Seminar Wissensbasierte Systeme: Semantik von ProMD

Matthias Mann

4. Dezember 1995

1 Einführung

In diesem Vortrag, soll es darum gehen, die Grundlagen regelbasierter Systeme kennenzulernen, indem die Semantik von Pro.M.D., einem Expertensystem für medizinische Diagnostik, vorgestellt wird.

Da bis jetzt *fallbasierte* Systeme im Mittelpunkt des Seminars standen, liegt die Motivation, etwas über *regelbasierte* Systeme zu erfahren, auf der Hand: Schließlich kann und sollte man einen neuartigen Ansatz im Vergleich zur „althergebrachten" Vorgehensweise betrachten.

Die Wortwahl war durchaus Absicht: Die ersten Expertensysteme, die in den 70er Jahren entwickelt wurden, stützten sich auf Wissensbasen, die vornehmlich aus Regeln und Fakten bestanden (siehe auch [1], „Expertensysteme"). Die *Regeln* besitzen ganz allgemein gesagt eine Form wie „Wenn Bedingung Dann Konsequenz". *Fakten* können damit als Regeln, deren Bedingung wahr ist, betrachtet werden.

Ein regelbasiertes Expertensystem geht dann vom Prinzip her *deduktiv* vor, indem es aus den gegebenen Regeln (einschließlich Fakten!) versucht, eine *Anfrage* abzuleiten. Die Darstellung der Wissensbasis und die Verwendung logischer Schlußtechniken stellten für die damaligen Programmiersysteme neuartige Probleme dar, die unter anderem zur Entwicklung von *Prolog* [1] führten, das die verwendeten Techniken besonders unterstützt. So ist es denn auch kein Wunder, daß das Expertensystem Pro.M.D. in Prolog implementiert wurde.

Trotzdem kann oder sollte man die regelbasierten Expertensysteme nicht nur historisch als Vorläufer der logischen Programmierung sehen. Denn Regeln und Fakten finden sich ja auch in logischen Programmen, z.B. in Prolog wieder und erlangen dort erst ihre theoretische Grundlage:

Wie bereits erwähnt, stellen Regeln im Prinzip nichts anderes als logische Implikationen dar. Eine Regel, deren Konsequenz B gültig ist, wenn die Prämissen $A_1 \ldots A_n$ allesamt gültig sind, können wir also schreiben als

$$B \Leftarrow A_1 \wedge \ldots \wedge A_n$$

Da Fakten nichts anderes als Regeln mit gültigen Prämissen sind, sehen sie folgendermaßen aus:

$$B \Leftarrow$$

An das System gestellte Anfragen werden dann als

[1] PROLOG steht für „PROgramming in LOGic".

$$\Leftarrow A_1 \wedge \ldots \wedge A_n$$

interpretiert. Normalerweise versucht nun das System auf die Anfrage hin, aus den Fakten und Regeln, die leere (Klausel[2]–) Menge herzuleiten [3]. Diese Sichtweise trifft auf unser Beispiel Pro.M.D. nicht ganz zu; weshalb, werden wir nach einem genaueren Blick auf die Syntax erkennen.

2 Syntax

Wir beschreiben die Syntax einer Untermenge von Pro.M.D. durch eine kontextfreie Grammatik in BNF:

```
<program>  ::=  IF <cond> THEN <action> (ε | <program>)
   <cond>  ::=  <b-expr>
 <b-expr>  ::=  NOT <b-expr> | <b-expr> <b-op> <b-expr> |
                F | T | UNKNOWN | <c-expr> | <b-var> |
                "("known? <b-expr>")" |
                "("known? <n-expr>")"
   <b-op>  ::=  AND | OR | EQUIV | IMPL
 <c-expr>  ::=  <n-expr> <n-cmp> <n-expr>
  <n-cmp>  ::=  GT | GE | EQ | LE | LT
 <n-expr>  ::=  <numerals> | <n-expr> <n-op> <n-expr> |
                UNKNOWN | <n-var>
   <n-op>  ::=  + | - | * | /
 <action>  ::=  <b-var> := <b-expr> | <n-var> := <n-expr>
```

Hiebei sollen `<numerals>` die üblichen Numerale sein, und `<b-var>` sowie `<n-var>` Bezeichner für — boolsche und numerische — Variablen.

Ein Pro.M.D. Programm ist also eine Folge von ein oder mehreren Regeln `IF <cond> THEN <action>`. Wir sollten nun unser Hauptaugenmerk auf die `<action>` legen. Denn diese besteht ja aus der Zuweisung eines Ausdrucks an eine Variable. Ob boolscher oder numerischer Ausdruck spielt hier erst einmal keine Rolle. Wichtig ist, daß wir es in Pro.M.D. mit *globalen Variablen* zu tun haben. Deren Wert ist entweder eine Zahl, wahr oder falsch oder, falls noch kein Wert ermittelt werden konnte, `UNKNOWN` [4]. An dieser Stelle liegt dann der Unterschied zur deklarativen Programmierung, die vom Prinzip her auf Prädikatenlogik basiert.

---

[2] Im Prinzip das gleiche wie Regel.

[3] Stichwort: Resolution mittels Unifikation, siehe dazu auch [3].

[4] → dreiwertige Logik.

