Wenn wir Constraint-Strukturen für arithmetische Operationen verwenden, können wir diese auch flexibel einsetzen, d.h. wie üblich in Prolog in verschiedene Richtungen verwenden. Betrachten wir hierzu die bekannte Fakultätsfunktion. Um diese in Prolog zu implementieren, müssen wir geschachtelte Funktionsanwendungen wie üblich entschachteln und durch Prädikate ersetzen und erhalten damit folgendes Programm:

```
:- use_module(library(clpfd)).

fac(0, 1).
fac(N, F) :- N #> 0, N1 #= N-1, F #= N*F1, fac(N1, F1).
```

Diese Definition können wir nun flexibel verwenden, wie folgende Anfragen zeigen:

```
?- fac(5,F).
F = 120
?- fac(N,720).
N = 6
?- fac(N,723).
no
?- fac(N,F).
N = 0,
F = 1 ? ;
N = 1,
F = 1 ? ;
N = 2,
F = 2 ? ;
N = 3,
F = 6 ? ;
N = 4,
F = 24 ? ;
N = 5,
F = 120 ? ;
N = 6,
F = 720
```

6.3.5 CLP(X): Ein Rahmen für Constraint Logic Programming

Mit CLP(X) wird ein allgemeiner Rahmen für die logische Programmierung mit Constraints bezeichnet, der in (Jaffar and Lassez, 1987) vorgeschlagen wird. Hierbei ist X eine beliebige, aber feste **Constraint-Struktur**, die aus folgenden Komponenten besteht:

- eine Signatur Σ, d.h. eine Menge von Funktions- und Prädikatssymbolen
- eine Struktur D über Σ, d.h. eine Wertemenge mit entsprechenden Funktionen und Prädikaten, die die Symbole in Σ interpretieren. Im Fall von CLP($\mathcal{R}$) könnten dies z.B. reelle Zahlen mit den üblichen Operationen und Relationen ($+$, $-$, $*$, $/$, $=$, $<$, $>$,...) sein.
- eine Klasse L von Formeln über Σ, d.h. Teilmenge der Formeln der Prädikatenlogik 1. Stufe mit Symbolen aus Σ.
- eine Theorie T, d.h. eine Axiomatisierung der wahren Formeln aus L, sodass ein Constraint $c \in L$ wahr ist gdw. $T \models c$.

Damit eine solche Constraint-Struktur mit der logischen Programmierung kombiniert werden kann, werden noch einige weitere Forderungen aufgestellt:

1. Σ enthält das binäre Prädikatsymbol $=$, das als Identität in D interpretiert wird.
2. Es gibt Constraints $\bot$ und $\top$ in L, die als falsch bzw. wahr in D interpretiert werden.
3. L ist abgeschlossen unter Variablenumbenennung, Konjunktion und Existenzquantifizierung.

Dann kann man CLP(X)-Programme ähnlich wie logische Programme definieren, indem man zusätzlich zu Literalen in Klauselrümpfen und Anfragen auch Constraints erlaubt. So haben **Klauseln** in CLP(X)-Programmen die Form

$$p(\bar{t}) \texttt{ :- } c, p_1(\overline{t_1}), \ldots, p_k(\overline{t_k})$$

wobei $\bar{t}, \overline{t_1}, \ldots, \overline{t_k}$ Folgen von Termen mit Funktionssymbolen aus Σ und c ein Constraint aus L sind.

Analog haben **Anfragen** in CLP(X)-Programmen die Form

$$\texttt{?- } c, p_1(\overline{t_1}), \ldots, p_k(\overline{t_k})$$

wobei c erfüllbar ist. Das **Resolutionsprinzip für CLP(X)** kann dann wie folgt definiert werden (wobei wir hier aus Vereinfachungsgründen die Selektionsregel von Prolog betrachten).

Definition 6.6 (Resolutionsprinzip für CLP(X)). *Die Anfrage*

$$\texttt{?-}\ c, p_1(\overline{t_1}), \ldots, p_k(\overline{t_k})$$

wird mit der Klausel

$$p_1(\overline{s})\ \texttt{:-}\ c', L_1, \ldots, L_m$$

reduziert zu der Anfrage

$$\texttt{?-}\ c \wedge c' \wedge \overline{t_1} = \overline{s}, L_1, \ldots, L_m, p_2(\overline{t_2}), \ldots, p_k(\overline{t_k})$$

falls der Constraint $c \wedge c' \wedge \overline{t_1} = \overline{s}$ erfüllbar ist.

Somit wird also die Unifikation durch einen Erfüllbarkeitstest für die jeweilige Constraint-Struktur ersetzt. Bezüglich dieser Erweiterungen werden in (Jaffar and Lassez, 1987) die wichtigsten Ergebnisse der logischen Programmierung auf den Rahmen CLP(X) übertragen:

> Die Standardresultate der Logikprogrammierung (d.h. die Korrektheit/Vollständigkeit der SLD-Resolution) sind auch gültig für CLP(X).

