# Partielle Auswertung von Prolog-Programmen am Beispiel des handgeschriebenen Compilergenerators LOGEN

Erik Steffen
Betreuer: Prof. Dr. Hanus

11. Dezember 2006

# Inhaltsverzeichnis

# 1 Einleitung

Bei der Implementierung von Programmen wählt man meist einen Zwischenweg aus einer möglichst allgemeinen, in mehreren Kontexten anwendbaren Lösung oder einer hoch spezialisierten, die nur für ein ganz bestimmtes Problem anwendbar ist. Spezielle Programme sind in der Regel sehr viel effizienter als möglichst allgmein einsetzbare. Dies induziert den Wunsch, ein Programm für ein oder mehrere Probleme zu spezialisieren, wenn es, obwohl vielfältig einsetzbar, in der Regel nur für eine kleine Menge bestimmter Probleme genutzt wird. In der Vergangenheit gab es viele Ansätze so genannte Spezialisierer zu schreiben. Diese liefern eine für ein bestimmtes Problem spezialisierte Version des Programms, welche dieses Problem möglichst effizient löst.

Ein weiterer Ansatz ist die Erstellung eines 'Spezialisierers' für das gegebene Programm. Dieser wird vor der eigentlichen Spezialisierungsphase generiert und ist in der Lage, das Programm effizient für eine bestimmt Art Problem zu spezialisieren. In Fällen, in denen ein und dasselbe Programm für verschiedene Probleme spezialisiert wird, relativiert sich so der zusätzliche Aufwand, den die Erstellung des Spezialisierers verursacht.

Im Folgenden wird die generische Vorgehensweise für eine partielle Auswertung logischer Programme vorgestellt und ein darauf aufbauender Spezialisierer, genannt LIX, für reine, logische Programme vorgestellt. Um auch Seiteneffekte, Negation und sonstige Erweiterungen betrachten zu können, wird LIX zum Compilergenerator LOGEN erweitert.

LOGEN benutzt eine Beschreibung, welche Art spätere Anfragen sein werden, welche Teile statisch sind und welche nicht, um einen spezialisierten Spezialisierer zu erstellen. Wird dieser Spezialisierer mit speziellen Werten für die statischen Teile ausgeführt, erhält man das spezialisierte Programm.

# 2 Grundlagen

Grundlage des Compilergenerators LOGEN ist eine spezielle Methode zur partiellen Auswertung reiner logischer Programme, genannt LIX. Dieser Abschnitt wird die grundlegende Funktionsweise von LIX aufzeigen, die später auch bei LOGEN Anwendung findet. Im Rahmen dieser Ausarbeitung lasse ich den Begriff der Sicherheit in Bezug auf Bindungstypanalyse, -klassifikation und Divisionen außer Acht, da die erforderlichen Beweise und die noch

strikter formal erforderliche Definition des Auswertungsprozesses den Rahmen dieser Ausarbeitung sprengen würden. An dieser Stelle sei nur gesagt, dass der Sicherheitsaspekt dafür zuständig ist, dass die am Ende der Spezialisierung erhaltenen Programme sich in Bezug auf eine Anfrage $A$ auch wirklich genau so verhalten wie das Urprungsprogramm.

## 2.1 Generische partielle Auswertung

Für ein logisches Programm $P$ und eine Anfrage $G$ berechnet eine partielle Auswertung ein neues Programm $P'$ welches auf die Auswertung von $G$ spezialisiert ist. Das Ziel ist es, dass $P'$ dies effizienter erledigt als $P$ für alle Instanzen der Anfrage $G$.

Zugrundeliegende Technik ist hier die Konstruktion eines möglicherweise unvollständigen, endlichen SLDNF-Baumes für die Anfrage $G$. Die Klauseln des spezialisierten Programmes $P'$ werden dann aus denjenigen Zweigen des SLDNF-Baumes abgeleitet, welche nicht fehlschlagen.

Eine *Entfaltungsregel* ist eine Funktion *unfold*, welche zu einem Programm $P$ und einer Anfrage $G$ einen nicht trivialen und möglicherweise unvollständigen SLDNF-Baum für $P \cup \{G\}$ liefert.

Sei $P$ ein Programm und $A$ ein Atom. Sei $\tau$ endlicher SLDNF-Baum für $P \cup \{\leftarrow A\}$. Seien $\leftarrow G_1, ..., \leftarrow G_n$ die Ziele in den Blättern der nicht fehlschlagenden Äste von $\tau$. Seinen $\theta_1, ..., \theta_n$ die berechneten Antworten der Ableitungen von $\leftarrow A$ nach $\leftarrow G_1, ..., \leftarrow G_n$, dann ist $resultants(\tau)$, die Menge der Resultierenden, definiert als die Menge $\{A\theta_1 \leftarrow G_1, ..., A\theta_n \leftarrow G_n\}$. Desweiteren definiert $leaves(\tau)$ die Menge der in den Blättern von $\tau$ vorkommenden Atome.

Beispiel: Programm P

```
nont(X,T,R) :- t(a,T,V), nont(X,V,R).
nont(X,T,R) :- t(X,T,R).
t(X,[X|R],R).
```

Sei $A :=$ `nont(c,T,R)`, $G1 :=$ `nont(c,V,R)`
Mit `nont(c,T,R)`

$\vdash$ `t(a,T,V),nont(c,V,R)`

$\vdash_{\{T \rightarrow [a|V]\}}$ `nont(c,V,R)`

Dann ist `nont(c,[a|V],R) :- nont(c,V,R).` die Resultierende zum Blatt $G1$ des SLDNF-Baumes, welcher durch Auffalten der Anfrage $A$ und Anwenden der Substitutionen $\theta_1$ auf $A$ entsteht. Auf diese Weise wir ein ganzer Ast des SLDNF-Baumes in einer einzigen Klausel zusammengefaßt.

