Analyse logisch-funktionaler Programme mit verzögerter Auswertung

Diplomarbeit am
Lehr- und Forschungsgebiet Informatik II
Prof. Dr. Michael Hanus

Ausarbeitung von

Michael Rüdelstein
Matrikelnummer 185681

Betreuer: Frank Zartmann

Aachen, Mai 1998

Hiermit versichere ich, daß ich die Arbeit selbstständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 25.Mai 1998

(Michael Rüdelstein)

Bedanken möchte ich mich für die freundliche Unterstützung all jener die mir bei Problemen und Fragen zur Seite standen. Auch Dank der ständigen Hilfsbereitschaft aller Angehörigen des Lehr- und Forschungsgebiets der Informatik II konnte ich diese Arbeit rechtzeitig fertigstellen. Besonderer Dank gilt Frank Zartmann für die Definition der denotationellen Analyse und für die Betreuung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Das erste Kapitel soll zunächst den Weg von der deklarativen Programmierung hin zur Programmanalyse logisch-funktionaler Programme beschreiben und damit die Motivation für das Thema dieser Arbeit erläutern. Dabei soll zum einen aufgezeigt werden, warum man sich überhaupt mit logisch-funktionaler Programmierung beschäftigt und zum anderen begründet werden, warum dann Programmanalysen sinnvoll sind. Vervollständigt wird diese Einführung durch einen kurzen Umriß über den Aufbau der Programmanalyse und deren Implementierung im Rahmen dieser Arbeit. In den weiteren Kapiteln soll ausführlich auf alle Aspekte von der Theorie bis hin zur Implementierung des Analysesystems eingegangen werden.

1.1 deklarative Programmierung

Deklarative Programmiersprachen bieten im Gegensatz zu den herkömmlichen imperativen Sprachen den Vorteil eines höheren Abstraktionsniveaus. Dies hat eine bessere Lesbarkeit, bessere Wartbarkeit und kürzere Sourcecodes zur Folge. Nachteil der deklarativen Programmiersprachen ist die aufwendigere Implementierung, sowie die zunächst weniger effiziente Auswertung. Daher ist man bestrebt, die Effizienz deklarativer Programmiersprachen durch die unterschiedlichsten Verbesserungen zu steigern, um mit imperativen Programmiersprachen konkurieren zu können. Einige dieser Verbesserungen seien kurz erwähnt:

- Fifth Generation Computer Project (u.a.Japan)
  - Entwicklung einer Hardware, die auf deklarative Programmiersprachen ausgelegt ist. (Projekt scheiterte u.a. an der Preis-/ Leistungsentwicklung von traditioneller Hardware)
- Compiler-Optimierungen
  - Durch last goal optimization oder tail recursion erreicht man bei deterministischen Berechnungen einen konstantem Speicherbedarf

  - Durch optimierte Registerzuteilung der WAM wird der Datenfluß verringert
  - Aufgrund des Fehlens von Seiteneffekten kann durch eine veränderte Unifikationsreihenfolge eine Optimierung erzielt werden

- verbesserte Auswertungsstrategien
  - Ausgehend von strikten Auswertungsstrategien hin zu lazy-Strategien und needed-Strategien, wodurch mehrfach Auswertungen vermieden und bessere Resultate in Bezug auf Vollständigkeit erzielt werden

- Sourcecode-Analysen (Striktheitsanalyse, abstrakte Interpretation)
  - zur Compilezeit ermitteln, wann man von der sicheren call-by-need Strategie zur effizienteren call-by-value Strategie wechseln kann, ohne die Vollständigkeit zu verlieren (funktionale Programmiersprachen)
  - zur Compilezeit ermitteln, welche Parameter zur Laufzeit als variabel, grundinstantiert, etc. auftreten können (logische Programmiersprachen)

Durch all diese Verbesserungen gelang es zumindest teilweise an die Effizienz imperativer Programmiersprachen heranzureichen (Sicstus, Haskell). In letzter Zeit ist man bemüht eine Kombination logischer und funktionaler Programmierkonzepte zu erreichen, um die Vorteile beider Paradigmen zu vereinigen. Vorteile der zwei Paradigmen sind dabei:

- logische Sprachen
  - logische Variablen
  - Suche nach Lösungen
  - partielle Datenstrukturen

- funktionale Programmiersprachen
  - höherer Determinismus in der Auswertung
  - Reduktion verschachtelter Ausdrücke
  - Funktionen höherer Ordnung

Vorteile logisch-funktionaler Programmiersprachen im Gegensatz zu rein logischen Programmiersprachen sind dabei die effizienteren Auswertungsmechanismen aufgrund der (deterministischen) Reduktion von Ausdrücken, sowie die Vermeidung nicht-deklarative Elemente (z.B. Cut(!)). Vorteile im Vergleich zu rein funktionalen Programmiersprachen wären eine höhere Ausdrucksstärke aufgrund logischer Variablen und eingebauter Suche.

In den letzten 10 Jahren wurden daher viele Versuche gestartet, diese beiden Paradigmen zu vereinigen. Aufgrund der Tatsache, daß es keine offensichtliche Methode gibt dies zu erreichen wurden viele unterschiedliche Vorschläge gemacht und es entstanden viele unterschiedliche Programmiersprachen (ALF, BABEL, Escher,K-Leaf, LPG, SLOG). Die vielversprechendsten operationalen Grundsätze sind Residuation und Narrowing.

1.2 Residuation und Narrowing

Residuation basiert auf der Verzögerung von Funktionsaufrufen solange, bis ihre Parameter insoweit instanziert sind, daß sie deterministisch ausgewertet werden können (s. Escher, Le Fun, Life, NUE-Prolog, Oz). Residuation stellt eine vernünftige Integration von funktionalen und logischen Paradigmen dar, da sie deterministische Reduktion mit partiellen Datenstrukturen verbindet und durch Synchronisation über logische Variablen paralelle Auswertungsmechanismen zulässt. Außerdem erlaubt sie eine einfache Einbindung externer Funktionen. Trotzdem birgt die Residuation-Methode einige Nachteile:

- Die nicht-deterministische Suche muß explizit durch Prädikate oder Disjunktionen codiert werden.
- Sie ist unvollständig, da sie nicht in der Lage ist Lösungen zu berechnen, wenn die Funktionsargumente während der Berechnung nicht ausreichend instanziert werden.
- Es ist unklar, ob Residuation besser ist als Prolog's Resolution, da es Beispiele gibt, bei denen Residuation einen unendlichen Suchraum hat, während die äquivalenten, abgeflachten Prologprogramme einen endlichen Suchraum durchlaufen.

Logisch-funktionale Programmiersprachen mit vollständigem operationellem Verhalten (ALF, BABEL und CURRY) basieren auf Narrowing, einer Kombination aus der Reduktion funktionaler Programmiersprachen und Unifikation zur Übergabe von Parametern. Dadurch vereint Narrowing Vollständigkeit im logischen Sinne (Lösungen werden berechnet, falls sie existieren) und im funktionalen Sinne (die Normalform wird berechnet, falls sie existiert). Um aber mit der Resolutionsstrategie von Prolog konkurrieren zu können muß man anspruchsvollere Methoden als einfaches Narrowing anwenden. Eager Narrowing erweist sich dabei als äquivalent zur Resolution. Eine erste Verbesserung stellt Normalisierendes Narrowing (ALF, LPG, SLOG) dar. Hier wird vor jedem Narrowingschritt eine Reduktion zur Normalform vorgenommen, wodurch geringere Laufzeiten, geringerer Speicherbedarf und eine erhebliche Einschränkung des Suchraums ermöglicht werden. Da aber moderne funktionale Programmiersprachen auf lazy Strategien aufbauen sucht man auch beim Narrowing nach einer 'lazy' Strategie. Bei naivem 'lazy Narrowing' kann es jedoch passieren, daß im Gegensatz zu funktionalen Programmiersprachen Argumente mehrmals ausgewertet werden. Needed Narrowing schließlich ist optimal bzl. der Länge der Ableitung und der Zahl berechneter Lösungen bei induktiv sequentiellen Termersetzungssystemen. Eine übersicht über die Entwicklung des Narrowing findet sich in [Han94].

Da beide Berechnungsmodelle ihre Vorteile haben liegt es Nahe beide zu einer neuen Strategie zu verschmelzen. Dabei stellt sich das Problem, das bei Residuation Ersetzungsschritte stets deterministisch durchgeführt werden und nicht-deterministische Schritte explizit mittels Prädikaten codiert werden, was wiederum den Nachteil hat, daß funktionale Abhängigkeiten verloren gehen. Narrowing erlaubt hingegen Nicht-Determinismus auch auf der Ebene von Funktionsaufrufen. Neuere logisch-funktionale Programmiersprachen erlauben nichtdeterministische Auswahlschritte durch explizite Disjunktionen (wie in klassischen logischen Programmiersprachen). Curry benutzt daher ein auf Narrowing basierendes Berechnungsmodell, welches Nicht-Determinismus durch explizite Disjunktionen realisiert. Zusätzlich können durch die Ausführung eines Narrowing-Schrittes neue Disjunktionen erzeugt werden.

1.3 Demand Analyse

Das Berechnungsmodell von Curry basiert auf definierenden Bäumen, welche die jeweilige Berechnungsvorschrift für alle definierten Funktionen darstellen. Aufbauend auf diesen Baumstrukturen soll eine Demand- oder Striktheitsanalyse für Curry entwickelt werden, d.h. es soll bereits zur Compilezeit bestimmt werden, welche Argumente einer Funktion 'demanded' sind, also zwingend ausgewertet werden müssen und bis zu welchem Grad dies der Fall ist. Diese Informationen können dann von einem Compiler verwendet werden, um Argumente sicher und effizient, strikt auszuwerten, wenn dies durch die Analyse aufgezeigt wird. Anderenfalls können Fallunterscheidungen über Top-Konstruktoren von Argumenten vereinfacht werden, wenn von vorneherein klar ist, welcher Konstruktor zum gewünschten Funktionsergebnis führen kann. Die Formulierung einer Analyse, welche direkt auf den definierenden Bäumen arbeitet, erweist sich als zu komplex. Daher werden die Baumstrukturen zunächst in Regelsysteme mit einfachem Regelaufbau umgewandelt (siehe Kapitel 4). Das dabei entstehende Regelsystem ist äquivalent zum ursprünglichen dahingehend, daß jede Narrowingberechnung im Ausgangsmodell einer Narrowingableitung im transformierten System entspricht und umgekehrt. Aufgrund der erheblich einfacheren Struktur der Regeln, im Vergleich zum definierenden Baum, ergibt sich aber eine erheblich einfachere Formulierung der denotationellen Semantik und damit eine einfachere Implementierung.

