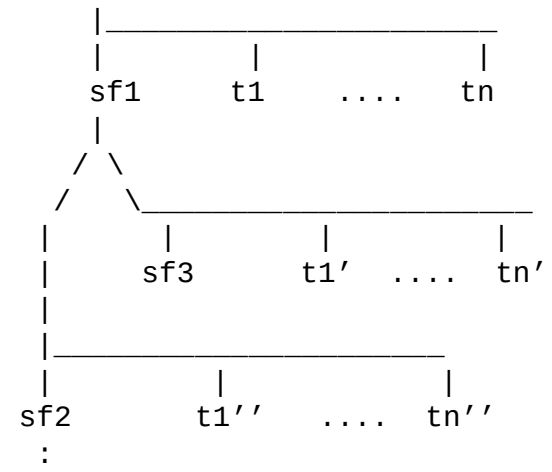
Nur ein Term enthalt einen T-Anteil

```
a)  t && (t' /// ta')  ==>  (t && t') /// ta'
b)  (t /// ta) && t'   ==>  (t && t') /// ta

    mit   &&&':  <t1..tn> &&&' <t1'..tn'> := <(t1 && t1')..(tn && tn')>
```

BEMERKUNG: Auch die Regeln unter 2a und 2b bedürfen noch einer genaueren Untersuchung, ob das Fehlen eines T-Anteils in einem der beiden Aeste nicht irgendwelche Auswirkungen auf den anderen T-Anteil hat.

Man ist nun versucht für den || Operator eine ähnliche Betrachtung durchzuführen. Dazu wieder ein Beispiel:

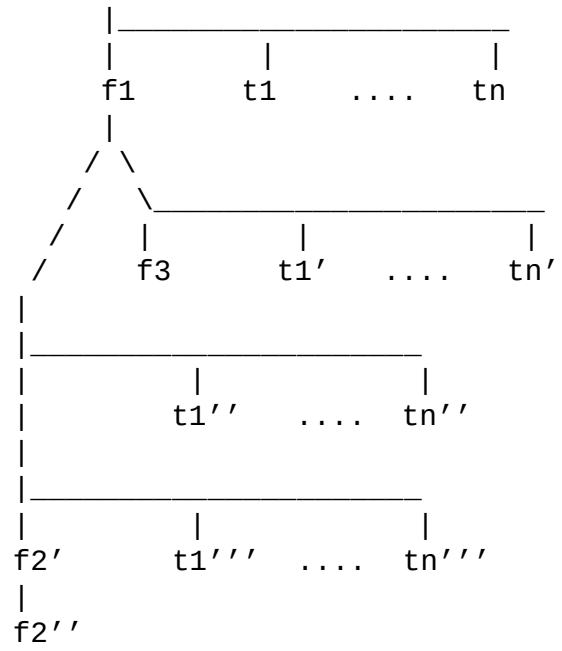


Wir wollen der Uebersicht halber unsere Betrachtungen jetzt nur auf die T-Anteile aus System I beschränken. Im ersten Moment koennte man vermuten, dass folgende Beziehung besteht:

```
sf1 & (sf2 | sf3)  ||  ((t1 & t1') | (t1 & t1'')) ....
```

Das ist auch nicht verkehrt, nur lässt sich das in diesem Fall nicht als allgemeine Regel fassen. Das Problem ist, das wir nicht wissen was die weitere Entwicklung von sf2 und sf3 noch hervorbringt.

Betrachten wir ein Beispiel für den Fall, das sf3 bereits atomar ist, also keiner weiteren Expandierung mehr unterliegt. sf2 wird nun folgendermassen expandiert:



Nach der obigen Annahme ergibt sich jetzt:

```
f1 && ((f2' && f2'') ||
       f3 /// <((t1 && t1') || (t1 && t1'')) && t1''' , ....
                              ---------------------------------
```

Für den unterstrichenen Teil ergibt sich die folgende Beziehung des T-Anteils im ersten System

```
((t1 & t1') & t1''') | ((t1 & t1'') & t1''')
                -------------------
```

t1"' steht jetzt fälschlicherweise auch mit t1' in einer Alpha Beziehung. In diese Lage sind wir nun geraten, weil wir zu früh T-Anteile aus einer Beta-Beziehung zusammengefügt haben.
Resultat aus diesen Betrachtungen ist, das die T-Anteile aus Beta-Zweigen erst zusammengefügt werden koennen, wenn gewährleistet ist, das der lokale Anteil des || Operators keine weiteren ///-Terme mehr enthält, und jede weitere Expandierung keine weiteren ///-Terme mehr hervorbringt. Die zweite Bedingung ist sehr schwer vorauszusehen, und daher verschärfen wir unsere Bedingung noch:
Gegeben ist ein Term

```
(t /// ta) || (t' /// ta')
```

Die T-Anteile dürfen erst dann in Beziehung gesetzt werden,wenn gewähr- leistet ist, das die Terme t und t' keine komplexen signierten Formeln mehr enthalten, und das t und t' keine Unterterme der Form (tt /// tta) enthalten.
Wir sehen das Teilterme einer Beta-Beziehung im Gegensatz zur Alpha Struktur komplett expandiert werden müssen. Es steht jetzt die Frage offen wie wir diese Eigenschafft eines Terms propagieren wollen, ohne einen Term auf diese Eigenschafft zu Testen. In anderen Worten, wie kann man mit einfacheren Eigenschaften ,wenigen Regeln sowie einem neuen Operator einen Term als ///-frei kennzeichnen.

1.4.3 T-Anteil freie Terme

Gehen wir schrittweise vor, und betrachten wir zunächst, wie wir einen solchen Term kennzeichnen:

```
{t}  :=   t ist ///-frei
```

Der kleinste Term von dem wir definitiv wissen das er `///` frei ist, ist die atomare signierte Formel , also

1. t ist atomare signierte Formel:

   ```
   =>  {t}
   ```

   sonst expandiere sf unter Anwendung des zugehoerigen Tableaus.

2. t und t' sind `///` frei :

   ```
   => {t} && {t'} -> {t && t'}
   ```

3. t und t' sind `///` frei :

   ```
   => {t} || {t'} -> {t || t'}
   ```

Wir sind nun in der Lage die Eigenschaft `///` frei zu Propagieren. Was uns noch fehlt sind die Regeln zum liften der T-Anteile aus Beta-Zweigen.

REGEL 3 :

Beide Teilterme des || Operators enthalten Transjunktive Anteile :

```
({t} /// ta) || ({t'} /// ta')  ==>  ({t || t'}) /// (ta |||' ta')
```

REGEL 4 :

Nur ein Term enthalt einen T-Anteil :

```
a)  {t} || ({t'} /// ta')  ->  {t || t'} /// ta'
b)  ({t} /// ta) || {t'}   ->  {t || t'} /// ta
```

mit `<t1..tn> |||' <t1'..tn'> := <(t1 || t1')..(tn || tn')>`

1.4.4 Geschachtelte /// - Terme

Was jetzt noch zu tun bleibt ist, wie man mit Termen der Form

```
(t /// ta) /// ta'
```

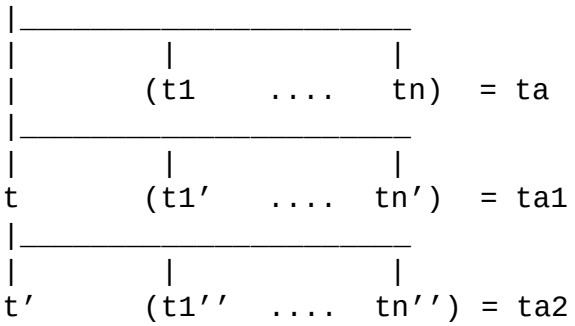
umzugehen hat. Zunächst analysieren wir wie diese Terme überhaupt entstehen. Wir gehen von der folgenden Situation aus:

```
(sf && sf') || ta
```

Nehmen wir an sf und sf' sin signierte Formrln die auf folgende Weise expandiert werden:

```
((t /// ta1) && (t' /// ta2)) /// ta
```

Das entspricht der folgenden Baumdarstellung



Wir wenden nun Regel (1) an und erhalten

```
(*)   ((t && t') /// (ta1 &&&' ta2)) /// ta
```

Aus der Baumdarstellung ist ersichtlich, das ta, ta1 und ta2 in konjunktiver Beziehung stehen

```
(*) -> (t && t') /// (ta1 &&&' ta2) &&&' ta
```

Wir wollen nun noch ein Beispiel für die disjunktive Beziehung betrachten

```
(sf || sf') /// ta
```

Wie schon oben nehmen wir an sf und sf' signierte Formeln sind,die auf folgende Weise expandiert werden:

```
      (({t} /// ta1) || ({t'} /// ta2)) /// ta
==>   (({t} || {t'}) /// (ta1 |||' ta2)) /// ta
```

Wir wissen das ta zu ta1 und zu ta2 in konjunktiver Beziehung steht, das aber beide Aeste disjunktive Alternativen repräsentieren. Sehen wir was passiert wenn wir in der gleichen Weise verfahren wie schon im letzten Beispiel:

```
      (([t] || [t']) /// (ta1 |||' ta2)) /// ta

==>   ([t] || [t']) /// (ta1 |||' ta2)) &&&' ta

==>   ([t] || [t']) /// (ta1 &&&' ta) |||' (ta2 &' ta)
```

Das ist genau das Resultat das wir uns vorgestellt haben. Wir koenen die Regel nun allgemein formulieren.

Regel 5: `(t /// ta) /// ta'  ->  t /// (ta &&&' ta')`

Wir haben jetzt alles was wir zur Darstellung der Beweisbäume benoetigen, und wissen wie wir sie in eine Art Normalform umformen kann aus der man dann die Belegungsinformation erhalten kann. Was wir noch nicht besprochen haben ist wie der Widerspruchsbegriffin unsere Darstellung einfliesst. Ich werde nun zunächst die bisher erarbeiteten Ergebnisse zusammenfassen, und dann zeigen, das dies mit den jetzigen Mitteln moeglich ist. Ich werde aber auch einen weiterreichenden Gedanken aufnehmen, der zu einer anderen Loesung führt, in der auch die `///` freien Terme einbezogen werden. Wir werden sehen das hier nur ein sehr kleiner Teil der entwickelten Regeln modifiziert werden muss, und das wir hierdurch zu einer sehr elaganten Darstellung der schon erwähnten DNF und des Widerspruchbegrifs gelangen.
Sie wird später im Baum durch den Konstruktor DNF gekennzeichnet und stellt im eine äquivalente Repräsentation der `///` freien Terme dar. Diese Information war noetig um gleich unsere Datenstruktur komplett aufzubauen.

1.4.5 Zusammenfassung der Regeln

Ein Beweisbaum lässt sich in der folgenden Datenstruktur zusammenfassen

```
  tableau_tree :=  Der leere Baum    => Empty
                |  Signierte Formel  => SF (truval,formel)
                |  Alpha Beziehung   => tableau_tree && tableau_tree
        |  Beta Beziehung    => tableau_tree || tableau_tree
                |  T-Anteile         => tableau_tree /// tableau_tree list
|  Dnf's (/// frei)  => DNF of dnf
```

Hier die Zusammenfassung der Regeln:

Regel 0 : Zuerst die Regeln zur Herstellung der DNF

```
          a)  a1 && (a2 || a3)  ->  (a1 && a2) || (a1 && a3)
          b)  (a1 || a2) && a3  ->  (a1 && a3) || (a2 && a3)
```

Regel I : Die Regeln für die alpha Beziehung von Termen mit T-Anteilen

```
    a) (t /// ta) && (t' /// ta')  ->  (t && t') /// (ta &&&' ta')
          b)  t && (t' // ta')           ->  (t && t') ///  ta'
          c)  (t /// ta) && t'           ->  (t && t') ///  ta

          <t1..tn> &&&' <t1'..tn'> := <(t1 && t1')..(tn && tn')>
```

Regel II : Die Regeln für die Beta Beziehung von Termen mit T-Anteilen

```
    a) ([t] /// ta) || ([t'] /// ta')  ->  [t || t'] /// (ta |||' ta')
          b) [t] || ([t'] /// ta')           ->  [t || t'] ///  ta'
          c) ([t] /// ta) || [t']            ->  [t || t'] ///  ta

          <t1..tn> |||' <t1'..tn'> := <(t1 | t1')..(tn | tn')>
```

Regel III: Die Regel für verschachtelte /// Terme

```
         (t /// ta) /// ta'  ->  t || (ta &&&' ta')
```

Regel IV : Die Regeln zur propagierung der /// freien Terme

```
a)  t ist atomare signierte Formel

    =>  [t]

        sonst  expandiere sf unter Anwendung des
        zugehoerigen Tableaus.

b)  t und t' sind /// frei

    => [t] && [t'] -> [t && t']

c)  t und t' sind /// frei

    => [t] || [t'] -> [t || t']
```

Die Regel 1a) kann man auch weglassen, da sie sich aus 1b) und 1c) und 3) herleiten lässt. Dieses gilt nicht für die Regel 2a) !!
Die Regeln für die Propagierung der /// freien Terme sind bewusst nicht auf die Regeln 1) und 2) verteilt worden. Dieser Teil wird den schon oben angesprochenen Modifikationen unterworfen. Hieraus ergibt sich sogar, das die Regeln unter 0. über sind. Dies wird im Folgenden unter dem Kontext des Widerspruchbegriffs erarbeitet.

1.5 Der Widerspruchbegriff

Nehmen wir erst nochmal den Widerspruchbegriff des klassischen Falls. Wir gehen von der DNF des Beweisbaumes aus, auf denen alle Beta's auf die oberste Ebene geliftet wurden. D.h. dass es keine konjunktiven Zweige mehr gibt, die disjunktive Verzweigungen aufweisen. Die konjunktiven Strukturen der verschiedenen beta-Zweige bilden die verschiedenen Belegungsalternativen. Eine Belegungsalternative heisst im klassischen Sinn geschlossen, wenn einer Variablen zwei unterschiedliche Wahrheitswerte zugewiesen werden.

```
A = a1 && a2 && .. && an
```

Definition A heißt geschlossen wenn ai und aj existieren, so daß
`ai = { T x }  und  aj = { F x }`

Der **Beweis** heißt geschlossen wenn alle Alternativen schliessen.
Wie kënnen wir nun den Begriff des schliessens propagieren ? Hier greifen wir auf die leere Alpha Menge zurück, und sagen, dass wenn eine Alternative schliesst sie gleich dieser leeren Menge ist ist. Da die Menge von Alternativen Alpha Mengen sind sagen wir, dass eine Beta Menge schliesst, wenn alle ihre Alpha Mengen geschlossen sind, d.h. also wenn sie leer ist.

```
{} := die leere Beta Menge
[] := die leere Alpha Menge
```

Die Regeln für die Handhabung mit leeren Mengen

```
=>  a) t || {} = t
    b) {} || t = t
    c) t && {} = {}
    d) {} && t = {}
```

Ein Beweis führt zum Widerspruch oder ist geschlossen, wenn er sich zu reduziert. Für den transklassischen Fall müssen wir den Widerspruchbegriff modifizieren.
Widersprüche ergen sich nur für Wahrheitswerte aus gleichen Systemen. Eine alternative schliesst also nicht schon bei verschiedenen Wertbelegungen derselben Variable, sondern erst wenn diese verschiedenen Wertbelegungen aus dem gleichen System stammen.
Für das Schliessen eines Beweises müssen wir nun zwei Fälle betrachten:
Das Resultat des Beweises ist

```
I.   {}                 => der Beweis schliesst
II.  {} /// <t1 .. tn> => der Beweis schliesst wenn auch alle
                           t1 .. tn schliessen
```

Wir sind also in der Lage den Begirff des Widerspruchs und der Belegungen auch im transklassischen Fall mit den jetzigen Mitteln zu formulieren.
Was hier allerdings stillschweigend mit eingeflossen ist, das irgendwie entschieden werden muss wann denn ein Beweisbaum (oder Term) in DNF ist, und wann denn zwei widersprüchliche Hypothesen in Alternativen vorliegen, sodass sich leere Mengen ergeben. Das erste Problem ergibt sich dadurch, das der Baum in DNF ist, wenn keine der Regeln mehr anwendbar ist. Der zweite Punkt ist leicht durch

das gegenseitige vergleichen der atomaren S-Formeln einer Alternative zu loesen. Das ist zwar sehr aufwendig, aber in diesem Fall unumgänglich.
Bevor ich zu einer alternativen Darstellung der DNF übergehe, will ich einen noch sehr wage angesprochenen Punkt aufgreifen. Hiermit meine ich die Expandierung von signierten Formeln. Dieser Begriff tauchte bislang häufig auf, ohne das näher auf ihn eingegangen wurde.
Zur Erinnerung :

```
(*)   t ist atomare signierte Formel SF(tv,var)   =>  [(tv,var)]
     (**)  sonst  => expandiere sf unter Anwendung des zugehoerigen Tableaus.
```

In (**) ist eine solche Regel schon informel enthalten. Wünschenswert wäre allerdings eine Regel in Form der bisherigen Termersetzung. Da es sehr mühselig ist, alle Tableauregeln als Regeln zu formulieren, werden wir diese Regeln implizit in einer Funktion verstecken.

REGEL VI :

```
SF(sf) -> expand(sf)

expand SF(tv,op(args)) = tableau_tree(op,tv,args)
expand SF(tv,var)      = [(tv,var)]
```

1.6 Die Darstellung der DNF

Wie schon gesehen werden die Belegungen erst in der DNF fassbar. Ein grosser Nachteil dieser Darstellung ist die Verschachtelung vieler Beta's und Alpha's. Natürlich werden auch viele Belegungen erzeugt die Widersprüche enthalten, die durch die Regeln unter (0) sogar noch dupliziert werden.
Die Idee ist nun schon von den Blättern des Baumes her DNF's zu erzeugen, und Regeln anzugeben wie diese DNF's disjunktiv bzw. konjunktiv zu verknüpfen sind. Die Regeln unter (4) werden der Kern der Betrachtungen sein

```
IV a)  SF(sf) -> expand(sf)
   b)  t und t' sind /// frei   ==> [t] && [t'] -> [t && t']
   c)  t und t' sind /// frei   ==> [t] || [t'] -> [t || t']
```

Aus den Mengenbetrachtungen hatten wir gesehen das die DNF eine Menge von Alpha Mengen ist, deren Gesammtheit wir mit der Beta Menge bezeichnet hatten. Wenn nun eine atomare sigierte Formel expandiert wird so erhalten wir die einelementige Beta Menge. Diese besteht aus der Alpha Menge die einzig und allein aus der atomaren signierten Formel besteht
Beispiel: expand SF(T1,Var X) => [(T1,Var X)]
Es müssen nun konjunktive und disjunktive Verknüpfungen auf diesen Mengen definiert werden werden, die zwei DNF's im Sinne der Regeln unter 0. zusammen- fügen.