2.1 Beispiel

Also schauen wir uns erst einmal ein Beispiel an, um überhaupt ein Gefühl für die Art und Weise der Wissensrepräsentation zu bekommen. Folgendes Programmm soll anhand von Ja–Nein–Fragen herausfinden, welches Tier sich der Benutzer gedacht hat [5]:

```
Fledermaus := 1
Wal        := 2
Tiger      := 3
Fisch      := 4
Vogel      := 5
Zongk      := 6

IF saeugetier_frage AND
   fliegen_frage THEN Tier := Fledermaus
IF saeugetier_frage AND
   NOT fliegen_frage THEN land_oder_meersaeuger := T

IF land_oder_meersaeuger AND
   schwimmen_frage THEN Tier := Wal
IF land_oder_meersaeuger AND
   NOT schwimmen_frage THEN Tier := Tiger

IF NOT saeugetier_frage AND
   schwimmen_frage THEN Tier := Fisch

IF NOT saeugetier_frage AND
   fliegen_frage THEN Tier := Vogel

IF stofftier_frage THEN Tier := Zongk
```

Dabei sollen die ..._frage für Variablen stehen, die ihren Wahrheitswert durch den Benutzer erhalten. Das kann geschehen, indem das System für jede dieser Variablen *einmal* eine Ja–Nein–Frage stellen darf, z.B. „Ist das Tier ein Säugetier?“

Nehmen wir einmal an, der Benutzer hätte die eben genannte Frage positiv und noch weitere Fragen wie folgt beantwortet:

[5]Um die Tiernamen quasi als symbolische Konstanten benutzen zu können, werden ihnen Zahlen zugeordnet. Hierbei soll z.B. `Fledermaus := 1` als Kurzform für `IF T THEN Fledermaus := 1` stehen.

1. „Kann das Tier fliegen“: negativ
2. „Kann das Tier schwimmen“: positiv
3. „Ist das Tier ein Stofftier?“: negativ

Damit seien die entsprechenden Variablen nun mit `T` oder `F` belegt. Das System kann dann die zweite Regel anwenden und lernt dabei, daß `land_oder_meersaeuger` wahr ist. Damit ist die Bedingung für die dritte Regel erfüllt, und das System führt die dazugehörige Aktion aus. Diese belegt die Variable `Tier` mit dem Wert `Wal`.

Faßt man als Ergebnis des Programmes den Inhalt der Variable `Tier` auf, so sieht man sofort, daß durch die Regeln nicht alle Fälle abgedeckt werden; antwortet man z.B. auf alle Fragen mit „Nein“, so wird die Variable `Tier` nie mit einem Wert belegt, sie ist dann eben `UNKNOWN`.

2.2 „conflicting facts“ und Ausführungsreihenfolge

Weiter kann man erkennen, daß das Programm nicht für alle möglichen Antworten auf die Fragen gewappnet ist. Gibt man nämlich auf die 3. Frage positive Antwort, so kann sowohl die letzte Regel ausgeführt werden, die `Tier` mit `Zongk` belegt, als auch wie zuvor schon die zweite und die dritte Regel, wobei `Tier` den Wert `Wal` erhält. Von Pro.M.D. wird dieser Fall nicht zugelassen, es betrachtet ihn als Programmierfehler und hält mit der Fehlermeldung *conflicting facts* an. Das ist ganz einleuchtend; denn das Tier kann *nicht zugleich* `Wal` und `Zongk` sein. Wir erkennen dadurch, daß das gegebene Programm in der Tat fehlerhaft ist. Die Regel

```
IF stofftier_frage THEN Tier := Zongk
```

muß ergänzt werden zu

```
IF NOT saeugetier_frage AND
   stofftier_frage THEN Tier := Zongk
```

Im Prinzip hat uns nun Pro.M.D. mit seiner Fehlermeldung gezwungen, unser Programm so umzuschreiben, daß wir *nie* [6] eine Variable mit zwei unterschiedlichen Werten belegen werden. Das ist auch gut so: Würde uns Pro.M.D nämlich nicht dazu zwingen, so hätte unser zuerst angegebenes, fehlerhaftes Programm die Eigenschaft, `Zongk` *oder* `Wal` als Ergebnis von

[6] Zumindest im Rahmen der vorhandenen vier Tierfragen.

`Tier` auszugeben, *je nachdem in welcher Reihenfolge* die Regeln durchlaufen werden. Umgekehrt scheint es intuitiv so zu sein, daß das Konzept der *statischen* Variablen [7] zu einer Unabhängigkeit des Programmes von der Ausführungsreihenfolge der Regeln führt. Ob die Bedingung statischer Variablen nicht nur notwendig, sondern auch hinreichend für die Unabhängigkeit der Ausführungsreihenfolge ist, können wir nur klären, wenn wir eine Semantik von Pro.M.D. entwickeln.

3 Semantik

3.1 Was ist Semantik?

Eine Semantik entwickeln heißt nun, die Bedeutung von Pro.M.D.–Programmen zu formalisieren; die zur Sprache von Pro.M.D. gehörenden Zeichenketten mit einem Sinn zu füllen. Die Syntax im Abschnitt 2 hatte uns ja nur angegeben, *wie* ein korrektes Pro.M.D.–Programm *auszusehen* hat, aber nicht, *welches Ergebnis* es liefert.