Bei der partiellen Auswertung werden nun die Resultierenden zu einer gegebenen Menge $S$ von Atomen genutzt, um das spezialisierte Programm zu erzeugen. Hierbei wird für jedes Atom ein eigener Satz spezieller Prädikate erzeugt. Unter der Annahme der Abgeschlossenheit und Unabhängigkeit ist die Korrektheit des spezialisierten Programmes gegeben; d.h. alle Blätter sind Instanzen eines Atoms in $S$ und keine zwei Atome in $S$ besitzen eine gemeinsame Instanz.

Abgeschlossenheit wird von der folgenden Prozedur sichergestellt, die ein Programm $P$ und eine initiale Menge $S_0$ an Atomen als Eingabe erhält und bei Termination eine Menge $S$ von Atomen zurückgibt.

**Prozedur 1**

**Initialisierung:** $S_{new} := generalize(S_0)$
**repeat**
$S_{old} := S_{new}$
$S_{new} := \{s_n | s_n \in leaves(unfold(P, s_0)) \wedge s_0 \in S_{old}\}$
$S_{new} := generalize(S_{old} \cup S_{new})$
**until** $S_{old} = S_{new}$ (modulo Variablenumbenennung)
**Ausgabe:** $S := S_{new}$

Eine *Generalisierungsoperation* *generalize* ist eine Funktion von einer Menge von Atomen in eine Menge von Atomen. Sei $S$ eine endliche Menge von Atomen, dann ist $S' := generalize(S)$ eine endliche Menge von Atomen mit den gleichen Prädikatssymbolen wie in $S$ und jedes Atom in $S$ ist eine Instanz eines Atoms in $S'$.

Innerhalb von *Prozedur 1* wird zwischen zwei verschiedenen Arten von Kontrolle unterschieden: Die Entfaltungsfunktion kontrolliert die Konstruktion der SLDNF-Bäume, was als *lokale Kontrolle* bezeichnet wird, während die Generalisierungsoperation *generalize* die Konstruktion der Mengen von Atomen kontrolliert, für die SLDNF-Bäume berechnet werden. Dies wird als *globale Kontrolle* bezeichnet.

Der sogenannte *off-line* Ansatz für das Kontrollproblem der partiellen

Ableitung benutzt eine Analysephase (*Binding-Time-Analysis*), die vor der eigentlichen Spezialisierung durchgeführt wird. Die Analysephase erstellt Annotationen, durch welche die eigentliche Spezialisierungsphase gesteuert wird.

Der *off-line*-Ansatz ist generell sehr effektiv, wenn ein und das selbe Programm mehrfach spezialisiert werden soll, da die Analyse nur einmalig durchgeführt werden muß und die eigentliche Spezialisierung sehr effizient und schnell mehrfach durchgeführt werden kann für unterschiedliche Anfragen.

## 2.2 Typendefinitionen

In logischen Programmen kann ein Typ als eine Menge von Termen angesehen werden, welche in Bezug auf Substitution abgeschlossen ist. Formal gesehen ist ein Typ entweder eine Typenvariable oder ein $n$-stelliger Typenkonstruktor, welcher wiederum auf $n$ Typen angewendet wird. Ein Typ, welcher keine Variablen enhält, wird Grundtyp genannt.

Eine Typendefinition für einen Typenkonstruktor hat die Form

$$c(V_1, \ldots, V_n) \rightarrow f_1(T_1^1, \ldots, T_1^{n_1}); \ldots; f_k(T_k^1, \ldots, T_k^{n_k})$$

wobei $k \geq 1$, $n, n_1, n_2, \cdots \geq 0$ und $f_i$ eindeutige Funktionssymbole sind.

Ein Typensystem $\Gamma$ ist eine Menge von Typendefinitionen und enhält genau eine Typendefinition für jeden $n_{\geq 1}$-stelligen Typenkonstruktor $c$. $Def_\Gamma(c)$ bezeichnet die Typendefinition von $c$ im Typensystem $\Gamma$. In logischen Programmen unterscheidet man drei Grundtypen, die induktiv wie folgt definiert sind.

- $t : \mathit{dynamic}$ für jeden term $t$
- $t : \mathit{static}$ für jeden Grundterm $t$
- $t : \mathit{nonvar}$ für jeden Term, der keine Variable ist.
- $f(t_1, \ldots, t_n) : c(\tau'_1, \ldots, \tau'_k)$ falls eine Grundinstanz der Typendefinition $Def_\Gamma(c)$ existiert der Form $c(\tau'_1, \ldots, \tau'_k) \rightarrow f(\tau_1, \ldots, \tau_n)$ mit $t_i : \tau_i$ für $1 \leq i \leq n$.

Schlußendlich heißt ein Typ $\tau$ *allgemeiner* als ein Typ $\tau'$ falls, wann immer $t : \tau'$ gilt, auch $t : \tau$ gilt.

## 2.3 Bindungstyp-Analyse und -Klassifikation

Der nächste Schritt, der vor der eigentlichen Spezialisierung nötig ist, ist die sogenannte *Bindingsbtypeanalyse* (BTA). Diese liefert genaue Informationen darüber, welche Werte während der späteren Spezialisierungsphase bekannt sind. Die BTA benötigt hierzu das zu spezialisierende Programm $P$ und eine Beschreibung, in welcher Art das Programm spezialisiert werden soll. Diese Beschreibung erfolgt mit Hilfe einer Division, welche den Argumenten eines Prädikats einen Typ zuweist.