Der Formulierung der denotationellen Semantik (Kapitel 5) folgen zunächst Aussagen über Korrektheit und Vollständigkeit der Semantik. Diese Erkenntnisse können dann, unter Beachtung der nötigen Vorraussetzungen, auf die Definition einer abstrakten Semantik übertragen werden. Die Abstraktion bezieht sich dabei auf die Konzentration auf wesentliche Aspekte der zu behandelnden Domains. Der (unendliche) Domain wird über einige wesentliche Eigenschaften auf den abstraketn (endlichen) Domain projeziert. Dadurch wird die Berechnung der Semantik zur Compilezeit ermöglicht. Die Definition der abstrakten Semantik soll dabei so weit wie möglich unabhängig von der konkreten Wahl des abstrakten Domains gehalten werden. Die Analyse wird dazu parametrisiert über fünf Operationen, welche direkt auf den Domain und seine Elemente zugreifen. Dies ermöglicht es verschiedene Domains zu definieren und wahlweise in das Analysesystem einzubinden, ohne selbiges verändern zu müssen.

Das in Prolog implementierte System ist zwecks einfacherer Bedienung von einer Shell umgeben. Diese läßt zum einen eine einfache Bestimmung der Eingabewerte (z.B. dem Domain) zu, wertet aber auch die Ergebnisse aus und steuert die wiederholte Anwendung der Analyse mit veränderten Parametern. Dies ermöglicht es eigene, neue Domains zu definieren und ins System einzubinden. Dazu wird im Anhang C ein Beispieldomain beschrieben, der als Vorlage für eigene Definitionen dienen kann.

Die Shell, sowie die zur Implementierung verwendeten Datenstrukturen und Rekursionskontrollen werden in Kapitel 6 beschrieben. Im Anschluß daran werden die Ergebnisse, welche mit der Analyse erzielt wurden aufgezeigt und bewertet. Dabei wird eine Erweiterung der Analyse, um sogenannte disjunktive Bereiche angeregt. Diese Erweiterung wird dann in Kapitel 8 im Hinblick auf Probleme und erzielbare Verbesserungen untersucht. Dazu werden Ergebnisse einfacher Analysen mit denen, die unter Einbeziehung der disjunktiven Bereiche entstanden, verglichen. Dieser Vergleich zeigt sowohl den Unterschied der Güte der Ergebnisse als auch den Unterschied in der Zeitkomplexität auf. Um diesen Vergleich zu erleichtern sollen auch die disjunktiven Bereiche derart ins System eingebunden werden, daß sie sich jederzeit sowohl einbeziehen als auch wieder ausklammern lassen.

Bevor mit der Beschreibung der Demand-Analyse begonnen wird sollen kurz einige Grundlagen geschaffen werden und ein Einblick in die Programmiersprache Curry als Motivation die Einleitung bilden.

Kapitel 2

Theoretische Grundlagen

Zu Beginn sollen zum besseren Verständnis der restlichen Ausführungen einige Definitionen und Konventionen vorgestellt werden, die im weiteren Verlauf zur Anwendung kommen.

2.1 Terme und Positionen

Sei $\mathcal{F}$, die Menge aller Funktionssymbole, disjunkt unterteilt in die Menge $\mathcal{D}$ der definierten Funktionen und die Menge $\mathcal{C}$ der Konstruktorsymbole. Jeder Funktion $f \in \mathcal{F}$ ist eine Stelligkeit $n \in \mathbb{N}_0$ zugeordnet, was mit f/n bezeichnet werden soll. Die unendliche Menge der Variablen sei $\mathcal{X}$. Die Menge $\mathcal{T}(\mathcal{F}, \mathcal{X})$ der Terme über $\mathcal{F}$ und $\mathcal{X}$ sei die kleinste Menge, welche $\mathcal{X}$ und für alle $f/n \in \mathcal{F}$ und $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ auch $f(t_1, \ldots, t_n)$ enthält. Die Menge der Konstruktorterme $\mathcal{T}(\mathcal{C}, \mathcal{X})$ ist die Menge aller Terme über Funktionen aus $\mathcal{C}$ und Variablen aus $\mathcal{X}$. Ein Pattern ist ein Term $f(t_1, \ldots, t_n)$, wobei $f/n \in \mathcal{D}$ und $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. Das Topsymbol $root(t)$ eines Terms t ist das Funktionssymbol $f \in \mathcal{F}$, falls $t = f(t_1, \ldots, t_n)$ oder t, falls $t \in \mathcal{X}$. Ein Term t mit $root(t) \in \mathcal{C}$ ist in Kopfnormalform.

Die Menge der Variablen, die im Term t vorkommen bezeichnen wir mit $\mathcal{V}ar(t)$. Eine Position p eines Teilterms t wird durch eine Folge natürlicher Zahlen repräsentiert und die Menge aller endlichen Folgen natürlicher Zahlen bildet die Menge $\mathcal{P}os$ aller Positionen. Den Teilterm von t an der Position p bezeichnen wir mit $t|_p$. Die Position des Topsymbols $root(t)$ des Terms t sei ϵ. Sei p eine Position im Term t und q die Position von u im Term s ($u = s|_q$). Ersetzten wir nun den Teilterm an der Stelle p in t durch s (i.Z. $t[s]_p$), so bezeichnen wir die Position von u in $t[s]_p$ mit der Konkatenation $p.q$. Zwei Positionen p und q heißen disjunkt, wenn weder eine Position p' mit $q = p.p'$ (q bezeichnet Unterterm von p) noch eine Position q' mit $p = q.q'$ (p bezeichnet Unterterm von q) existiert.

2.2 Substitutionen

Eine Substitution σ aus der Menge $\mathcal{S}ub$ aller Substitutionen ist eine Abbildung von der Menge $\mathcal{X}$ der Variablen in die Menge $\mathcal{T}(\mathcal{F},\mathcal{X})$, welche zu einem Homomorphismus von $\mathcal{T}(\mathcal{F},\mathcal{X})$ in sich selbst erweitert werden kann. Wir beschreiben Substitutionen durch eine Menge von Ersetzungen $\{x \to \sigma(x) | x \in \mathcal{D}om(\sigma)\}$, wobei $\mathcal{D}om(\sigma) = \{x | x \in \mathcal{X}, \sigma(x) \neq x\}$ den Domain von σ bezeichnet. Sei A eine Teilmenge der Variablenmenge $\mathcal{X}$, dann ist $\sigma|_A$ die Einschränkung der Substitution σ auf die Variablen in der Menge A. Die Komposition zweier Substitutionen σ und τ sei $\sigma\tau$ mit $(\sigma\tau)(x) = \sigma(\tau(x))$.

Ein Term t ist eine Instanz des Terms s, falls eine Substitution σ existiert mit $s = \sigma(t)$. Wir sagen, daß sich ein Term s mit einem Term t matchen läßt, wenn eine Substitution σ existiert mit $\sigma t = s$. Eine Substitution σ ist ein Unifikator für die Terme s und t, falls $\sigma s = \sigma t$. σ ist ein allgemeinster Unifikator für s und t, falls für jeden Unifikator τ für s und t gilt, daß σ allgemeiner als τ ist, d.h. es gibt eine Substitution ρ mit $\tau = \rho\sigma$.

2.3 Termersetzung und Narrowing

Eine Termersetzungsregel $l \to r$ ist ein Paar von Termen l und r, wobei l ein Pattern ist und $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ gilt. Dabei heißt l die linke und r die rechte Regelseite. Eine Menge $\mathcal{R}$ von Termersetzungsregeln heißt Termersetzungssystem. Konstruktorbasiert heißt ein Termersetzungssystem, falls alle linken Regelseiten des Systems Pattern sind. Ein Termersetzungsschritt matcht eine linke Regelseite aus $\mathcal{R}$ mit einem Teilterm von t und ersetzt diesen Teilterm durch die instanziierte rechte Seite der Regel. Dies soll für eine Position p in t und eine Regel $l \to r$ aus $\mathcal{R}$ durch $t \to_{p,l\to r} s$ bezeichnet werden, wobei $\sigma l = t|_p$ und $s = t[\sigma r]_p$. Sind die Position p und die Regel $l \to r$ aus dem Kontext heraus klar, oder nicht von Interesse, so bezeichnen wir diesen Ersetzungsschritt auch mit $t \to s$. Schwach orthogonale Termersetzungssysteme enthalten nur Regeln mit trivialen Überlappungen, d.h. falls sich die linken Seiten l_1 und l_2 zweier Varianten von Regeln des Termersetzungssystems mittels einer Substitution σ unifizieren lassen, dann gilt auch $\sigma(r_1) = \sigma(r_2)$. Ein Term heißt reduzibel, wenn sich noch eine Regel auf ihn anwenden läßt und irreduzibel oder in Normalform, wenn er nicht reduzibel ist. Die Reduzierbarkeitsrelation $\to^*$ ist der reflexive, transitive Abschluß der Relation $\to$. Wir nehmen an, daß $\to^*$ konfluent ist, d.h. falls $t \to^* s_1$ und $t \to^* s_2$, dann existiert auch s_3 mit $s_1 \to^* s_3$ und $s_2 \to^* s_3$.

Eine Termersetzungsregel heißt Variante einer anderen Regel, wenn sie aus dieser durch eine eindeutige Variablenumbenennung entsteht. Der Term t läßt sich mittels eines Narrowingschrittes zu s ableiten ($t \rightsquigarrow_{p,l\to r,\sigma} s$), falls $s = \sigma(t[r]_p)$ für eine nicht-variable Position p in t, eine Variante der Regel $l \to r$ aus $\mathcal{R}$ und für einen allgemeinsten Unifikator σ von $t|_p$ und l. Eine Ableitung A ist eine Folge von $n \in \mathbb{N}(n > 0)$ Narrowingschritten, d.h. $A : t_0 \rightsquigarrow_{p_1,l_1\to r_1,\sigma_1} t_1 \rightsquigarrow_{p_2,l_2\to r_2,\sigma_2} t_2 \cdots \rightsquigarrow_{p_n,l_n\to r_n,\sigma_n} t_n$. Wie schon bei Termersetzungsschritten schreiben wir auch hier manchmal nur $t_0 \rightsquigarrow t_1 \rightsquigarrow t_2 \cdots \rightsquigarrow t_n$. Eine Folge syntaktischer Objekte $t_1, \ldots, t_n$ schreiben wir auch kurz als $\overline{t}$.

Die hier vorgestellten Definitionen und Konventionen entsprechen den allgemein gebräuchlichen in der Literatur und finden sich teilweise in vielen Arbeiten zur deklarativen Programmierung wieder. Einige abkürzende Schreibweisen wurden dabei aus [Zar97]. übernommen, um für eine tiefergehende Lektüre der theoretischen Grundlagen dieser Arbeit das gleiche Rüstzeug zur Verfugung zu stellen.

