Monotone bedingte Verzweigungen in Logikprogrammen

Ulrich Neumerkel
Technische Universität Wien
ulrich@complang.tuwien.ac.at

Stefan Kral
Fachhochschule Wiener Neustadt
stefan.kral@fhwn.ac.at

Zusammenfassung

Einfache bedingte Verzweigungen sind in den meisten Programmiersprachen ein elementares Sprachmittel. In monotonen Logikprogrammen war ihre Verwendung bisher auf nur sehr wenige Bereiche beschränkt, was oft zu unnatürlich komplexen oder inkorrekten Formulierungen führte. Wir stellen ein monotones Kontrollkonstrukt `if_/3` vor, das in vielen Fällen kompakte, korrekte und effiziente Definitionen direkt in ISO-Prolog erlaubt. Dies erreichen wir durch die explizite Darstellung von Wahrheitswerten mit Prädikaten höherer Ordnung. Zudem können auch Constraints unmittelbar einbezogen werden. Insbesondere werden dadurch Programme, die syntaktische Ungleichheit (`dif/2`) verwenden, wesentlich vereinfacht.

1 Einführung

Das Programmieren in purem, monotonem Prolog ist seit jeher ein wenig beachteter Bereich der Logikprogrammierung. Die meisten Prologprogramme bestehen noch immer aus unnötigerweise prozeduralen Elementen. Mit ein Grund dieses Missstandes sind oft Effizienzüberlegungen, die jegliche deklarative Sichtweise vernebeln. Um für pures Prolog eine brauchbare Programmiermethodik zu entwickeln, benötigen wir Konstrukte, die punkto Effizienz mit den nicht-deklarativen Methodiken vergleichbar sind. Wir richten dabei unsere Aufmerksamkeit ausschließlich auf monotone Elemente, die in vielerlei Hinsicht Vorteile bieten. So können alternative Beweisverfahren, wie etwa *iterative deepening* auf monotone Programme unmittelbar angewendet werden. Ähnlich verhält es sich mit deklarativen Debugging-Techniken und Program Slicing [5, 10]. Ebenso wird der Constraintprogrammierung dadurch ein pureres Umfeld bereitet. Unsere Bemühungen zielen in eine ähnliche Richtung, wie es die Funktionale Programmierung erfolgreich vorgeführt hat: Fort von einer befehlsorientierten Sichtweise hin zum eigentlichen puren Kern des Paradigmas.

Viele Entwicklungen der letzten Jahre haben dazu beigetragen einen deklarativeren Programmierstil zu fördern. Ein besonders großer Fortschritt war die Klärung von Prologs ISO-Norm [8] durch Cor.2:2012 [13], das Laufzeitfehler semantisch abgeglichen hat [12] und das insbesondere die Programmierung Höherer Ordnung über `call/N` auf feste, normative Beine stellte und damit weitergehende Verwendungen erlaubte [11]. Während das Konstrukt 1982 in ersten vagen Formen sondiert wurde [4] und bereits 1984 zum ersten Mal vorgeschlagen wurde [6], benötigte die präzise Definition offenbar ihre Zeit, um allgemein anerkannt und kodifiziert zu werden. Leider sind jedoch viele nichtdeklarative Konstrukte derzeit noch alternativlos. Ihre Verwendung für deklarative Zwecke ist zwar prinzipiell möglich, aber der Aufwand um dabei noch im puren Bereich zu verbleiben ist so hoch, dass er kaum in Kauf genommen wird.

Wir betrachten dazu zunächst die Schwächen von Prologs if-then-else Konstrukt und wenden uns dann jenen des Prädikats `member/2` zu, um daraufhin verbesserte Fassungen dieses Prädikats vorzustellen, die zur Einführung von Reifikation und letztlich eines monotonen if-then-else führen. Zu guter Letzt betrachten wir einige Beispiele zur allgemeinen Reifikation, die vor allem die Unterschiede zur konstruktiven Negation hervorheben.

2 Die Grenzen von Prologs if-then-else

Prologs if-then-else Konstrukt wurde erstmals um 1978 im Interpreter des DEC10 Prologsystems implementiert [3], es wurde aber nicht durch den Compiler unterstützt. Erst spätere Systeme wie etwa Quintus Prolog verfügten über eine effiziente Implementierung, die schließlich zur Aufnahme in die ISO-Norm führte.