Eine *Division* für ein n-stelliges Prädikat hat die Form $p(\tau_1, ..., \tau_n)$, wobei die $\tau_i$ Grundtypen sind. Die *Division eines Programmes* $P$ ist eine Menge von Divisionen für die Prädikate des Programms mit maximal einer Division für jedes Prädikat. Eine Division heißt *einfach*, wenn sie nur die Typen *static* und *dynamic* enthält. Eine Division $\Delta$ heißt *allgemeiner* als eine Division $\Delta'$ falls gilt:

$$\forall p(\tau'_1, \ldots, \tau'_n) \in \Delta' \ : \ \exists p(\tau_1, \ldots, \tau_n) \in \Delta \text{ mit } \tau_i \text{ allgemeiner als } \tau'_i \ \forall i.$$

Auf die Division und die initiale Beschreibung aufbauend liefert die BTA eine Programmannotation, welche den späteren Entfaltungsprozess steuert und kann prinzipiell als eine spezielle Entfaltungsregel angesehen werden.

Eine Bindungstypanalyse generiert zu einem gegebenen Programm $P$ und einer initialen Division $\Delta_0$ für $P$ ein Tupel $(U, \Delta)$, wobei $U$ eine Entfaltungsregel und $\Delta$ eine allgemeinere Division für $P$ ist als $\Delta_0$. Das Tupel $\beta = (U, \Delta)$ heißt *Bindungstypklassifikation* (BTC).

Die Information, in welcher Art $P$ später spezialisiert werden soll, wird hier durch $\Delta_0$ bereitgestellt und wird in der Regel von Hand erstellt. Die Division $\Delta$ gibt an, welchen Typs Atome sein können, die auf globaler Ebene auftauchen können.

## 2.4 LIX

Die Basis auf der der Compilergenerator LOGEN aufgebaut ist, ist eine Methode zur partiellen Ableitung von reinen logischen Programmen, genannt LIX. Sie benutzt eine einfache Entfaltungsregel, welche wie folgt definiert ist.

Eine *Annotation A* markiert jedes Literal im Rumpf jeder Klausel von P als *reduzierbar* oder *nicht-reduzierbar*. Ein Programm $P$ zusammen mit einer Annotation $A$ wird als *annotiertes Programm* $P_A$ bezeichnet. Für $P_A$ bezeichnet $U_A$ die Entfaltungsregel, welche für eine Anfrage $G$ den SLD-Baum $U_A(P, G)$ berechnet indem iterativ immer das am weitesten links stehende, und als reduzierbar markierte Atom in $G$ entfaltet wird, bis nur noch als nicht-reduzierbar markierte Atome in den Blättern des resultierenden SLD-Baumes vorkommen.

Syntaktisch werden reduzierbare Literale durch im Programmtext unterstrichene Prädikatssymbole gekennzeichnet. Um eine konkrete Instanz von *Prozedur 1* zu erhalten, wird nun nur noch die Generalisierungsoperation benötigt, welche durch folgende Familie $gen_\tau$ von Funktionen definiert ist.

- $gen_{static}(t) = t$ für jeden Term $t$
- $gen_{dynamic}(t) = V$ für jeden Term $t$ und eine frische Variable $V$
- $gen_{nonvar}(f(t_1, \ldots, t_n)) = f(V_1, \ldots, V_n)$ mit frischen Variablen $V_1, \ldots, V_n$
- $gen_{c(\tau'_1,\ldots,\tau'_k)}(f(t_1, \ldots, t_n)) = f(gen_{\tau_1}(t_1), \ldots, gen_{\tau_n}(t_n))$ falls in $Def_\Gamma(c)$ eine Grundinstanz existiert der Form $c(\tau'_1, \ldots, \tau'_k) \rightarrow \ldots; f(\tau_1, \ldots, \tau_n); \ldots$

Für eine Division $\Delta$ eines Programms $P$ ist die partielle Funktion $gen_\Delta$ von Atomen auf Atome definiert durch

- $gen_\Delta(p(t_1, \ldots, t_n)) = p(gen_{\tau_1}(t_1), \ldots, gen_{\tau_n}(t_n))$ falls $p(\tau_1, \ldots, \tau_n) \in \Delta$ existiert mit $p(t_1, \ldots, t_n) : p(\tau_1, \ldots, \tau_n)$

Die eigentliche Generalisierungsoperation $generalize_\Delta$ ist dann wie folgt definiert: Für eine Menge $S$ von Atomen ist $generalize_\Delta(S) =: S_1$ eine minimale Menge von Atomen aus $S_2 := \{gen_\Delta(s) | s \in S\}$, so dass für jedes Element aus $S_2$ eine Variante in $S_1$ existiert.

**Beispiel**

Sei $\Delta = \{p(static, dynamic), q(dynamic, static, nonvar)\}$

Dann ist $gen_\Delta(p(a, b)) = p(a, X)$ und $gen_\Delta(q(a, b, f(c))) = q(Y, b, f(Z))$

Es folgt $generalize_\Delta(\{p(a,b), q(a,b,f(c))\}) = \{p(a,X), q(Y,b,f(Z))\}$

Um als letzten Schritt das spezialisierte Programm zu erhalten, wird noch eine Funktion benötigt, die jedem spezialisierten Atom der Form $gen_\Delta$ einen eindeutigen Bezeichner zuweist. Diese Aufgabe übernimmt die Funktion $filter_\Delta$.

Sei $\|\cdot\|$ eine Funktion, die Atome auf natürliche Zahlen abbildet, so dass $\|A\| = \|B\|$ wenn $A$ und $B$ gleich sind, modulo Variablenumbenennung.

Die Funktion $filter_\Delta$ wird durch $filter_\Delta(A) = p_{\|gen_\Delta(A)\|}(V_1\theta, \ldots, V_2\theta)$ definiert, wobei $A = gen_\Delta(A)\theta$, $p$ das Prädikatssymbol von $A$ und $V_1, \ldots, V_k$ die Variablen sind, die in A vorkommen.

Im Kontext des obigen Beispiels erhalten wir so:

$\|p(a,X)\| = 1$ und $\|q(Y,b,f(Z))\| = 2$

$filter_\Delta(p(a,b)) = p_1(b)$ und $filter_\Delta(q(a,b,f(c))) = q_2(a,c)$

Jetzt kann folgende Methode zur Spezialisierung reiner logischer Programme angegeben werden.