Kapitel 3

Überblick über Curry

In diesem Abschnitt soll kurz die logisch-funktionale Programmiersprache *Curry* vorgestellt werden. Curry wurde entwickelt in internationaler Zusammenarbeit für Forschung, Lehre und Anwendung logisch-funktionaler Sprachen. Ein Curry-Programm besteht aus einer Menge von Typdeklarationen, die die Domains definieren, und einer Menge von Funktionsdeklarationen, welche die Operationen auf diesen Domains beschreiben. Eine genauere Sprachdefinition von Curry findet sich unteranderem in [HKMN95] und [Han98].

3.1 Definition von Datentypen und Funktionen

Curry bietet ein polymorphes Typsystem ähnlich dem von Haskell (Hindley / Milner), eine Erweiterung auf Typklassen (s.Haskell) steht aber noch aus. Dieses erlaubt es viele Programmierfehler bereits zur Compilezeit zu erkennen. Die Menge $\mathcal{F}$ der Funktionen wird in zwei disjunkte Mengen eingeteilt. Zum einen die Menge $\mathcal{C}$ der Konstruktoren, die diejenigen Funktionen enthält, die Datentypen konstruieren und zum anderen die Menge $\mathcal{D}$ der definierten Funktionen, die auf diesen Datentypen arbeiten. Konstruktoren werden durch Datentyp Deklarationen der Form

$$\texttt{data}\ T\alpha_1 \cdots \alpha_n = C_1\ \ \tau_{11} \cdots \tau_{1n_1}\ \ | \cdots |\ \ C_k\ \ \tau_{k1} \cdots \tau_{kn_k}$$

definiert. Diese stellen einen neuen Konstruktortyp T/n mit k neuen Konstruktoren $C_1/n_1, \ldots, C_k/n_k$ mit dem jeweiligen Typ $\tau_{i_1} \to \cdots \to \tau_{in_i} \to T\alpha_1 \cdots \alpha_n$ bereit. Als Beispiele für solche Datentyp-Definitionen seien der Typ `Bool` der Boole'schen Werte, der Typ `Nat` der natürlichen Zahlen, sowie die Deklarationen von Listen (`List`) und Bäumen (`Tree`) angegeben.

```
data Bool = true | false

data Nat = 0 | s Nat
```

```
data List = [ ] | a : List a

data Tree A = leaf A | node (tree A) A (tree A)
```

Der Datentyp `bool` wird danach durch die nullstelligen Konstruktoren `true` und `false`, die natürlichen Zahlen durch den Konstruktor `0` und den Sukzessor `s/1` gebildet. Listen entstehen durch die leere Liste [ ] oder durch Konkatenation (:) eines Elements des Typs a mit einer Liste von Elementen des Typs a. Analog ensteht ein Baum über dem Typ A aus Blättern (`leaf`) des Typs A oder aus Knoten (`node`), welche zwei Bäume des Typs A mit einer Beschriftung vom Typ A zu einem neuen Baum zusammenfügen. Da Curry Funktionen höherer Ordnung erlaubt, werden Funktionen durch Typdefinitionen in curriefizierter Schreibweise deklariert:

$$f : \tau_1 \to \tau_2 \to \dots \tau_n \to \tau$$

wobei $\tau_1, \dots, \tau_n, \tau$ polymorphe Typen sind und τ kein Funktionstyp ist. In diesem Fall heißt n die Stelligkeit der Funktion.

Jedes Objekt in Curry ist eindeutigt getypt und der Typ von Variablen und Operationen kann weggelassen werden, wodurch das Typinferenzsystem die Herleitung des Typs übernimmt. Dieser Deklaration folgen bedingte Regeln, die die eigentliche Funktionsdefinition darstellen. Diese Regeln haben die Form

$$f\ t_1, \dots, t_n \mid \{C\} = e$$

Die linke Seite der Regel muß jeweils ein Pattern sein und darf keine doppelten Variablenvorkommen enthalten. Der bedingte Teil C ist optional und stellt eine Konjunktion aus Boole'schen Ausdrücken und strikten Gleichungen der Form $l = r$ dar. Strikte Gleichungen sind erfüllbar, wenn linke und rechte Seite zu unifizierbaren Konstruktortermen reduziert werden können. Boole'sche Ausdrücke werden aus boole'schen Funktionen aus $\mathcal{D}$ gebildet. Im Gegensatz zu funktionalen Programmiersprachen sind zusätzliche Variablen in C erlaubt (Variablen, die nicht auf der linken Regelseite auftauchen). Diese extra Variablen stellen die Möglichkeiten der Logikprogrammierung zur Verfügung, da eine Suche nach Belegungen dieser Variablen notwendig ist, um die bedingte Regel ausführen zu können. Nicht-Determinismus in Funktionsdefinitionen erreicht man durch überlappende linke Regelseiten oder durch Benutzung freier Variablen. Ein Beispiel für überlappende linke Seiten ist die Definition der Multiplikation mit Null :

`mult x 0 = 0` und
`mult 0 x = 0`

Beim Versuch der Auswertung von `mult 0 0 = ?` wird hier nicht-deterministisch eine Regel ausgewählt und angewendet.

3.2 Freie Variablen

Da Curry auch logische Programmierparadigmen abdecken soll besteht die Möglichkeit, wie in logischen Sprachen, freie Variablen in Ausdrücken zu verwenden. Dabei sollen Werte für die freien Variablen berechnet werden, derart, daß Ausdrücke zu Datentermen reduzierbar werden oder Constraints ausgewertet werden können. In logischen Sprachen sind Variablen stets implizit auf Toplevel existenzquantifiziert. In Curry werden die Variablen in dem Gültigkeitsbereich deklariert in dem sie auch auftauchen, daher sollte man freie Variablen in Curry durch `free` entweder in `let`-Ausdrücken oder `where`-Klauseln deklarieren.

`hasdaughter m | let c free in {ismother m = c} = True` oder
`hasdaughter m | {ismother m = c} = True where c free`

Um einen besseren Stil für solch logische Konstrukte zu erreichen werden Variablen, deren Name mit '_' (underscore) beginnen implizit als freie Variablen deklariert. Damit läßt sich obiges Beispiel in Curry einfacher schreiben als:

```
hasdaughter m | {ismother m = _c} = True
```

Hier besteht auch die Möglichkeit die freie Variable `_c`, ähnlich zu Prolog, durch eine anonyme Variable ('_') zu ersetzen.

3.3 Lokale Deklarationen

Um nicht alle definierten Einheiten global sichtbar zu machen besteht die Möglichkeit den Gültigkeitsbereich z.B. einer Funktionsdeklaration einzuschränken. Dazu stehen ebenfalls die Konstrukte `let` und `where` zur Verfügung. Während `let` in Gleichungen der Form

$$\begin{array}{ll} \texttt{let k} & = 1024 \\ \texttt{in m} & = 640 * \texttt{k} \end{array}$$

benutzt wird, kann `where` dazu dienen Gültigkeitbereiche von Funktionen einzuschränken. In folgender Definition:

```
perm([ ],[ ]).
perm([E|L],[F|M]) = select(F,[E|L],N), perm(N,M)
    where select(E,[E|L],L).
          select(E,[F|L],[F|M]) = select(E,L,M)
```

ist der Gültigkeitsbereich der Funktion `select` auf die Definition der Klausel von `perm` beschränkt. Durch solche `where`-Konstrukte lasen sich zusätzlich nicht-deklarative Aspekte der Logikprogrammierung vermeiden (assert,retract) und eine bessere Strukturierung der Programme erzielen.

3.4 Höhere Ordnung

Als Programmiersprache höherer Ordnung bietet Curry die Möglichkeit funktionale Programmiertechniken, wie partielle Applikation und Lambdaabstraktion zu benutzen. Da die hier behandelte Analysemethode aber keine Programme mit Konstrukten höherer Ordnung unterstütz sei der interessierte Leser auf [Han98] oder [HKMN95] verwiesen.

3.5 Constraints

Die Bedingung C (Constraint) in einer bedingten Regel muß gelöst werden, um die Regel anwenden zu können. Die einfachste Form eines Constraints ist dabei die Gleichung $e_1 = e_2$ zweier Ausdrücke mit Basistyp. Die Gleichung $e_1 = e_2$ ist dabei erfüllt, wenn beide Ausdrücke zum gleichen Grundterm reduziert werden können. Diese als strikte Gleichheit bezeichnete Form der Gleichheit ist die einzig vernünftige im Zusammenhang mit nicht-terminierenden Funktionen und wird daher auch in funktionalen Programmiersprachen verwendet. Dies bedeutet, sollte einer der Ausdrücke e_1, e_2 nicht terminieren, so ist der Constraint nicht erfüllt.

Desweiteren können einzelne Constraints $C_1, \ldots, C_n$ mittels $\{C_1, \ldots C_n\}$ zu einer Konjunktion von Constraints kombiniert werden, welche concurrent ausgewertet werden. Dies bedeutet z.B. für eine sequentielle Implementierung, daß zunächst versucht wird C_1 zu erfüllen. Wird diese Berechnung suspendiert, startet man die Auswertung des Constraints C_2, um vielleicht zu einem späteren Zeitpunkt wieder mit C_1 fortzufahren. Zusätzlich zur Notation $\{C_1, \ldots, C_n\}$ können konjunktive Constraints auch mittels des Infix-Operators /\ definiert werden ($C_1/\backslash C_2 = \{C_1, C_2\}$).

Eine detailliertere Beschreibung der Programmiersprache Curry mit Behandlung der vordefinierten Sprachelemente, dem Modulkonzept, der monadischen Ein/-Ausgabe, der eingekapselten Suche und weiteren Eigenschaften findet sich wie angesprochen in [Han98] oder [HKMN95].

3.6 Beispiele

Einige in Curry implementierte Beispielfunktionen befinden sich im Anhang E, wo die Beispielprogramme für die Ergebnisse der Analyse aufgelistet sind. Aber auch in [Han98] sind viele kleinere Programmierbeispiele enthalten.

3.7 Zusammenfassung

Curry beseitigt einige unsauberen (nicht-deklarativen) Aspekte der Logikprogrammierung:

- Cuts und andere Operatoren zur Beschneidung des Suchbaums finden in Curry keine Entsprechung, da Funktionen deterministisch ausgewertet werden.

- das call-Prädikat wird durch die Möglichkeiten zur Definition von Funktionen höherer Ordnung überflüssig.

- Viele Situationen in denen in logischen Sprachen `assert` und `retract` verwendet werden mußten, lassen sich durch Implikation und Quantifizierung verhindern.

- Die I/O-Funktionen von Prolog wurden durch das monadische Konzept der funktionalen Programmierung ersetzt.