1.6.1 Modifikation der Regeln

```
IV'  a)  SF(sf) -> expand(sf)
     b)  DNF(d) && DNF(d') -> DNF(t |--| t')
     c)  DNF(d) || DNF(d') -> DNF(t |++|  t')
```

Die Information der `///` freien Terme wird jetzt mit Hilfe der DNF propagiert. Das bedeutet eine Modifikation der Terminologie der Regeln unter (2).

```
==> 2'a) (DNF(t)///ta)||(DNF(t')///ta') => DNF(t |++| t')///(ta |||' ta')

        b) DNF(t) || (DNF(t')///ta')  => DNF(t |++| t')///ta'

        c) (DNF(t)///ta) || (DNF(t')  => DNF(t |++| t')///ta
```

Was jetzt noch fehlt ist die Spezifikation von `|++|` und `|--|`.

1.6.2 Die konj. verknüpfung von Dnf's $|--|$

Die Funktion `|--|` ist vom folgenden Typ

```
|--|: dnf*dnf -> dnf
```

Wir werden nun ihre Operationalität erarbeiten. Nehmen wir die beiden folgenden DNF's:

```
dnf1 :=  { [(T1,p),(T1,q)] , [(F1,p),(F1,q)] }

dnf2 :=  { [(T1,p)] , [(F1,q)] }
```

Nehmen wir weiterhin an, die beiden DNF's stehen in einer konjunktiven Beziehung. Dies entspreche in der Baumdarstellung einer alpha Beziehung von beta-Beziehungen.

```
dnf1 && dnf2
```

Daraus ergibt sich das jedes Alpha Menge aus dnf1 ,mit jeder Menge aus dnf2 in einer alpha-Beziehung steht. Dies besagt auch die folgende Aussagenlogische Formel

```
   (a or b) and (c or d)
==> ((a and c) or (a or d) or (b and c) or (b and d))
```

In gleicher Weise werden wir die beiden DNF's zusammenfügen.

```
=>  {
      [T1 p , T1 q , T1 p] ,   (1)
      [T1 p , T1 q , F1 q] ,   (2)
      [F1 p , F1 q , T1 p] ,   (3)
      [F1 p , F1 q , F1 p]     (4)
    }
```

Hier sind einige Besonderheiten zu bemerken. Zum einen sind in einigen konjunktiven Aesten ((1) + (2))doppelte atomare Formeln. Diese koennen eliminiert werden. A und A bleibt halt A. Der nächste Punkt sind die konjunktiven Aeste (3) und (4). Diese enthalten Widersprüchliche Hypothesen und Reduzieren sich daher zur leeren Menge, sodass sie eliminiert werden koennen. Diese Mengen korrespondieren zu den Widersprüchlichen Aesten im Beweisbaum.
Die obige DNF sieht jetzt folgendermassen aus =>

```
{  [T1 p , T1 q ] ,  [F1 p , F1 q ]  }
```

Wir wollen jetzt die Operationalität allgemeiner erfassen. Gegeben sind die beiden DNF's

```
dnf1 := { [ a1 ]  , .. , [ an ] }
```

```
dnf2 := { [ b1 ]  , .. , [ bm ] }


dnf1 |--| dnf2 :=
     reduce( {
              [a1]@[b1] , .. , [a1]@[b1] ,
                        ......           ,
              [an]@[bm] , .. , [an]@[bm]
              } )
{} |--| dnf = {}
dnf |--| {} = {}
```

Der Operator @ hat die beiden folgenden Aufgaben:

- Vereinigen der beiden Mengen unter Eliminierung der doppelten Hypothesen
- Das entdecken von Widersprüchen, worauf sich `[a]@[b]` zu [] reduziert. Diese leeren Mengen koennen in der DNF weggelassen werden. Die leere DNF stellt einen Widerspruch dar.

Die Funktion eliminiert redundante Alpha Mengen.

1.6.3 Die disj. Verknüpfung `|++|`

Wie schon bei join werden wir hier ihre operationalität erarbeiten. Nehmen wir wiederum die beiden folgenden DNF's

```
dnf1 :=  { [(T1,p),(T1,q)] , [(F1,p),(F1,q)] }
dnf2 :=  { [(T1,p)] , [(F1,q)] }
```

und nehmen wir jetzt an, die beiden DNF's stünden in einer disjunktiven Beziehung. Dies entspreche in der Baumdarstellung einer beta-Beziehung von beta-Beziehungen. Dies wäre aber nichts anderes als eine Alternative von Alternativen

```
(a or b) or (c or d) =>  a or b or c or d

{ [T1 p,T1 q] , [F1 p, F1 q] , [T1 p] , [F1 q] }
```

Dies entspricht also nur der Vereinigung zweier Mengen. Das ist auch schon alles was app zu leisten hat. Allgemein:

```
dnf1 := { [ a1 ]  , .. , [ an ] }
dnf2 := { [ b1 ]  , .. , [ bm ] }

dnf1 |++| dnf2 := reduce( {[a1],..,[an],[b1],..,[bm]} )
{} |++| dnf = {}
dnf |++| {} = {}
```

1.6.4 Die beiden Operatoren &&&' und |||'

Zur Spezifikation der Termrsetzungsregeln hatten wir die Operatoren &&&' und |||' verwendet, sie aber wage beschrieben. Sie dienen der disj bzw. konj. Verknüpfung der T-Anteile unserer Termdarstellung. Wir hatten gesagt, dass die T-Anteile alle Subsysteme beinhalte, und deren Verknüpfung das Resultat der jeweiligen Verknüpfung der Subsysteme ist. Subsysteme die in den Tableau-Regeln nicht enthalten sind werden durch den leeren Baum repräsentiert.

```
&&&' :  (tableau_tree list) * (tableau_tree list) -> tableau_tree list

        <t1..tn> &&&' <t1'..tn'> := <(t1 &'' t1')..(tn &'' tn')>

                    t &'' Empty := t
                    Empty &'' t := t
                    t &'' t'    := t & t'

     |||' :  (tableau_tree list) * (tableau_tree list) -> tableau_tree list


        <t1..tn> |||' <t1'..tn'> := <(t1 | t1')..(tn | tn')>
```

HIER NOCHEINMAL DER WICHTIGE HINWEIS. DIE LETZTEN BEIDEN REGELN ZUR VERKNUEPFUNG DER T-ANTEILE SIND NUR EINE ERSTE ANNAHME. SIE SETZT VORAUS, DASS DIE VERKNUEPFUNGEN IN DEN EINZELNEN SUBSYSTEMEN VOLLKOMMEN UNABHAENGIG VONEINANDER STATTFINDEN, UND BERUCKSICHTIGT AUCH KEINEN SYSTEMWECHSEL DER FORMELN IN DEN EINZELNEN SPALTEN.
An dieser Stelle haben wir alles zur Implementierung des Beweismechanismus noetige beisammen. Diese Informationen werden im Implementierungsteil aufgegriffen und weiter spezifiziert.
———- X-Sun-Data-Type: default X-Sun-Data-Description: default X-Sun-Data-Name: bwm.txt X-Sun-Content-Lines: 1775

2 Entwicklung eines Tableaubeweismechanismus für aussagenlogische Formeln der PKL

Im folgenden beschreibe ich die Entwicklung eines Beweismechanismus für aussagenlogische Formeln der PKL.
Dies umfasst den Entwurf

- einer Darstellung von Tableaubeweisbäumen ,
- einer Menge von Reduktionsregeln ,
- einer Menge von Expansionsregeln.

Die Reduktionsregeln überführen einen Beweisbaum in eine Normalform, die logische Interpretationen über die ursprüngliche Formel erlauben (z.B , daß es sich bei der Formel um eine Tautologie oder eine Kontradiktion handelt). Bei den Expansionsregeln geht es um die Darstellung von Tableauregeln. Hier müssen Formeln des Beweisbaumes in die entsprechende Baumdarstellung transformiert werden.
Dieses Kapitel gibt zunächst eine kleine Einführung in Tableaus und Tableaubeweisbäume im klassischen Sinne. Danach wird schrittweise eine Regelsatz für Reduktionen und Expansionen entwickelt. Diese werden danach unter dem Kontext der PKL erweitert.

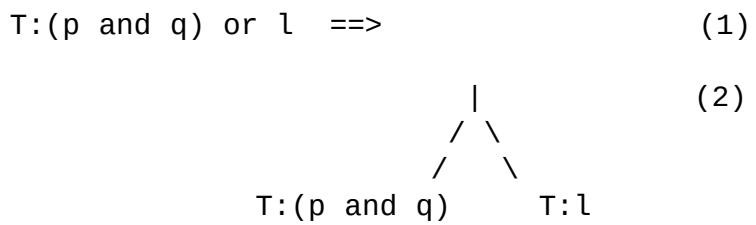
Der hier beschriebene Regelsatz ist ein erster Ansatz und ist noch nicht vollständig und korrekt . Er dient aber dazu ein erstes Verständnis für einen Reduktionsmechnismus zu gewinnen. Diese Erfahrungen bilden die Grundlagen für weitere Modifikationen und Verfeinerungen des Regelsatzes.

2.1 Tableau-Beweisbäume

Ein Tableaubeweisbaum im klassischen Sinne repräsentiert die disjunktiven und konjunktiven Verzweigungen eines Tableaubeweises. Zum besseren Verständnis gebe ich eine kurze Einführung in die Technik von Tableaubeweisen.
Beispiel: (Für einen klassischen Tableaubeweis)

```
T : (p and q) or l
```

Wir fragen hier für welche Belegungen der Variablen p und q die Gesammtformel den Wert T (true) annehmen kann. Der Beweis besteht aus der Zerlegung der Formel in seine Bestandteile oder besser Teilformeln.



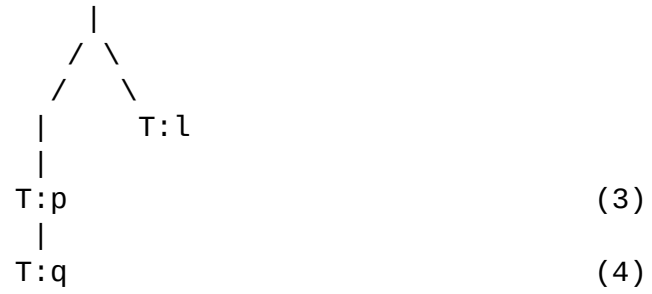
Die Verzweigung des Baumes stellt zwei mögliche Alternativen von Belegungen dar,d.h. die Formel kann den Wert true annehmen, wenn entweder die Teilformel (p and q) den Wert true annimmt oder die Teilformel l den Wert true annimmt. Die Formel (1) wird nicht mehr benoetigt da sie jetzt durch den entstandenen Baum repräsentiert wird.
Der Rechte Ast des Baumes zeigt eine atomare signierte Formel, die als Belegung der Variablen l interpretiert werden kann.

Definition 1: Unter einer signierten Formel versteht man einen Ausdruck von der Form

```
tv: fml
```

tv ist ein Element aus einer Wahrheitsmenge und fml ist Element einer Formelmenge.

Eine signierte Formel wie `"T:(p and q)"`, ist also eine Formel die mit einer Frage nach einem Wahrheitswert versehen ist.
Zurück zu unserem Beispiel. Der linke Ast des Baumes kann noch weiter abgebaut werden:

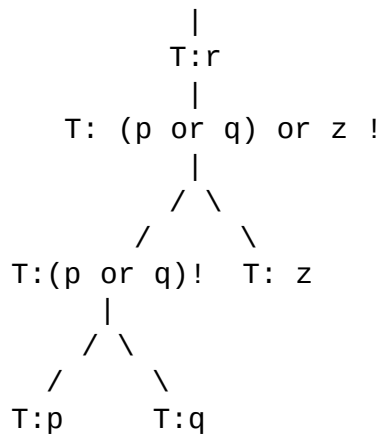


Der Abbau der Teilformel (2) führt zu einem konjunktiven Ast, d.h. die Bedingungen (3) und (4) müssen beide erfüllt werden, was leicht zu erkennen ist.
Die obige Formel nimmt den Wert T an wenn entweder p und q den Wert T annehmen, oder l gleich T, man erhält also zwei alternative Wertbelegungen der Variablen.
Allgemein ist jetzt zu sagen, das man alle Alternativen (disj.) Belegungen erhält , indem man die atomaren Formeln jedes Weges von der Wurzel des Baumes zu einem Blatt zusammenfast. Jede dieser Mengen bildet eine konjunktion von Belegungen.
Für das obige Beispiel erhält man die beiden Mengen:

```
M1 = {p=T,q=T}
M2 = {l=T}
```

Ein etwas größeres Beispiel:

```
T: r and ((p or q) or z)  ==>
```



Der Übersicht halber sind hier die Teilformeln der Formel an den Wurzeln der Teilbäume beibehalten worden (mit ! markiert). Hier erhält man die folgenden Mengen von alternativen Belegungen:

```
M1 = { r=T , p=T }
M2 = { r=T , z=T }
M3 = { r=T , q=T }
```

"r" ist in jeder Menge enthalten, da es auf jedem Weg von der Wurzel zu einem Blatt liegt.

2.2 Allgemeingültigkeit und Widersprüche

Ein wichtiges Ziel von Tableaubeweisen ist die Allgemeingültigkeit von Formeln zu zeigen. Dies wird im klassischen Fall dadurch bewältigt, indem man zeigt, daß die Frage nach dem Wert false nur Mengen von Belegungen liefert, die pro Menge mindestens einen Widerspruch aufweisen. D.h. das es ein Belegungspaar in jeder Menge gibt von der Form

```
{ ... , p=T , .... , p=F }
```

Eine Variable die beide Wahrheitswerte annehmen soll führt im klassischen Sinn zum Widerspruch. Beispiel:

```
F:p or (not p) !              (1)
      |
     F:p                      (2)
      |
  F:(not p) !                 (3)
      |
     T:p                      (4)
```

Dieser Baum enthält nur einen Ast. Die Menge von Belegungen die man erhält, enthält nur das folgende Element

```
M = {F:p,T:p}
```

Diese enthält einen Widerspruch, also ist unsere Formel eine Tautologie.
Ich fasse noch einmal kurz zusammen. Ein Beweis führt entweder zum Widerspruch, indem alle seine disjunktiven Aeste einen Widerspruch enthalten, oder er liefert eine Menge von alternativen Belegungen der Variablen. Die Evaluierung der Formel unter diesen Belegungen liefert dann jeweils den erfragten Wahrheitswert des Beweises.

Definition 2: Ein Tableaubeweisbaum heißt widersprüchlich im klassischen Sinne, wenn ALLE Mengen von alternativen Belegungen mindesten einen Belegungspaar der Form

```
p=T und p=F
```

für beliebige Variablen p enthält. Man sagt hier auch, der Beweisbaum schliesst.

2.3 Repräsentation eines Beweisbaumes

Zur Darstellung eines Baumes werde ich eine Notation wählen die die verschiedenen Verzweigungen eines Baumes und dessen Bestandteile kennzeichnet. Nennen wir den Typ eines Beweisbaumes tableau_tree.

- Man benoetigt die Moeglichkeit den leeren Baum darzustellen. Wir bezeichnen ihn mit

```
=> Empty
```

- Die elementaren Daten des Baumes sind signierte Formeln. Im Baum wird dieser durch den Typkonstruktor SF gekennzeichnet.

```
=> SF (truethvalue,formula)
```

  Man koennte auch sagen SF konstruiert mit seinen Bestandteilen eine signierte Formel.

- Als nächstes benoetigen wir die Darstellungen der konj. sowie disj. Verzweigungen. Wir gehen von unmittelbaren Darstellung durch den oben illustrierten Baum weg, und ziehen die Termdarstellung (oder Konstruktordarstellung) vor. Nachteil dieser Darstellung ist, daß sie nicht so anschaulich ist wie die gezeigten Bäume. Sie erlaubt aber eine kompaktere Handhabung und unter Zunahme der später entwickelten Regeln lässen sich mit dieser Darstellung algebraische Termumformungen durchführen.

  – Die konj. Verzweigung nennen wir Alpha und bezeichnen sie mit dem Konstruktor && . Die Komponennten sind natürlich vom Typ tableau_tree. Binäre Konstruktoren werden infix notiert.

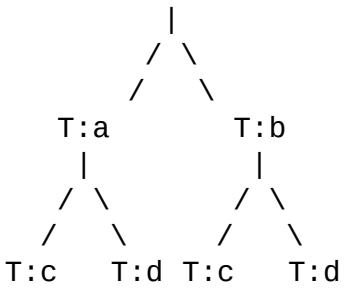
```
=>  tableau_tree && tableau_tree
```

  – die disj. Verzweigung nennen wir Beta und bezeichnen sie mit dem Konstruktor || .

```
=>  tableau_tree || tableau_tree
```

Ein Beispiel: (zunächst für die Baumdarstellung)

```
T: (a or b) and (c or d)  ==>
```



Die Expansion unserer Termdarstellung sieht folgendermaßen aus: (wir starten mit der signierten Formel)

```
SF(T , (a or b) and (c or d))    (1)

==> SF(T,(a or b))  &&  SF(T,(c or d))

==> (SF(T,a) || SF(T,b))  &&  (SF(T,c) || SF(T,d))
```

Offen ist jetzt wie man Belegungsmengen gelangt. Ein erster Anstatz ist, zunächst alle ||'s(Betas) auf oberste Ebene zu ziehen. Das heißt, daß wir einen Term der Form

```
A1 || A2 || ... || An
```

erhalten, wobei es kein Ai (i aus 1 . . . n) gibt das einen Teilterm der Form (t1 || t2) enthält. t1 und t2 sind vom Typ tableau_tree. Dies ist Vergleichbar mit der Umformung eines Ausdruckes in disjunktive Normalform. Man kann nun durch reine Termtransformationen zu alternativen Belegungen gelangen. Jetzt zunächst die Regeln, wie man die 'DNF' herstellt. Zur Abkürzung werden Signierte Formeln der Form SF(tf,fml) durch (tf:fml) dargestellt.