3.2 Arten von Semantiken

Wie kann man nun die Semantik eines Programmes betrachten? Folgende Arten sind dabei möglich:

- axiomatische Semantik
- operationale Semantik
- denotationale Semantik

Axiomatische Semantiken gehen mit Hilfe der Prädikatenlogik vor. Aufgrund der oben schon genannten Abweichung Pro.M.D.'s von deklarativer Programmierung und Prädikatenlogik werden wir sie hier nicht benutzen. Als weiteres gutes Beispiel für ein Anwendungsgebiet von axiomatischen Semantiken seien formale Korrektheitsbeweise von Programmen, z.B. mit den Methoden von *Hoare* bzw. *Dijkstra*, genannt.

[7] Damit ist gerade gemeint, daß nur eine unbelegte bzw. mit `UNKNOWN` belegte Variable einen Wert erhalten, und dann keinen *anderen* (wohl aber denselben) mehr zugewiesen bekommen darf.

3.3 Operationale Semantik für Pro.M.D.

Eine operationale Semantik haben wir unbewußt schon kennengelernt, als wir das Beispielprogramm erklärten. Denn in einer operationalen Semantik versuchen wir, eine konkrete Berechnungsvorschrift, wie wir aus der Eingabe effektiv durch eine Folge von Rechenschritten die Ausgabe erzeugen, also einen Algorithmus zur Auswertung unseres Programmes, anzugeben. Intuitiv hatten wir sogar einen solchen Algorithmus angewendet, als wir das Beispielprogramm in Gedanken ausführten. Nur, anstatt zu sagen: „Die Bedingung der dritten Regel ist wahr, also führen wir die dazugehörige Aktion aus“, hätten wir genauer spezifizieren müssen, wie wir eine auszuführende Regel finden, z.B. folgendermaßen:

1. Initialisierung: Markiere die erste Regel und setze `flag := t`.
2. Nimm die erste Regel aus der Wissensbasis.
3. Falls es die makierte Regel ist:
   (a) Falls `flag = t`: Setze `flag := f`.
   (b) Ansonsten: Stop.
4. Falls die Bedingung der Regel wahr ist:
   (a) Falls die Aktion die Variable ändert: Führe die Aktion aus und setze `flag := t`.
   (b) Ansonsten: Setze die Regel wieder ans Ende der Wissensbasis.
5. Ansonsten: Setze die Regel wieder ans Ende der Wissensbasis.
6. Fahre mit dem 2. Schritt fort.

Den obigen Algorithmus könnten wir fast verwenden, um einen Interpreter für Pro.M.D.–Programme zu schreiben. Wir hätten aber auch einen anderen Algorithmus angeben können, indem wir zum Beispiel die Regeln in *umgekehrter* Reihenfolge durcharbeiteten. In diesem Fall handelt es sich aber auch um eine *andere* operationale Semantik. Für *jede* mögliche Reihenfolge erhalten wir im Prinzip *eine* operationale Semantik.

Damit sind wir nun wieder beim Thema, nämlich der Ausführungsreihenfolge. Was wir erreichen wollten, ist ja, daß die Reihenfolge, in der die Regeln ausgeführt werden, keine Rolle spielt. Wir müssen also zeigen, daß all diese möglichen operationalen Semantiken gleich sind.

Es bleibt die Frage, wie wir diese Aufgabenstellung am besten lösen könnten. Wir werden im folgenden versuchen, einem Pro.M.D.–Programm ein abstraktes mathematisches Objekt zuzuordnen, welches unabhängig von der Anordnung der Regeln im Pro.M.D.–Programm sein wird. Einen solchen Zugang nennt man *denotationale* Semantik.

3.4 Denotationale Semantiken

Zu diesem Zweck beschäftigen wir uns erst einmal allgemein mit denotationalen [8] Semantiken.

Die oben erwähnte Zuordnung bezeichnet man auch als *semantische Funktion*; sie ordnet jedem syntaktischen Konstrukt eine Bedeutung zu. Zusammen mit der Syntax spricht man dann von einem *interpretierten System*. Dafür wollen wir einmal ein Beispiel angeben. Sei $\Sigma = \{0, 1\}$. Wir definieren eine Sprache L durch $L := \Sigma^*$. Als semantische Funktion $I : L \to \mathbb{N}$ könnten wir nun z.B. folgende angeben:

$$\begin{aligned} I(\varepsilon) &:= 0 \\ I(0) &:= 0 \\ I(1) &:= 1 \\ I(uv) &:= 2 \cdot I(u) + I(v) \quad \text{für } u \in L \text{ und } v \in \Sigma \end{aligned}$$

Wir interpretieren also Strings über $\{0, 1\}^*$ durchaus wie gewöhnlich als Binärdarstellung der natürlichen Zahlen [9]. Als interpretiertes System können wir dann $S = (\Sigma, L, I, \mathbb{N})$ angeben.