Wir betrachten für unsere Zwecke lediglich Ziele der Form `( If_0 -> Then_0 ; Else_0 )`, wobei alle Teile einfache Ziele vor- oder benutzerdefinierter Prädikate sind. In diesem Fall ist die Ausführung dieses Ziels äquivalent mit `( once(If_0) -> Then_0 ; Else_0 )`. Es wird also die erste Antwort von `If_0` genommen, weitere Antworten werden ignoriert. Weiters kommt `Else_0` nur dann in Betracht, wenn `If_0` scheitert. Das Konstrukt ist geeignet, um Prologs Negation zu implementieren, andererseits bedeutet das auch, dass zumindest dieselben Probleme auftreten: Nichtkommutativität der Konjunktion sowie Nichtmonotonie.

Weitere Verbesserungsversuche sind das sogenannte *soft cut*, das alle Antworten von `If_0` betrachtet. Es wird von einigen Systemen als `if/3` oder `(*->)/2` angeboten. Dadurch wird zwar die willkürliche Beschränkung auf die erste Antwort aufgehoben, und damit die Menge an Lösungen vollständiger und Antworten unabhängig von der Reihenfolge der Klauseln — modulo Termination und Fehler. Die eigentlichen, tieferliegenden Probleme der Nichtkommutativität und Nichtmonotonie bleiben jedoch bestehen.

Um die bisherigen Konstrukte sicher verwenden zu können, kommen für `If_0` nur Ziele in Betracht, die selbst zusichern, dass sie hinreichend instanziert sind. So etwa Prologs Arithmetikprädikate, die in Arithmetikausdrücken keine Variable zulassen, indem sie Instanzierungsfehler melden. Es gibt aber nicht viele weitere Prädikate, die sich in ähnlich sicherer Weise verwenden lassen. Insbesondere kommen Constraints dafür nicht in Frage. Darüber hinausgehende Ziele für `If_0` können nur in speziellen, kaum dokumentierten und ungeprüften Modi verwendet werden.

3 Die Schwächen von `member/2`

Auch an sich pure Definitionen weisen problematische Eigenschaften auf. Wir erläutern dies anhand von `member/2`, das für ein Ziel `member(X, Es)` wahr ist, wenn `X` Element der Liste `Es` ist.

```
member(X, [X|_Es]).
member(X, [_E|Es]) :-
   member(X, Es).
```

Die bekannte Relation ist etwas zu allgemein gefasst, da sie auch für Nicht-Listen erfüllt ist, die einen Listenpräfix mit dem passenden Element besitzen. So gilt etwa `member(a, [a|non_list])`. Derartige Verallgemeinerungen werden in Prolog jedoch gern in Kauf genommen, weil man sich dadurch eine effizientere Ausführung erwartet. Für die erste Antwort muss nur der Anfang der Liste bis zum ersten passenden Element betrachtet werden. Allerdings wird bei Wiedererfüllung dann doch noch die gesamte Liste betrachtet. Man bleibt also trotz Verallgemeinerung auf den Kosten zum Besuch der gesamten Liste sitzen. Dies ist einfach schon dadurch begründet, dass die verbleibende Liste ja tatsächlich noch ein weiteres passendes Element besitzen könnte.

```
?- member(1, [1,2,3,4,5]).
   true
;  false.
```

```
?- member(1, [1,2,1,4,5]).
   true
;  true
;  false.
```

Ein Ziel `member/2` wird praktisch nie deterministisch sein und wird für die gesamte Dauer des Beweises Platz benötigen. Es ist naheliegend sich in dieser Situation nicht-deklarativer Hilfsmittel zu bedienen. Man beschränkt sich etwa auf die erste Antwort — ungeachtet der dadurch bedingten Unvollständigkeit. Ein weit verbreitetes Bibliotheksprädikat dazu ist `memberchk/2`.

```
memberchk(X, Es) :-
   once(member(X, Es)).
```

```
?- X = 2, memberchk(X, [1,2]), X = 2.
   X = 2.

?-        memberchk(X, [1,2]), X = 2.
   false.    % unerwartetes Scheitern
```

Offenbar wird so nicht nur die Monotonieeigenschaft verletzt, es gibt nun überhaupt keine deklarative Erklärungsmöglichkeit mehr. Als einzige Erklärung verbleibt das schrittweise Nachvollziehen des prozeduralen Ablaufs. Häufig wird diese Unzulänglichkeit kaschiert, indem man nur `memberchk/2`-Ziele zulässig erklärt, die hinreichend instanziert (*sufficiently instantiated*) sind, ohne sich allerdings auf eine genaue Definition dieses Kriteriums festzulegen und ohne Rückmeldung in Form eines Laufzeitfehlers. Man kann sich also nur bei variablenfreien Zielen sicher sein, dass `memberchk/2` korrekt verwendet wird.