REGEL 0 :

```
1)  a1 && (a2 || a3)  ->  (a1 && a2) || (a1 && a3)
2)  (a1 || a2) && a3  ->  (a1 && a3) || (a2 && a3)
```

Anwendung auf das letzte Beispiel:

```
     ( T:a || T:b ) && ( T:c || T:d )

==>  (T:a && ( T:c || T:d ))  ||          (Regel 2)
     (T:b && ( T:c || T:d ))
```

```
==>  (T:a && T:c) ||                    (Regel 1)
     (T:a && T:d) ||
     (T:b && T:c) ||
     (T:b && T:d)
```

Betaklammerungen sind assoziativ und kommutativ und daher stehen die Alpha Terme alle in einer disjunktiven Beziehung. Die Belegungen sind also einfach nur die Argumente der Betas.
In diesem Fall:

```
M1 = { a=T , c=T }
M2 = { a=T , d=T }
M3 = { b=T , c=T }
M4 = { b=T , d=T }
```

2.4 Die Tableauregeln

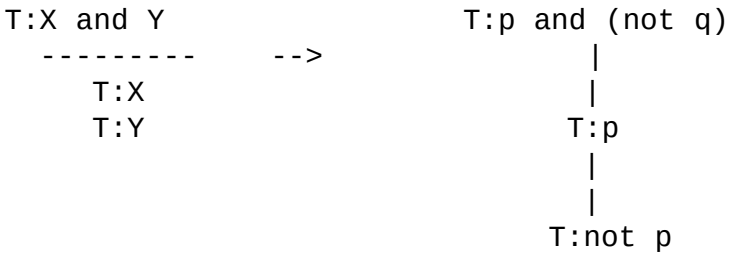
Bei den bisherigen Erläuterungen haben wir uns auf rein klassischem Boden bewegt. Um nun auf neuen Bestandteile hinzugelangen, wird zunächst ein Thema einfügt, das bei der Behandlung der Tableau Methode noch nicht erwähnt wurde. Hierbei handelt es sich um die Tableauregeln. Sie geben an welche Art der Expansion einer signierten Formel auszuführen ist. Im klassischen Fall gibt es für jeden Junktor zwei Regeln, d.h. für jeden Wahrheitswert eine.
Wir betrachten die Regeln für `'and'` und `'or'` und die Negation`'not'` :

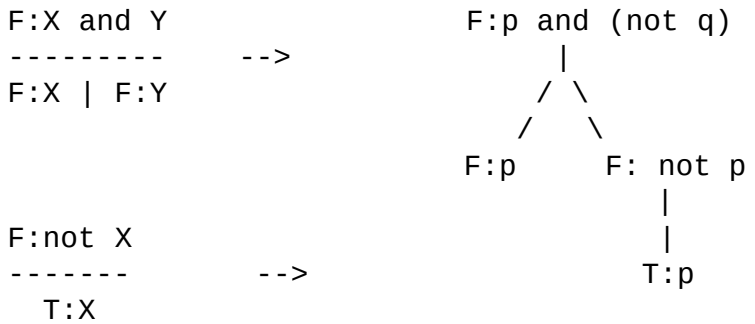
```
1.              T:X and Y         F:X and Y            (1)
                ---------         ---------
                   T:X           F:X | F:Y
                   T:Y

2.              T:X or Y          F:X or Y
                ---------         --------
                T:X | T:Y            F:X
                                     F:Y

3.              T:not X           F:not X
                --------          --------
                  F:X               T:X
```

Man liest die obigen Tableaus für T folgendermassen. Die Formel "X and Ynimmt genau dann den Wert T an, wenn sowohl die Teilformel X als auch die Teilformel Y den Wert T annehmen. Das Untereinanderschreiben hat bezeichnet das konjunktive Verhältnis zweier Teilformeln X und Y zueinander, Nebeneinanderschreiben das disjunktive Verhältnis. Nebeneinander und Untereinanderschreiben korrespondieren also zu den || und && Konstruktoren.
Die Regeln finden nun die folgenden Anwendung innerhalb eines Tableaubeweises:





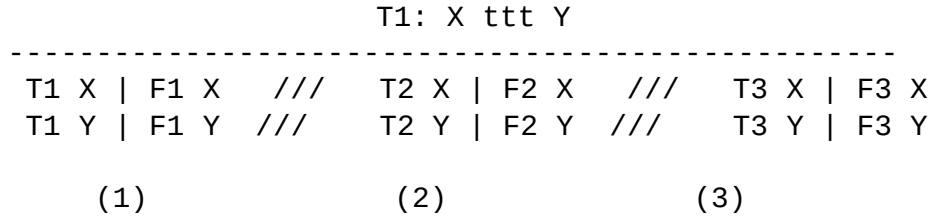
Dies ist im wesentlichen alles was für die weitere Entwicklung der Regeln unter dem Kontext der PKL benoetigt wird.

2.4.1 Übergang zur PKL

Die Tableaus der PKL sehen komplexer aus, und enthalten neben Alpha und Beta Beziehungen, die hier auch gemischt innerhalb der Regeln vorkommen, auch transjunktive Beziehungen, d.h. das es für eine Anfrage für einen Wert aus einem System es auch Belegungsmoeglichkeiten aus anderen Systemen geben kann.
Bei einem n-stelligen System kann es hier natürlich bis zu n-1 transjunktive Alternativen geben.

Definition 3: Als transjunktiven Anteile werden alle Wertangebote bezeichnet, die nicht aus dem gleichen System stammen wie der erfragte Wert.

Die transjunktiven Anteile werden im Tableau durch einen Dreifachstrich getrennt.In folgenden werde ich transj. Anteile auch häufiger mit T-Anteile bezeichnen. Hier nun ein Beispiel für die Transjunktion ttt mit dem Tableau für T1.



Die erste Alpha-Beta Komponente (1) bezeichne ich den ’lokalen Anteil’, da er im selben System bleibt. (2) und (3) sind die transjunktiven Anteile.
Lange Zeit hat uns die Frage beschäftigt, wie die Transjunktion im Tableau darzustellen ist. So stand zur Frage ob sie als herkoemmliche Disjunktion, also als Beta aufzufassen sei. Wir sind nun zu dem ersten Entschluss gekommen, das die Transjunktiven Anteile zunächst unabhängige Alternativen zum lokalen Anteil bilden, und zu den Belegungen des resultierenden Beweises des lokalen Anteils keine Beziehung haben.
Allerdings sind sie auch keine richtigen Betas , da sie ansonsten mit Teilen des Baumes eine Beziehung hätten die es eigentlich nicht gibt.
Wir führen hier eine explizite Darstellung für T-Anteile in Beweisbäumen, sowie einen erweiterten Regelsatz ein.
Beispiel für einen Ausdruck mit Transjunktion

```
H = (p ttt q) aaa (p ooo q)

    ( aaa ist hier die dreistellige Konjunktion und
      ooo die dreistellige Disjunktion )

                  t1:H
                    |
               t1:(p ttt q)
                    |
               t1:(p ooo q)
```

Teilformeln werden von nun in der Art expandiert indem sie an Ort und Stelle durch ihre regelbestimmte Alpha-Beta Struktur ersetzt werden.
Wir erhalten zunächst

```
          T1:(p ttt q)
                |
           T1:(p ooo q)          (*)

==>
          T1:(p ttt q)
                |  (!)
               / \
              /   \
          T1:p    T1:q            (DTB)
```

`(!)` ist die Anwendung der Tableauregel für (*).
Durch die verschachtelte Alpha-Beta Struktur müssen für ttt mehrere Expansionsschritte gleichzeitig vorgenommen werden. Der Übersichtlichkeit halber wird jedes weitere Vorkommen des Disjunktiven Teilbaumes mit `(!)` abgekürzt. Die transjunktiven Anteile vermerke ich an dem Knoten der Regelanwendung. Der Grund hierfür wird später hoffentlich klar.

```
 ==>
                L(1)                  T(2)                  T(3)
                  ____________________________________________
                 /\                    /\                    /\
                /  \                  /  \                  /  \
           T1:p      F1:p        T2:p      F2:p      T3:p      F3:p
             |         |           |         |         |         |
           T1:q      F1:q        T2:q      F2:q      T3:q      F3:q
             |         |
            (!)       (!)
            / \       / \
       T1:p T1:q T1:p T1:q
```

Hier wäre der Beweis abgeschlossen, und neben den Belegungen des eigenen Systems sind hier noch zusätzliche Belegungen aus den anderen Systemen. Der Widerspruchbegriff muss hier natürlich daraufhin erweitert werden das auch alle alternativen transj. Äste schließen müssen.
Ich werde jetzt zunächst auf unsere Darstellung der transj. Anteile in unserer Baum(Term)darstellung eingehen, und dann letzteres Beispiel nochmal unter leicht modifiziertem Gesichtspunkt aufgreifen.

2.4.2 Repräsentation der T-Anteile

Wir benoetigen eine Struktur, die die T-Anteile an gegebener Stelle festhällt bzw kennzeichnet. Sie enthält zum einen den weiteren lokalen Teilbaum sowie eine Liste der T-Anteile, die ja selbst auch wieder Bäume sind. Wir nennen den Konstruktor Trans und bezeichnen ihn mit `///` .

```
=>  tableau_tree /// tableau_tree list
```

Ich greife das letzte Beispiel nocheinmal auf , und bearbeite es unter dieser Darstellung. Hierzu gebe ich das Tableau für die Transjunktion nochmal in leicht modifizierter Darstellung an.
Die Liste t1,t2,. . .,tn von T-Anteilen wird durch <t1,. . .,tn> abgekürzt.
Beispiel:

```
        H = (p ttt q) aaa (p ooo q)

        T1: H

==>     T1: (p ttt q) && T1:(p ooo q)

==>     T1: (p ttt q) && (T1:p || T1:q)

==>     (1 /// <2,3>) && (T1:p || T1:q)
```

1,2 und 3 bezeichnen hier die drei Anteile in der Tableauregel für ttt. Die Reduktion gerät hier ins stoppen, da wir hier keine Tableau Regel mehr anwenden koennen, aber auch keine Regel haben, die Betas auf oberste Ebene transportieren kann.
Wir hatten nun angenommen, dass die T-Anteile unabhängig von Rest des Baumes seien.Gesucht sind nun Regeln, die den Baum in die folgende Form bringen:

```
dnf /// <t1..tn>
```

`"dnf"` ist der lokale Baum in disjunktiver Normalform.Er enthält selbst keine Äste die T-Anteile enthalten.
Eine mögliche Regel für das obige Beispiel wäre :

```
(a /// list) && b   ==>   (a && b ) /// list
```

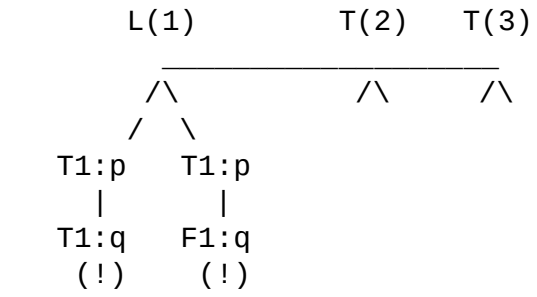
Es ist nun Aufgabe Regeln zu finden, die auch die Beziehungen zwischen T-Anteilen aus verschiedenen Aesten mit einbeziehen. Hierzu betrachten wir ein geeignetes Beispiel.

```
H = (p ttt q) aaa (p ttt r)
```

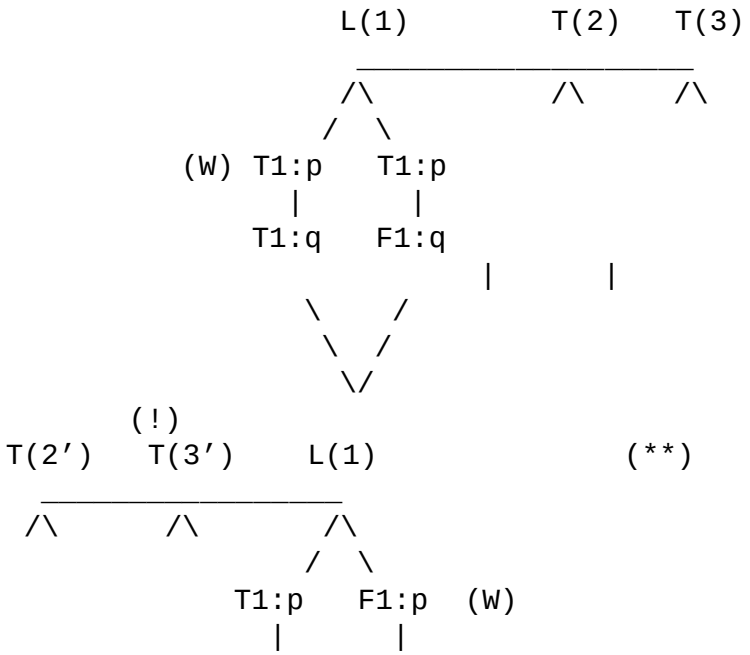
Wir betrachten die ersten Expansionsschritte:

```
T1:(p ttt q)       (1)
     |
T1:(p ttt r)       (2)
```

Expandiert wird nun zunächst Teilbaum (1). Vorkommen von Teilbaum (2) stelle ich wie schon vorhin abkekürzt durch (!) dar. Auch die T-Anteile kennzeichne ich nur noch durch abstrakte Bezeichner wie z.B. T(2). Es ergibt sich der folgende Baum:



Durch Expandierung von Teilbaum (2) kommen wir zu der folgenden Darstellung:



T1:r F1:r

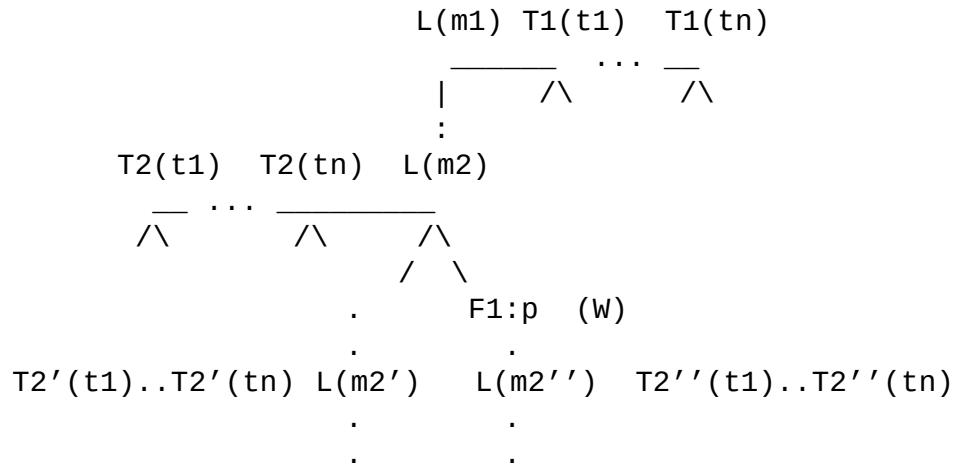
Nach unseren bisherigen Annahmen sind die T-Anteile T(2),T(2'), T(3) und T(3') völlig unabhängig voneinander. Für die T-Anteile T(2) und T(3) sollte dies auch der Fall sein, da wir hier gesagt hatten, das dieses Alternativen aus verschiedenen Systemen sind. Ebenso bei T(2') und T(3').
Betrachtet man aber die Beziehung zwischen T(2) und T(2') sowie zwischen T(3) und T(3') so stellen wir fest, das sie im Baum in einer Art Alpha Beziehung stehen. Dies ist auch nicht verwunderlich, da diese Beziehung ja aus den Termen (1) und (2) entstanden ist die ja auch in dieser Beziehung zueinander standen.
Es liegt nicht fern, daß es berechtigt ist anzunehmen, daß T-Anteile die im Baum einer konjunktiven Beziehung unterliegen diese auch in den einzelnen Subsystemen aufweisen. Hierbei ist darauf zu achten das dies nur für gleiche Subsysteme gilt. In Bezug auf das obige Beispiel heißt das, daß T(2) und T(2') in einer konjunktiven Beziehung stehen. Unter verschiedenen Subsystemen besteht nach wie vor keine Beziehung.

BEMERKUNG: Es ist wichtig an dieser stelle zu sagen, daß die letzten Annahmen auf reiner Intuition beruhen, und an dieser Stelle noch durch keine analytischen Beobachtungen begründet sind !!!

Wir müssen hier zunächst abwarten ob die weitere Arbeit unter diesen Annahmen zu schlüssigen Ergebnissen gelangt Wir hoffen das aber auch schon der Aufbau des formalen Gerüsts unter diesen Bedingungen weitere Einblicke gewährt.

Unklar sind in diesem Kontext noch die Beziehungen der verschiedenen Systeme eines T-Anteils zueinander, die hier zunächst einmal als unabhängig voneinander angenommen werden.

Der Aufmerksame Leser wird gemerkt haben, das in dem obigen Beispiel die T-Anteile T(1'),T(2') in zwei alternativen Aesten auftauchen, und sich warscheinlich fragen in welcher Beziehung denn diese nun stehen moegen. Hier sei nun zunächst nur gesagt, dass diese scheinbare Beziehung in diesem Fall nur eine Pseudo-Beziehung ist. Sie ergibt sich aus der disjunktiven Aufspaltung, wodurch sich zwei redundante Alternativen der T-Anteile ergeben. Diese redundante Darstellung ist auf die Darstellung durch den Baum zurückzuführen. Wir werden sehen, dass dies in der Termdarstellung nicht mehr der Fall sein wird.
Es gibt aber tatsächlich auch disjunktive Beziehung zwischen den transjunktiven Anteilen. Ich moechte hierfür ein Beispiel konstruieren. Nehmen wir an wir hätten schon folgenden Beweisbaum vorliegen:



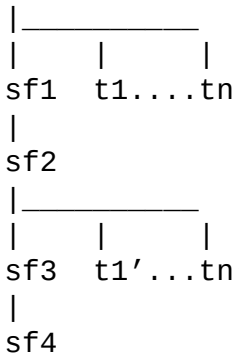
Nehmen wir weiterhin an, das alle Teilbaume unterhalb von L(m2) nur durch Abbau des Teilterms der L(m2) hervorgebracht hat entstanden sind. Dann sollten auch die T-Anteile T2'(t1) und T2"(t1) keine trivialen Redundanzen wie oben aufweisen, was hier heist, das sie also in einer nichttrivialen disjunktiven Beziehung stehen. Es ergibt sich die folgende Struktur der T-Anteile, unter der Annahme das diese die einzigen im obigen Baum sind.
Für alle i aus $\{1 \ldots n\}$ gilt :

```
T1(ti) && (T2(ti) && (T2'(ti) || T2''(ti)) )
                     ========================
```

Man erkennt hier, das die T-Anteile also ihre Struktur im Baum beibehalten, was sie also den gleichen Regeln unterwirft, wie wir sie schon bei den herkoemmlichen Alpha-Beta Strukturen kennengelehrnt haben.
Der Einfachheit halber wollen wir annehmen, das der T-Anteil immer alle Subsysteme umfasst, sodas sich die obigen Beziehungen gleichfoermig für alle T-Anteile konstruieren lassen. Natürlich ist diese Forderung durch die Tableauregeln nicht zu halten. Wir behelfen uns hier dadurch , das alle nicht vorkommenden Subsysteme im T-Anteil durch die leere Alpha-Beta Struktur Empty dargestellt werden. Diese leere Struktur bildet bezüglich dem Alpha und Beta Operatoren (wenn man sie Semantisch interpretiert) ein Neutrales Element, sodass gilt:

```
a && Empty  = a
Empty && a  = a
a || Empty  = a
Empty || a  = a
```

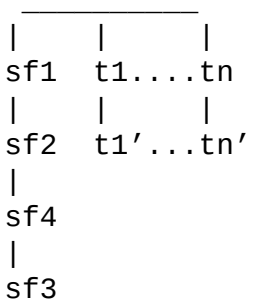
Wir wollen uns nun den Regeln des || Operators zuwenden. Betrachten wir zunächst den folgenden Baum sowie dessen Termrepräsentation:



Die sf's bezeichnen signierte Formeln , die t(a)'s die T-Anteile. In der Termschreibweise sieht der Baum folgendermassen aus:

```
( (sf1 /// <t1..tn>)    &&  sf2 )  &&
( (sf3 /// <t1'..tn'>)  &&  sf4 )
```

Selbst wenn eine der sfn's zu einer beta-Struktur expandiert würde, blieben die Alpha Beziehungen der T-Anteile bestehen.Aus diesen Betrachtungen koennen wir jetzt den Schluss ziehen, dass T-Anteile aus geschachtelten Alpha Termen extrahiert werden koennen. Das Resultat ist dann ein reiner Alpha Term sowie die konjunktiv verknüpften T-Anteile der einzelnen Subsysteme. Für den obigen Baum sieht das folgendermassen aus:



Das Resultat der Termnotation ist

```
(sf1 && sf2 && sf3 && sf4) /// <(t1 && t1')..(tn && tn')>
```

Wir sind nun in der Lage erste Regeln für den && Operator aufzustellen.