Das gegebene Beispiel ist aber sehr einfach und verdeckt daher den Unterschied zur operationalen Semantik. Die angegebene Funktion I liefert nämlich sozusagen eine Berechnungsvorschrift gleich mit. Wir schauen uns daher noch ein anderes Beispiel an, diesmal sei folgende rekursive Definition einer Funktion `f` gegeben:

```
f x   =   if x == 0 then 1
          else (if x == 1 then (f 3)
                else (f (x - 2)))
```

[8] *to denote (engl.)* – bezeichnen, bedeuten.

[9] Dabei ist selbstverständlich ein Unterschied zu machen zwischen der „0“ auf der linken Seite z.B. in „I(0)“ — die ist nämlich ein String — und der auf der rechten Seite — die ist eine natürliche Zahl!

Wir erkennen schnell, daß bei „wörtlicher“ Implementierung wie oben angegeben, die Funktion `f` für ungerade `x` nicht terminieren würde. Nichtterminierung interpretieren wir ja allgemein als undefiniert, weshalb wir behaupten können, obiges Programm berechnete folgende Funktion:

$$f(x) = \begin{cases} 1 & \text{falls } x \text{ gerade} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Aber auch die Funktion

$$f_1(x) = 1$$

widerspricht nicht der rekursiven Definition. Daher stellt sie genauso eine Lösung dar wie sogar die Funktionenschar:

$$f_a(x) = \begin{cases} 1 & \text{falls } x \text{ gerade} \\ a & \text{sonst} \end{cases}$$

Nun haben wir mehrere Lösungen zur Auswahl. Intuitiv ziehen wir natürlich die erste vor, da sie uns auch zuerst eingefallen war. Die letzten beiden erscheinen ein bißchen seltsam, weil sie den Kern des Problems nicht ganz treffen. In der Tat ist es ja so, daß uns eigentlich nur die Definition für gerade `x` interessiert. Die Anzahl der Möglichkeiten für ungerade `x` ist uns bei der letzten Definition schlicht zu groß, und anderseits ist jede einzelne Lösung zu speziell. Wir wählen also die erste, weil sie die allgemeinste Lösung darstellt. Allgemeinste Lösung ist hier gleichbedeutend mit der Lösung, die im *Informationsgehalt* am schwächsten ist. Um unsere Lösungen nun überhaupt bezüglich ihres Informationsgehaltes vergleichen zu können benötigen wir eine *Ordnung*.

3.4.1 Ordnungen und Verbände

Definition 3.1: Eine Relation $R \subseteq M \times M$ heißt *partielle Ordnung* genau dann wenn R folgende Eigenschaften besitzt:

- Reflexivität: $\forall a \in M : \; aRa$
- Transitivität: $\forall a, b, c \in M : \; ((aRb \wedge bRc) \Rightarrow aRc)$
- Antisymmetrie: $\forall a, b \in M : \; ((aRb \wedge bRa) \Rightarrow a = b)$

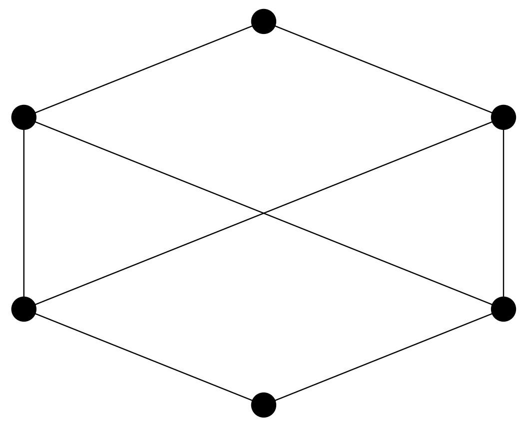

Abbildung 1: Hasse–Diagramm einer partiellen Ordnung

Man spricht dann auch von M als *partiell geordnete Menge.*

In Abbildung 1 ist eine partielle Ordnung zu sehen, graphisch dargestellt durch ein sogenanntes *Hasse–Diagramm.* Gewöhnlich schreibt man für eine partielle Ordnung „$\leq$". Übernehmen wir diese Notation so können wir wir die Lösungsfunktionen aus dem vorigen Beispiel anordnen als $f \leq f_a \forall a$ bezüglich des Informationsgehaltes. Das geschieht an dieser Stelle aber noch rein intuitiv; wir haben aber noch keine Vorstellung davon, wie wir den Informationsgehalt *berechnen* könnten.

Eine partielle Ordnung ist hierfür als Modell zu schwach. Wir benötigen eine Ordnung, in der wir von Teilmengen *eindeutig* bestimmen können, welche Elemente zwar einerseits ausreichend Information enthalten anderseits aber möglichst schwach in ihrem Informationsgehalt sind. Die mathematische Struktur, welche dies gewährleistet ist ein *Verband.*

Definition 3.2: Sei M eine partiell geordnete Menge. Dann definieren wir für eine Teilmenge $N \subseteq M$ von M:

- Die Menge der oberen Schranken [10] von N durch $UB(N) := \{x \in M \mid \forall y \in N : \ y \leq x\}$
- Die Menge der kleinsten oberen Schranken [11] von N durch $LUB(N) := \{x \in UB(N) \mid \forall y \in UB(N) : \ x \leq y\}$

[10] *upper bounds.*

[11] *least upper bounds.*

- Die Menge der unteren Schranken [12] von N durch $LB(N) := \{x \in M \mid \forall y \in N: \; x \leq y\}$
- Die Menge der größten unteren Schranken [13] von N durch $GLB(N) := \{x \in LB(N) \mid \forall y \in LB(N): \; y \leq x\}$

Gilt nun für jede endliche Teilmenge $N \subseteq M$ von M, daß

- $\exists \; LUB(N)$ und $|LUB(N)| = 1$
- $\exists \; GLB(N)$ und $|GLB(N)| = 1$

dann heißt $(M, \leq)$ ein (vollständiger) *Verband* und wir schreiben $lub(N)$ bzw. $glb(N)$ für die eindeutig bestimmte kleinste obere bzw. größte untere Schranke von N.