Auch wenn nicht jedes 'unsaubere' Prolog-Programm direkt in Curry übertragbar ist, so lassen sich doch alle rein logischen Prolog-Programme durch Transformation von Klauseln der Form $p \Leftarrow p_1, \ldots, p_n$ in die Notation $p \mid \{p_1, \ldots, p_n\} = true$ und der Einschränkung der Auswertungsstrategie für die Konjunktion ',' auf $1 : (true \Rightarrow 2)$ [erstes Argument muß zu *true* Ausgewertet werden, bevor die Auswertung des zweiten Arguments beginnt] direkt in Curry übertragen.

Damit sollte die Programmiersprache Curry ausreichend vorgestellt sein, um eigene kleinere Programme zu entwickeln und diese duch die im folgenden beschriebene Analyse bearbeiten zu können.

Kapitel 4

Definierende Bäume

Wie schon erwähnt dienen definierende Bäume in Curry zur Beschreibung der Berechnungsvorschrift einer jeden definierten Funktion. In diesem Abschnitt soll gezeigt werden, was ein definierender Baum ist und wie man auf definierenden Bäumen eine Narrowingstrategie definieren kann. Desweiteren wird die schon angesprochene Transformation von definierenden Bäumen in einfache Termersetzungssysteme beschrieben und an Beispielen verdeutlicht.

4.1 Berechnungsmodell mit definierenden Bäumen

Ein definierender Baum T ist eine hierarchische Struktur, die endlich ist und als Blätter alle Regeln der Funktion enthält. T ist ein definierender Baum mit Pattern π, wenn einer der Fälle 1) - 4) vorliegt.

1. $T = rule(l \to r)$, wobei $l \to r$ Variante einer Regel mit $l = \pi$ ist oder
2. $T = branch(\pi, o, r, T_1, ..., T_k)$, wobei o eine Variablenposition in π, r $\in$ {rigid,flex} und $c_1, \ldots, c_k$ sind Konstruktoren des Typs von $\pi|_o$ und $\forall i \in \{1, \ldots, k\}; k > 0$ gilt: T_i ist ein definierender Baum mit Pattern $\pi[c_i(X_1, \ldots, X_n)]|_o$, wobei n die Stelligkeit von c_i ist und $X_1, \ldots, X_n$ sind neue Variablen.
3. $T = or(T_1, T_2)$, wobei T_1 und T_2 definierende Bäume mit Pattern π sind.
4. $T = and(T_1, T_2)$, wobei T_1 und T_2 definierende Bäume mit Pattern π sind.

Ein 'definierender Baum für die n-stellige Funktion f' ist ein definierender Baum mit Pattern $f(X_1, \ldots, X_n)$, wobei $X_1, \ldots, X_n$ neue Variablen sind, so daß für jede Regel $l \to r$ mit $l = f(t_1, \ldots, t_n)$ (t_i Terme) ein Knoten $rule(l' \to r')$ in T existiert derart, daß l Variante von l' ist. Eine Funktion f/n heißt induktiv sequentiell, falls ein definierender Baum von f mit Pattern $f(X_1, \ldots, X_n)$ mit paarweise verschiedenen Variabeln $X_1, \ldots, X_n$ existiert. Ein Termersetzungssystem $\mathcal{R}$ heißt induktiv sequentiell, falls alle $f \in \mathcal{R}$ induktiv sequentiell sind.

Im Folgenden soll kurz erklärt werden, welche Berechnungen durch die einzelnen Knoten eines definierenden Baums spezifiziert werden.

- Liegt ein *rule*-Knoten vor, so wird die entsprechende Regel angewendet.
- Liegt ein $branch(\pi, o, r, T_1, \ldots, T_n)$-Knoten vor, so wird der Unterterm an der Stelle o bis zur Kopfnormalform ausgewertet und mit dem, dem Topkonstruktor entsprechenden, Teilbaum T_i fortgefahren. Ist der Unterterm an der Stelle o eine Variable, so gibt es zwei Möglichkeiten
  1. r=rigid: die Auswertung des Funktionsaufrufs wird verzögert, bis die Variable gebunden wurde und es wird zunächst ein anderer Funktionsaufruf ausgewertet (= Residuationschritt).
  2. r=flex: Die Variable wird an die verschiedenen Konstruktoren der Teilbäume gebunden und entsprechend fortgefahren (= Narrowingschritt).
- Liegt ein *or*-Knoten vor, so gibt es kein eindeutiges Argument, um zu entscheiden, welcher Teilbaum zu bearbeiten ist (Disjunktion).
- Liegt ein *and*-Knoten vor, so ist es nötig mehr als eine Argumentposition zu betrachten. Suspendiert dabei ein Argument, so wird mit dem nächsten fortgefahren. Da es stets möglich ist einen definierenden Baum ohne and-Knoten zu konstruieren werden diese nur eingefügt, um die zweistellige Boolean-Funktion and/2 zu realisieren oder um parallele Auswertung zu ermöglichen.

(Eine detaillierte Definition der Berechnungsvorschrift findet sich in [AEH94] und [Han97])

Die Abbildung 4.1 auf Seite 24 zeigt am Beispiel der Funktionen $append/2$ und $\leq /2$ von Curry erstellte definierende Bäume.

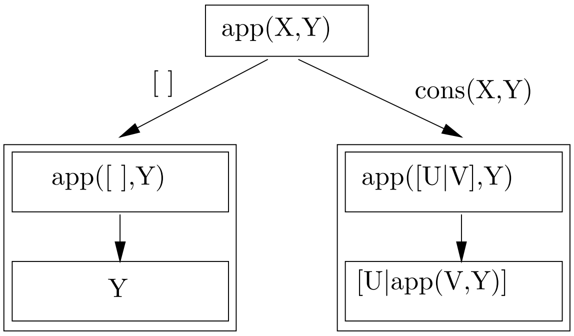
Die oben angesprochenen Modelle 'Needed Narrowing' und 'Residuation' lassen sich durch einfache Einschränkungen aus dem hier gezeigten Modell herleiten:

1. Enthalten die definierenden Bäume jeder Funktion nur *rule*- und *branch*-Knoten und jeder *branch*-Knoten ist als 'flex' markiert, so ist obige Strategie äquivalent zum 'Needed Narrowing'.
2. Das Verhalten der Residuation Strategie kann einfach dadurch simuliert werden, daß alle nicht-boolean Funktionen durch *branch*-Knoten mit der Kennzeichnung 'rigid' realisiert werden.

Hier sieht man auch, daß der Unterschied zwischen 'Narrowing' und 'Residuation' im vorgestellten Berechnungsmodell lediglich in einem flag liegt, in der Kennzeichnung 'rigid' oder 'flex'. Die automatische Generierung von definierenden Bäumen aus einer Regelmenge $\mathcal{R}$ ($\mathcal{R}$ ist konstruktorbasiertes, schwach-orthogonales TES) ist für folgende strategischen Annahmen in [Han97] beschrieben:

Beispiele:

app([ ],L)=L.
app([H|T],L)=[H|app(T,L)].



≤ (0,X)=true.
≤ (s(X),0)=false.
≤ (s(X),s(Y))= ≤ (X,Y).

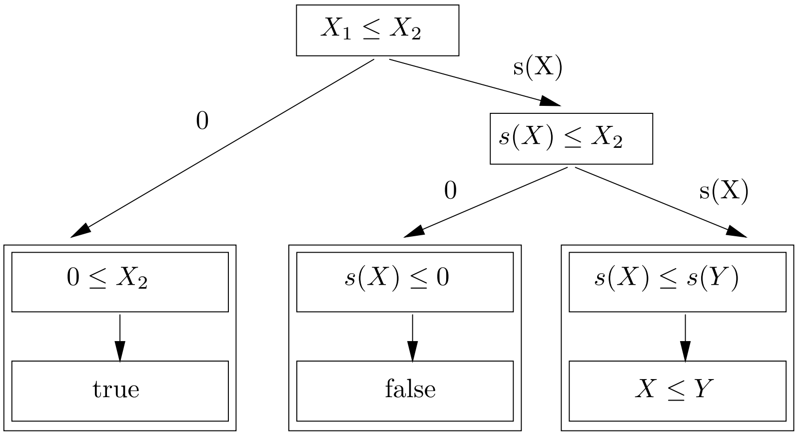


Abbildung 4.1: Beispielbäume für append und ≤

1. Pattern-matching wird von links nach rechts durchgeführt.

2. *or*-Knoten werden generiert, wenn an einer Argumentposition ein Konflikt zwischen Konstruktor und Variable auftritt.

Diese Strategie ist vernünftig, da sie für die meisten funktionalen Programme ohne *or*-Knoten auskommt und in vielen aktuellen funktionalen Programmiersprachen eingesetzt wird. Dennoch kann es sein, daß diese Strategie nicht den optimalen definierenden Baum liefert.

4.2 Erweiterung des Berechnungsmodells

Hier sollen noch kurz einige Erweiterungen angesprochen werden

- Funktionen höherer Ordnung: Hierfür ist es ausreichend eine *apply*-Funktion ($apply f x = f x$) einzuführen, deren erstes Argument den Modus 'rigid' hat. λ-Abstraktionen in rechten Regelseiten können ähnlich behandelt werden.

- bedingte Regeln: Man kann eine bedingte Regel $l \to r \Leftarrow c$ als syntaktische Vereinfachung für die Regel $l \to (c \Rightarrow r)$ mit Guard-Ausdruck auf der rechten Seite betrachten. Die operationelle Bedeutung von $(c \Rightarrow r)$ wird dabei durch die einzige Regel $(true \Rightarrow X) \to X$ definiert. Kann die Bedingung c zu *true* ausgewertet werden, so wird sie durch r ersetzt. Kann sie nicht zu *true* ausgewertet werden und wird sie nicht suspendiert, so wird das disjunktive Glied in dem sie auftaucht gelöscht.

4.3 Narrowing und definierende Bäume

Es liegt ein Narrowingschritt $t \rightsquigarrow t'$ bzgl. eines definierenden Baumes T und einer Substitution σ vor, wenn

1. $T = rule(l \to r) : t = \sigma(l), t' = \sigma(r)$ oder

2. $T = branch(\pi, o, r, T_1, \dots, T_n)$, bzgl. des Unterterms $t|_o$ betrachten wir folgende 3 Fälle

   (a) $t|_o = f(\dots)$ mit $f \in \mathcal{D}$, dann führe einen Narrowingschritt bzgl. des definierenden Baumes von f durch und wende die Substitution σ auf den ganzen Term an.

   (b) $t|_o = c(...)$ mit $c \in \mathcal{C}$, dann führe einen Narrowingschritt mit t bzgl. desjenigen T_j durch, dessen Pattern sich mit t matchen läßt. Existiert kein solches Pattern erhalten wir einen Fehlschlag.