Wie man aus dem erweiterten Resolutionsprinzip ersehen kann, ist es zur Implementierung von CLP(X) notwendig, die Erfüllbarkeit von Constraints bezüglich der Constraint-Struktur X zu testen. Die Implementierung hiervon wird auch als **Constraint-Löser** bezeichnet. Damit nicht bei jedem Schritt erneut ein kompletter Erfüllbarkeitstest durchgeführt werden muss, muss der Constraint-Löser *inkrementell* (!) arbeiten.

Betrachten wir z.B. $X = \mathcal{R}$ (d.h. arithmetische Constraints über den reellen Zahlen): Als Constraint-Löser wird hier neben der Termunifikation eine inkrementelle Version vom Gauß'schen Eliminationsverfahren (Lösen von Gleichungen) und eine inkrementelle Simplexmethode (Lösen von Ungleichungen) eingesetzt.

Eine Abschwächung dieses allgemeinen Schemas erfolgt oft bei „harten Constraints", d.h. Constraints, die nur mit großem Aufwand exakt zu lösen sind: Hier erfolgt kein voller Erfüllbarkeitstest, sondern es wird z. B. wie folgt vorgegangen:

- Verzögere die Auswertung von „harten Constraints" (z. B. Verzögerung von nichtlinearen Constraints in CLP($\mathcal{R}$), bis sie linear werden).
- Führe bei endlichen Bereichen (CLP(FD)) nur eine lokale Konsistenzprüfung durch, da ein globaler Erfüllbarkeitstest für die Constraints in der Regel NP-vollständig ist (siehe (Van Hentenryck, 1989)).

Betrachten wir z. B. CLP(FD), d. h. die Constraint-Programmierung über endlichen Bereichen. Hier werden als Lösungsalgorithmen Methoden aus dem Operations Research zur lokalen Konsistenzprüfung (Knoten-, Kantenkonsistenz) eingesetzt, d. h. es ist nicht sichergestellt, dass die Constraints immer erfüllbar sind (da ein solcher Test NP-vollständig wäre). Aus diesem Grund erfolgt die konkrete Überprüfung einzelner Lösungen durch Aufzählen (Prinzip: "constrain-and-generate"). Dies ist der Grund, warum man nach der Aufzählung aller Constraints das `labeling`-Prädikat angibt.

6.4 Datenbanksprachen

Eine relationale Datenbank ist im Prinzip eine Menge von Fakten ohne Variablen und ohne Funktoren. Eine Anfrage à la SQL ist eine prädikatenlogische Formel, die in eine Prolog-Anfrage mit Negation transformiert werden kann.

Aus diesem Grund ist es sinnvoll und konzeptuell einfach möglich, relationale Datenbanken in Prolog-Systeme einzubetten. Der Vorteil einer solchen Einbettung ist die Erweiterung der Funktionalität eines Datenbanksystems durch Methoden der Logikprogrammierung. Dies führt zu dem Konzept der **deduktiven Datenbanken**. Eine deduktive Datanbank ist im Prinzip eine Mischung aus relationalen Datenbanken und Prolog, wobei meist aber nur eine Teilsprache namens **DATALOG** (Prolog ohne Funktoren) betrachtet wird.

Der Vorteil ist, dass man nur Basisrelationen in der Datenbank speichern muss und abgeleitete Relationen als Programm darstellen kann. Hierbei ist es wichtig anzumerken, dass das Programm auch rekursiv sein kann, wodurch die Mächtigkeit im Vergleich zu grundlegendem SQL erhöht wird (neuere Versionen von SQL erlauben ebenfalls die Definition rekursiver Abfragen).

Beispiel 6.9 (Deduktive Datenbanken)**.** Wir betrachten folgende Basisrelationen:

- Lieferanten-Teile: `supp_part(Supp,Part)`
- Produkt-Teile: `prod_part(Prod,Part)`

Nun wollen wir die folgende abgeleitete Relation definieren: `prod_supp` (Produkt-Lieferant).

```
prod_supp(P,S) :- prod_part(P,Part), supp_part(S,Part).
prod_supp(P,S) :- prod_part(P,Part), prod_supp(Part,S).
                  % Ein Produkt kann andere Produkte enthalten!
```

Dies ist eine rekursive Relation, die so nicht direkt in grundlegendem SQL formulierbar ist.

Interessante Aspekte von DATALOG sind:

- Anfrageoptimierung: Effizientes Finden von Antworten (top-down- oder bottom-up-Auswertung)
- Integritätsprüfung: Integritätsbedingungen sind logische Formeln über Datenbank-Relationen, deren Erfüllbarkeit nach jeder Datenbank-Änderung (effizient!) geprüft werden muss.