REGEL 1 :

Beide Teilterme des && Op's enthalten Transjunktive Anteile

```
(t /// ta) && (t' /// ta')  ==>  (t && t') /// (ta &&&' ta')
```

REGEL 2 :

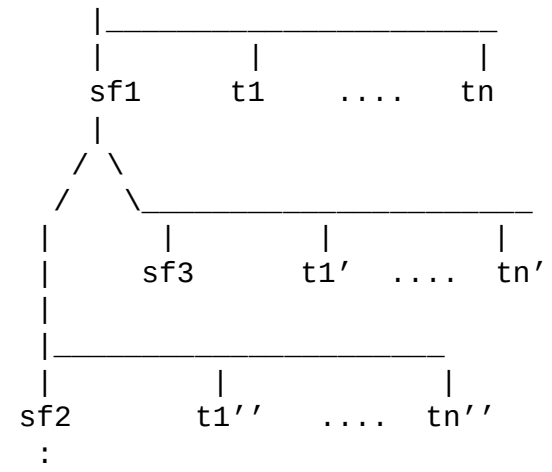
Nur ein Term enthalt einen T-Anteil

```
a)  t && (t' /// ta')  ==>  (t && t') /// ta'
b)  (t /// ta) && t'   ==>  (t && t') /// ta

    mit   &&&':  <t1..tn> &&&' <t1'..tn'> := <(t1 && t1')..(tn && tn')>
```

BEMERKUNG: Auch die Regeln unter 2a und 2b bedürfen noch einer genaueren Untersuchung, ob das Fehlen eines T-Anteils in einem der beiden Aeste nicht irgendwelche Auswirkungen auf den anderen T-Anteil hat.

Man ist nun versucht für den || Operator eine ähnliche Betrachtung durchzuführen. Dazu wieder ein Beispiel:

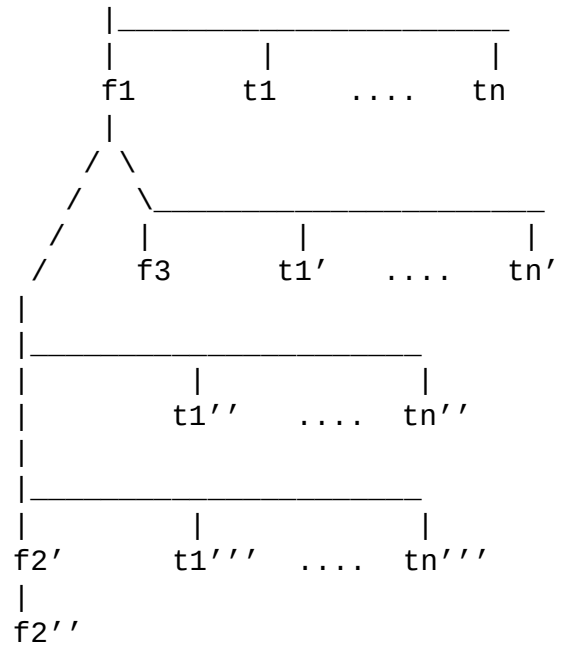


Wir wollen der Uebersicht halber unsere Betrachtungen jetzt nur auf die T-Anteile aus System I beschränken. Im ersten Moment koennte man vermuten, dass folgende Beziehung besteht:

```
sf1 & (sf2 | sf3)  ||  ((t1 & t1') | (t1 & t1'')) ....
```

Das ist auch nicht verkehrt, nur lässt sich das in diesem Fall nicht als allgemeine Regel fassen. Das Problem ist, das wir nicht wissen was die weitere Entwicklung von sf2 und sf3 noch hervorbringt.

Betrachten wir ein Beispiel für den Fall, das sf3 bereits atomar ist, also keiner weiteren Expandierung mehr unterliegt. sf2 wird nun folgendermassen expandiert:



Nach der obigen Annahme ergibt sich jetzt:

```
f1 && ((f2' && f2'') ||
       f3 /// <((t1 && t1') || (t1 && t1'')) && t1''' , ....
                              ---------------------------------
```

Für den unterstrichenen Teil ergibt sich die folgende Beziehung des T-Anteils im ersten System

```
((t1 & t1') & t1''') | ((t1 & t1'') & t1''')
                -------------------
```

t1"' steht jetzt fälschlicherweise auch mit t1' in einer Alpha Beziehung. In diese Lage sind wir nun geraten, weil wir zu früh T-Anteile aus einer Beta-Beziehung zusammengefügt haben.
Resultat aus diesen Betrachtungen ist, das die T-Anteile aus Beta-Zweigen erst zusammengefügt werden koennen, wenn gewährleistet ist, das der lokale Anteil des || Operators keine weiteren ///-Terme mehr enthält, und jede weitere Expandierung keine weiteren ///-Terme mehr hervorbringt. Die zweite Bedingung ist sehr schwer vorauszusehen, und daher verschärfen wir unsere Bedingung noch:
Gegeben ist ein Term

```
(t /// ta) || (t' /// ta')
```

Die T-Anteile dürfen erst dann in Beziehung gesetzt werden,wenn gewähr- leistet ist, das die Terme t und t' keine komplexen signierten Formeln mehr enthalten, und das t und t' keine Unterterme der Form (tt /// tta) enthalten.
Wir sehen das Teilterme einer Beta-Beziehung im Gegensatz zur Alpha Struktur komplett expandiert werden müssen. Es steht jetzt die Frage offen wie wir diese Eigenschafft eines Terms propagieren wollen, ohne einen Term auf diese Eigenschafft zu Testen. In anderen Worten, wie kann man mit einfacheren Eigenschaften ,wenigen Regeln sowie einem neuen Operator einen Term als ///-frei kennzeichnen.

2.4.3 T-Anteil freie Terme

Gehen wir schrittweise vor, und betrachten wir zunächst, wie wir einen solchen Term kennzeichnen:

```
{t}  :=   t ist ///-frei
```

Der kleinste Term von dem wir definitiv wissen das er `///` frei ist, ist die atomare signierte Formel , also

1. t ist atomare signierte Formel:

   ```
   =>  {t}
   ```

   sonst expandiere sf unter Anwendung des zugehoerigen Tableaus.

2. t und t' sind `///` frei :

   ```
   => {t} && {t'} -> {t && t'}
   ```

3. t und t' sind `///` frei :

   ```
   => {t} || {t'} -> {t || t'}
   ```

Wir sind nun in der Lage die Eigenschaft `///` frei zu Propagieren. Was uns noch fehlt sind die Regeln zum liften der T-Anteile aus Beta-Zweigen.

REGEL 3 :

Beide Teilterme des || Operators enthalten Transjunktive Anteile :

```
({t} /// ta) || ({t'} /// ta')  ==>  ({t || t'}) /// (ta |||' ta')
```

REGEL 4 :

Nur ein Term enthalt einen T-Anteil :

```
a)  {t} || ({t'} /// ta')  ->  {t || t'} /// ta'
b)  ({t} /// ta) || {t'}   ->  {t || t'} /// ta
```

mit `<t1..tn> |||' <t1'..tn'> := <(t1 || t1')..(tn || tn')>`

2.4.4 Geschachtelte /// - Terme

Was jetzt noch zu tun bleibt ist, wie man mit Termen der Form

```
(t /// ta) /// ta'
```

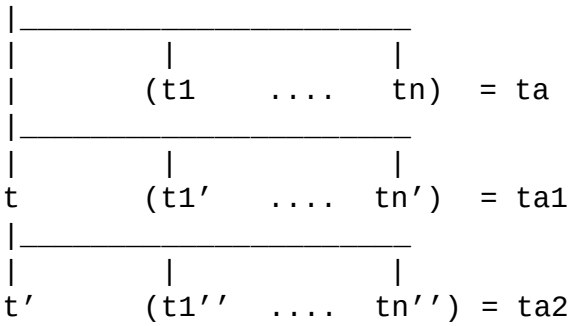
umzugehen hat. Zunächst analysieren wir wie diese Terme überhaupt entstehen. Wir gehen von der folgenden Situation aus:

```
(sf && sf') || ta
```

Nehmen wir an sf und sf' sin signierte Formrln die auf folgende Weise expandiert werden:

```
((t /// ta1) && (t' /// ta2)) /// ta
```

Das entspricht der folgenden Baumdarstellung



Wir wenden nun Regel (1) an und erhalten

```
(*)   ((t && t') /// (ta1 &&&' ta2)) /// ta
```

Aus der Baumdarstellung ist ersichtlich, das ta, ta1 und ta2 in konjunktiver Beziehung stehen

```
(*) -> (t && t') /// (ta1 &&&' ta2) &&&' ta
```

Wir wollen nun noch ein Beispiel für die disjunktive Beziehung betrachten

```
(sf || sf') /// ta
```

Wie schon oben nehmen wir an sf und sf' signierte Formeln sind,die auf folgende Weise expandiert werden:

```
      (({t} /// ta1) || ({t'} /// ta2)) /// ta
==>   (({t} || {t'}) /// (ta1 |||' ta2)) /// ta
```

Wir wissen das ta zu ta1 und zu ta2 in konjunktiver Beziehung steht, das aber beide Aeste disjunktive Alternativen repräsentieren. Sehen wir was passiert wenn wir in der gleichen Weise verfahren wie schon im letzten Beispiel:

```
      (([t] || [t']) /// (ta1 |||' ta2)) /// ta

==>   ([t] || [t']) /// (ta1 |||' ta2)) &&&' ta

==>   ([t] || [t']) /// (ta1 &&&' ta) |||' (ta2 &' ta)
```

Das ist genau das Resultat das wir uns vorgestellt haben. Wir koenen die Regel nun allgemein formulieren.

Regel 5: `(t /// ta) /// ta'  ->  t /// (ta &&&' ta')`

Wir haben jetzt alles was wir zur Darstellung der Beweisbäume benoetigen, und wissen wie wir sie in eine Art Normalform umformen kann aus der man dann die Belegungsinformation erhalten kann. Was wir noch nicht besprochen haben ist wie der Widerspruchsbegriffin unsere Darstellung einfliesst. Ich werde nun zunächst die bisher erarbeiteten Ergebnisse zusammenfassen, und dann zeigen, das dies mit den jetzigen Mitteln moeglich ist. Ich werde aber auch einen weiterreichenden Gedanken aufnehmen, der zu einer anderen Loesung führt, in der auch die `///` freien Terme einbezogen werden. Wir werden sehen das hier nur ein sehr kleiner Teil der entwickelten Regeln modifiziert werden muss, und das wir hierdurch zu einer sehr elaganten Darstellung der schon erwähnten DNF und des Widerspruchbegrifs gelangen.
Sie wird später im Baum durch den Konstruktor DNF gekennzeichnet und stellt im eine äquivalente Repräsentation der `///` freien Terme dar. Diese Information war noetig um gleich unsere Datenstruktur komplett aufzubauen.

2.4.5 Zusammenfassung der Regeln

Ein Beweisbaum lässt sich in der folgenden Datenstruktur zusammenfassen

```
  tableau_tree :=  Der leere Baum    => Empty
                |  Signierte Formel  => SF (truval,formel)
                |  Alpha Beziehung   => tableau_tree && tableau_tree
        |  Beta Beziehung    => tableau_tree || tableau_tree
                |  T-Anteile         => tableau_tree /// tableau_tree list
|  Dnf's (/// frei)  => DNF of dnf
```

Hier die Zusammenfassung der Regeln:

Regel 0 : Zuerst die Regeln zur Herstellung der DNF

```
          a)  a1 && (a2 || a3)  ->  (a1 && a2) || (a1 && a3)
          b)  (a1 || a2) && a3  ->  (a1 && a3) || (a2 && a3)
```

Regel I : Die Regeln für die alpha Beziehung von Termen mit T-Anteilen

```
    a) (t /// ta) && (t' /// ta')  ->  (t && t') /// (ta &&&' ta')
          b)  t && (t' // ta')           ->  (t && t') ///  ta'
          c)  (t /// ta) && t'           ->  (t && t') ///  ta

          <t1..tn> &&&' <t1'..tn'> := <(t1 && t1')..(tn && tn')>
```

Regel II : Die Regeln für die Beta Beziehung von Termen mit T-Anteilen

```
    a) ([t] /// ta) || ([t'] /// ta')  ->  [t || t'] /// (ta |||' ta')
          b) [t] || ([t'] /// ta')           ->  [t || t'] ///  ta'
          c) ([t] /// ta) || [t']            ->  [t || t'] ///  ta

          <t1..tn> |||' <t1'..tn'> := <(t1 | t1')..(tn | tn')>
```

Regel III: Die Regel für verschachtelte /// Terme

```
         (t /// ta) /// ta'  ->  t || (ta &&&' ta')
```

Regel IV : Die Regeln zur propagierung der /// freien Terme

```
a)  t ist atomare signierte Formel

    =>  [t]

        sonst  expandiere sf unter Anwendung des
        zugehoerigen Tableaus.

b)  t und t' sind /// frei

    => [t] && [t'] -> [t && t']

c)  t und t' sind /// frei

    => [t] || [t'] -> [t || t']
```

Die Regel 1a) kann man auch weglassen, da sie sich aus 1b) und 1c) und 3) herleiten lässt. Dieses gilt nicht für die Regel 2a) !!
Die Regeln für die Propagierung der /// freien Terme sind bewusst nicht auf die Regeln 1) und 2) verteilt worden. Dieser Teil wird den schon oben angesprochenen Modifikationen unterworfen. Hieraus ergibt sich sogar, das die Regeln unter 0. über sind. Dies wird im Folgenden unter dem Kontext des Widerspruchbegriffs erarbeitet.

2.5 Der Widerspruchbegriff

Nehmen wir erst nochmal den Widerspruchbegriff des klassischen Falls. Wir gehen von der DNF des Beweisbaumes aus, auf denen alle Beta's auf die oberste Ebene geliftet wurden. D.h. dass es keine konjunktiven Zweige mehr gibt, die disjunktive Verzweigungen aufweisen. Die konjunktiven Strukturen der verschiedenen beta-Zweige bilden die verschiedenen Belegungsalternativen. Eine Belegungsalternative heisst im klassischen Sinn geschlossen, wenn einer Variablen zwei unterschiedliche Wahrheitswerte zugewiesen werden.

```
A = a1 && a2 && .. && an
```

Definition A heißt geschlossen wenn ai und aj existieren, so daß
`ai = { T x }  und  aj = { F x }`

Der **Beweis** heißt geschlossen wenn alle Alternativen schliessen.
Wie kënnen wir nun den Begriff des schliessens propagieren ? Hier greifen wir auf die leere Alpha Menge zurück, und sagen, dass wenn eine Alternative schliesst sie gleich dieser leeren Menge ist ist. Da die Menge von Alternativen Alpha Mengen sind sagen wir, dass eine Beta Menge schliesst, wenn alle ihre Alpha Mengen geschlossen sind, d.h. also wenn sie leer ist.

```
{} := die leere Beta Menge
[] := die leere Alpha Menge
```

Die Regeln für die Handhabung mit leeren Mengen

```
=>  a) t || {} = t
    b) {} || t = t
    c) t && {} = {}
    d) {} && t = {}
```

Ein Beweis führt zum Widerspruch oder ist geschlossen, wenn er sich zu reduziert. Für den transklassischen Fall müssen wir den Widerspruchbegriff modifizieren.
Widersprüche ergen sich nur für Wahrheitswerte aus gleichen Systemen. Eine alternative schliesst also nicht schon bei verschiedenen Wertbelegungen derselben Variable, sondern erst wenn diese verschiedenen Wertbelegungen aus dem gleichen System stammen.
Für das Schliessen eines Beweises müssen wir nun zwei Fälle betrachten:
Das Resultat des Beweises ist

```
I.   {}                 => der Beweis schliesst
II.  {} /// <t1 .. tn> => der Beweis schliesst wenn auch alle
                           t1 .. tn schliessen
```

Wir sind also in der Lage den Begirff des Widerspruchs und der Belegungen auch im transklassischen Fall mit den jetzigen Mitteln zu formulieren.
Was hier allerdings stillschweigend mit eingeflossen ist, das irgendwie entschieden werden muss wann denn ein Beweisbaum (oder Term) in DNF ist, und wann denn zwei widersprüchliche Hypothesen in Alternativen vorliegen, sodass sich leere Mengen ergeben. Das erste Problem ergibt sich dadurch, das der Baum in DNF ist, wenn keine der Regeln mehr anwendbar ist. Der zweite Punkt ist leicht durch

das gegenseitige vergleichen der atomaren S-Formeln einer Alternative zu loesen. Das ist zwar sehr aufwendig, aber in diesem Fall unumgänglich.
Bevor ich zu einer alternativen Darstellung der DNF übergehe, will ich einen noch sehr wage angesprochenen Punkt aufgreifen. Hiermit meine ich die Expandierung von signierten Formeln. Dieser Begriff tauchte bislang häufig auf, ohne das näher auf ihn eingegangen wurde.
Zur Erinnerung :

```
(*)   t ist atomare signierte Formel SF(tv,var)   =>  [(tv,var)]
     (**)  sonst  => expandiere sf unter Anwendung des zugehoerigen Tableaus.
```

In (**) ist eine solche Regel schon informel enthalten. Wünschenswert wäre allerdings eine Regel in Form der bisherigen Termersetzung. Da es sehr mühselig ist, alle Tableauregeln als Regeln zu formulieren, werden wir diese Regeln implizit in einer Funktion verstecken.