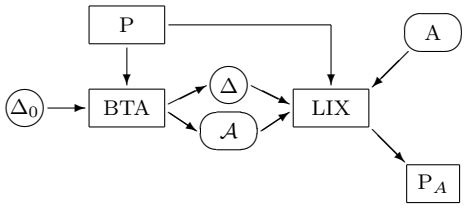



Abbildung 1: Übersicht über den *mix*-Ansatz

**Prozedur 2**

1. Führe eine BTA durch (möglicherweise von Hand) und erhalte BTC $(U_A, \Delta)$
2. Führe Prozedur 1 unter Benutzung von $U_A$ als Entfaltungsregel und $generalize_\Delta$ als Generalisierungsoperation durch. Die initiale Menge an

Atomen $S_0$ sollte nur Atome $p(t_1, ..., t_n)$ enthalten für die ein $p(\tau_1, ..., \tau_n)$ $\in \Delta$ existiert mit $t_i : \tau_i$.

3. Konstruiere das spezialisierte Programm $P'$ mit Hilfe der Funktion $filter_\Delta$ und der Rückgabe von Prozedur 1 wie folgt:
$P' = \{filter_\Delta(A)\theta \leftarrow filter_\Delta(B_1), \ldots, filter_\Delta(B_n) \mid$
$A\theta \leftarrow B_1, \ldots, B_n \in resultants(U_A(P, A)) \wedge A \in S$

## 2.5 Beispiel

Es folgt ein kurzes Beispiel, wie mit Hilfe von Prozedur 1 und 2 ein einfaches Programm $P$ (siehe Beispiel S.4) spezialisiert werden kann. Ausgehend von der initialen Division $\Delta_0 := \{nont(static, dynamic, dynamic)\}$ könnte eine BTA eine BTC $\beta = (U_A, \Delta)$ liefern mit $\Delta = \{nont(static, dynamic, dynamic)$, $t(static, dynamic, dynamic)\}$ und $A$ wie unten gezeigt (hierbei steht ein unterstrichenes Prädikatssymbol für ein reduzierbares Prädikat).

Eine partielle Auswertung mit einer initialen Menge von Atomen $S_0 = (nont(c, T, R))$ liefert den folgenden SLD-Baum.

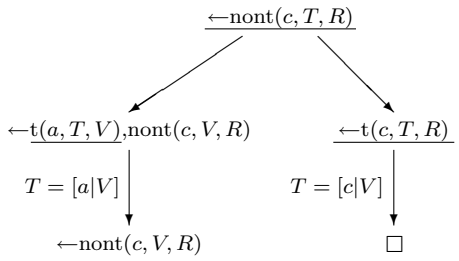



Abbildung 2: Entfalten des Beispiels

Wie man sieht, ist $\{nont(c, V, R)\}$ das einzige Atom in den Blättern und wir erhalten $S_{old} = S_{new}$ (modulo Variablenumbenennung).

Das spezialisierte Programm sieht dann folgendermaßen aus:

Vor Filterung:

```
nont(c,[a|V],R) :- nont(c,V,R).
nont(c,[c|R],R).
```

Nach Filterung:

```
nont_1([a|V],R) :- nont_1(V,R).
nont_1([c|R],R).
```

Das Beispiel zeigt, dass bei dem spezialisierten Programm immer nur eine einzige Regel angewendet werden kann. Der Verwaltungsaufwand für das Backtracking wird enorm reduziert. Diese Eigenschaft ist maßgeblich für die Effizienzsteigerung des Residualcodes gegenüber dem Originalcode verantwortlich.

# 3 Hauptteil - der Compilergenerator LOGEN

Ausgehend von der Methode der Spezialisierung von rein logischen Programmen, welche im vorherigen Kapitel vorgestellt wurde, wird im Folgenden der Compilergenerator LOGEN vorgestellt. Im Grunde wird der Ansatz der partiellen Auswertung hierbei um einen Schritt erweitert. Anstelle ein Programm $P$ direkt für eine bestimmte Anfrage $A$ zu spezialisieren, wird eine Spezialisierungserweiterung $SE^P_{A,\Delta}$ für $P$ erstellt, welche sich vorerst nur auf die von der BTA generierte BTC stützt. Die erstellte Spezialisierungserweiterung ist dann in der Lage zu einer Anfrage $A$ eine spezialisierte Version des Programmes $P$ zu generieren.

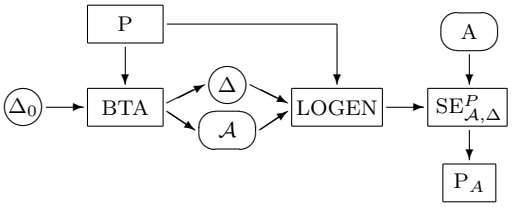



Abbildung 3: Übersicht über den *cogen*-Ansatz

## 3.1 Die lokale Kontrolle

Die grundlegende Idee der Spezialisierungerweiterungen ist die Einführung von Entfaltungs- und Memo-Prädikaten. Ein Entfaltungsprädikat zu einem Prädikat $p/n$ ist ein $n+1$-stelliges Prädikat $p_u/n+1$ welches die Entfaltung des Prädikates $p/n$ durchführt. Hierbei ist $p_u$ im Prinzip mit $p$ identisch, jedoch um ein weiteres Argument erweitert, in welchem das Ergebnis des Entfaltungsprozesses gesammelt wird. Desweiteren werden die anderen Argumente wie gewohnt instantiiert.

```
nont_u(X,T,R,(V1,V2)) :- t_u(a,V,T,V1), nont_m(X,V,R,V2).
nont_u(X,T,R,V1) :- t_u(X,V,R,V2).
t_u(X,[X|R],R,true).
```

Ein Aufruf von $p_u$ wendet also die Entfaltungsregel $U_A$ auf die Zweige des SLDNF-Baumes an. Die $p_m$ sind die oben erwähnten Memo-Prädikate. Diese werden für all diejenigen Prädikate erstellt, welche auf globaler Ebene auftauchen können. Ein Aufruf von $p_m$ stoppt an dieser Stelle den Entfaltungsprozess und hinterlässt das Atom für die globale Kontrolle. Die Memo-Prädikate liefern dabei eine gefilterte und umbenannte Version des Aufrufs im letzten Argument zurück.