Ein Beispiel für einen Verband ist in der Abbildung 2 zu sehen. Dort werden alle Teilmengen der Menge $\{a, b, c\}$ bezüglich $\subseteq$ geordnet. Wir erkennen, daß sogar ein *größtes* und ein *kleinstes* Element existieren, nämlich $\{a, b, c\}$ bzw. $\{\}$.

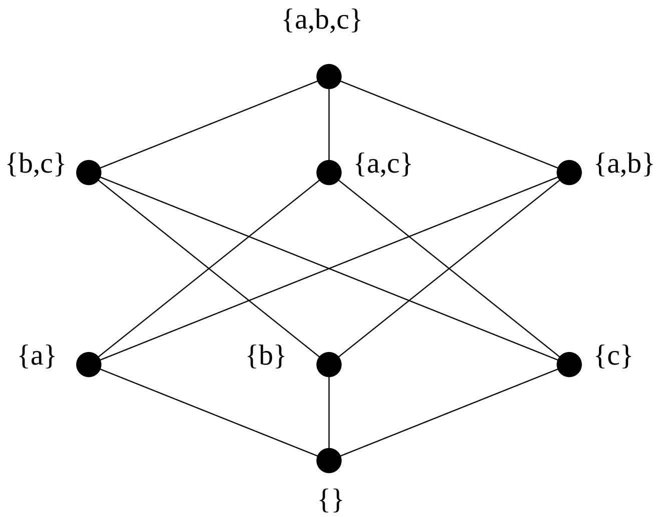

Abbildung 2: Hasse–Diagramm eines Verbandes

Dies ist bei jedem Verband so, aber die Umkehrung gilt nicht, wie ein Blick auf die partielle Ordnung aus Abbildung 1 zeigt. In Abbildung 3 ist

[12] *lower bounds.*
[13] *greatest lower bounds.*

diese noch einmal zu sehen: Dabei existieren zwar auch größtes und kleinstes Element nicht aber die $lub(N)$ der dort eingezeichneten Teilmenge N.

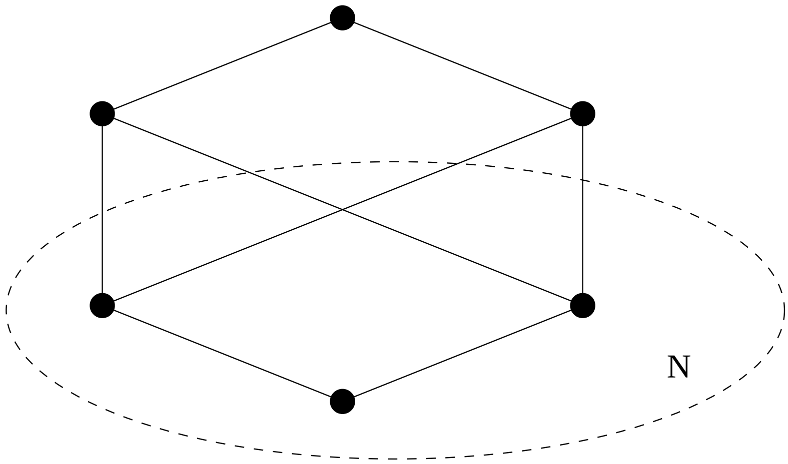

Abbildung 3: Größtes und kleinstes Element machen noch keinen Verband

Wir bezeichnen größtes bzw. kleinstes Element oft der Einfachheit halber als „Top" $\top$ bzw. „Bottom" $\perp$. Benutzen wir nun einen Verband zur Darstellung von Informationsgehalten bzw. von Wissen, so interpretieren wir $\perp$ als kein Wissen und $\top$ als widersprüchliches Wissen.

3.4.2 Verbände als semantische Wertebereiche

Das leuchtet ein, wenn wir uns erst einmal die Ordnung vorstellen, die außer dem kleinsten Element $\perp$ nur noch die logischen Werte `true` und `false` enthält. Dann nämlich soll weder `false` $\leq$ `true` noch `true` $\leq$ `false` [14] gelten. Anderseits ist aber $\perp \leq$ `true` und $\perp \leq$ `false`. Diese Ordnung ist in Abbildung 4 zu sehen.

Wir könnten eine solche Ordnung z.B. dazu verwenden, die möglichen Werte einer boolschen Variablen darzustellen. Übertragen auf Pro.M.D. hieße das, daß eine boolsche Variable entweder `F`, `T` oder `UNKNOWN` sein kann. Damit könnten wir schon die `land_oder_meersaeuger` sowie die `..._frage`–Variablen darstellen. Nehmen wir außerdem noch die ganzen Zahlen hinzu, so können wir auch den Wert der `Tier`–Variablen repräsentieren.