REGEL VI :

```
SF(sf) -> expand(sf)

expand SF(tv,op(args)) = tableau_tree(op,tv,args)
expand SF(tv,var)      = [(tv,var)]
```

2.6 Die Darstellung der DNF

Wie schon gesehen werden die Belegungen erst in der DNF fassbar. Ein grosser Nachteil dieser Darstellung ist die Verschachtelung vieler Beta's und Alpha's. Natürlich werden auch viele Belegungen erzeugt die Widersprüche enthalten, die durch die Regeln unter (0) sogar noch dupliziert werden.
Die Idee ist nun schon von den Blättern des Baumes her DNF's zu erzeugen, und Regeln anzugeben wie diese DNF's disjunktiv bzw. konjunktiv zu verknüpfen sind. Die Regeln unter (4) werden der Kern der Betrachtungen sein

```
IV a)  SF(sf) -> expand(sf)
   b)  t und t' sind /// frei   ==> [t] && [t'] -> [t && t']
   c)  t und t' sind /// frei   ==> [t] || [t'] -> [t || t']
```

Aus den Mengenbetrachtungen hatten wir gesehen das die DNF eine Menge von Alpha Mengen ist, deren Gesammtheit wir mit der Beta Menge bezeichnet hatten. Wenn nun eine atomare sigierte Formel expandiert wird so erhalten wir die einelementige Beta Menge. Diese besteht aus der Alpha Menge die einzig und allein aus der atomaren signierten Formel besteht
Beispiel: expand SF(T1,Var X) => [(T1,Var X)]
Es müssen nun konjunktive und disjunktive Verknüpfungen auf diesen Mengen definiert werden werden, die zwei DNF's im Sinne der Regeln unter 0. zusammen- fügen.

2.6.1 Modifikation der Regeln

```
IV'  a)  SF(sf) -> expand(sf)
     b)  DNF(d) && DNF(d') -> DNF(t |--| t')
     c)  DNF(d) || DNF(d') -> DNF(t |++|  t')
```

Die Information der `///` freien Terme wird jetzt mit Hilfe der DNF propagiert. Das bedeutet eine Modifikation der Terminologie der Regeln unter (2).

```
==> 2'a) (DNF(t)///ta)||(DNF(t')///ta') => DNF(t |++| t')///(ta |||' ta')

        b) DNF(t) || (DNF(t')///ta')  => DNF(t |++| t')///ta'

        c) (DNF(t)///ta) || (DNF(t')  => DNF(t |++| t')///ta
```

Was jetzt noch fehlt ist die Spezifikation von `|++|` und `|--|`.

2.6.2 Die konj. verknüpfung von Dnf's $|--|$

Die Funktion `|--|` ist vom folgenden Typ

```
|--|: dnf*dnf -> dnf
```

Wir werden nun ihre Operationalität erarbeiten. Nehmen wir die beiden folgenden DNF's:

```
dnf1 :=  { [(T1,p),(T1,q)] , [(F1,p),(F1,q)] }

dnf2 :=  { [(T1,p)] , [(F1,q)] }
```

Nehmen wir weiterhin an, die beiden DNF's stehen in einer konjunktiven Beziehung. Dies entspreche in der Baumdarstellung einer alpha Beziehung von beta-Beziehungen.

```
dnf1 && dnf2
```

Daraus ergibt sich das jedes Alpha Menge aus dnf1 ,mit jeder Menge aus dnf2 in einer alpha-Beziehung steht. Dies besagt auch die folgende Aussagenlogische Formel

```
   (a or b) and (c or d)
==> ((a and c) or (a or d) or (b and c) or (b and d))
```

In gleicher Weise werden wir die beiden DNF's zusammenfügen.

```
=>  {
      [T1 p , T1 q , T1 p] ,   (1)
      [T1 p , T1 q , F1 q] ,   (2)
      [F1 p , F1 q , T1 p] ,   (3)
      [F1 p , F1 q , F1 p]     (4)
    }
```

Hier sind einige Besonderheiten zu bemerken. Zum einen sind in einigen konjunktiven Aesten ((1) + (2))doppelte atomare Formeln. Diese koennen eliminiert werden. A und A bleibt halt A. Der nächste Punkt sind die konjunktiven Aeste (3) und (4). Diese enthalten Widersprüchliche Hypothesen und Reduzieren sich daher zur leeren Menge, sodass sie eliminiert werden koennen. Diese Mengen korrespondieren zu den Widersprüchlichen Aesten im Beweisbaum.
Die obige DNF sieht jetzt folgendermassen aus =>

```
{  [T1 p , T1 q ] ,  [F1 p , F1 q ]  }
```

Wir wollen jetzt die Operationalität allgemeiner erfassen. Gegeben sind die beiden DNF's

```
dnf1 := { [ a1 ]  , .. , [ an ] }
```

```
dnf2 := { [ b1 ]  , .. , [ bm ] }


dnf1 |--| dnf2 :=
     reduce( {
              [a1]@[b1] , .. , [a1]@[b1] ,
                        ......           ,
              [an]@[bm] , .. , [an]@[bm]
              } )
{} |--| dnf = {}
dnf |--| {} = {}
```

Der Operator @ hat die beiden folgenden Aufgaben:

- Vereinigen der beiden Mengen unter Eliminierung der doppelten Hypothesen
- Das entdecken von Widersprüchen, worauf sich `[a]@[b]` zu [] reduziert. Diese leeren Mengen koennen in der DNF weggelassen werden. Die leere DNF stellt einen Widerspruch dar.

Die Funktion eliminiert redundante Alpha Mengen.

2.6.3 Die disj. Verknüpfung `|++|`

Wie schon bei join werden wir hier ihre operationalität erarbeiten. Nehmen wir wiederum die beiden folgenden DNF's

```
dnf1 :=  { [(T1,p),(T1,q)] , [(F1,p),(F1,q)] }
dnf2 :=  { [(T1,p)] , [(F1,q)] }
```

und nehmen wir jetzt an, die beiden DNF's stünden in einer disjunktiven Beziehung. Dies entspreche in der Baumdarstellung einer beta-Beziehung von beta-Beziehungen. Dies wäre aber nichts anderes als eine Alternative von Alternativen

```
(a or b) or (c or d) =>  a or b or c or d

{ [T1 p,T1 q] , [F1 p, F1 q] , [T1 p] , [F1 q] }
```

Dies entspricht also nur der Vereinigung zweier Mengen. Das ist auch schon alles was app zu leisten hat. Allgemein:

```
dnf1 := { [ a1 ]  , .. , [ an ] }
dnf2 := { [ b1 ]  , .. , [ bm ] }

dnf1 |++| dnf2 := reduce( {[a1],..,[an],[b1],..,[bm]} )
{} |++| dnf = {}
dnf |++| {} = {}
```

2.6.4 Die beiden Operatoren &&&' und |||'

Zur Spezifikation der Termrsetzungsregeln hatten wir die Operatoren &&&' und |||' verwendet, sie aber wage beschrieben. Sie dienen der disj bzw. konj. Verknüpfung der T-Anteile unserer Termdarstellung. Wir hatten gesagt, dass die T-Anteile alle Subsysteme beinhalte, und deren Verknüpfung das Resultat der jeweiligen Verknüpfung der Subsysteme ist. Subsysteme die in den Tableau-Regeln nicht enthalten sind werden durch den leeren Baum repräsentiert.

```
&&&' :  (tableau_tree list) * (tableau_tree list) -> tableau_tree list

        <t1..tn> &&&' <t1'..tn'> := <(t1 &'' t1')..(tn &'' tn')>

                    t &'' Empty := t
                    Empty &'' t := t
                    t &'' t'    := t & t'

     |||' :  (tableau_tree list) * (tableau_tree list) -> tableau_tree list


        <t1..tn> |||' <t1'..tn'> := <(t1 | t1')..(tn | tn')>
```

HIER NOCHEINMAL DER WICHTIGE HINWEIS. DIE LETZTEN BEIDEN REGELN ZUR VERKNUEPFUNG DER T-ANTEILE SIND NUR EINE ERSTE ANNAHME. SIE SETZT VORAUS, DASS DIE VERKNUEPFUNGEN IN DEN EINZELNEN SUBSYSTEMEN VOLLKOMMEN UNABHAENGIG VONEINANDER STATTFINDEN, UND BERUCKSICHTIGT AUCH KEINEN SYSTEMWECHSEL DER FORMELN IN DEN EINZELNEN SPALTEN.
An dieser Stelle haben wir alles zur Implementierung des Beweismechanismus noetige beisammen. Diese Informationen werden im Implementierungsteil aufgegriffen und weiter spezifiziert.

3 Die Möglichkeiten des PKLBEWEISERS Version 1.0

In diesem Kapitel sollen die Möglichkeiten des Beweisersystems erläutert werden. Es ist nicht als Benutzerhandbuch zu verstehen. Die Darstellung gliedert sich in fünf Punkte :

1. Definition und Darstellung von Tableauregeln und Formeln
2. Beweisbäume und Termdarstellung
3. Baumoperationen
4. Beweiskonstruktion und Entwicklung neuer Regeln
5. Lokale Variablendarstellung

3.1 Definition von Tableauregeln und Definition von neuen Regeln basierend auf schon existierenden

Tableauregeln können auf zwei Arten definiert werden. Zum einen eben durch Tableaus, und zum anderen per Konstruktion durch schon bekannte Junktoren. Hierzu wurden drei Syntaktische Konstrukte definiert, die eine lesbare strukturierte Eingabe erlauben:

1.
```
<Formel>  := variable
          | "(" <Formel> ")"
          | "(" junktor <Formelliste> ")"
          | <Formel> junktor <Formel>

<Formelliste> := <Formel> "," <Formelliste>
               | <Formel>
```

Beispiele:

```
(X laa Y) aaa (Y laa X)
( ttt (X aaa Y) , X )
```

Junktoren werden generel klein geschrieben und können auch numerische Zeichen beinhalten:

```
( junktor := [a..z] {[a..z0..9]}* )
```

Variablen fangen mit einem Grossbuchstaben an und enthalten nur Buchstaben [a..zA..Z] :

```
( variable := [A..Z] {[a..zA..Z]}* ).
```

2.

```
<Tableauregel> := junktor <n> "(" <variablelist> ") :=" <RuleList>
        <RuleList>     := <Rule> <RuleList>
                        | <Rule>
        <Rule>         := "[ truval : AlphaBeta "]"
                        | "[ truval : AlphaBeta <TransList> "]"
        <TransList>    := "///" <AlphaBeta> <TransList>
                        | "///" <AlphaBeta>
        <AlphaBeta>    := <AlphaBeta> "&&" <AlphaBeta>
                        | <AlphaBeta> "||" <AlphaBeta>
                        | "(" <AlphaBeta> ")"
                        | truval ":" variable
                        | "[]"


        Beispiele ...

        aaa(X,Y) :=

        [T1:   T1: X && T1: Y ]  [F1:   F1: X || F1: Y ]
        [T2:   T2: X && T2: Y ]  [F2:   F2: X || F2: Y ]
        [T3:   T3: X && T3: Y ]  [F3:   F3: X || F3: Y ]


        laa(X,Y) :=

        [T1: (T1: X && T1: Y) || (T1: X && F1: Y)             ]
        [F1:  F1: X && F1: Y                                 ]
        [T2:  T2: X && T2: Y /// F1: X && F1: Y  /// [] /// [] ]
        [F2:  F2: X || F2: Y /// F1: X && T1: Y  /// [] /// [] ]
        [T3:  T3: X && T3: Y /// T1: X && T1: Y  /// [] /// [] ]
        [F3:  F3: X || F3: Y /// F1: X && T1: Y  /// [] /// [] ]

        n1 3 (X):=

        [T1:  F1:X] [F1:  T1:X]
        [T2:  T3:X] [F2:  F3:X]
        [T3:  T2:X] [F3:  F2:X]

        n2 3 (X):=

        [T1:  T3:X] [F1:  F3:X]
        [T2:  F2:X] [F2:  T2:X]
        [T3:  T1:X] [F3:  F1:X]
```

Die Zahlenangabe bei n1 und n2 ist nötig um dem System mitzuteilen das es sich um eine Definition in einem System mit drei Subsystemen handelt. Die Angabe ist optional. Als Defaultwert wird die Anzahl der Zeichen des Junktors verwendet. Die Angabe der Zahl ist bei der Definition basierend auf schon existierenden Operatoren wichtig(siehe die folgenden Beispiele). Hier muss

entschieden werden für welche Wahrheitswerte Tableauregeln generiert werden müssen. Wie im letzten Beispiel zu sehen können auch die Negationen durch Tableaus definiert werden.

3.
```
<TableauDef> := junktor <n> "(" <variablelist> ") :=" <Formel>
```

```
Beispiele ...

taa (X,Y) := (X laa Y) aaa (Y laa X),
n3 3 (X) := (n1 (n2 X)),
n4 3 (X) := (n2 (n1 X)),
n5 3 (X) := (n1 (n2 (n1 X)))
```

Für n = 3 würden Tableaus für die Wahrheitswerte [T1,F1,T2...F3] generiert. Für n=6 Tableaus für [T1,F1,T2,F2,T3...F6].

Die Definitionen werden auf semantische Fehler hin kontrolliert:

Fehler I Es werden nichtdefinierte Operatoren im rechten Teil der Definition verwendet.

```
xyz (X,Z) := (xxx (n1 X),(n2 Z))
                 ===
```

Fehler II Aritätsfehler, d.h. Operatoren werden zu viel bzw zu wenig Argumente übergeben. Die Arität eines Operators wird aus der Variablenliste bestimmt.

```
xxyz 3 (X,Y,Z) := (xyz X,Y,Z)
                        =====
```

Fehler III Es werden Variablen im rechten Teil der Definition verwendet, die in der Variablenliste nicht definiert worden sind.

```
xyz (X,Z) := (xxx (n1 X),(n2 Y))
                            ===
```

In allen drei Fällen können die Daten entweder über Tastatur oder per Fall in das System geladen werden. Der Systemzustand kann gespeichert werden. Hierbei werden keine Daten sondern ein neues komplettes System mit den neu definierten Regeln gespeichert.

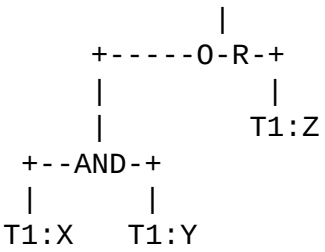
3.2 Beweisen von Aussagenlogischen Formeln der Pkl

Der Beweismechanismus ist nur teilweise automatisiert, sodass der Benutzer einen Beweis tatkräftig unterstützen muss. Er hat hierbei die Möglichkeit in den Beweisbaum hinabzusteigen und Beweisschritte partiell ausführen zu lassen. Die einzelnen Beweisschritte werden mitdokumentiert und lassen sich Tracen, sodas zum Schluss ein vollständiger den Abläufen entsprechender Beweis entsteht.

- Der Mechanismus führt alle nächstmöglichen Reduktionen in einem Schritt aus. Hierbei hält er sich an die in Kapitel I entwickelten Regeln.
- Der Benutzer muss Schrittweise den Beweis zu Ende führen.
- Es besteht die Möglichkeit eigene Regeln per Cut and Paste Modifikationen auf dem Baum auszuführen.

3.3 Baum-Darstellung sowie Term-Darstellung von Beweisbäumen

Bei der Darstellung von Beweisbäumen hat man die Auswahl zwischen zwei Repräsentationen. Zum einem eine eben Baumartige Darstellung, und zum anderen die etwas kompaktere Termdarstellung. Bei der Baumdarstellung handelt es sich allerdings nicht um die klassischen Tableaubäume sondern vielmehr um die Baumrepräsen- tation der Terme. Man kann während eines Beweises zwischen den Darstellungen hin und her schalten.
Beispiele ...
Formel 1: `T1: (X aaa Y) ooo Z`
Baumdarstellung 1: (nach einem Beweisschritt)



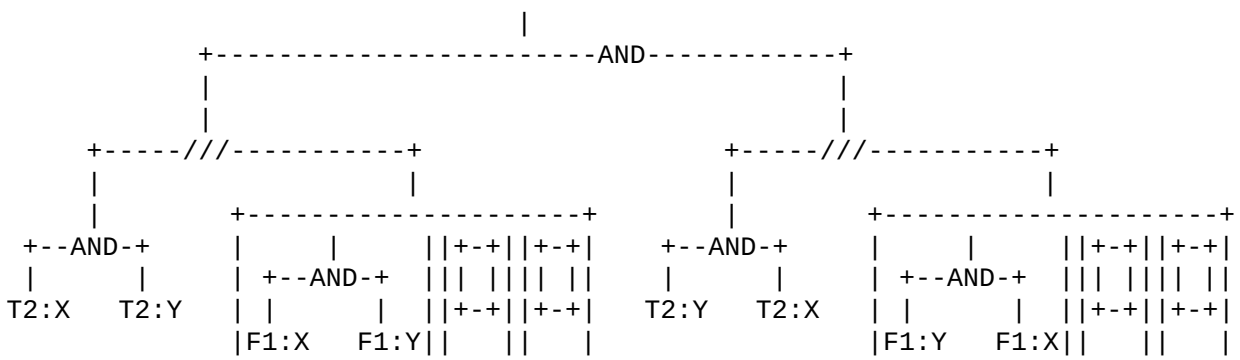
Termdarstellung 1:

```
((T1:X) & (T1:Y)) | (T1:Z)
```

Formel 2:

```
T2: (X laa Y) aaa (Y laa X)
```

Baumdarstellung 2: (nach mehreren Beweisschritten)



Termdarstellung 2:

```
(((t2:X) & (t2:Y)) //(f1:X) & (f1:Y)/LB/LB//) &
(((t2:Y) & (t2:X)) //(f1:Y) & (f1:X)/LB/LB//)
```

Bei den Darstellungen lassen sich noch die Darstellungstiefen beeinflussen, und die T-Anteile können abstahiert werden, d.h. sie werden im Baum sowie im Term durch TA"dargestellt.

3.4 Durchwandern der Beweisbäume mit der Möglichkeit zum partiellen Ausführen von Beweisschritten

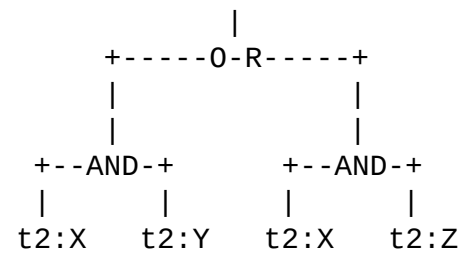
Wie schon erwähnt kann man den aktuellen Beweisbaum durchwandern und dann Teilbäume partiell reduzieren. Dies ist hilfreich um den Ablauf eines Beweises besser zu verstehen. Beim durchwandern des Baumes ist die Darstellung auf den aktuellen Teilbaum eingeschränkt.