Nehmen wir an, $nont_m$ führe weder globale Kontrolle, Filterung noch Umbenennung durch und setzen oben `nont_m(X,V,R,nont(X,V,R))` ein. Führt man für die Anfrage `nont_u(c,T,R,Leaves)` den oben angegebenen Code aus, so werden zwei Antworten berechnet, welche genau den beiden Ästen des SLD-Baumes aus dem Beispiel entsprechen.

```
> ?-nont_u(c,T,R,Leaves).
   T = [a|_A], Leaves = true,nont(c,_A,R) ? ;
   T = [c|R], Leaves = true ? ;
no
```

## 3.2 Die globale Kontrolle

Wie oben schon angedeutet, ist der Code für $P_U^A$ noch nicht vollständig da noch keine globale Kontrolle erfolgt. Wie oben zu sehen ist, liefert ein Aufruf von $p_u$ z.B. nur die Atome in *einem* Ast des SLDNF-Baumes. Um alle Äste auszuwerten, bedient sich LOGEN des `findall` Prädikats von Prolog.

Ein Aufruf von `findall(V,Call,Res)` findet alle Antworten $\theta_i$ von `Call`, wendet diese auf `V` an und instantiiert `Res` zu einer Liste welche umbenannte Versionen aller `V`$\theta_i$s enthält. Wird also `findall(B,nont_u(c,T,R,B),Bs)` aufgerufen, so wird `Bs` zu `[[true,nont(c,_48,_49)],[true]]` instantiiert.

Dies entspricht nun genau den beiden Ästen im obigen Beispiel. Ein Aufruf von `findall( clause( nont(c,T,R),Bdy), nont_u(c,T,R,Bdy), Cs)` liefert außerdem in `Cs` die beiden Resultanten aus dem Beispiel ohne Filterung:

```
[nont(c,[a|V],R):-nont(c,_48,_49), nont(c,[c|R],R).]
```

Nun, wo alle Resultierenden gefunden sind, müssen diese als nächstes mittels $gen_\Delta$ generalisiert und danach entfaltet werden, falls dies noch nicht geschehen ist. Dies geschieht durch die Memo-Prädikate $p_m$. Diese bilden die globale Kontrolle, indem für jedes Atom $p/n$ bei Aufruf von $p_m(t_1, .., t_n, R)$ $R$ zur gefilterten Version von $p(t_1, .., t_n)$ instantiiert wird. Der resultierende Aufruf von $p(a, b, X)$ könnte somit $p_1(X)$ sein falls $p(X, Y, Z)$ vom Typ $p(static, static, dynamic)$ ist.

```
p_m( v_i, R ) :- ( find_pattern( p(v_i), R ) -> true
      ; ( generalise( p(v_i), p(g_i) ),
          insert_pattern( p(g_i), Hd ),
          findall( clause( Hd, Bdy ), p_u( g_i, Bdy ), Cs ),
          pp( Cs ),
          find_pattern( p(v_i), R ).
```

Zur gleichen Zeit überprüft $p_m$ mittels `find_pattern`, eines 'seen-before'-Checks, ob das Argument $p(v_i)$ schon früher spezialisiert wurde. Ist dies der Fall, so wird das zweite Argument von `find_pattern` zu dem umbenannten und gefilterten Aufruf $filter_\Delta(p(v_i))$ instantiiert. Zu diesem Zweck benutzt `find_pattern` eine im Hintergrund geführte Liste in der alle spezialisierten Aufrufe zusammen mit ihrer gefilterten Version gespeichert werden. Falls sich der Aufruf nicht in der Liste befindet, wird der andere Fall des Konditionals ausgeführt.

Der Aufruf `generalise( p(`$v_i$`), p(`$g_i$`) )` berechnet $p(g_i) = den_\Delta(p(v_i))$.

Das Prädikat `insert_pattern( p(`$g_i$`), Hd )` fügt ein neues Atom ( hier $p(g_i)$ ) in die Liste der bekannten Atome ein und instantiiert Hd zu der umbenannten und gefilterten Version $filter_\Delta(p(g_i))$ des verallgemeinerten Atoms. Der Aufruf von `insert_pattern` ist als erster aufgerufen, um zu verhindern dass ein und das selbe Atom immer und immer wieder spezialisiert wird.

Abschließend wird das Prädikat `pp` (pp für 'pretty-print') verwendet, um

die Klausen des Residualprogramms ansehnlicher zu machen. Der letzte anschließende Aufruf von `find_pattern` instantiiert das Ausgabeargument `R` zum Residualaufruf $filter_{\Delta}(p(v_i))$ des Atoms $p(v_i)$.

Die Spezialisierungserweiterung wird wie folgt aufgerufen: Soll ein Atom $p(v_i)$ spezialisiert werden, so wird die SE einfach mit $p_m(v_i, R)$ aufgerufen. Zu beachten ist, dass hierbei die Verallgemeinerung und die Spezialisierung instantan mit dem Aufruf von $p_m(v_i, R)$ geschieht und nicht erst der gesamte SLDNF-Baum aufgebaut wird.

Im Prinzip könnte die oben angeführte Implementation von $p_m$ schon für die SE verwendet werden, bei LOGEN wird eine weiter verbesserte Version der Memo-Prädikate verwendet.