Da sich das gesamte Wissen eines Pro.M.D–Programmes aber aus den Werten *aller* globalen Variablen ergibt, benötigen wir für jede Variable eine

[14] `true` und `false` sind *nicht vergleichbar* in unserer partiellen Ordnung.

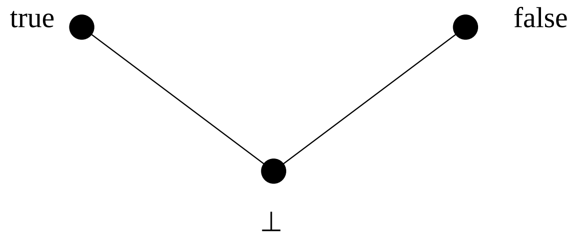

Abbildung 4: Ordnung zur Darstellung der logischen Werte

eigene Ordnung, wie sie oben beschrieben wurde; bezeichnen wir diese mit D. Um jetzt den Informationsgehalt von n Variablen ausdrücken zu können, bilden wir das n–fache kartesische Produkt

$$D^n = \underbrace{D \times D \times \ldots \times D}_{n\text{–mal}}$$

und legen die Ordnung darauf durch

$$(v_1, \ldots, v_n) \leq (w_1, \ldots, w_n) :\Longleftrightarrow \forall (1 \leq i \leq n) : v_i \leq w_i$$

fest. D^n können wir sogar durch die Einführung von $\top$ zu einem Verband ergänzen; dafür schreiben wir dann $(D_n)^\top$. Gilt für $v = (v_1, \ldots, v_n)$ und $w = (w_1, \ldots, w_n)$ aus $(D_n)^\top$, daß $lub(v, w) = \top$, so widersprechen sich die Informationsgehalte aus v und w. Das liefert die Grundlage für die Interpretation von $\top$ als inkonsistentes Wissen.

3.5 Denotationale Semantik für Pro.M.D.

Wir haben also nun eine Struktur kennengelernt, die es uns erlaubt, das in den Variablen eines Pro.M.D.–Programms gespeicherte Wissen zu repräsentieren. Als nächstes sollten wir uns dann überlegen, wie wir die Regeln von Pro.M.D. realisieren, so daß sie die Werte von Variablen beeinflussen können.

Es sei noch einmal daran erinnert, daß die Variablen, die in unseren Programmen verwendet werden, *statisch* sind. Außerdem kann *eine* Regel nur *eine* Variable verändern, daher müssen wir uns den Informationsgehalt unserer n Variablen vor und nach der Ausführung der Regel [15] nur diesbezüglich

[15]Nehmen wir einmal an, daß die Bedingung wahr war; ansonsten passiert ja nichts.

anschauen. Seien $v_1, \dots v_n$ unsere n Variablen, dann können wir die Aktion $v_i := x$ in einer Regel so auffassen, daß das Tupel $(v_1, \dots, v_n)$ durch $(v_1, \dots, v_{i-1}, x, v_{i+1}, \dots v_n)$ ersetzt wird, falls gilt:

$$(v_1, \dots, v_n) \leq (v_1, \dots, v_{i-1}, x, v_{i+1}, \dots v_n)$$

Denn wegen der Forderung nach statischen Variablen dürfen wir ja nur an Information hinzugewinnen. Wir können dies konkreter als Suche nach der *kleinsten oberen Schranke* formulieren, was uns zu folgender Definition führt:

$$\begin{aligned} &IFTHEN_i(cond, t)(w) \\ &\quad := \begin{cases} lub(w, (v_1, \dots, v_{i-1}, t(w), v_{i+1}, \dots v_n) & \text{falls } cond(w) = true \\ w & \text{sonst} \end{cases} \end{aligned}$$

Dabei soll $w = (v_1, \dots, v_n)$ das Tupel unserer n Variablen vor der Ausführung der Regel bezeichnen, $cond : (D_n)^\top \to \{\perp, true, false\}$ eine Funktion, die die Bedingung der Regel zu `UNKNOWN`, `T` oder `F` auswertet [16]. Weiter ist $t : (D_n)^\top \to D$ eine Funktion, die den Wert in der rechten Seite der Aktion $v_i := x$ berechnet [17]. Wir sehen, daß die Bedingung $w \leq IFTHEN_i(cond, t)(w)$ trotzdem notwendig bleibt, damit wir nicht z.B. einer Variablen, die den Wert 1 hat, den Wert 2 zuweisen können, da ja weder $1 \leq 2$ noch $2 \leq 1$ gilt [18]. Gelten für eine Funktion $\varphi : (D_n)^\top \to (D_n)^\top$ folgende Eigenschaften:

$$\begin{aligned} \varphi\top &= \top \\ v \leq w &\implies \varphi v \leq \varphi w \end{aligned}$$

so nennen wir sie *monoton*. Erfüllt sie darüberhinaus, die eben schon genannte Voraussetzung, daß $v \leq \varphi v$, so heißt sie *streng monoton*. In der Tat ist auch die Voraussetzung der Monotonie für $IFTHEN_i$ sinnvoll, da das Wissen eines Programms durch die Anwendung einer Regel nicht größer werden kann, als wenn diese Regel schon auf mehr Wissen angewendet wurde. Außerdem bleibt inkonsistentes Wissen immer inkonsistent. Es ist also so,

[16] Und dabei wenn nötig die Werte aller n Variablen heranzieht, daher als Definitionsbereich $(D_n)^\top$.