(c) t_o ist eine Variable, dann wähle nicht-deterministisch einen Teilbaum T_j aus, binde $t|_o$ an $\pi_j|_o$ (π_j ist das Pattern von T_j) und führe einen Narrowingschritt für $t[t|_o \to \pi_j|_o]$ bzgl. T_j aus. Für die neue Substitution σ' gilt $\sigma' = \{t|_o \to \pi_j|_o\} \cup \sigma_j$, wobei σ_j die Substitution des Schrittes bzgl. T_j ist.

Ein needed Narrowingschritt $t \rightsquigarrow_\sigma t'$ ist ein Narrowingschritt für t bzgl. des definierenden Baumes von $root(t)$. Die Substitution σ ist die Substitution der Ableitung eingeschränkt auf $Var(t)$. Ist t ein Konstruktorterm, so wähle mittels leftmost-outermost den nächsten Funktionsaufruf für einen needed Narrowingschritt.

4.4 Transformation von definierenden Bäumen

Hier soll nun beschrieben werden, durch welche Transformation sich definierende Bäume, die die Berechnungsgrundlage von Curry bilden, in Termersetzungssysteme mit einfachen Regeln überführen lassen. Zunächst sei die Definition der Transformation Π: Tree → TES: T → $\mathcal{R}_E$, welche einen definierenden Baum aus der Menge Tree der definierenden Bäume abbildet auf ein Termersetzungssystem $\mathcal{R}_E$, angegeben. Diese Transformation ist eine Erweiterung der in [Zar97] angegebenen, Transformation. Hier werden zusätzlich aber auch *or*-Knoten behandelt.

- $\Pi(or(T_1, \ldots, T_n)) = \bigcup\limits_{i=1}^{n} \Pi(T_i)$
- $\Pi(rule(\pi, f(X) \to r)) = \{f_\pi(X) \to r\}$
- $\Pi(branch(\pi, o, r, \overline{T})) = \bigcup\limits_{1 \le i \le n} \Pi(branch(\pi, o, T_i)$
- $\Pi(branch(\pi, o, r, T) = \{f_\pi(\overline{X}, c(\overline{Y}), \overline{Z}) \to r\}$, falls $T = rule(\pi, l \to r)$
- $\Pi(branch(\pi, o, r, T)) = \{f_\pi(\overline{X}, c(\overline{Y}), \overline{Z}) \to f_{\pi'}(\overline{X}, \overline{Y}, \overline{Z}) \cup \bigcup\limits_{i=1}^{n} \Pi(branch(\pi', o', T_i))$, falls $T = branch(\pi', o', \overline{T'})$
- $\Pi(branch(\pi, o, r, or(\overline{T})) = \{f_\pi(\overline{X}, c(\overline{Y}), \overline{Z}) \to f_{\pi'}(\overline{X}, \overline{Y}, \overline{Z})\} \cup \bigcup\limits_{i=1}^{n} \Pi(T_i)),$
  für alle $T_i = rule(\pi', l \to r)$
  und alle $T_i = branch(\pi', o', \overline{T'})$

hierbei sind: $\pi'|_o = c(\overline{Y})$, $\overline{X}$ und $\overline{Y}$ sind die Variablen vor bzw. nach der Position o in π' in leftmost-innermost Reihenfolge. $\overline{T} = (T_1, ..., T_m)$ für $m \in \mathbb{N}$. Abschließend können noch die f_π durch f ersetzt werden, bei denen das Pattern π dem Top-Pattern des definierenden Baumes entspricht. Sei $\mathcal{P}$ die Menge aller Pattern π für die eine Regel $f_\pi \to r$ im entstandenen Regelsystem $\mathcal{R}_E$ existiert.

Wir definieren eine Nummerierung der Elemente in $\mathcal{P}$ mittels einer bijektiven Abbildung $num : \mathcal{P} \rightarrow \{2, \ldots, \mid \mathcal{P} \mid +1\} : num(\pi) \rightarrow n$ und ersetzen alle Symbole f_π in $\mathcal{R}_E$ durch $f_{num(\pi)}$. Sei $\mathcal{D}_E$ die Menge der neuen Funktionssymbole in $\mathcal{R}$, dann erweitern wir $\mathcal{D}$ auf $\mathcal{D} \cup \mathcal{D}_E$. Das entstandene Termersetzungssytem $\mathcal{R}_E$ enthält nun ausschließlich Regeln der Form:

- $f_n(X_1, \ldots, X_n) \rightarrow r$ mit $X_i \neq X_j$ für $i \neq j, n \in \mathbb{N}$ oder
- $f_n(\overline{X}, c(\overline{Y}), \overline{Z}) \rightarrow r$ mit $\overline{X} = \{X_1, \ldots, X_n\}, \overline{Y} = \{Y_1, \ldots, Y_n\}$ und $\overline{Z} = \{Z_1, \ldots, Z_n\}$ sind disjunkte Mengen paarweise verschiedener Variabeln. Die c aus $\mathcal{C}$ sind verschieden für verschiedene Regeln einer Funktion f.

Der Index n am Funktionsnamen entfällt bei Funktionen, die nicht erst durch die Transformation eingebunden wurden. Es soll nun anhand eines Beispiels die Arbeitsweise der Transformation Π vorgestellt werden. Betrachten wir dazu die Funktion `dotmult/2`, welche zwei Vektoren (Listen) von Zahlen multipliziert.

```
dotmult([ ],[ ]) = 0.
dotmult([X|Xs],[Y|Ys]) = X*Y + dotmult(Xs,Ys)
```

Der von Curry berechnete definierende Baum für diese Funktion `dotmult/2` sieht folgendermaßen aus:

```
branch (dotmult(A,B),{1},rigid,
       {branch (dotmult([ ],B),{2},rigid,
               {rule(dotmult([ ],[ ]),0)}),
        branch (dotmult(:(C,D),B),{2},rigid,
               {rule(dotmult(:(E,F),:(G,H)),+(*(E,G),dotmult(F,H)))})
       })
```

Er besteht also aus einem `branch`-Knoten gefolgt von einem weiteren, der dann nur die erste Regel enthält. Den zweiten Teilbaum des äusseren `branch`-Knotens bildet wiederum ein weiteres `branch` mit einer Regel. Es werden hier zur Transformation also außer den Fällen, die `or`-Knoten behandeln, alle Fälle der Transformation Π benötigt und es ergibt sich das folgende Termersetzungssystem $\mathcal{R}_E$:

$$\begin{array}{ll}
dotmult([\,], X_2) & \rightarrow dotmult_2(X_2) \\
dotmult(:(X_3, X_4), X_2) & \rightarrow dotmult_3(X_3, X_4, X_2) \\
dotmult_2([\,]) & \rightarrow 0 \\
dotmult_3(X_5, X_6, :(X_7, X_8)) & \rightarrow +(*(X_5, X_7), dotmult(X_6, X_8))
\end{array}$$

Das entstandene Termersetzungssystem $\mathcal{R}_E$ verhält sich bei Narrowing-Ableitungen für f genauso, als würden Ableitungen mittels Narrowingschritten bzgl. des definierenden Baumes für f vollzogen (s. [Zar97]). Die Informationen der Baumstruktur sind durch die Einführung der neuen Funktionen f_i nun auch in den Regeln von $\mathcal{R}_E$ verfügbar. Die neuen Funktionssymbole verschwinden, wenn die behandelten Terme bis zu konstruktorbasierten Termen oder Variabeln reduziert werden.

Damit sind die Informationen der definierenden Bäume in einfach strukturierte Termersetzungssysteme umgewandelt. Aufbauend auf diesen Systemen soll nun in den nächsten Kapiteln eine denotationelle Semantik defniert und abstrahiert werden, mit dem Ziel eine Demandanalyse definieren zu können.

Kapitel 5

Demand Analyse

Wie angedeutet soll nun die Demand Analyse, aufbauend auf der denotationellen Semantik beschrieben werden. Zunächst wird dazu die denotationelle Semantik definiert, auf Korrektheit und Vollständigkeit hin überprüft und schließlich zur denotationellen Analyse abstrahiert. Die Definitionen der folgenden Ausführungen entstammen dem Artikel [Zar97] und wurden an gegebenen Stellen durch weitere Erläuterungen ergänzt.

5.1 denotationelle Semantik

Jetzt soll ein Formalismus eingeführt werden, um Narrowing Ableitungen zu beschreiben. Die denotationalle Semantik bildet Terme auf Werte ab, die einige Aspekte der Ableitung des Terms wiedergeben. Später wird dann die denotationelle Semantik durch Änderung der Interpretation zur abstrakten Semantik. Dieser Ansatz hat den Vorteil, daß durch Angabe einer anderen Interpretation, die bestimmte Bedingungen erfüllt, ohne weitere Beweise die Korrektheit der Analyse sichergestellt ist. Die Semantik wird beschrieben durch eine Funktion

$$[\![\cdot]\!] : \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{DP} \rightarrow \wp(\mathcal{AP}) \text{ mit } \mathcal{DP} = \{t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \mid t \text{ ist linear}\},$$

welche einem Term unter Angabe des gewünschten Auswertungsgrads eine Menge von Paaren zuordnet. Die erste Komponente dieser Paare gibt jeweils den Ergebnisterm der Auswertung bzgl. des angegebenen Auswertungsgrades an. Die zweite Komponente ist die, während der Herleitung berechnete, Substitution eingeschränkt auf die Variablen des Ausgangsterms. Die Paare im Ergebnis der Semantik bezeichnen wir als Antwortpaare. ($\mathcal{AP} = \mathcal{T}(\mathcal{F}, \mathcal{X}) \times Sub$). Der Auswertungsgrad wird formell durch einen linearen Konstruktorterm beschrieben, dem Demand Pattern. Die Menge der Demand Pattern bezeichnen wir mit $\mathcal{DP}$. Der Demand Pattern DP hat dabei folgende Bedeutung:

- $DP = x$, mit $x \in Var$, fordert keinerlei Auswertung.

- $DP = c(s_1, \ldots, s_n)$ fordert die Auswertung eines Terms t, hin zu einem Term $c(t_1, \ldots, t_n)$ mit dem gleichen Konstruktor c an der Wurzel und der Auswertung der Unterterme $t_1, \ldots, t_n$ entsprechend $s_1, \ldots, s_n$.