4 Ein grundüberholtes `member/2`

Das eigentliche Problem von `member/2` ist nicht sosehr Prologs Ausführungsmechanismus als die ursprüngliche Definition selbst, die bei einer gefundenen Antwort noch weitere redundante Antworten zulässt. Durch die folgende äquivalente, alternative Formulierung tritt die Ursache der redundanten Antwort, die durch die erste subsumiert wird, etwas deutlicher zutage. Die erste Alternative `X = E` wird in der zweiten nicht ausgeschlossen. Es ist also gut möglich, dass `X = E` auch für die zweite Alternative gilt. Genau an dieser Stelle muss also die Ungültigkeit von `X = E` und damit die syntaktische Ungleichheit der Terme zugesichert werden. In der neuen Definition `memberd/2` verwenden wir dazu das zu Anbeginn in Prolog 0 [1] vorhandene und leider ab Prolog I [2] für lange Zeit vergessene Prädikat `dif/2`, welches bisher nicht in die ISO-Norm von Prolog aufgenommen wurde. Für unsere Zwecke ist es unerheblich, ob `dif/2` wie vorgesehen Constraints verwendet, oder durch ISO-konforme Instanzierungsfehler falsche Antworten meidet. Wird `dif/2` nicht als implementierungsspezifische Erweiterung der ISO-Norm [8] bereitgestellt, genügt die Definition im Anhang.

```
member(X, [E|Es]) :-
   (  X = E
   ;  member(X, Es)
   ).
```

```
?- member(1, [1,X]).
   true
;  X = 1. % redundante Antwort
```

```
memberd(X, [E|Es]) :-
   (  X = E
   ;  dif(X, E),
      memberd(X, Es)
   ).
```

```
?- memberd(1, [1,X]).
   true
;  false. % nichtdet. Scheitern

?- memberd(1, [1,2,3]).
   true
;  false. % nichtdet. Scheitern
```

In dieser Formulierung von `memberd/2` treten nun keine redundanten Antworten mehr auf, dennoch verbleiben überflüssige Wahlpunkte, die **nichtdeterministisches Scheitern** verursachen und durch `; false` angezeigt werden. Dieses Problem tritt oft auf, wenn man versucht, mittels `dif/2` pure Programme zu definieren.

5 Reifizierung der Gleichheit

Das Problem dahinter liegt hier darin, dass durch unterschiedliche Alternativen miteinander zusammenhängende Fälle beschrieben werden. In einer Alternative ist Gleichheit und in der anderen Ungleichheit derselben Terme beschrieben. Durch Komprimierung dieser beiden Fälle in ein einziges

Prädikat `(=)/3` hat nun eine Implementierung die Freiheit, Wahlpunkte, so möglich, zu vermeiden. Im Anhang findet sich dazu eine normkonforme Definition, die bereits viele überflüssige Wahlpunkte sofort entfernt.

```
memberd(X, [E|Es]) :-
   =(X, E, T),
   (  T = true
   ;  T = false,
      memberd(X, Es)
   ).
```

```
=(X, X, true).
=(X, Y, false) :-
   dif(X, Y).
```

Diese Implementierung ist noch etwas verbesserungswürdig. Einige Prologsysteme sind nicht in der Lage die Disjunktion effizient zu implementieren. Zudem sollten auch fehlerhafte Werte für `T` erkannt werden. Weiters ist die Verwendung der Hilfsvariable `T` zur Darstellung des Wahrheitswertes besonders fehleranfällig. All diese Probleme werden wir durch einen neuen Ansatz lösen.