Eine der Verbesserungen behandelt den Aufruf der Verallgemeinerungsfunktion `generalise` im Falle eines Prädikats dessen Division simpel ist, d.h. nur die Typen *static* und *dynamic* enthält. Sei zum Beispiel $\Delta = \{p(static, dynamic)\}$ und $p(v_i) =$ `p(X,Y)`, dann muß der *cogen* keinen Aufruf von `generalise/2` generieren sondern kann `p(X,Z)` innerhalb des Codes für `p_m(X,Y,R)` für $p(g_i)$ verwenden, wobei `Z` eine frische Variable ist. Die Spezialisierungserweiterung wird weiterhin die statischen Werte in `X` behalten und die dynamischen in `Y` abstrahieren.

Desweiteren ist es manchmal unnötig, für jedes Prädikat $p$ ein Memo-Prädikat $p_m$ zu erstellen, da einige Prädikate niemals auf globaler Ebene auftauchen. Ein Beispiel hierfür ist das Beispiel am Ende des ersten Abschnitts, wo das Prädikat t/3 immer reduzierbar ist und niemals direkt durch den Benutzer spezialisiert wird. Da außerdem die Division hier simpel ist, kann `generalise` im Vorfelde ausgewertet werden. Die resultierende, optimierte Version der Spezialisierungerweiterung für dieses Beispiel ist somit:

```
nont_m(B,C,D,FilteredCall) :-
   (find_pattern( nont(B,C,D), FilteredCall) -> true
      ; (insert_pattern( nont(B,F,G), FilteredHead),
         findall( clause( FilteredHead, Body),
            nont_u(B,F,G,Body), SpecClause),
         pp(SpecClause),
         find_pattern( nont(B,C,D), FilteredCall)
   )).
nont_u(B,C,D,(E,F)) :- t_u(a,C,G,E), nont_m(B,G,D,F).
nont_u(H,I,J,K) :- t_u(H,I,J,K).
t_u(L,[L|M],M,true).
```

Die Arbeit, welche LOGEN jetzt noch leisten muß, um eine Spezialisierungserweiterung für ein Programm $P$ zu generieren, beläuft sich also nur noch auf die beiden Abschnitte der lokalen- und globalen Kontrolle. Das Prädikat `memo_clause` generiert die Memo-Klauseln der nicht reduzierbaren Prädikate während `unfold_clause` und `body` für die Übersetzung der Originalprädikate in Entfaltungsprädikate für die lokale Kontrolle zuständig sind.

## 3.3 Behandlung von Sprachbesonderheiten

In der Form, in welcher LOGEN nun vorliegt, ist dieser noch nicht in der Lage, etwas anderes als rein logische Programme zu behandeln. Da solche Programme aber in der Regel recht selten sind, wurde LOGEN erweitert, um auch mit Spracherweiterungen und nicht-deklarativen Sprachkonstrukten umgehen zu können.

### 3.3.1 Deklarative Primitiven

Die erste Erweiterung betrifft deklarative Erweiterungen und externe, vom Benutzer definierte, deklarative Prädikate. Da in diesen Fällen der Code nicht bekannt ist, können auch keine Entfaltungsprädikate für diese erstellt werden. LOGEN bleiben also zwei Möglichkeiten, solche Fälle zu behandeln.

- Den Aufruf komplett auswerten (reduzierbarer Fall)
- Den Aufruf direkt als Residualaufruf zurückgeben (nicht-reduzierbarer Fall)

### 3.3.2 Nicht-deklarative Primitiven

Im Prinzip könnte bei nichtdeklarativen Prädikaten die gleiche Vorgehensweise angewendet werden. Jedoch treten hierbei zwei Probleme in den Vordergrund: Die sogenannten Seiteneffekte, und die propagations-sensitiven Spracherweiterungen.

Betrachtet man die Klausel `t :- print(a),2=3.` so darf diese zu `t :- print(a),fail.` spezialisiert werden, nicht aber zu `t :- fail, print(a).` oder gar `t:- fail.` Das Vorgehen, wie bei deklarativen Primitiven, würde

hier das Entfaltungsprädikat

```
t :- print(a),2=3. ⇝ t_u(print(a)) :- fail.
```

liefern, dieses erzeugt aber das leere Programm.

Das zweite Problem wird bei der Betrachtung von `var/1` deutlich, da `(X=a,var(X))` $\not\equiv$ `(var(X),X=a)`. In diesem Fall würde obiges Vorgehen

```
t(X) :- var(X), X=a. ⇝ t_u(X,var(X)) :- X=a.
```

liefern, was immer fehlschlägt. Um diesen Problemen Herr zu werden, benutzt LOGEN ein spezielles Prädikat `hide_nf` welches verhindert, dass Bindungen und Fehlschläge weiterpropagiert werden. Auf diese Weise werden aber Seiteneffekte, wie oben im Beispiel mit `print` nicht verhindert.

### 3.3.3 'Teure' Prädikate / nicht-deterministische Entfaltung

Das eben vorgestellte Prädikat `hide_nf` hat noch einen nützlichen Nebeneffekt. Es verhindert eine rechts-Propagation der Bindungen. Diese Eigenschaft macht man sich dort zu Nutze, wo vollständig deklarative Prädikate, welche jedoch auf Grund von zum Beispiel nicht-Termination nicht entfaltet werden können. In diesem Fall wird bei der Annotation dieses Prädikat als nicht entfaltbar markiert und die anderen Prädikate im Rumpf in der Regel in ein `hide_nf` gekapselt:

Programm

```
p(X) :- expensive_predicate(X), q(X),r(X).
q(a).    q(b).    q(c).
r(a).    r(b).
```

Annotation

```
ann_clause(1,p(X),( rescall( expensive_predicate(X) ),
        hide_nf( (unfold(q(X)), unfold(r(X)) ) ) )).
```

Residualprogramm

```
p__0(B) :- expensive_predicate(B), ( B = a ; B = b ).
```

Eine solchen Annotation erlaubt es, die Prädikate `q/1` und `r/1` zu entfalten und eine rechts-Propagation der Bindungen von `q/1` auf `r/1` zu ermöglichen, während der Aufruf des teuren Prädikats komplett in den Residualcode übergeht.