Für die *append*-Funktion auf Listen, sowie für die kleiner-gleich Operation auf den natürlichen Zahlen

$$\begin{array}{lll} append([\,], L) & \to & L & (R_1) \\ append([X|Xs], L) & \to & [X|append(Xs, L)] & (R_2) \end{array}$$

$$\begin{array}{llll} 0 \leq N & \to & true & (R_3) \\ s(M) \leq s(N) & \to & M \leq N & (R_4) \\ s(M) \leq 0 & \to & false & (R_5) \end{array}$$

seien nun einige Beispiele angegeben

$$\begin{array}{l} [\![append(X, Y)]\!] \; [V_1|V_2] = \{ \; ([Y1|Ys], \{X \to [\,], Y \to [Y1|Ys]\}), \\ \qquad\qquad\qquad\qquad\qquad ([X1|append(Xs, Y)], \{X \to [X1|Xs]\}), \ldots \} \end{array}$$

$$\begin{array}{l} [\![s(U) \leq V]\!] \; true = \{ \; (true, \{U \to 0, V \to s(V_1)\}), \\ \qquad\qquad\qquad\qquad (true, \{U \to s(0), V \to s(s(V_2))\}), \ldots \} \end{array}$$

Dabei bezeichnet das erste Beispiel die Auswertung von $append(X, Y)$ bis als Top-Konstruktor ein *cons* steht. Dazu kann man die erste Regel anwenden, wodurch X an die leere Liste gebunden wird und L als rechte Seite dem Demand Pattern genügen muß. Oder man benutzt die zweite Regel, wo die rechte Seite sofort nach Unifikation der gewünschten Form entspricht. Im zweiten Beispiel soll der Term $s(U) \leq V$ zu $true$ ausgewertet werden. Dies ist immer dann der Fall, wenn V an eine um mindestens eins grösere Zahl gebunden wird als V.

Zur formalen Beschreibung der denotationellen Semantik definieren wir die Hilfsfunktion DS (von disagreement Set, s. Abbildung 5.1), welche als Parameter ein Demand Pattern π und einen Term t erhält und entweder eine Position liefert, die weiterer Auswertung bedarf, oder durch das eindeutige Element $\diamondsuit$ angibt, das keine weitere Auswertung von t bzgl. π notwendig ist.

Wir definieren nun die Semantikfunktion $[\![.]\!]$ als kleinsten Fixpunkt eines Funktionals Φ über der Funktionsmenge $\mathcal{T}(\mathcal{F}, \mathcal{X})) \times \mathcal{DP} \to \wp(\mathcal{AP})$. Die Funktionen dieser Menge sind dabei mittels der Mengeninklusion auf ihren Bildern geordnet. Der kleinste Fixpunkt existiert dabei stets, da $(\wp(\mathcal{AP}), \subseteq)$ eine vollständige partielle Ordnung bildet und Φ stetig ist.

Das Funktional Φ (s. Abbildung 5.2) erhält als Argumente das bisherige Ergebnis δ mit: $\delta : \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{DP} \to \wp(\mathcal{AP})$ sowie den zu analysierenden Term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ und das übergebene Demand Pattern $\pi \in \mathcal{DP}$. Ist nun $DS(\pi, t) = \diamondsuit$, so wird ein Antwortpaar $(t, \{\})$ berechnet. Andernfalls sei $p = DS(\pi, t)$ mit $p \in \mathcal{P}os$ und es wird zwischen verschiedenen Fällen unterschieden. Haben die Teilterme an der Position p unterschiedliche Konstruktoren, so erhält man die

Domains:
$AP = \mathcal{T}(\mathcal{F},\mathcal{X}) \times Sub$, die Menge der Antwortpaare
$DP = \{t \in \mathcal{T}(\mathcal{C},\mathcal{X}) |\ t$ ist linear $\}$, die Menge der Demand Pattern

Hilfsfunktion DS:
$DS : \mathcal{DP} \times \mathcal{T}(\mathcal{F},\mathcal{X}) \to Pos \cup \{\diamondsuit\}$, mit

$$
\begin{array}{ll}
DS(\pi, t) & = \diamondsuit; \pi \in \mathcal{X} \\
& \{\text{Variable fordert keine Auswertung}\} \\
DS(c(\overline{\pi}), c(\overline{t})) & = j.p; DS(\pi_i, t_i) = \diamondsuit, i = 1, \ldots, j-1, DS(\pi_j, t_j) = p \\
& \{\text{Term } j \text{ ist erster Unterterm, für weitere Auswertung }\} \\
DS(c(\overline{\pi}), c(\overline{t})) & = \diamondsuit; DS(\pi_i, t_i) = \diamondsuit \text{ für alle i} \\
& \{\text{kein Unterterm bedarf weiterer Auswertung}\} \\
DS(c(\overline{\pi}), d(\overline{t})) & = \epsilon; c, d \in \mathcal{C} \text{ mit } c \neq d \text{ oder } c \in \mathcal{C} \text{ und } d \in \mathcal{D} \\
& \{\text{Topsymbole stimmen nicht überein}\} \\
DS(c(\overline{\pi}), x) & = \epsilon; c \in \mathcal{C} \text{ und } x \in \mathcal{X}
\end{array}
$$

Abbildung 5.1: Die Hilfsfunktion DS

Funktionen:
$[\![\,]\!] : \mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{DP} \to \wp(\mathcal{AP})$
$\Phi : (\mathcal{T}(\mathcal{F},\mathcal{X})) \times \mathcal{DP} \to \wp(\mathcal{AP}) \to (\mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{DP} \to \wp(\mathcal{AP}))$
$step : \mathcal{T}(\mathcal{F},\mathcal{X}) \times Pos \times \mathcal{R}_\epsilon \to \wp(\mathcal{AP})$ { Einzelschrittrelation }

Gleichungen:

$$
\begin{array}{lll}
[\![t]\!]\pi = (lfp\ \Phi)(t,\pi) & & \\
\Phi(\delta, t, \pi) = \{(t, \{\})\} & DS(\pi, t) = \diamondsuit, \text{ sonst } p = DS(\pi, t) & \\
\Phi(\delta, t, \pi) = \{\} & root(t|_p), root(\pi|_p) \in \mathcal{C}, root(t|_p) \neq root(\pi|_p) & \\
\Phi(\delta, t, \pi) = \displaystyle\bigcup_{(s,\beta)\in\Phi(\delta,\xi t,\pi)} \{(s, \beta\xi|_{\mathcal{V}ar(t)})\} & t|_p \in \mathcal{X}, \xi = \{t|_p \to c(\overline{Y})\}, \pi|_p = c(\overline{\pi}) & \\
\Phi(\delta, t, \pi) = \displaystyle\bigcup_{\substack{R\in\mathcal{R}_\epsilon \\ (s,\xi)\in step(t,p,R) \\ (r,\beta)\in\delta(s,\pi)}} \{(r, \beta\xi|_{\mathcal{V}ar(t)})\} & t|_p \text{ ist Funktionsaufruf} &
\end{array}
$$

$$
\begin{array}{ll}
step(t, p, l \to r) = \{\} & root(t|_p) \neq root(l), \text{ sonst} \\
step(t, p, l \to r) = \displaystyle\bigcup_{R\in\mathcal{R}_\epsilon} step(t, p.k, R) & l|_k = c(\overline{Y}), t|_{p.k} \text{ ist Funktionsaufruf} \\
step(t, p, l \to r) = \{(\sigma(t[r]_p), \sigma|_{\mathcal{V}ar(t)})\} & \sigma \in mgu(l, t|_p) \\
step(t, p, l \to r) = \{\} & fail = mgu(l, t|_p), root(t|_p) = root(l)
\end{array}
$$

Abbildung 5.2: Die denotationelle Semantik leere Menge, da sich der Term t nicht bzgl. π auswerten läßt. Ist $t|_p$ eine Variable, so wird diese an den Top-Konstruktor von $\pi|_p$, angewendet auf frische Variablen $\overline{y}$, gebunden. Ist $t|_p$ ein Funktionsaufruf, so versucht die Funktion *step* einen Narrowingschritt auszuführen und die Funktion δ wird angewendet. Im letzten Fall werden die einzelnen Substitutionen zusammengesetzt und auf Variablen von t eingeschränkt. Man beachte, daß aus $(s, \alpha) \in [\![t]\!]\ \pi$ folgt, daß s eine Instanz des Demand Pattern π ist.

Das folgende Theorem belegt die Vollständigkeit und Wohldefiniertheit dieser denotationellen Sematik.

Theorem
Es existiert eine Needed-Narrowing Ableitung $s = t \rightsquigarrow^*_\sigma$ true bzgl. des Termersetzungssystems $\mathcal{R}$ genau dann, wenn ein Antwortpaar $(true, \sigma)$ in $[\![s = t]\!] true$ existiert.

Nicht jedes Antwortpaar entspricht dabei einer Ableitung, da z.B. bei einem Term $x \in \mathcal{X}$ und einem Demand Pattern $c(t_1, \dots, t_n)$ mit $c \in \mathcal{C}$ eine Bindung von x an den Term $c(t_1, \dots, t_n)$ stattfindet, die nicht zu einem Narrowing-Schritt gehört.

Im folgenden soll die Frage beantwortet werden, wie man die Semantik eines Terms aus der Semantik seiner Unterterme bzw. wie man die Ableitung eines Terms aus den Ableitungen seiner Unterterme erhält. Enthält ein Funktionsaufruf keine Variablen und der Auswertungsgrad der Parameter ist bekannt so können wir die Auswertung der einzelnen Parameter getrennt betrachten. Dieser Fall entspricht der Termersetzung. Im Fall des Narrowings jedoch enthalten die verschiedenen Unterterme eventuell gemeinsame Variablen und die Auswertung der Unterterme liefert eventuell unvereinbare Substitutionen für diese Variablen. Da die Semantik Informationen über die berechneten Substitutionen enthält ist es möglich eine Operation $\sqcup$ zu definieren, welche angewendet auf eine Menge von Substitutionen eine neue Substitution liefert, welche alle Bindungen der Ausganssubstitutionen vereinigt.

Definition:
Sei A eine endliche, nichtleere Menge von Substitutionen, dann definieren wir $\sqcup(A)$ durch: $\sqcup(A) = mgu\{(x, ax) | a \in A, x \in dom(a)\}$

Wir wollen also die Semantik eines Terms t durch die Semantik seiner Unterterme berechnen. Dies bedeutet wir wollen $(u, \sigma) \in [\![t]\!]\pi$ aus den Paaren $(v, \sigma_2) \in [\![t[Z]|_p]\!]\pi$ und $(w, \sigma_3) \in [\![t|_p]\!]\sigma_2(Z)$ errechnen. Dabei taucht ein Problem auf, welches durch doppelte Variablenvorkommen auf rechten Regelseiten verursacht wird.