6 Das monotone `if_/3`

Die angeführten Probleme lassen sich allesamt durch ein neues Prädikat `if_(If_1, Then_0, Else_0)` beheben. Die Bedingung wird nun nicht mehr durch ein einfaches Ziel dargestellt, sondern durch ein partielles Ziel eines reifizierten Prädikats, dem noch ein weiteres Argument zum vollständigen Ziel fehlt. Auf diese Art wird die Hilfsvariable für den Wahrheitswert versteckt. Damit gelangen wir nun zur endgültigen Fassung von `memberd/2`:

```
memberd(X, [E|Es]) :-
   if_( X = E
      , true
      , memberd(X, Es)
      ).
```

```
?- memberd(1, [1,X]).
   true.

?- memberd(1, [1,2,3]).
   true.
```

Diese Fassung vermeidet bereits überflüssige Wahlpunkte. Weitere Optimierungen sind durch partielle Auswertung möglich, um sämtliche Metacalls in `if_/3` durch Ziele zu ersetzen.

Die Wahl, ein dreistelliges Prädikat zu verwenden und nicht mehrere binäre Operatoren, wie etwa bei Prologs if-then-else, war vor allem den semantisch sehr problematischen Nebeneffekten von if-then-else in ISO-Prolog geschuldet. So überschneidet sich das Konstrukt mit der Disjunktion, beide besitzen denselben äußeren Funktor `(;)/2`. Weiters ist `(->)/2` für sich ein eigenes Konstrukt und führt damit zu einer sehr fragilen Semantik, die oft vom genauen Zeitpunkt abhängt, wann ein Term ein if-then-else Konstrukt beschreibt. Bei einem dreistelligen Prädikat können diese Probleme nicht auftreten. Dafür sind mehrfache Verzweigungen eher umständlich zu schachteln.

Aufbauend auf `if_/3` lassen sich nun viele der üblichen Prädikate höherer Ordnung definieren - wobei nun wesentlich allgemeinere Verwendungen ermöglicht werden.

```
?- tfilter(=(X), [1,2,2], Fs).
   X = 1, Fs = [1]
;  X = 2, Fs = [2, 2]
;  Fs = [], dif(X, 2), dif(X, 1).
```

```
duplicate(X, Xs) :-
   tfilter(=(X), Xs, [_,_|_]).

?- duplicate(X, [1,2,2,1,3]).
   X = 1
;  X = 2
;  false.
```

7 Allgemeine Reifizierung

Bisher haben wir lediglich ein einziges reifiziertes Prädikat verwendet — für syntaktische Gleichheit. Für jede weitere neue Bedingung benötigen wir eine eigene Definition. Es obliegt also dem Programmierer, eine entsprechend reifizierte Fassung eines Prädikats zu erarbeiten. Damit unterscheiden wir uns grundsätzlich von allgemeineren Verfahren, wie etwa der Konstruktiven Negation [7, 9], die mehr oder minder automatisch versucht, die Negation eines Ziels zu bilden. Ein Ansatz [7] hängt von richtig gesetzten delay-Deklarationen ab. Kann also keine unendliche Anzahl von Antworten erzeugen für Prädikate, die als Bedingungen gelten. Jedenfalls sind derartige Techniken verwendbar, um reifizierte Prädikate automatisch zu erzeugen. Effizient werden sie vermutlich jedoch nicht sein, da für den negativen Teil der positive nochmals in einem eigenen, vermutlich meta-interpretierten Ausführungsmodus behandelt wird und da die Gemeinsamkeiten zwischen positivem und negativem Fall nicht geteilt werden können; zumindest also dieser Teil redundant ist. Wir werden dies nun anhand der Reifikation von `memberd/2` erörtern.

```
memberd_t(X, Es, true) :-
   memberd(X, Es).
memberd_t(X, Es, false) :-
   maplist(dif(X), Es).
```