[17] Eben wieder, wenn nötig, aus den Werten aller n Variablen.

[18] Ganze Zahlen sind bezüglich des Informationsgehaltes nicht vergleichbar.

daß die Funktionen $IFTHEN_i$ streng monoton sein müssen, um Regeln von Pro.M.D. abzubilden.

Wie sieht es nun mit der Reihenfolge der Ausführung von Regeln aus? Ordnen wir jedem Pro.M.D.–Programm als Semantik eine Menge von Funktionen $IFTHEN_i$ zu, so ist intuitiv einleuchtend, daß die Berechnung der $lub()$ in den $IFTHEN_i$ an keine Reihenfolge gebunden ist und immer das gleiche Ergebnis liefern wird. Dies trifft aber nur soweit zu, als das wir sicherstellen, daß jede Regel die Chance erhält ausgeführt zu werden. Betrachten wir dafür die Menge aller möglichen Informationen, die ein Pro.M.D.–Programm erzeugen kann. Seien die Startwerte — die Eingabe — der n Variablen durch w_0 gegeben und Φ die Menge der Funktionen $IFTHEN_i$, die wir den Regeln zugeordnet haben. Dann bezeichnen wir mit $W = \Phi^*(w_0)$ die Menge, die aus der unendlich häufigen Anwendung aller $IFTHEN_i$ hervorgeht, genauer $w_0 \in W$ und $\forall \varphi \in \Phi : \varphi(w) \in W$ falls $w \in W$. In einer Folge von Anwendungen der Funktionen $\varphi \in \Phi$ auf $w \in W$ *muß* jede Funktion unendlich oft auftreten. Eine solche Folge nennen wir *fair*. Sie stellt sicher, daß — auch bei nur endlich häufiger Anwendung — jede Funktion an die Reihe kommt. Ein Interpreter für Pro.M.D., der diese Eigenschaft nicht erfüllte, wäre auch wenig sinnvoll.

Betrachten wir noch einmal die Menge $W = \Phi^*(w_0)$. Potentiell kann diese unendlich sein, dann würde durch jede Anwendung einer Regel immer wieder Information hinzugewonnen. Dies tritt ein, falls das Programm ein Intervall approximiert und es von Schritt zu Schritt immer wieder ein Stück verfeinert [19]. Betrachten wir nur die logischen Werte und die ganzen Zahlen, so muß wegen der strengen Monotonie aller $\varphi \in \Phi$ die Menge W endlich sein. An irgendeinem Punkt $w \in W$ kann keine Information mehr hinzugewonnen werden, es gilt dann $\varphi(w) = w \forall \varphi$. Wir nennen w einen Fixpunkt für Φ in W. Einen solchen Fixpunkt kann man in einem Interpreter programmtechnisch leicht finden, wie die operationale Semantik für Pro.M.D. gezeigt hat. Das Ergebnis des Programmes stellt das bis dahin in den Variablen gesammelte Wissen dar. Für die Formulierung auf Basis von $\Phi^*(w_0)$ entspricht dies der Definition von $lub(\Phi^*(w_0))$ als Ausgabe des Programmes.

Fassen wir noch einmal zusammen: Die Semantik eines Pro.M.D.–Programmes ist eine Menge streng monotoner Funktionen $IFTHEN_i$. Dabei haben wir im Prinzip die semantische Funktion umgangssprachlich formuliert: Sie bildet jede Regel der Form `IF <cond> THEN` v_i `:= t` auf eine Funktion $IFTHEN_i(cond, t)(w)$ ab. Ein Pro.M.D.–Programm ist dann unabhängig von der Reihenfolge, in der die Regeln ausgeführt werden, vorausgesetzt die

[19]Dazu müssen natürlich Intervalle in $(D_n)^\top$ aufgenommen worden sein.

Reihenfolge ist fair.

4 Weiterführende Betrachtungen

4.1 Nebenläufige Ausführung von Regeln

Wie wir gesehen haben, spielt die Ausführungsreihenfolge von Regeln keine Rolle für das Ergebnis eines Pro.M.D.–Programmes. Es stellt sich nun die interessante Frage, ob Regeln auch *parallel* ausgeführt werden können. Und zwar in einer Art und Weise, die nicht ohne Weiteres sequentialisiert werden kann; also z.B. in einem Mehrprozessorsystem mit lokalem Speicher, in dem jeder Prozessor eine Kopie aller Variablen bearbeitet. In diesem Fall könnten wir nicht unbedingt garantieren, daß die Ausgabe des Programmes noch $lub(\Phi^*(w_0))$ entspricht. Dies liegt daran, daß wir den Prozessoren sinnvollerweise außer der Ausführung einer Regel noch erlauben müssen, mit einem anderen Prozessor zu kommunizieren [20] oder einen neuen Prozessor zu starten. Anderseits müssen wir die Prozessoren „zwingen", daß sie nicht nur diese zusätzlichen Operationen ausführen, sondern auch ausreichend oft eine Regel. Unser Kriterium einer fairen Folge von Regelanwendungen aus Abschnitt 3.5 greift hier nicht mehr.