### 3.3.4 Negation

Die Negation `not/1` wird von LOGEN im Prinzip genau wie eine deklarative Primitive behandelt, mit dem Unterschied, dass LOGEN im Falle eines Residualcodes *not( C )* trotzdem den Code innerhalb der *not*-Anweisung spezialisiert. Hierbei muß sichergestellt werden, dass der Code, welcher später von der Spezialisierungserweiterung ausgeführt wird, niemals fehlschlagen kann.

## 3.4 Anwendungsbeispiel: PyLogen

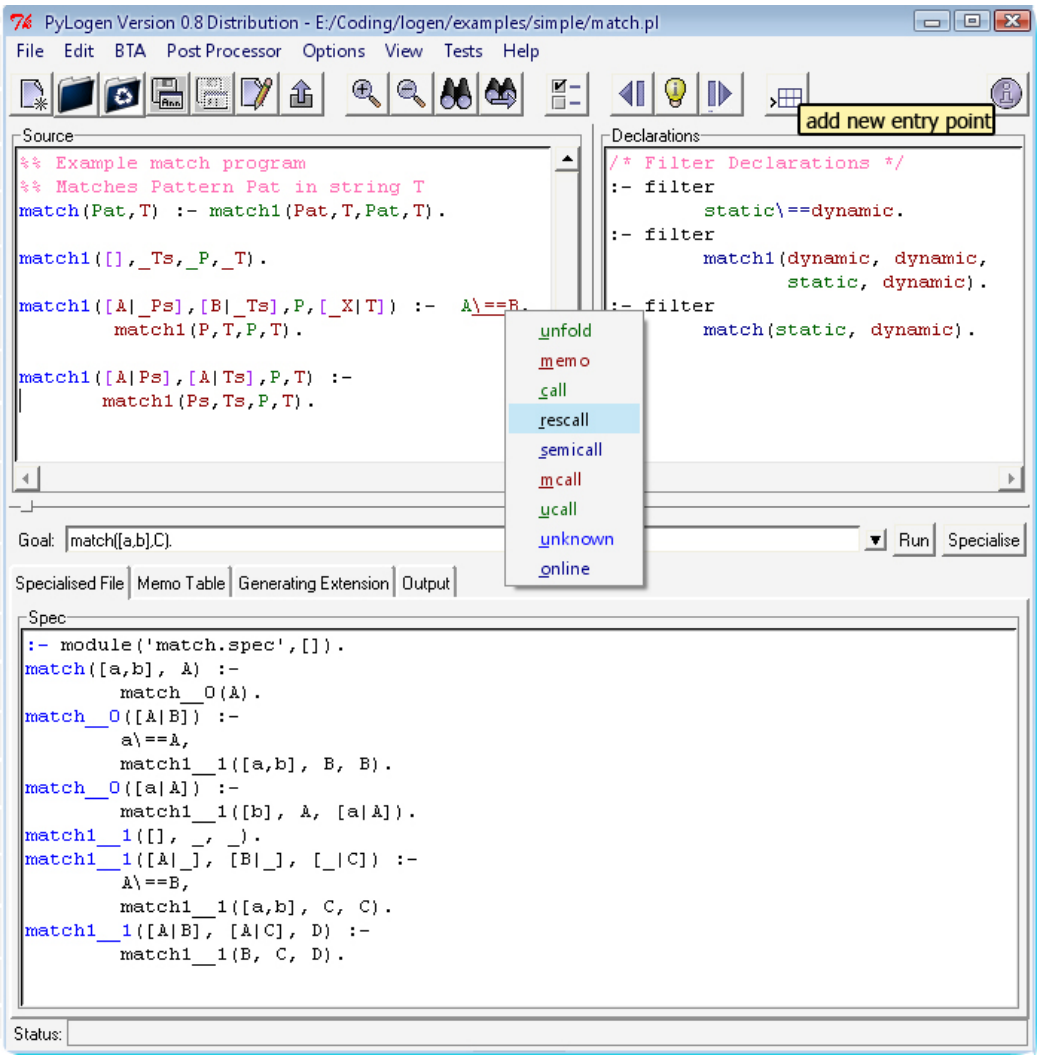
Mit PyLogen steht dem Anwender ein recht komfortables Werkzeug zur Seite. Es ist ein grafisches Interface für die Python-Variante von LOGEN und liegt in der Version 0.8 von 2004 vor. Entwickelt wurde es von Leuschel, Joergensen und Craig. Es arbeitet mit Tcl/TK / Python 2.3.4. PyLogen unterstützt den Anwender bei der Erstellung der Programmannotationen und beinhaltet auch einen Algorithmus für eine vollständig automatisierte Bindungszeitanalyse [3] . In der derzeitigen Version wurde jedoch jeder Variable der Typ *dynamic* zugewiesen. Ein Handbuch ist auf der, seit 2004 nicht mehr gepflegten, Internetseite des Projektes nicht zu finden.

Im Folgenden wird demonstriert wie ein einfaches Programm, welches ein simples Pattern-matching implementiert, mit PyLogen spezialisiert werden kann.

```
match(Pat,T) :- match1(Pat,T,Pat,T).
match1([],_Ts,_P,_T).
match1([A|_Ps],[B|_Ts],P,[_X|T]) :- A\==B, match1(P,T,P,T).
match1([A|Ps],[A|Ts],P,T) :- match1(Ps,Ts,P,T).
```

Nach Eingabe bzw. Laden des Quellcodes werden alle Prädikate in den Rümpfen der Klauseln als `unknown` markiert und farblich hinterlegt. Der Anwender hat nun komfortabel die Möglichkeit den Typ des Prädikats festzulegen (`mcall,ucall,rescall,..`).