Die Definition besteht für den negativen Fall darauf, dass `Es` eine wohlgeformte Liste ist, deren Elemente alle von `X` verschieden sind. Dies zeigt schon einen klaren Unterschied zur Konstruktiven Negation. Während in unserer Definition `memberd_t(X, non_list, T)` einfach nur scheitert, also weder wahr noch falsch ist, müsste diese bei konstruktiver Negation mit `T = false` erfolgreich sein. Einfach, weil `memberd(X, non_list)` scheitert, muss die Negation also gelten. Hingegen verlangt unsere Definition bis zu einem gewissen Grad einen Listentyp. Es gibt auch Fälle, in denen unsere Definition für Nicht-Listen erfüllt ist. Etwa gilt `memberd_t(1, [1|non_list], true)`. Die genaue Entscheidung, welcher Fall einem Wahrheitswert zugeordnet wird und welcher nicht, lässt sich nur aus rein implementierungstechnischen Erwägungen ersehen. Weder Typsysteme noch konstruktive Negation lassen ein solches Ermessen zu. Es ist diese Freiheit, die uns eine sehr effiziente Implementierung gestattet. Unmittelbar ist unsere Definition nicht sonderlich effizient, vor allem weil beide Klauseln völlig unabhängig voneinander sind, und zur Erzeugung von überflüssigen Wahlpunkten führen. Allerdings können wir nun die Gemeinsamkeiten beider Alternativen herausheben: Beide Alternativen bestehen zumindest auf einer gemeinsamen Präfixliste, welche an Stellen endet, an denen das gesuchte Element `X` vorkommt.

```
memberd_t(X, Es, T) :-
   l_memberd_t(Es, X, T).

l_memberd_t([], _, false).
l_memberd_t([E|Es], X, T) :-
   if_( X = E, T = true, l_memberd_t(Es, X, T) ).

firstduplicate(X, [E|Es]) :-
   if_( memberd_t(E, Es), X = E, firstduplicate(X, Es) ).
```

```
?- firstduplicate(1, [1,2,3,1]).
true.

?- firstduplicate(X, [1,2,2,1,3]).
X = 1.
```

```
?- firstduplicate(X, [A,B,C]).
   X = A, A = B
;  X = A, A = C, dif(C, B)
;  X = B, B = C, dif(A, C), dif(A, C)
;  false.
```

Komplexere Strukturen erschweren die Identifikation des gemeinsamen Teils. Bei einer linearen Liste kann ja nur der gemeinsame Teil aus einem Präfix bestehen, damit gibt es praktisch keine Freiheiten. Bereits ein binärer Baum gibt viel weniger vor, da jene Zweige, die nicht das betrachtete Element enthalten nun je nach Implementierung von Relevanz sind oder nicht. Die erste Fassung fordert hier nur das Minimum, während die verbesserte Fassung die Relation etwas einschränkt.

```
treemember(E, t(E, _, _)).    tree_non_member(_, nil).
treemember(E, t(_, L, R)) :-  tree_non_member(E, t(F, L, R)) :-
   (  treemember(E, L)           dif(E, F),
   ;  treemember(E, R)           tree_non_member(E, L),
   ).                            tree_non_member(E, R).

treemember_t(E, Tr, true) :-
   treemember(E, Tr).
treemember_t(E, Tr, false) :-
   tree_non_member(E, Tr).

?- treememberd_t(2, t(1,non_tree,t(2,non_tree,non_tree)), T).
   T = true.
```

In der folgenden verbesserten Implementierung verwenden wir bereits die reifizierte Disjunktion. Während Disjunktion in purem, monotonem Prolog modulo Nichttermination und Laufzeitfehler kommutativ ist, gilt die Kommutativität der reifizierten Disjunktion nur bedingt.

```
treememberd_t(_, nil, false).
treememberd_t(E, t(F, L, R), T) :-
   call(
     (  E = F
     ;  treememberd_t(E, L)
     ;  treememberd_t(E, R)
     ),
   T).

?- treememberd_t(2, t(1,non_tree,t(2,non_tree,non_tree)), T).
   false.                                  % Einschränkung
?- treememberd_t(2, t(1,     nil,t(2,non_tree,non_tree)), T).
   T = true
;  false.
```

8 Schluss

Wir haben einen neuen, besonders einfachen Ansatz zur monotonen, bedingten Verzweigung vorgestellt, der bereits in seiner ersten Implementierung kostspielige Wahlpunkte effektiv vermeidet. Alle verwendeten Mittel sind zwar schon seit langem bekannt, die konkrete Zusammenstellung gab es bislang jedoch nicht.

Die vorgestellten Programme wurden in den letzten Jahren von den Autoren auf comp.lang.prolog und stackoverflow.com schrittweise entwickelt. Die wesentlichen Eckpunkte waren:

ISO-`dif/2` comp.lang.prolog 2009-10-15
Reification of term equality/inequality, stackoverflow.com/q/13664870 2012-12-01.
`memberd/2` stackoverflow.com/a/21971885 2014-02-23.
`tfilter/3` stackoverflow.com/a/22053194 2014-02-23.
`if_/3` stackoverflow.com/a/27358600 2014-12-09.

Weitere Beispiele finden sich mit `stackoverflow.com/search?q=[prolog]+if_`

Literatur

[1] A. Colmerauer, H. Kanoui, Ph. Roussel, R. Pasero. Un système de communication homme-machine en Français, Rapport de recherche, CRI 72-18. U.E.R de Luminy. Université d'Aix-Marseille. 1972-1973.