3.5 Lokale Variablendarstellung

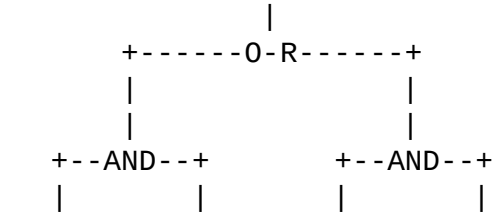
Variablen einer Formel werden mit einer Nummer gekennzeichnet. Mehrmaliges Vorkommen einer Variablen in einer Formel bedeutet, dass jedes Vorkommen der Variablen mit einer anderen Nummer versehen wird. Das ist im Laufe eines Beweises hilfreich, um sehen zu können welche Teile einer Formel an welchen Auswirkungen beteiligt waren.

```
(X aaa Y) ooo (X aaa Z)    ==>  (X1 aaa Y1) ooo (X2 aaa Z1)

Beispiel (Normal) ...
```




```
Beispiel mit lokaler Darstellung ...
```



```
t2:X1   t2:Y1   t2:X2   t2:Z1
```

3.6 BEMERKUNGEN

In dieser Version wurde zunächst Wert auf die Funktionalität der Beweismechanismen und auf eine Benutzerfreundliche Methode zur Spezifikation von Tableauregeln gelegt. Benutzerschnittstelle sowie Menusystem sind daher noch etwas mager und auf das Nötigste eingeschränkt.
underlineEtwas Zum Beweis-Mechanismus
In dieser Version wurde noch mit der Annahme von unabhängigen Systemen innerhalb von Transjunktiven Anteilen gearbeitet.Die Untersuchungen haben ergeben,dass dem nicht so ist, und das ein Systemwechsel einer Variablen in einem T-Anteil Auswirkungen auf das System hat,in das gewechselt wurde.
Weiterhin unklar ist noch der Umgang mit geschachtelten T-Anteilen. D.h. T-Anteile in T-Anteilen.
Ein weiteres Problem, dass aufgefallen ist, ist das konjunktive Zusammenfügen von T-Anteilen, wenn z.B. in einem der beiden Alpha-Aeste kein T-Anteil vorhanden ist. Uebr diese Regeln muss auch nochmal intensiv nachgedacht werden.
In diesem Zusammenhang ist auch die Unabhängig von lokalen Anteilen und T-Anteilen zu erwähnen. Hier bin ich mir gar nicht mehr so sicher ob man generell von Unabhängigkeit,bzw Abhängigkeit reden kann. Hier müssten meiner Meinung nach noch viel genauere Untersuchungen stattfinden, wobei man Abhängigkeit und Unabhängigkeit sowie Mischformen davon noch im Rahmen der jetzigen Regelbeschreibung formulieren kann. Ich tendiere im augenblick zu einer Mischform. Auf diese Probleme und deren soll etwas näher im nächsten Kapitel eingegangen werden.
Vor dem Angriff neuer Regeln sollten folgende Punkte untersucht werden:

- Unabhängigkeit/Abhängigkeit von lokalen und transjunktiven Anteilen
- Das Zusammenfügen von T-Anteilen
- Welche Auswirkungen 1. und 2. auf den angesprochenen Punkt der Systemwechsel haben

Eine neue Implementierung, die die Systemwechsel berücksichtigt, wird zumindest auf Punkt 2 keinen positiven Einfluss ausüben.
———- X-Sun-Data-Type: default X-Sun-Data-Description: default X-Sun-Data-Name: info.txt X-Sun-Content-Lines: 427

4 Die Möglichkeiten des PKLBEWEISERS Version 1.0

In diesem Kapitel sollen die Möglichkeiten des Beweisersystems erläutert werden. Es ist nicht als Benutzerhandbuch zu verstehen. Die Darstellung gliedert sich in fünf Punkte :

1. Definition und Darstellung von Tableauregeln und Formeln
2. Beweisbäume und Termdarstellung
3. Baumoperationen
4. Beweiskonstruktion und Entwicklung neuer Regeln
5. Lokale Variablendarstellung

4.1 Definition von Tableauregeln und Definition von neuen Regeln basierend auf schon existierenden

Tableauregeln können auf zwei Arten definiert werden. Zum einen eben durch Tableaus, und zum anderen per Konstruktion durch schon bekannte Junktoren. Hierzu wurden drei Syntaktische Konstrukte definiert, die eine lesbare strukturierte Eingabe erlauben:

1.
```
<Formel>  := variable
           | "(" <Formel> ")"
           | "(" junktor <Formelliste> ")"
           | <Formel> junktor <Formel>

<Formelliste> := <Formel> "," <Formelliste>
               | <Formel>
```

Beispiele:

```
(X laa Y) aaa (Y laa X)
( ttt (X aaa Y) , X )
```

Junktoren werden generel klein geschrieben und können auch numerische Zeichen beinhalten:

```
( junktor := [a..z] {[a..z0..9]}* )
```

Variablen fangen mit einem Grossbuchstaben an und enthalten nur Buchstaben [a..zA..Z] :

```
( variable := [A..Z] {[a..zA..Z]}* ).
```

2.
```
<Tableauregel> := junktor <n> "(" <variablelist> ") :=" <RuleList>
<RuleList>     := <Rule> <RuleList>
                | <Rule>
<Rule>         := "[ truval : AlphaBeta "]"
                | "[ truval : AlphaBeta <TransList> "]"
<TransList>    := "///" <AlphaBeta> <TransList>
                | "///" <AlphaBeta>
<AlphaBeta>    := <AlphaBeta> "&&" <AlphaBeta>
                | <AlphaBeta> "||" <AlphaBeta>
                | "(" <AlphaBeta> ")"
                | truval ":" variable
```

```
                | "[]"
```

```
Beispiele ...

aaa(X,Y) :=

[T1:   T1: X && T1: Y ]  [F1:   F1: X || F1: Y ]
[T2:   T2: X && T2: Y ]  [F2:   F2: X || F2: Y ]
[T3:   T3: X && T3: Y ]  [F3:   F3: X || F3: Y ]


laa(X,Y) :=

[T1: (T1: X && T1: Y) || (T1: X && F1: Y)             ]
[F1:  F1: X && F1: Y                                  ]
[T2:  T2: X && T2: Y /// F1: X && F1: Y  /// [] /// [] ]
[F2:  F2: X || F2: Y /// F1: X && T1: Y  /// [] /// [] ]
[T3:  T3: X && T3: Y /// T1: X && T1: Y  /// [] /// [] ]
[F3:  F3: X || F3: Y /// F1: X && T1: Y  /// [] /// [] ]

n1 3 (X):=

[T1:  F1:X] [F1:  T1:X]
[T2:  T3:X] [F2:  F3:X]
[T3:  T2:X] [F3:  F2:X]

n2 3 (X):=

[T1:  T3:X] [F1:  F3:X]
[T2:  F2:X] [F2:  T2:X]
[T3:  T1:X] [F3:  F1:X]
```

Die Zahlenangabe bei n1 und n2 ist nötig um dem System mitzuteilen das es sich um eine Definition in einem System mit drei Subsystemen handelt. Die Angabe ist optional. Als Defaultwert wird die Anzahl der Zeichen des Junktors verwendet. Die Angabe der Zahl ist bei der Definition basierend auf schon existierenden Operatoren wichtig(siehe die folgenden Beispiele). Hier muss entschieden werden für welche Wahrheitswerte Tableauregeln generiert werden müssen. Wie im letzten Beispiel zu sehen können auch die Negationen durch Tableaus definiert werden.

3.

```
<TableauDef> := junktor <n> "(" <variablelist> ") :=" <Formel>
```

```
Beispiele ...

taa (X,Y) := (X laa Y) aaa (Y laa X),
n3 3 (X) := (n1 (n2 X)),
n4 3 (X) := (n2 (n1 X)),
n5 3 (X) := (n1 (n2 (n1 X)))
```

Für n = 3 würden Tableaus für die Wahrheitswerte [T1,F1,T2...F3] generiert. Für n=6 Tableaus für [T1,F1,T2,F2,T3...F6].

Die Definitionen werden auf semantische Fehler hin kontrolliert:

Fehler I Es werden nichtdefinierte Operatoren im rechten Teil der Definition verwendet.

```
xyz (X,Z) := (xxx (n1 X),(n2 Z))
                  ===
```

Fehler II Aritätsfehler, d.h. Operatoren werden zu viel bzw zu wenig Argumente übergeben. Die Arität eines Operators wird aus der Variablenliste bestimmt.

```
xxyz 3 (X,Y,Z) := (xyz X,Y,Z)
                        =====
```

Fehler III Es werden Variablen im rechten Teil der Definition verwendet, die in der Variablenliste nicht definiert worden sind.

```
xyz (X,Z) := (xxx (n1 X),(n2 Y))
                             ===
```

In allen drei Fällen können die Daten entweder über Tastatur oder per Fall in das System geladen werden. Der Systemzustand kann gespeichert werden. Hierbei werden keine Daten sondern ein neues komplettes System mit den neu definierten Regeln gespeichert.

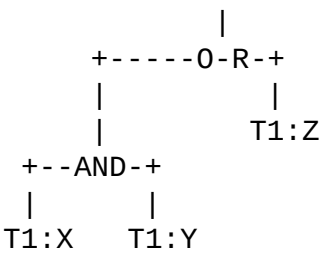
4.2 Beweisen von Aussagenlogischen Formeln der Pkl

Der Beweismechanismus ist nur teilweise automatisiert, sodass der Benutzer einen Beweis tatkräftig unterstützen muss. Er hat hierbei die Möglichkeit in den Beweisbaum hinabzusteigen und Beweisschritte partiell ausführen zu lassen. Die einzelnen Beweisschritte werden mitdokumentiert und lassen sich Tracen, sodas zum Schluss ein vollständiger den Abläufen entsprechender Beweis entsteht.

- Der Mechanismus führt alle nächstmöglichen Reduktionen in einem Schritt aus. Hierbei hält er sich an die in Kapitel I entwickelten Regeln.
- Der Benutzer muss Schrittweise den Beweis zu Ende führen.
- Es besteht die Möglichkeit eigene Regeln per Cut and Paste Modifikationen auf dem Baum auszuführen.

4.3 Baum-Darstellung sowie Term-Darstellung von Beweisbäumen

Bei der Darstellung von Beweisbäumen hat man die Auswahl zwischen zwei Repräsentationen. Zum einem eine eben Baumartige Darstellung, und zum anderen die etwas kompaktere Termdarstellung. Bei der Baumdarstellung handelt es sich allerdings nicht um die klassischen Tableaubäume sondern vielmehr um die Baumrepräsen- tation der Terme. Man kann während eines Beweises zwischen den Darstellungen hin und her schalten.
Beispiele ...
Formel 1: `T1: (X aaa Y) ooo Z`
Baumdarstellung 1: (nach einem Beweisschritt)



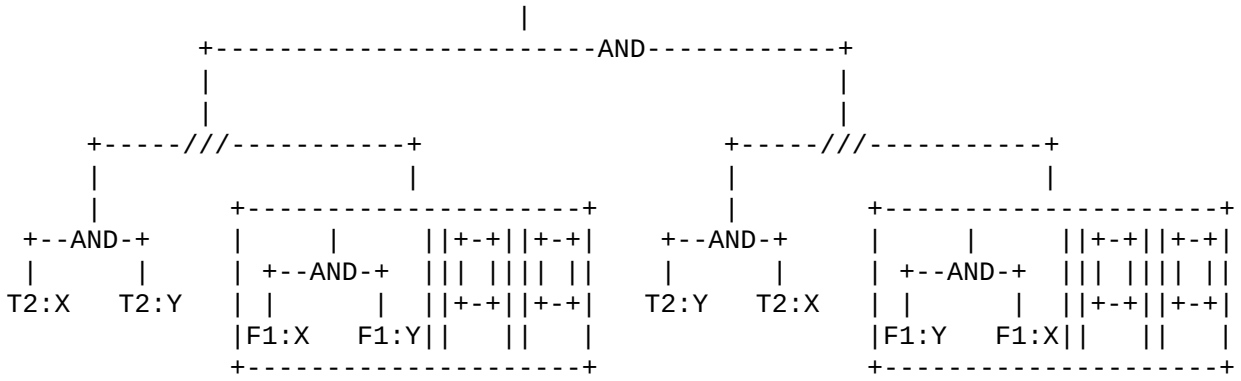
Termdarstellung 1:

```
((T1:X) & (T1:Y)) | (T1:Z)
```

Formel 2:

```
T2: (X laa Y) aaa (Y laa X)
```

Baumdarstellung 2: (nach mehreren Beweisschritten)



Termdarstellung 2:

```
(((t2:X) & (t2:Y)) //(f1:X) & (f1:Y)/LB/LB//) &
(((t2:Y) & (t2:X)) //(f1:Y) & (f1:X)/LB/LB//)
```

Bei den Darstellungen lassen sich noch die Darstellungstiefen beeinflussen, und die T-Anteile können abstahiert werden, d.h. sie werden im Baum sowie im Term durch TA"dargestellt.

4.4 Durchwandern der Beweisbäume mit der Möglichkeit zum partiellen Ausführen von Beweisschritten

Wie schon erwähnt kann man den aktuellen Beweisbaum durchwandern und dann Teilbäume partiell reduzieren. Dies ist hilfreich um den Ablauf eines Beweises besser zu verstehen. Beim durchwandern des Baumes ist die Darstellung auf den aktuellen Teilbaum eingeschränkt.

4.5 Lokale Variablendarstellung

Variablen einer Formel werden mit einer Nummer gekennzeichnet. Mehrmaliges Vorkommen einer Variablen in einer Formel bedeutet, dass jedes Vorkommen der Variablen mit einer anderen Nummer versehen wird. Das ist im Laufe eines Beweises hilfreich, um sehen zu können welche Teile einer Formel an welchen Auswirkungen beteiligt waren.

```
(X aaa Y) ooo (X aaa Z)    ==>  (X1 aaa Y1) ooo (X2 aaa Z1)

Beispiel (Normal) ...

                |
        +-----O-R-----+
        |             |
        |             |
    +--AND-+      +--AND-+
    |      |      |      |
   t2:X   t2:Y   t2:X   t2:Z



Beispiel mit lokaler Darstellung ...


                |
         +------O-R------+
         |               |
         |               |
     +--AND--+       +--AND--+
     |       |       |       |
   t2:X1   t2:Y1   t2:X2   t2:Z1
```

4.6 BEMERKUNGEN

In dieser Version wurde zunächst Wert auf die Funktionalität der Beweismechanismen und auf eine Benutzerfreundliche Methode zur Spezifikation von Tableauregeln gelegt. Benutzerschnittstelle sowie Menusystem sind daher noch etwas mager und auf das Nötigste eingeschränkt.
underlineEtwas Zum Beweis-Mechanismus
In dieser Version wurde noch mit der Annahme von unabhängigen Systemen innerhalb von Trans-junktiven Anteilen gearbeitet.Die Untersuchungen haben ergeben,dass dem nicht so ist, und das ein

Systemwechsel einer Variablen in einem T-Anteil Auswirkungen auf das System hat,in das gewechselt wurde.
Weiterhin unklar ist noch der Umgang mit geschachtelten T-Anteilen. D.h. T-Anteile in T-Anteilen.
Ein weiteres Problem, dass aufgefallen ist, ist das konjunktive Zusammenfügen von T-Anteilen, wenn z.B. in einem der beiden Alpha-Aeste kein T-Anteil vorhanden ist. Uebr diese Regeln muss auch nochmal intensiv nachgedacht werden.
In diesem Zusammenhang ist auch die Unabhängig von lokalen Anteilen und T-Anteilen zu erwähnen. Hier bin ich mir gar nicht mehr so sicher ob man generell von Unabhängigkeit,bzw Abhängigkeit reden kann. Hier müssten meiner Meinung nach noch viel genauere Untersuchungen stattfinden, wobei man Abhängigkeit und Unabhängigkeit sowie Mischformen davon noch im Rahmen der jetzigen Regelbeschreibung formulieren kann. Ich tendiere im augenblick zu einer Mischform. Auf diese Probleme und deren soll etwas näher im nächsten Kapitel eingegangen werden.
Vor dem Angriff neuer Regeln sollten folgende Punkte untersucht werden:

- Unabhängigkeit/Abhängigkeit von lokalen und transjunktiven Anteilen
- Das Zusammenfügen von T-Anteilen
- Welche Auswirkungen 1. und 2. auf den angesprochenen Punkt der Systemwechsel haben

Eine neue Implementierung, die die Systemwechsel berücksichtigt, wird zumindest auf Punkt 2 keinen positiven Einfluss ausüben.

5 KONZEPTE UND STRUKTUR DER IMPLEMENTIERUNG

Das folgende Kapitel soll die Konzepte und die Strukturierung der Implementierung tranzparent machen. Es ist als ein erster Einstieg zur Einarbeitung in das Programmsystem vorauszusetzen. Im zweiten Schritt sollen die Schnittstellen zu den wichtigsten Modulen dargestellt und erklärt werden. Die implementierung der Signaturen soll nicht mehr Bestandteil des Berichts sein. Hier wird auf die dokumentierten Sourcen verwiesen.

5.1 Das Konzept

Ein Ziel der Implementierung war es, möglichst elementare Operationen zur Verfügung zu stellen. Aus diesen sollen dann auf einer höheren Ebene alle weiteren komplexeren Operationen konstruiert werden. Aus dieser Vorgabe ergibt sich die folgende Struktur der Sourcen.

5.1.1 Operationale Kern

Formeln Elementare Operationen auf Formeln

Tableaus Darstellung sowie Operationen auf Tableaus sowie Operationen zur Unterstützung des Beweismechanismus, sowie Verwaltung der Regeln.

Beweismechanismus Operationen zum schrittweisen Beweisen (nach Kapitel I) unter der Verwendung der Operationen der TAbleaus

TabDef Operationen , die verschiedene Darstellungen von Tableauregeln in die innere Repräsentation der Regeln transformieren.

TabWalk Operationen zum durchwandern der Tableaubäumen sowie Operationen zur partiellen Manipulation von Bäumen

TabPic Konverter zur Ascci-Darstellung von Tableaubäumen und Regeln, d.h also Generierung der Termdarstellung aus der internen Repräsentation

Parser Zur Benutzerfreundlichen Eingabe von Tableauregeln und Formeln

5.1.2 Benutzerschnittstelle

Operationen, die das experimentieren mit Tableaubeweisen für den Benutzer erlauben. Diese werden in Menüs zusammengefast, die die Schnitstelle zum Benutzer bieten.