Zur Verdeutlichung diene folgendes Beispiel mit den drei Regeln:

$$\begin{array}{llll} pair(X) & \rightarrow & c(X, X) & (R_1) \quad \text{!!! } kritischeRegel \\ 0 + N & \rightarrow & N & (R_2) \\ s(M) + N & \rightarrow & s(M + N) & (R_3) \end{array}$$

Elementare Terme

$$
\begin{array}{llll}
[\![X]\!] & \pi & = \{(\pi, \{X \to \pi\})\} & \pi \notin \mathcal{X}, X \in \mathcal{X} \\
[\![c(\overline{X})]\!] & d(\overline{t}) & = \{\} & d \neq c \\
[\![c(\overline{X})]\!] & c(\overline{\pi}) & = \{(c(\overline{s}, \nu) | (s_i, \beta_i) \in [\![X_i]\!]\pi_i, \forall i, \nu \in \sqcup\overline{\beta}\} & c \in \mathcal{C} \\
[\![f(\overline{X})]\!] & c(\overline{\pi}) & = \bigcup_{\substack{l \to r \in \mathcal{R}_\epsilon \\ \sigma \in mgu(l, f(\overline{X}))}} \{(s, \beta\sigma|_{\overline{x}} \,|(s, \beta) \in [\![\sigma r]\!]\pi\} & f \in \mathcal{D}_\epsilon, c \in \mathcal{C}
\end{array}
$$

Komposition

$$
\begin{array}{llll}
[\![t]\!] & \pi & = \{(t, \{\})\} & \pi \in \mathcal{X} \\
[\![f(\overline{t})]\!] & \pi & = \{(\nu s, \nu|_{Var(f(\overline{t}))}) \mid (s, \alpha) \in [\![f(\overline{Z})]\!]\,\pi & \pi \notin \mathcal{X}, \\
 & & \quad (u_i, \beta_i) \in [\![t_i]\!]\,\alpha Z_i, i = 1, \ldots, n & f/n \in \mathcal{F}_\epsilon \\
 & & \quad \nu \in \sqcup\{\beta_i, \alpha, \{Z_i \to u_i\} | i = 1, \ldots, n\} &
\end{array}
$$

Abbildung 5.3: Die Komposition der Semantik

mit der zu berechnenden Semantik $[\![pair(Y+0)]\!]\ c(0,V)$.
Dabei ergibt sich folgendes Problem:

$$
\begin{array}{lcl}
[\![pair(Z)]\!]\ c(0,V) & = & \{(c(0,0), \{Z \to 0\})\} \text{ und} \\
[\![Y+0]\!]\ 0 & = & \{(0, \{Y \to 0\})\} \\
\text{liefern als Ergebnis} & & \\
[\![pair(Y+0)]\!]\ c(0,V) & = & \{(c(0,0), \{Y \to 0\})\} \\
\text{korrekt wäre aber} & & \\
[\![pair(Y+0)]\!]\ c(0,V) & = & \{(c(0,0+0), \{Y \to 0\})\}.
\end{array}
$$

Dieses Problem läßt sich mit Hilfe von Markierungen gemeinsamer Teilterme lösen (s. [Zar97]). Dadurch ist man in der Lage folgendes Theorem aufzustellen, welches die Korrektheit der Komposition der Semantik eines Terms t aus der Semantik seiner Teilterme belegt.

Theorem (Komposition):
Sei $(t, \pi) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{DP}$, Z eine neue Variable und $p \in \mathcal{P}os(t)$. Wenn $t|_p$ und $t[Z]_p$ keine gemeinsamen Markierungen im Sinne von [Zar97] haben, dann gilt:

$$
\begin{array}{ll}
[\![t]\!]\ \pi = \{(\nu s, \nu|_{Var(t)}) \mid & (s, \alpha) \in [\![t[Z]_p]\!]\ \pi \\
 & (u, \beta) \in [\![t|_p]\!]\ \alpha Z, \\
 & \nu \in \sqcup\{\alpha, \beta, \{z \to u\}\}\ \}
\end{array}
$$

Damit läßt sich nun die Berechnung der Semantik eines Terms t aus der Berechnung der Semantik seiner elementaren Unterterme angeben (s. Abbildung 5.3). Als elementare Unterterme gelten dabei Terme, die entweder Variablen oder Funktionssymbole angewendet auf unterschiedliche Variablen sind.

5.2 Denotationelle Analyse

Die Definition der denotationellen Semantik wurde motiviert mit dem Ziel einen Rahmen für statische Analysen zu schaffen. Hier soll nun am Beispiel einer Demand-Analyse dieses Ziel umgesetzt werden. Eine Demand-Analyse setzt sich zum Ziel aus dem geforderten Grad der Auswertung eines Terms t auf den nötigen Grad der Auswertung seiner Unterterme zu schließen. Im hier gezeigten Rahmen wird der gewünschte Grad der Auswertung durch ein Demand Pattern π gegeben und für den Unterterm and der Stelle $p \in \mathcal{P}os(t)$ ist dann eine Auswertung bzgl. $\{\alpha z | (s, \alpha) \in [\![t[Z]|_p^*]\!]\ \pi\}$ gefordert. Dies wird durch das Kompositions-Theorem garantiert.

Um das Antwortpaar für $[\![t]\!]\ \pi$ zu erhalten muß $t|_p$ bzgl. αZ ausgewertet werden. Für eine Analyse wird also Information über αZ benötigt. Diese soll mittels einer abstrakten Interpretation gewonnen werden. Also müssen alle semantischen Objekte durch endliche Objekte beschrieben werden, d.h die Semantik $[\![\cdot]\!]$ wird durch die abstrakte Semantik $[\![\cdot]\!]_A$ mit: $[\![t]\!]_A = \mathcal{DP}_A \rightarrow \mathcal{AP}_A$ ersetzt. Die Demand Pattern $\pi \in \mathcal{DP}$ werden durch abstrakte Pattern $\pi_A \in \mathcal{DP}_A$ ersetzt. Dabei charakterisiert jedes Element π_A eine Menge von Demand Pattern aus $\mathcal{DP}$. Den Übergang vom abstrakten Pattern zur entsprechenden Menge konkreter Pattern bewirkt die Funktion $\gamma : \mathcal{DP}_A \rightarrow \wp(\mathcal{DP})$.

Wir nehmen an, daß die Menge $\mathcal{DP}_A$ eine endliche partielle Ordnung bildet, deren Ordnung der Inklusion der abstrahierten Mengen entspricht, d.h. $\pi_A \leq \pi'_A \iff \gamma(\pi_A) \subseteq \gamma(\pi'_A)$. Der Domain $\mathcal{DP}_A$ muß dabei endlich sein, da es für eine abstrakte Interpretation notwendig ist, daß der abstrakte Domain frei von unendlichen aufsteigenden Ketten ist. Dies sichert auch hier die Existenz des Fixpunktes. Zusätzlich benötigen wir zwei ausgewiesene Elemente $\top$ (Top) und $\bot$ (Bottom) mit $\gamma(\top) = \mathcal{DP}$ und $\gamma(\bot) = \{\}$, die das größte bzw. kleinste Element des Domains darstellen. Nun fehlt noch die Definition eines abstrakten Domains für die Antwortpaare. Dazu abstrahieren wir lediglich den zweiten Teil, die Substitution α zu einer abstrakten Substitution φ_A. Eine Abstrahierung der berechneten Lösung ist nicht nötig, da sie wie oben schon einmal angemerkt lediglich eine Instanz des Demand Pattern ist. Der Domain der Substitution α ist eine Teilmenge von $\mathcal{V}ar(t) = \{X_1, \ldots, X_n\}$. Eine abstrakte Substitution φ_A steht für eine Menge von Substitutionen und hat die Form:

$$
\begin{array}{ll}
\varphi_A = \bigwedge_{i=1}^{n} X_i^{a_i} & \mathcal{V}ar(t) \neq \{\}, a_i \in \mathcal{DP}_A, \text{ mit} \\
 & \gamma(\varphi_A) = \{\alpha \in Sub \mid dom(\alpha) \subseteq \{X_1, \ldots, X_n\}, \alpha(X_i) \in \gamma(a_i), \forall i\} \\
\text{oder} & \\
\varphi_A = true | false & \mathcal{V}ar(t) = \{\}, \text{ mit } \gamma(true) = \{\{\}\} \text{ und } \gamma(false) = \{\}.
\end{array}
$$

Wir erweitern die Ordnung auf $\mathcal{DP}_A$ kanonisch zu einer Ordnung der abstrakten Substitutionen und mit einem festen endlichen Vorrat an Variablen erfüllt auch diese Ordnung die Bedingung, daß keine unendlichen aufsteigenden Ketten existieren. Zusätzlich müssen noch die abstrakten Operationen definiert werden,

$$\begin{array}{rll} proj^i(c, c(\overline{t}) & \in \gamma(proj^i_A(c, \pi_A)) & c(\overline{t}) \in \gamma(\pi_A), c/n \in \mathcal{C}, 1 \leq i \leq n \\ \varphi(t) & \in \gamma(apply_A(t, \varphi_A)) & \varphi \in \gamma(\varphi_A) \\ \sqcup(\varphi_i)_i & \subseteq \gamma(\sqcup_A(\varphi^i_A)_i) & \varphi_i \in \gamma(\varphi^i_A) \forall i \in I \\ \{\varphi_i | i \in I\} & \subseteq \gamma(union_A(\varphi^i_A)_i) & \varphi_i \in \gamma(\varphi^i_A) \forall i \in I \end{array}$$

Abbildung 5.4: Die Eigenschaften der abstrakten Operationen

Abstrakte Semantik

$[\![t]\!]\ \pi_A = (\bigsqcup_{n \geq 0})\ (t, \pi_A)$wobei $\Phi^0_A(t, \pi_A) = false$, für $\mathcal{V}ar(t) = \{\}$ oder

$\Phi^0_A(t, \pi_A) = X^\perp_1 \wedge \ldots \wedge X^\perp_n$, für $\mathcal{V}ar(t) = \{X_1, \ldots, X_n\}$

Einfache Terme

$$\begin{array}{lll} \Phi^{n+1}_A(X, \pi_A) & = X^{\pi_A} & X \in \mathcal{X} \\ \Phi^{n+1}_A(c(\overline{X}, \pi_A) & = \bigwedge_i X_i^{proj^i_A(c,\pi_A)} & c \in \mathcal{C} \\ \Phi^{n+1}_A(f(\overline{X}), \pi_A) & = union_A(match(\Phi^n_A(r_i, \pi_A), \{\overline{t_i}\}))_{f(\overline{t}_i) \rightarrow r_i \in \mathcal{R}_\epsilon} & f \in \mathcal{D}_\epsilon \end{array}$$

Komposition

$$\begin{array}{lll} \Phi^{n+1}_A(t, \pi_A) & = true & \gamma(\pi_A) \cap \mathcal{X} \neq \{\}, \mathcal{V}ar(t) = \{\} \\ \Phi^{n+1}_A(t, \pi_A) & = X^\top_1 \wedge \ldots \wedge X^\top_n & \gamma(\pi_A) \cap \mathcal{X} \neq \{\}, \mathcal{V}ar(t) = \{X_1, \ldots, X_n\} \\ \Phi^{n+1}_A(f(\overline{t}), \pi_A) & = \sqcup_A(\Phi^{n+1}_A(t_i, a_i))_i & f \in \mathcal{F}_\epsilon, \Phi^{n+1}_A(f(\overline{Z}), \pi_a) = \bigwedge_i Z_i^{a_i} \end{array}$$

Abbildung 5.5: Die abstrakte Interpretation

welche die konkreten simulieren sollen. Wir wollen die abstrakte Interpretation in Abhängigkeit von den abstrakten Operationen $proj^i_A$, $apply_A$, $\sqcup_A$ und $union_A$ definieren. Dabei realisiert $apply_A$ die Anwendung einer abstrakten Substituion φ_A auf einen Term t. $proj^i_A$ simuliert die Projektion auf das i-te Element t_i eines Terms $c(t_1, \ldots, t_n)$. Die Operation $\sqcup_A$ abstrahiert die Operation $\sqcup$, berechnet also aus verschiedenen abstrakten Substitutionen eine neue abstrakte Substitution, welche die Eigenschaften der ursprünglichen vereinigt. $union_A$ schließlich simuliert die Vereinigung auf Mengen von abstrakten Substitutionen. Zusätzlich sei noch die Operation $match(\{X_1, \ldots, X_n\}, \psi, \{t_1, \ldots, t_n\}) = \bigwedge_{i=1}^{n} X_i^{apply_A(t_i, \psi)}$ definiert, welche die abstrakte Substitution liefert, die aus der Anwendung der abstrakten Substitution ψ auf eine Menge von Teiltermen $\{t_1, \ldots, t_n\}$ entsteht. Diese Operationen sollten zusätzlich zur Monotonie noch die in Abbildung 5.4 aufgeführten Eigenschaften erfüllen.