[2] Ph. Roussel, Prolog, manuel de référence et d'utilisation. Groupe d'Intelligence Artificielle de Marseille-Luminy, 1975.

[3] L. M. Pereira, F. C. N. Pereira, D. H. D. Warren. User's Guide to DECsystem-10 Prolog. 1978.

[4] D. H. D. Warren. Higher-Order Extensions to Prolog - Are They Needed?, Machine Intelligence 10. 1982. Originally: International Machine Intelligence Workshop, Cleveland, April 1981, DAI Research Paper 154.

[5] M. Weiser. Programmers Use Slices When Debugging. CACM 25(7): 446-452, 1982.

[6] R. O'Keefe. Draft Proposed Standard for Prolog Evaluable Predicates. 1984. Kopie unter: http://www.complang.tuwien.ac.at/ulrich/iso-prolog/okeefe.txt

[7] D. Chan. An Extension of Constructive Negation and its Application in Coroutining. NACLP 1989.

[8] ISO/IEC 13211-1:1995 Programming languages - Prolog - Part 1: General core.

[9] W. Drabent. What is failure? An approach to constructive negation. Acta Informatica 32(1):27-59, 1995.

[10] U. Neumerkel, St. Kral. Declarative program development in Prolog with GUPU. 12th Workshop on Logic Programming Environments (WLPE), Copenhagen 2002.

[11] U. Neumerkel. Lambdas und Schleifen in monotonen Logikprogrammen. KPS 2009.

[12] U. Neumerkel, M. Triska. An error class for unexpected instantiations. ISO/IEC JTC1 SC22 WG17 N226. 2010. http://www.complang.tuwien.ac.at/ulrich/iso-prolog/error_k

[13] ISO/IEC 13211-1:1995/Cor 2:2012. Second Technical Corrigendum for Programming languages - Prolog - Part 1: General core.

Anhang

Alle folgenden Definitionen sind strikt normkonform und benötigen daher keinerlei Constraintmechanismus. Sie finden sich in `library(reif)` für SICStus Prolog und andere Prologsysteme.

Die Definition von `dif/2` ist nur dann erforderlich, wenn ein Prologsystem keine eigene Implementierung von `dif/2` als implementierungsspezifische Erweiterung aufweist. Alle Fälle, in denen keine korrekte Antwort möglich ist, der Constraintmechanismus also benötigt werden würde, werden durch einen Laufzeitfehler angezeigt. Inkorrekte Antworten werden somit vermieden.

`if_/3` besteht auf einen konkreten Wahrheitswert für `T` in `call(If_1, T)`. Im Falle von Nichtdeterminismus sollte `If_1` zuerst `true` und erst danach `false` als Antwort liefern.

`(=)/3` ist bereits völlig deterministisch in allen Fällen, die durch syntaktische Gleichheit und Ungleichheit eindeutig sind. Möglicherweise ist es auch hier interessant, den Wahrheitswert mit dem Fehler `type_error(boolean, T)` abzusichern. Eine effizientere interne Implementierung könnte die bis zu vier Traversierungen der beiden Terme auf bestenfalls eine reduzieren.

`(',')/3` und `(;)/3` sind reifizierte Fassungen der Konjunktion und Disjunktion.

```
dif(X, Y) :-
   X \== Y,
   ( X \= Y -> true ; throw(error(instantiation_error,_)) ).

% :- meta_predicate(if_(1, 0, 0)).

if_(If_1, Then_0, Else_0) :-
   call(If_1, T),
   (  T == true -> call(Then_0)
   ;  T == false -> call(Else_0)
   ;  nonvar(T) -> throw(error(type_error(boolean,T),_))
   ;  /* var(T) */ throw(error(instantiation_error,_))
   ).

=(X, Y, T) :-
   (  X == Y -> T = true
   ;  X \= Y -> T = false
   ;  T = true, X = Y
   ;  T = false,
      dif(X, Y)                              % ISO extension - mit dif/2
      % throw(error(instantiation_error,_)) % ISO strict - ohne dif/2
   ).

',',(A_1, B_1, T) :-
   if_(A_1, call(B_1, T), T = false).

;(A_1, B_1, T) :-
   if_(A_1, T = true, call(B_1, T)).
```