Faßt man die Kommunikation zwischen zwei Prozessoren als Berechnung der $lub()$ ihres Wissens auf, so kann man Bedingungen für eine *faire parallele Folge* von Regelanwendungen finden. Werden diese Kriterien eingehalten, so bleibt die Semantik des Pro.M.D.–Programmes auch in einer verteilten Umgebung die gleiche.

4.2 Voraussetzungen für strenge Monotonie von Regeln

Die zentrale Anforderung an die Funktionen $IFTHEN_i$, die wir den Regeln eines Pro.M.D.–Programmes zuordnen, war ja die strenge Monotonie gewesen. In Abschnitt 3.5 haben wir dies mit dem Konzept der statischen Variablen in Verbindung gebracht. Die Entsprechung gilt aber nur insoweit, als daß sie den Aktionsteil einer Regel betrifft.

Wie müssen die Funktionen $IFTHEN_i$ nun allgemein gestaltet sein, um streng monoton zu bleiben? Als erste Anforderung scheint die Monotonie der Funktion t, die den Wert für die Zuweisung in einer Aktion berechnet, notwendig zu sein. Denn ist t nicht monoton, so auch $IFTHEN_i$ nicht. Für den Bedingungsteil einer Regel kommt noch hinzu, daß die Funktion *cond* einmal zu wahr evaluiert, immer zu wahr evaluieren sollte. Das heißt, für

[20] Ohne Kommunikation hat ein Mehrprozessorsystem gar keinen Zweck.

$v \leq w$ soll gelten $cond(v) = true \Longrightarrow cond(w) = true$. Diese Eigenschaft nennen wir *wahrheitbewahrend.* Beide Eigenschaften zusammen sind dann schon hinreichend dafür, daß $IFTHEN_i(cond, t)$ streng monoton ist.

Betrachten wir in diesem Zusammenhang das Meta–Prädikat `known?`, so stellen wir fest, daß es zwar wahrheitserhaltend nicht aber monoton ist. `known?` darf also nicht im Aktionsteil einer Regel verwendet werden. Im Gegensatz dazu sind die üblichen arithmetischen und boolschen Operationen monoton. Monotone Funktionen sind außerdem wahrheitserhaltend und darüberhinaus unter gewissen Voraussetzungen auch die Zusammensetzung boolscher Operationen.

4.3 Regeln für Vorgabewerte

Manchmal ist es wünschenswert, für eine Variable einen Vorgabewert angeben zu können. Dieser soll benutzt werden, falls keine Regel, in welcher diese Variable einen Wert zugewiesen bekäme, angewendet werden kann. Man ist versucht, für diesen Fall die Konstruktion `IF NOT (known? (v)) THEN v := t` zu verwenden, mit der Intention „Falls die Variable noch `UNKNOWN` ist, dann weise ihr jetzt einen Wert zu“. Problematisch ist dabei, daß die Funktion $\neg(known?)$ nicht wahrheitsbewahrend ist. Das führt dazu, daß die Reihenfolge von Regelanwendungen nicht mehr unerheblich ist.

Um dies zu vermeiden, muß man Regeln, die Vorgabewerte angeben sollen, in einen separaten Programmteil „verbannen“. Erst wenn die Ausführung der gewöhnlichen Regeln nicht mehr möglich ist, oder keinen Informationsgewinn mehr bringt [21], werden die Regeln aus dem Vorgabeteil ausgeführt. Trotzdem muß man dann sicherstellen, daß sich diese Regeln nicht widersprechen können, z.B. hängt das Ergebnis von

```
IF NOT (known? (v)) THEN v := t1
IF NOT (known? (v)) THEN v := t2
```

wieder von der Ausführungsreihenfolge ab. Lassen wir die Werte aller Variablen fest, während wir den Vorgabeteil auswerten und dort alle Ergebnisse aufsammeln, bevor wir aus diesem gesamten Wissen die $lub()$ berechnen, so werden wir Konflikte der oben geschilderten Art leicht feststellen können. Im Hinblick darauf, daß der Vorgabeteil mit Vorsicht zu genießen ist, kann man sich dafür aussprechen, nach der Abarbeitung desselben das Programm zu beenden [22] und nicht mehr in den gewöhnlichen Teil zurückzukehren.

[21] Also das Programm dort terminiert hat.

[22] Auf diese Art betrachten wir den Vorgabeteil als Einbahnstraße und die Semantik des gewöhnlichen Regelteils bleibt sauber.

Inhaltsverzeichnis

Literatur

[1] Hermann Engesser (Hrsg.): *DUDEN Informatik, Dudenverlag, Mannheim, 1988*

[2] Manfred Schmidt-Schauß: *On the Semantics and Interpretation of Rule Based Programs with Static Global Variables, Fachbereich Informatik der JWG–Universität Frankfurt*

[3] Manfred Schmidt-Schauß: *Einführung in die logische Programmierung, Vorlesung SS 1995, Fachbereich Informatik der JWG–Universität Frankfurt*

[4] Joseph E. Stoy: *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory, MIT Press, Massachusetts, 1977*