Menü Tool Operationen zur Konstruktion von (zyklischen) Menüs

Menüs (Benutzerschnittstellen) :

- Eingabe und Verwaltung von Tableauregeln
- Schrittweise und partielle Beweisführung sowie Möglichkeiten zur Baummanipulation
- Verwaltung der Darstellung von Tableaubäumen, Regeln und Formeln

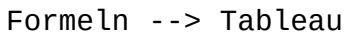
5.2 Hilfsmittel (Benutzte Tools)

Die jetzigen Sourcen sind in SML/NJ geschrieben und unter der Version 0.75 entwickelt worden. Zum schreiben der Parser wurden der Scannergenerator sml-lex und der Parsergenerator sml-yacc verwendet. Zur Implementierung des Beweismechanismus wurde sml-twig benutzt. Hiermit ist es möglich durch einfache Regelangaben Programme zur Termreduktion zu generieren.
Zur besseren Verwaltung der Sourcen wird das Make System unter SML verwendet, wodurch sich Sourcen zu logischen Moduleinheiten gliedern lassen.

5.3 Modularer Aufbau und Abhängigkeit der Sourcen

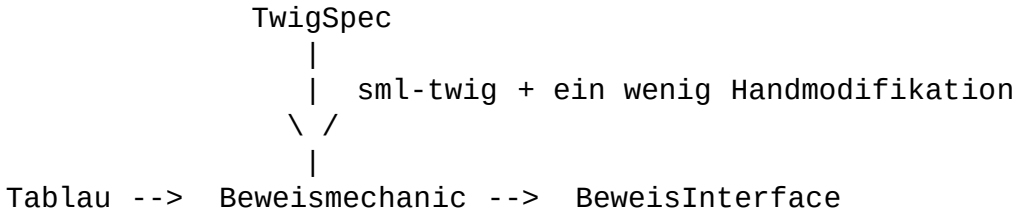
Der Modulare Aufbau der Sourcen ist im wesentlichen wie unter I. und II. organisiert, und spiegelt sich auch so in der Directorie-Struktur wieder.
Im folgenden soll die Abhängigkeit der einzelnen Moduleinheiten voneinander dargestellt werden, was auch das Strukturierte Durcharbeiten der Sourcen ermöglichen soll.

1. **Tableau**

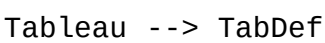


2. **Beweise**

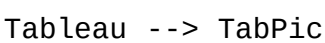
   Dieser Teil der Implementierung sollte im nächste Schritt durch einen eigenen Reduktionsmechanismus ersetzt werden, sodass der Twigs nicht mehr benötigt wird. Dadurch gelänge man zu einer saubereren und transparenteren Schnittstelle.



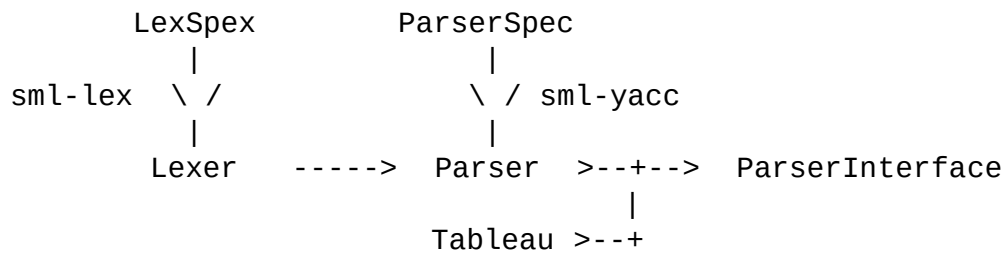
3. **TabDef**



4. **TabPic**

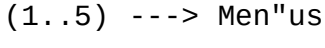


5. **Parser**



6. **Menüs**

   Die Menüs fassen die elementaren Operationen des Operationalen Kerns zu komplexeren Aufgaben zusammen, und sind daher von allen elementaren Modulen abhängig.



5.4 Ein kleiner Wegweiser durch die Folderstruktur

```
Tableau    ->  formel.sml
               tableau.sml  (Implementierung von Tableau)

Beweis     ->  Spec         -> tabtwig
                                ( TwigSpezifikation )
                                tabtwig.sml
                                ( Beweismechanic von sml-twig generiert)
               beweis.sig
               beweis.sml  ( Beweise-Interface )

TabDef     ->  tabdef.sml  (Spezifikation von Tableauregeln)

TabWalk    ->  tabwalk.sml (I'm awanderer ...)

TabPic     ->  tabpic.sml

Parser     ->  formel      -> lex         (LexerSpezifikation)
                              grm         (ParserSpezifikation)
                              lex.sml     (generieter Lexer)
                              grm.sig
                              grm.sml     (generierter Parser)
                              parser.sml  (Parser Interface)
               tabdef      -> ...         (gleiche Struktur wie unter Formel
               tableaus    -> ...         (  s.o.  )

Men"uTool  ->  menu.sml  (Tool zur Konstruktion der Men"us )

Men"us     ->  userMnu.sml    ( Hauptmenu  )
               tabMnu.sml     ( Tableau Handhabung )
               beweisMnu.sml  ( experimentieren mit Beweisen )


Make       ->  ...   ( Zusammenf"ugen der Sourcen )
```

———- X-Sun-Data-Type: default X-Sun-Data-Description: default X-Sun-Data-Name: impl.txt X-Sun-Content-Lines: 217

6 KONZEPTE UND STRUKTUR DER IMPLEMENTIERUNG

Das folgende Kapitel soll die Konzepte und die Strukturierung der Implementierung tranzparent machen. Es ist als ein erster Einstieg zur Einarbeitung in das Programmsystem vorauszusetzen. Im zweiten Schritt sollen die Schnittstellen zu den wichtigsten Modulen dargestellt und erklärt werden. Die implementierung der Signaturen soll nicht mehr Bestandteil des Berichts sein. Hier wird auf die dokumentierten Sourcen verwiesen.

6.1 Das Konzept

Ein Ziel der Implementierung war es, möglichst elementare Operationen zur Verfügung zu stellen. Aus diesen sollen dann auf einer höheren Ebene alle weiteren komplexeren Operationen konstruiert werden.

Aus dieser Vorgabe ergibt sich die folgende Struktur der Sourcen.

6.1.1 Operationale Kern

Formeln Elementare Operationen auf Formeln

Tableaus Darstellung sowie Operationen auf Tableaus sowie Operationen zur Unterstützung des Beweismechanismus, sowie Verwaltung der Regeln.

Beweismechanismus Operationen zum schrittweisen Beweisen (nach Kapitel I) unter der Verwendung der Operationen der TAbleaus

TabDef Operationen , die verschiedene Darstellungen von Tableauregeln in die innere Repräsentation der Regeln transformieren.

TabWalk Operationen zum durchwandern der Tableaubäumen sowie Operationen zur partiellen Manipulation von Bäumen

TabPic Konverter zur Ascci-Darstellung von Tableaubäumen und Regeln, d.h also Generierung der Termdarstellung aus der internen Repräsentation

Parser Zur Benutzerfreundlichen Eingabe von Tableauregeln und Formeln

6.1.2 Benutzerschnittstelle

Operationen, die das experimentieren mit Tableaubeweisen für den Benutzer erlauben. Diese werden in Menüs zusammengefast, die die Schnitstelle zum Benutzer bieten.

Menü Tool Operationen zur Konstruktion von (zyklischen) Menüs

Menüs (Benutzerschnittstellen) :

- Eingabe und Verwaltung von Tableauregeln
- Schrittweise und partielle Beweisführung sowie Möglichkeiten zur Baummanipulation
- Verwaltung der Darstellung von Tableaubäumen, Regeln und Formeln

6.2 Hilfsmittel (Benutzte Tools)

Die jetzigen Sourcen sind in SML/NJ geschrieben und unter der Version 0.75 entwickelt worden. Zum schreiben der Parser wurden der Scannergenerator sml-lex und der Parsergenerator sml-yacc verwendet. Zur Implementierung des Beweismechanismus wurde sml-twig benutzt. Hiermit ist es möglich durch einfache Regelangaben Programme zur Termreduktion zu generieren.
Zur besseren Verwaltung der Sourcen wird das Make System unter SML verwendet, wodurch sich Sourcen zu logischen Moduleinheiten gliedern lassen.

6.3 Modularer Aufbau und Abhängigkeit der Sourcen

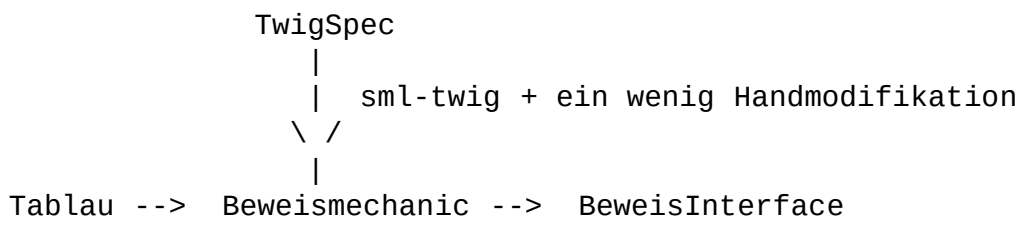
Der Modulare Aufbau der Sourcen ist im wesentlichen wie unter I. und II. organisiert, und spiegelt sich auch so in der Directorie-Struktur wieder.
Im folgenden soll die Abhängigkeit der einzelnen Moduleinheiten voneinander dargestellt werden, was auch das Strukturierte Durcharbeiten der Sourcen ermöglichen soll.

1. **Tableau**

```
Formeln --> Tableau
```

2. **Beweise**

   Dieser Teil der Implementierung sollte im nächste Schritt durch einen eigenen Reduktionsmechanismus ersetzt werden, sodass der Twigs nicht mehr benötigt wird. Dadurch gelänge man zu einer saubereren und transparenteren Schnittstelle.



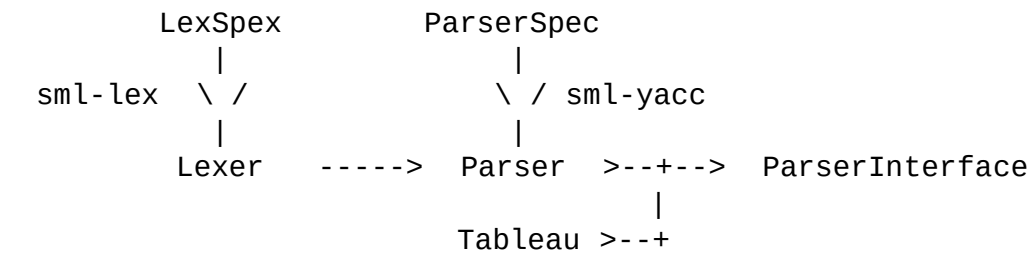
3. **TabDef**

```
Tableau --> TabDef
```

4. **TabPic**

```
Tableau --> TabPic
```

5. **Parser**



6. **Menüs**

   Die Menüs fassen die elementaren Operationen des Operationalen Kerns zu komplexeren Aufgaben zusammen, und sind daher von allen elementaren Modulen abhängig.

```
(1..5) ---> Men"us
```

6.4 Ein kleiner Wegweiser durch die Folderstruktur

```
Tableau    ->  formel.sml
               tableau.sml  (Implementierung von Tableau)

Beweis     ->  Spec         -> tabtwig
                                ( TwigSpezifikation )
                                tabtwig.sml
                                ( Beweismechanic von sml-twig generiert)
               beweis.sig
               beweis.sml  ( Beweise-Interface )

TabDef     ->  tabdef.sml  (Spezifikation von Tableauregeln)

TabWalk    ->  tabwalk.sml (I'm awanderer ...)

TabPic     ->  tabpic.sml

Parser     ->  formel      -> lex         (LexerSpezifikation)
                              grm         (ParserSpezifikation)
                              lex.sml     (generieter Lexer)
                              grm.sig
                              grm.sml     (generierter Parser)
                              parser.sml  (Parser Interface)
               tabdef      -> ...         (gleiche Struktur wie unter Formel
               tableaus    -> ...         (  s.o.  )

Men"uTool  ->  menu.sml  (Tool zur Konstruktion der Men"us )

Men"us     ->  userMnu.sml    ( Hauptmenu  )
               tabMnu.sml     ( Tableau Handhabung )
               beweisMnu.sml  ( experimentieren mit Beweisen )


Make       ->  ...   ( Zusammenf"ugen der Sourcen )
```

7 DOKUMENTATION DER WICHTIGSTEN SCHNITTSTELLEN DER MODULE

7.1 Das Modul Der AUSSAGENLOGISCHEN Formeln

Struktur der Formeln und der unteliegenden Logik der Tableaus !!
Hier werden die elementaren Datentypen der Logik spezifiziert, sowie Operationen zu deren Gemerierung und Manipulation. Hierzu gehören die Formeln und deren Bestandteile (z.B. Wahrheitswerte, Variablen usw). Von diesen elementaren Objekten einer Formel wird abstrahiert. Sie werden nur als Typen definiert.

7.1.1 Spezifikation der elementaren Datentypen

Wie vielleicht schon aus der Aussagenlogik bekannt bestehen Formeln aus elementaren Einheiten, aus denen dann komplexere Formeln (induktiv) konstruiert werden können. Was eine Logik in der Regel besitzt sind Wahrheitswerte, und diese werden unter dem Typ truval zusammengefasst. Formeln werden nun aus Variablen, Operatoren (Junktoren) und auch dem gesondert betrachteten Negationsoperator zusammengesetzt. In der PKL gibt es hiervon mehr als einen.
Die Typen noch mal zusammengefasst:

```
type truval
        type var
type negation
type operator
```

7.1.2 Der Aufbau der Formeln

Unter dem Kontext der PKL wird nun zunächst ein sehr einfacher Datentyp Formel definiert. Formeln sind zum einen Variablen, oder für Formeln f1,..,fn ist für einen beliebigen Operator op auch op(f1,. . . ,fn) eine Formel. Für binäre Operatoren wird häufig auch f1 op f2 geschrieben.
Der Datentyp der Formeln:

```
datatype formula =
 Var  of var
       | Conn of (operator*(formula list))
```

7.1.3 Operationen auf den atomaren Daten von Formeln

Um nicht direkt auf die elementaren Daten zugreifen zu müssen werden Operationen auf ihnen definiert. Dies hat den Vorteil, das eine Strukturelle Aenderung an den Daten nicht sofort zu einer Aenderung der Quelltexte führt, in denen diese benutzt werden.
Ein elementarer Begriff ist der Widerspruchbegriff, der hier derart angelegt ist, das er bei der Angabe zweier Variablen und derer Belegungen sagt ob eine Widerspruch vorliegt.

```
val widerspruch : (truval*var)*(truval*var) -> bool
```

Die nächste Operation liefert eine Menge von Wahrheitswerten aus der Unterliegenden Logik. Hier wird angenommen, das auf diesen Wahrheitswerten eine Ordnung definiert ist, und das ein Parameter N angibt, das die ersten N Wahrheitswerte dieser Ordnungt generiert werden.

```
val truvalGen   : int -> truval list
```

Der nächste Typ definiert eine Ausnahme, die von partiellen Funktionen zurückgegeben wird, falls ein Argument nicht im Definitionsberich der Funktion liegt. Dieser Typ beinhaltet eine Fehlernummer und einen kurzen Informationstext.

```
exception FormulaError of int*string
```

7.1.4 Weitere Datengeneratoren

Die folgenden Funktionen wandeln Zeichenketten in die entsprechenden Datentypen um. Alle elementaren Datentypen haben eine spezielle Syntax. Nur diese wird von diesen Funktionen akzeptiert.

```
val string2truval   : string -> truval
val string2operator : string -> operator
val string2var      : string -> var
```

Die folgenden Funktionen sind die Umkehrung der letzten drei, sie wandeln Datentypen wieder in Strings um.

```
val truval2string   :  truval -> string
val operator2string :  operator -> string
val var2string      :  var -> string
```

Für die Umwandlung von Zeichenketten in Formeln gibt es einen eigenen Parser, ist also nicht Bestandteil dieses Moduls. Es gibt allerdings eine Funktion, die Formeln wieder in Zeichenketten umwandelt.

```
val formula2string :  formula -> string
```

7.1.5 Ordnung auf den Wahrheitswerten und Operatoren

Auf den Operatoren und Wahrheitswerten muss eine Ordnung (<=) definiert werden, die besagt, ob ein Datum "kleiner ist als des andere.

```
val truvalOrder    : truval*truval -> bool
val operatorOrder  : operator*operator -> bool
```

Die folgenden Funktionen betreffen die Generierung und Darstellung der Variablen. Jede Variable wird mit einer Kennung versehen. Die gleiche Variable X bekommt pro Vorkommen in einer Formel eine Kennung.
Beispiel

```
(X or Y)  and X      ohne Kennung
(X2 or Y1)  and X2   mit Kennung
```

Hierzu werden folgende Funktionen gebraucht:

1. `varEqual : var*var -> bool`

   Gleichheittest auf Variablen, unabhängig von der Kennung.

2. `varReset : int -> unit`

   Die Kennungen sind fortlaufende. Diese werden mit dieser Funktion auf einen Initialen Wert zurückgesetzt. (Sinnvoll nach dem einlesen einer kompletten Formel)

3. ```
   val varFull  : unit -> unit
   val varShort :unit -> unit
   ```

   Diese Operationen beeinflussen die Umwandlungen der Variablen in Zeichenketten. Sie können mit Kennung (varFull) oder ohne (varShort) dargestellt werden.

7.2 Grundlegende Module zur Spezifikation des Beweisers

Die Implementierung der Grundlagen lässt sich in folgende Punkte aufgliedern.

FORMELN(LOGIK) Die Struktur beinhaltet alle wichtigen elementaren Daten und Operationen der Logik, wie z.B. Wahrheitswerte und Wiederspruchsbegriff. Jede Formel- struktur muss der Signature FORMULA genügen (MODUL Formeln). Alle weiteren Strukturen bauen auf auf dieser Struktur auf, und benutzen nur die in der Signatur spezifizierten Datentypen und Operationen. Dies ermöglicht eine schnelle und einfache Konstruktion eines neuen Beweisers mit neuer unterliegender Basis-Logik.

TABLEAUS Darstellung von Tableau Beweisbäumen und Tableauregeln. (Im wesentlichen wie sie im Rahmen des Beweismechanismus beschrieben wurden)

TABLEAUREGELNVERWALTUNG Operationen zur Verwaltung von Tableauregeln in Tabellen.

GRUNDLEGENDES
Erster Schritt ist die Definition der Darstellung von Beweisbäumen. Hierin wird der Typ der Formeln verlangt, die aus der Struktur Formula importiert wird.

7.2.1 DIE TABLEAUS

Der Funktor enthält eine Struktur Formula, eine Instanz des Moduls der Formeln, sowie die Darstellung der Tableaubeweisbäume, der vom Typ der Formeln abhängig ist.