Bei der Spezialisierung des Beispielprogrammes markierte ich den Aufruf des Prädikats `match1/4` im Rumpf von `match` mit `unfold`, `\==` mit `rescall` und die restlichen Aufrufe von `match1` mit `memo`. Die rekursiven Aufrufe von `match1` sollte man hier nicht auf `unfold` setzen, da dies in einem Speicherüberlauf endet, welcher sich jedoch leicht erklären läßt. LOGEN scheint hier zu versuchen die unendlichen Listen rekursiv zu entfalten.




Die initiale Division wird über '`add new entry point`' definiert. Hier werden alle Prädikate mit ihrer Stelligkeit nochmals aufgeführt und es kann ein so genannter Filter angelegt werden. Hier bleibt es dem Benutzer frei an diesem Punkt eine automatische BTA zu nutzen, um die BTC zu erstellen oder die Annonationen komplett eigenhändig zu erstellen.

Anschließend muss nur noch ein Aufruf (`Goal`) spezifiziert werden, für welchen das Programm spezialisiert werden soll und LOGEN erstellt die Spezialisierungserweiterung und das spezialisierte Programm. Die SE wird nur dann neu erstellt, wenn die BTC geändert wurde.

Für die Spezialisierung wählte ich ein statisches Pattern `[a,b]` und den Aufruf `match([a,b],X)`. Die Annotationen stellte ich so ein, dass das Pattern stets den Typ `static` besitzt. Das System benötigte dann 15ms für die Erstellung der Spezialisierungserweiterung.

Neben PyLogen gibt es noch ein Online-Interface von LOGEN, genannt Weblogen, welches bequem über Webbrowser bedienbar ist. Erreichbar ist es unter `http://www.stups.uni-duesseldorf.de/~pe/weblogen/index.php` und ist von der Benutzung her mit PyLogen vergleichbar. Der Benutzer bekommt jedoch nicht so viel Hilfe bei der Erstellung der Annotationen und der Code der Spezialisierungserweiteung ist im Vergleich schlechter lesbar als bei PyLogen.

## 3.5 Experimentelle Resultate und Benchmarks

Wie aus den Behandlungen der Spracherweiterungen zu erahnen ist, wird die Güte der Spezialisierung maßgeblich durch die Qualität der erstellten Programmannotationen bestimmt. Dies ist bei weitem nicht trivial und erfordert ein gutes Wissen über die Interna der Programmiersprache. In LOGEN ist deswegen eine Erweiterung integriert, welche die BTA automatisiert und Annotationen zu einem Programm erstellt. Diese Annotationen haben jedoch selten die Qualität der von Hand erstellten. Aus diesem Grunde wurden bei den Benchmarks, die Leuschel et al. mit LOGEN durchführten fast ausnamslos von Hand annotierte Programme verwendet. Verglichen wurde LOGEN mit MIXTUS [4] (Version 0.3.6) und mit ECCE [2].

Die Tests zeigen, dass LOGEN die Testprogramme im Schnitt 30 bis 100 mal schneller spezialisiert, obwohl LOGEN jedes mal zuerst die SE und anschließend das spezialisierte Programm generiert. Den anderen Systemen muß hierbei allerdings zu gute gehalten werden, dass MIXTUS und ECCE vollautomatisch laufen und Leuschel et al. bei LOGEN die BTC von Hand durchführten.

Die Güte des spezialisierten Codes ist bei den drei getesteten Systemen etwa gleich; auch wenn LOGEN dort leicht hinter den anderen Systemen liegt. Der spezialisierte Code lief bei allen drei Systemen etwa 8,5 mal schneller als der Originalcode. Seine wahre Stärke zeigt LOGEN jedoch bei dem Bench-

mark, bei dem der Originalcode für mehrere Anfragen spezialisiert wird. Im Gegensatz zu seinen Mitstreitern profitiert LOGEN hier deutlich, da die Spezialisierungserweiterung nur ein einziges mal erstellt wird und die benötigte Zeit für die dreifache Spezialisierung des Codes somit im Vergleich fast gedrittelt wird.

# 4 Fazit

LOGEN ist ein erstaunlich kompaktes light-weight Tool zum Spezialisieren von Logikprogrammen. Der Code beläuft sich ohne die Erweiterungen für eine automatische BTA auf gerade mal 150 Zeilen. Die Güte des Codes ist nur ganz marginal schlechter als von konkurrierenden Systemen. Die wahre Stärke von LOGEN kommt allerdings nur da zum Vorschein, wo ein Programm mehrfach für unterschiedlich strukturierte Anfragen spezialisiert wird. Bei allen Anwendungen ist jedoch Voraussetzung, dass ein wirklich erfahrener Benutzer die BTA am besten von Hand durchführt. Am interessantesten ist sicher die Möglichkeit, LOGEN direkt in eine Anwendung zu integrieren. Hier könnten Programmteile bei Bekanntwerden der Eingabe vor Ausführung erst spezialisiert werden, wenn davon auszugehen ist, dass das spezialisierte Programm für eine Eingabe mehrfach ausgeführt wird.

# Literatur

[1] Michael Leuschel, Jesper Jørgensen, Wim Vanhoof, Maurice Bruynooghe, Offline specialisation in Prolog using a hand-written compiler generator, *Theory and Practice of Logic Programming*, v.4 n.2, p.139-191, January 2004.

[2] Michael Leuschel, The ECCE partial deduction system, In G.Puebla, editor, *ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, Port Jefferson USA, Oct 1997, Universidad Politécnica de Madrid Tech. Rep. CLIP7/97.1.

[3] Stephen-John Craig , John P. Gallagher, Michael Leuschel and Kim S. Henriksen, Fully Automatic Binding-Time Analysis for Prolog, *Lecture Notes in Computer Science*, vol. 3573/2005, Springer Berlin / Heidelberg, p.53-68.

[4] Dan Sahlin, `MIXTUS`: an automatic partial evaluator for full Prolog, *New Generation Computing*, Volume 12, Issue 1 (1993), p.7-51, ISSN:0288-3635, Tokyo, Japan.