```
structure Formula

datatype tableau_tree =
    Empty
  | SF  of Formula.truval*Formula.formula
```
```

```
| DNF of (Formula.truval*Formula.var) list list
| &&  of tableau_tree * tableau_tree
| ||  of tableau_tree * tableau_tree
| /// of tableau_tree * (tableau_tree list)
```

Tableau-Regeln

Eine Tableau-Regel eines Operators ist eine Funktion vom Typ:

```
formula list -> tableau_tree
```

Die 'formula list' beinhaltet die aktuellen Operanden des obesrten Operators einer Formel. Zurück bekommt mann die Tableau-Tree Darstellung der Formel.
Beispiel:

```
Tableau    F1: X aaa Y
           ===========
           F1:X || F1:Y
```

Bezeichnen wir diese Regel mit t1-aaa. Unsere aktuelle signierte Formel ist

```
SF ( F1 , ((X ooo Y) aaa Z) )
```

mit den Teilformeln (X ooo Y) und Z. Die Anwendung der Regeln ergibt folgendes:

```
t1-aaa [(X ooo Y), Z] =>  F1:(X ooo Y) || F1:Z
```

Verwaltung der Tableau-Regeln

Die Tableau-Regeln müssen in irgendeiner Form verwaltet werden. Wünschenswert ist es neue Regeln zu generieren, diese abzulegen, sie zu modifizieren und sie abzufragen.
Hier wird mit zwei Tabellen gearbeitet:

1. eine Tabelle, die die Regeln eines Operators zusammenfast,sortiert nach Wahrheitswerten. Die Einträge der Tabelle bestehen also aus einem Paar (Wahrheitswert,Regel) mit Suchindex Wahrheitswert.

```
type tableauTable
```

2. Eine Tabelle die die Regeln aller definierten Operatoren umfast, also sortiert nach Operatoren. Jeder Eintrag zu einem Operators beinhaltet eine Tabelle aus

```
=> (Operator, (n,VariablenListe,tabl"auTable)).
```

Die anderen beiden Einträge haben folgende Bedeutung:

n ist die Anzahl der Systeme für die der Operator definiert ist. Diese Zahl wird bei der Definition von Regeln explizit angegeben. Wenn nicht wird die Grösse des Identifiers defaultmässig eingesetzt.(Dieser Wert hat bislang nur Informativen gehalt)

VarListe dient zum einen zur Aritätsbestimmung eines Operators, zum anderen aber auch für Tableaurepräsentation. Hier wird ebenfalls die Liste aus der Definition übernommen.

```
type opTable
```

Wir benötigen die folgenden Operationen für beide Kategorien von Tabellen :

```
  empty   -> Die leere Table
  update  -> Modifizieren/einf"ugen eines Eintrages und liefern der neuen
             Tabelle
  lookup  -> Liefern der Daten eines Eintrages

  Beide liefern eine Exception beim Misserfolg einer Suche *)


Operationen zu I.

  exception TableauLookup
  val emptyTableau  : tableauTable
  val lookupTableau : tableauTable -> Formula.truval  ->
                      (Formula.formula list -> tableau_tree)
  val updateTableau : tableauTable -> Formula.truval  ->
                      (Formula.formula list -> tableau_tree) ->
tableauTable

Operationen zu II.

  exception OpLookup
  val emptyOp   : opTable
  val lookupOp  : opTable -> Formula.operator  ->
          (int * (Formula.var list) * tableauTable)
  val updateOp  : opTable -> Formula.operator  ->
         (int * (Formula.var list) * tableauTable) ->
  opTable
```

7.2.2 TABLEAU-BEWEISER

Die folgenden Operationen sind genauer im Rahmen des Beweismechanismus dokumentiert.

7.2.3 Zusammenfügen von Dnf's

- Konjunktiv

```
    val |--|   : (Formula.truval * Formula.var) list list *
```

```
                  (Formula.truval * Formula.var) list list
            ->  (Formula.truval * Formula.var) list list
```

- Disjunktiv

```
      val |++|   : (Formula.truval * Formula.var) list list *
                   (Formula.truval * Formula.var) list list
                     ->  (Formula.truval * Formula.var) list list
```

Expandierung von signierten Formeln (SF(sg,fml))

```
val expand : tableau_tree -> tableau_tree
```

Das Zusammenfügen der Transjunktiven Anteile

HINWEIS: Hier werden sicherlich weitere Funktionen und modifikationen notwendig sein, um Systemwechsel in T-Anteilen zu Implementieren.

```
val &&&    : tableau_tree list * tableau_tree list -> tableau_tree list
     val |||    : tableau_tree list * tableau_tree list -> tableau_tree list
```

Die Tabelle der Tableau-Regeln

Die letzten Operationen werden in dem von Twig generierten Sourcen verwendet, u.a. expand, dass auf eine Regeltabelle zugreifen muss. Mir wäre es lieber gewesen expand diese Tabelle als Parameter zu übergeben. Es ist aber leider noch nicht möglich dem Twig mitzuteilen, das man mehrere Parameter an seine Schnittstellenfunktion übergeben möchte, sodas keine Möglichkeit besteht eine Tabelle von aussen her an expand weiterzugeben.
Ich habe mich also für folgende Lösung entschieden.
Die Tabelle der Tableauregeln wird schon an dieser Stelle definiert, damit sie in dem vom Twig generierten Sourcen verfügbar ist. Sie ist allerdings modifizierbar, das heist sie ist durch eine andere ersetzbar. Auf dieser Ebene sind Erweiterungen der Tabelle möglich.
Die folgenden beiden Routienen erlauben den Zugriff auf die aktuelle Beweistabelle:

```
     val get_table: unit -> opTable
     val set_table: opTable -> unit
```

7.3 Modul TABDEF zur Generierung von Tableauregeln und Formel

Dieses Modul ist die Schnittstelle zwischen Formel und Tableau- definitionsparser und der Generirung der eigentlichen Daten. Es volzieht die semantischen Analysen auf den abstrakten Darstellungen von Formeln und Tableauregeln, und generiert bei Erfolg die entsprechenden internen Darstelungen.
Die folgenden Datentypen aus anderen Modulen werden benutzt:

```
type var
type truval
type operator
type formula
type tableau_tree
```

Bei Fehlern wird der folgende Typ zurückgegeben:
exception TabDefError of string

7.3.1 Formeln

Die erste Funktion bekommt eine Liste von Formeln (Typ formula) und testet die beiden folgenden Dinge:

- Ob jeder Operator(Junktor) der Formel bereits definiert wurde.
- Wenn ein Operator definiert wurde, ob er denn auf die richtige Anzahl von Argumenten angewandt wurde.

Es werden die entsprechenden Fehlermeldungen ausgegeben. Die interne Struktur der Formeln ist auch vom Typ formula, also werden die fehlerfreien Formeln wieder zurückgegeben.

```
==>    fmls2fmls:  (formula list) -> (formula list)
```

7.3.2 Tableaus

Zur Definition von Tableaus stehen zwei Möglichkeiten zur Verfügung. Zum einen die Definition per Tableauregeln, zum anderen die per Definition aufbauend auf bereits definierten Operatoren.
I) Tableauregeln
Die Definition einer Regel beinhaltet die folgenden Bestandteile:

- Name des Operators (Junktor) vom Typ operator
- Seine Arität, also die Anzahl seiner Argumente (Typ int)
- Die Variablenliste der Definition (Typ var list)
- Die Tableauregeln der entsprechenden Wahrheitswerte `(Typ (truval*tableau_tree) list)`

Die Funktion bekommt eine Liste von Definitionen übergeben, und testet ob alle Variablen der Tableauregeln auch in der Variablenliste der Definition enthalten sind. Ist das der Fall, so werden alle Regeln in die interne Darstellung der Regeln umgewandelt und sofort in die Tabelle der Regeln eingetragen. (Seiteneffekt !!) Ansonsten werden die entsprechenden Fehlermeldungen zurückgegeben.

```
tabs2rules: (operator *
                (int *
                 (var list) *
                 ((truval*tableau_tree) list))
            ) list     ->   unit
```

II) Tableaudefinition

Die Definition beinhaltet die folgenden Bestandteile:

- Name des Operators (Junktor) vom Typ operator
- Seine Arität, also die Anzahl seiner Argumente (Typ int)
- Die Variablenliste der Definition (Typ var list)
- Eine Formel vom Typ formula

```
defs2rules: (operator *
               (int *
               (var list) *
               formula)
             ) list  -> unit
```

Die Funktion testet Testet die folgenden Dinge:

- Ob jeder Operator(Junktor) der Formel definiert ist (also ein Eintrag in der internen Tabelle existiert).
- Ob Operatoren auf richtige Anzahl von Argumenten angewandt wird.
- Ob alle Variablen innerhalb der Formel auch in der Variablenliste der Definition enthalten sind.

Ist das der Fall, so werden Regeln für alle Wahrheitswerte erzeugt, und in der internen Tabelle abgelegt.

7.4 Modul zum Durchwandern der Bäume

Dieses Modul gibt die Möglichkeit Tableaubäume zu durchwandern und dabei Teilbäume zu modifizieren, und das ganze ohne Seiteneffekte! Man kann also Teilbäume, deren Brüder, Söhne oder Väter besuchen.
Die Datenstruktur mit der dies möglich ist heisst

```
type tabwalk
```

Als erstes mus die Datenstruktur initialisiert werden. Dieses leistet die Funktion

```
init_tabwalk: tableau_tree -> tabwalk .
```

Diese Funktion bettet einen tableau_tree in die Datenstruktur ein. Man befindet sich an der Wurzel des Baumes. Mit den folgenden vier Funktionen kann die jeweilige Position im Baum verändert werden.

```
val left  : tabwalk -> tabwalk
val right : tabwalk -> tabwalk
val up    : tabwalk -> tabwalk
val down  : tabwalk -> tabwalk
```

Funktionsweise:

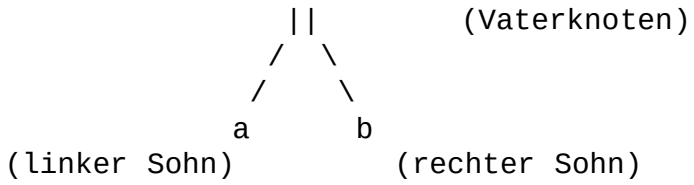
left aktuelle Position wird der linke Bruder des aktuellen Teilbaumes, falls dieser existiert. Falls nicht bleibt die aktuelle Position bestehen.

right aktuelle Position wird der rechte Bruder des aktuellen Teilbaumes, falls dieser existiert.

up aktuelle Position wird der Vaterknoten des aktuellen Teilbaumes, falls dieser existiert.

down aktuelle Position wird der linke Sohn des aktuellen Teilbaumes, falls dieser existiert

Ein Beispiel soll die Funktionsweise verdeutlichen. Nehmen wir den folgenden tableau_tree:



Unter anderem ist a linker Bruder von b und b rechter Bruder von a. Wir befinden uns am Vaterknoten und führen die Operation up aus. Wir nehmen an, das der Vaterknoten auch die Wurzel des Baumes ist. Die Operation hat also keine Wirkung, an der aktuellen Position hat sich nichts verändert. (genauso bei left und right, => keine Brüder) Wir führen nun die Operation down aus und landen bei Teilbaum a.

Aktuelle Position ist also jetzt a. Die Operation left hätte nun keine Wirkung, da a keinen linken Bruder hat. Wenn a ein Blatt wäre hätte auch down keine Wirkung. Die Operation right bringt uns nun zu a's rechten Bruder

Aktuelle Position ist also Teilbaum b. (Ein weiteres right hätte keine Wirkung, left würde uns wieder nach a bringen) Um wieder nach || zu gelangen führen wir up aus und sind am Ursprung.

Nun werden durch die letzten Operationen nur Positionen im Baum gewechselt. Wir benötigen noch eine Funktion, die einem den aktuellen Teilbaum aus der Struktur extrahiert.

```
val actual: tabwalk -> tableau_tree
```

Eine weiter Funktion bietet die möglichkeit den aktuellen Teilbaum durch einen anderen zu substituieren:

```
val map_actual  : (tableau_tree -> tableau_tree) ->
                          tabwalk -> tabwalk
```

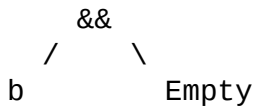
Hierbei wird eine Funktion f übergeben, die den aktuellen Teilbaum ta durch f(ta) ersetzt. Sagen wir im obigen Beispiel befänden wir uns an der aktuellen Position bei Teilbaum b.

```
fun addEmpty tb =  tb && Empty
```

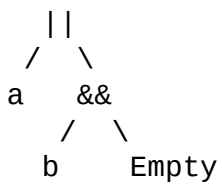
Der tabwalk heisse tw, und wir führen

```
map_actual addEmpty tw
```

aus. Als Ergebnis erhalten wir einen tabwalk der an der aktuellen Position den Teilbaum



besässe. Die Operation left würde die Position nach Teilbaum a wechseln, und nach einem weiteren up sähe der aktuelle Baum so aus:



Bei der Ausführung der nächste Operation befindet man sich an der Wurzel des Baumes.

```
val up_to_top   : tabwalk -> tabwalk
```

7.5 Modul zur Umwandlung von Bäumen und Formeln in Ascii Repräsentation

In diesem Modul werden Routienen zur Verfügung gestellt, die Beweisbäume und Regeln sowie Dnf's in Ascii-Darstellungen umwandeln.
Für Beweisbäume sind zwei Formen der Repräsentation möglich:

- Termdarstellung

  Diese ist kompakter und erlaubt bei grösseren Beweisbäumen einen besseren Ueberblick.

- Baumdarstellung

Zwischenrepräsentation

Die Daten werden nicht unmittelbar in Asciidarstellungen transformiert. Die Baumdarstellungen werden zunächst in den Typ picture umgewandelt, auf dem dann weitere Bildoperationen mit dem Modul Picture möglich sind. Terme werden in den Typ term umgewandelt, der dann durch das Modul Pretty weiter behandelt werden kann. Diese Module befinden sich unter pkl/tools/ .

7.5.1 Die Operationen

```
1) setPrintDepth : int -> unit

   Einstellung der Darstellungstiefe von B"aumen und Termen.


2) toggleFullTerm: unit -> unit

   Ein und Ausblenden der transjunktiven Anteile. Die T-Anteile werden
   entweder voll oder nur durch "TA" dargestellt.
```

3) tabtree2pic : tableau_tree -> picture

Baumdarstellung eines Beweisbaumes. Ergebnis ist die Zwischenrepr"asentation, die mit dem Modul Picture weiterverarbeitet werden kann.

4) tabtree2term : tableau_tree -> term

Termdarstellung eines Beweisbaumes. Hier ist das Ergebnis die Zwischenrepr"asentation eines Prettyprinters, die mit dem Modul Pretty weiterverarbeitet werden kann.

5) tabtable2pic : operator * truval * (var list) *
(formula list -> tableau_tree) -> picture

Darstellung von Tableauregeln, in der bekannten Weise.

6) dnf2pic : (truval*var) list list -> picture

Darstellung von Dnf’s.

8 Die Signaturen

Zum Überblick hier die Signaturen der Sourcen.

```
  signature FORMULA =
      sig
         type   var
 eqtype truval
 eqtype negation
 eqtype operator

         datatype formula =
          Var  of var
| Conn of (operator*(formula list))


        val widerspruch : (truval*var)*(truval*var) -> bool

        exception FormulaError of int*string

        val truvalGen   : int -> truval list


        val string2truval   : string -> truval
        val string2operator : string -> operator
        val string2var      : string -> var

        val truval2string   :  truval -> string
        val operator2string :  operator -> string
        val var2string      :  var -> string

        val formula2string :  formula -> string


        val truvalOrder    : truval*truval -> bool
        val operatorOrder  : operator*operator -> bool

        val varEqual       : var*var -> bool
        val varReset       : int -> unit
        val varFull        : unit -> unit
        val varShort        :unit -> unit
    end;
```

```
signature TABLEAU =
    sig
     structure Formula : FORMULA

     datatype tableau_tree =
            Empty
          | SF  of Formula.truval*Formula.formula
          | DNF of (Formula.truval*Formula.var) list list
          | &&  of tableau_tree * tableau_tree
          | ||  of tableau_tree * tableau_tree
          | /// of tableau_tree * (tableau_tree list)


     type tableauTable
     type opTable

     exception TableauLookup
     val emptyTableau  : tableauTable
     val lookupTableau : tableauTable -> Formula.truval  ->
                         (Formula.formula list -> tableau_tree)
     val updateTableau : tableauTable -> Formula.truval  ->
                         (Formula.formula list -> tableau_tree) ->
  tableauTable

      exception OpLookup
      val emptyOp   : opTable
      val lookupOp  : opTable -> Formula.operator  ->
              (int * (Formula.var list) * tableauTable)
      val updateOp  : opTable -> Formula.operator  ->
             (int * (Formula.var list) * tableauTable) ->
      opTable


      exception ExpandError of string

      val |--|   : (Formula.truval * Formula.var) list list *
                   (Formula.truval * Formula.var) list list
           ->  (Formula.truval * Formula.var) list list

      val |++|   : (Formula.truval * Formula.var) list list *
                   (Formula.truval * Formula.var) list list
                   ->  (Formula.truval * Formula.var) list list

      val expand : tableau_tree -> tableau_tree

      val &&&    : tableau_tree list * tableau_tree list -> tableau_tree list
      val |||    : tableau_tree list * tableau_tree list -> tableau_tree list

      val get_table: unit -> opTable
      val set_table: opTable -> unit
end
```

```
signature TABDEF =
    sig
        type var
        type truval
        type operator
        type formula
        type tableau_tree

        exception TabDefError of string

        val fmls2fmls:  (formula list) -> (formula list)

        val defs2rules: (operator * (int * (var list) * formula)) list  ->
                        unit

        val tabs2rules: (operator *
                         (int * (var list) *
                          ((truval*tableau_tree) list))) list  -> unit
    end;
```

```
signature TABLEAU_WALK =
    sig
        type tabwalk
        type tableau_tree

        val init_walk : tableau_tree -> tabwalk

        val left  : tabwalk -> tabwalk
        val right : tabwalk -> tabwalk
        val up    : tabwalk -> tabwalk
        val down  : tabwalk -> tabwalk

        val actual      : tabwalk -> tableau_tree
        val up_to_top   : tabwalk -> tabwalk
        val map_actual  : (tableau_tree -> tableau_tree) -> tabwalk -> tabwalk
    end;
```