Auf Grundlage dieser domainabhängigen Funktionen können wir nun die in Abbildung 5.5 gezeigte abstrakte Semantik definieren.

Die Sequenz der Φ^n bildet eine aufsteigende Liste abstrakter Substitutionen und stabilisiert sich aufgrund der Eigenschaft, daß keine unendlichen aufsteigenden Ketten existieren. Die Korrektheit der abstrakten Semantik wird durch folgendes Theorem sichergestellt.

Theorem 7 (Korrektheit)
Seien $\pi \in \gamma(\pi_A), t \in \mathcal{T}(\mathcal{F}, \mathcal{X}), \overline{x} = Var(t)$ und $[\![t]\!]\ \pi_A = \bigwedge_i x_i^{a_i}$, dann gilt:

$$(s, \alpha) \in [\![t]\!]\ \pi \ \Rightarrow\ \alpha x_i \in \gamma(a_i) \text{ für alle i.}$$

Damit sind die mathematischen Grundlagen zur Implementierung einer Demand-Analyse geschaffen. Aufbauend auf den Definitionen dieses Kapitels soll nun ein Analysesystem für die Programmiersprache Curry entwickelt werden, welches die gezielte Analyse von einzelnen Funktionen eines Curryprogramms erlaubt. Dieses System soll eine Grundlage bilden für die spätere Entwicklung eines Compilers, der die Informationen der Analyse zur Codeoptimierung einsetzt. Daher soll das zu implementierende Analysesystem seine Ergebnisse in einer Datei zur Verfügung stellen, die dann als Eingabe für den Compiler genutzt werden kann. Wie man effiziente lazy Narrowing Strategien mit Hilfe einer Demand Analyse definieren kann zeigt unter anderem [MNKMC+93].

Kapitel 6

Implementierung

In diesem Kapitel sollen die einzelnen Teile der Implementierung der vorgestellten Demandanalyse besprochen werden. Dabei soll besonders auf die Steuerung der Fixpunktberechnung eingegangen werden, um darzulegen, welche Datenstrukturen weshalb nötig sind, um die korrekte Berechnung des Fixpunktes zu garantieren. Aufgrund der Tatsache, daß die Implementierung in Prolog erfolgte werden detailliert beschriebene Programmteile und Aspekte der Datenstrukturen in Prolog-Notation angegeben. Das Analysesystem verfügt über einen Protokollierungsmodus, der alle internen Abläufe ausgibt. Dies ermöglicht stets eine genaue Analyse dessen, was berechnet wird und gibt dem Benutzer die Möglichkeit, das System zu verstehen und bei Bedarf zu erweitern. Um das nötige Wissen bereitzustellen den Protokollierungsmodus zu nutzen werden stets die vom System gespeicherten Informationen detailliert beschrieben. Dies gibt die Möglichkeit die einzelnen Arbeitsschritte an der Eingabeaufforderung des Prologsystems nachzuvollziehen. Als Compiler dienten bei der Implementierung der Demandanalyse der SWI-Compiler für MS-DOS und der Sicstus-Compiler für Unix.

6.1 Die Eingabedatei

Wie bereits erwähnt dienen die vom Curry-Interpreter bereitgestellten Dateien der definierenden Bäume als Eingabe für die Analyse. Die Dateien enthalten zusätzlich zu den definierenden Bäumen auch Informationen über verwendete Konstruktoren, sowie eine Liste der benutzten Funktionen und ihre Stelligkeiten. Die Datei muß dabei im folgenden Format vorliegen.

constructors

<Liste der benutzten Konstruktoren>

functions

<Liste der definierten Funktionen>

deftrees

<Liste der definierenden Bäume>

Die einzelnen Teile der Datei sollen im folgenden genauer beschrieben werden. Zunächst der Abschnitt mit der Definition der Konstruktoren. Nach dem Schlüsselwort *constructors* folgen in jeder Zeile ein Konstruktortyp gefolgt von der Liste der Konstruktoren dieses Typs und deren Stelligkeiten. So gibt zum Beispiel die Zeile `bool {true 0 false 0 }` an, daß zwei nullstellige Konstruktoren des Typs `bool` vorliegen, `true` und `false`. Da der Typ der Konstruktoren für diese abstrakte Interpretation nicht von Interesse ist, werden lediglich die Namen und die Stelligkeiten der Konstruktoren gespeichert. Damit diese Informationen sowohl beim Einlesen der definierenden Bäume als auch später während der Analyse zur Verfügung stehen, werden sie mittels *assert* als *constructor(Name,Stelligkeit))* in Prologs Database aufgenommen. Im obigen Beispiel bedeutet dies für die Konstruktoren `true` und `false`, daß sie als *constructor(true,0)* und *constructor(false,0)* abgespeichert werden.

Nach dem Schlüsselwort *functions* folgen Angaben darüber, welche definierten Funktionen im Programmodul benutzt werden. Hier enthält jede Zeile der Datei den Name und die Stelligkeit einer Funktion. So gibt die Zeile `append 2` an, daß eine zweistellige Funktion *append* definiert wurde. Die Angaben über die definierten Funktionen werden ähnlich zu denen über die Konstruktoren als *function(Name,Stelligkeit)* in die Database aufgenommen. Im Fall der append-Funktion würde dieser Eintrag also *function(append,2)* lauten.

Nach der Angabe der Funktionen folgen nun nach dem Schlüsselwort *deftrees* die definierenden Bäume aller definierten Funktionen. Dabei steht wiederum in jeder Zeile der Datei die Information für eine definierte Funktion. Dem Namen der Funktion folgt der definierende Baum in folgender Notation:

branch(Name[(Argumentliste)],{Pos},Modus,{Teilbäume}) oder

rule(Name[(Argumentliste)],RechteSeite)

Dabei steht $Name$ für den Namen der definierten Funktion. Ist die definierte Funktion nullstellig, so entfällt die nachfolgende Liste der Argumente, ansonsten enthält sie die Terme an den einzelnen Argumentpositionen. *Pos* bezeichnet die Position im vorstehenden Term, an der der Narrowingschritt angewendet werden soll. $Modus$ gibt an, ob es sich um einen $rigid$ oder $flex$ Knoten handelt. Nach dem Modus folgt, eingeschlossen in Mengenklammern, die Liste der Teilbäume des branch-Knotens, die wieder der gleichen Notation unterliegen. Die *RechteSeite* einer *rule* gibt den Term der rechten Regelseite der bezeichneten Regel an. Als Verdeutlichung diene das folgende Beispiel, welches den Dateieintrag über die übliche *append*-Funktion wiedergibt

append branch(append(A,B),{1},rigid,

{ rule(append([],C),C),

rule(append(:(D,E),F),:(D,append(E,F))) })

Die eingelesenen definierenden Bäume werden in eine günstigere interne Darstellung umgewandelt und als *getTree(Name,Stelligkeit, DefTree)* ebenfalls in die Database aufgenommen. *Name* und *Stelligkeit* geben an, welche definierte Funktion behandelt wird und *DefTree* enthält den definierenden Baum in der internen Darstellung.

Damit ist die Beschaffung der Eingabedaten abgeschlossen und es folgt die Transformation der definierenden Bäume mittels der in Abschnitt 4.4 ab Seite 26 beschriebenen Transformation Π in ein äquivalentes Termersetzungssystem.

6.2 Die Transformation Π

Da die Beschreibung der Analyse auf Basis der definierenden Bäume zu umfangreich wäre und sich damit auch die Implementierung erheblich schwerer gestalten würde, werden die definierenden Bäume mittels der Transformation Π in Termersetzungssysteme umgewandelt. Dazu werden nacheinander alle definierenden Bäume in Listen von Regeln transformiert. Die dabei auftretenden zusätzlichen Funktionen f_i werden durch anhängen des Indexes i an den Funktionsnamen f benannt und genau wie die definierten Funktionen als *function(Name,Stelligkeit)* in die Database aufgenommen. Die Liste aller Termersetzungsregeln einer Funktion wird mittels *newRules(Name, Stelligkeit, RegelListe)* anstelle der Einträge für die definierenden Bäume in die Database aufgenommen.

Nach dieser Transformation ist die Aufbereitung der Eingaben für die Analyse abgeschlossen. Die Database hält nun über die Prädikate *constructor*, *funktion* und *newRules* alle Informationen bereit, die zur Berechnung der Demandanalyse benötigt werden.

6.3 Fixpunktberechnung

Den zentralen Aspekt der Implementierung stellt die Fixpunktberechnung dar, daher soll nun detailliert die Implementierung der Berechnung des Funktionals Φ erläutert werden. Dazu wird zunächt der prinzipielle Ablauf der Analyse anhand eines Beispiels erläutert. Anschließend sollen die Probleme und Schwierigkeiten, die sich dabei ergeben, aufgezeigt und ihre Lösung in der Implementierung vorgestellt werden. Prinzipiell baut die Implementierung auf dem Rahmen von [Bru91] auf und nutzt teilweise Lösungsansätze von Analysesystemen wie PLAI [KM92] oder GAIA [CH94], die für Prolog entwickelt wurden.

6.3.1 Der Ablauf der Analyse

Wie schon früher angedeutet wird bei der Analyse die Semantik eines Terms aus der Semantik seiner Teilterme berechnet. Die Berechnungsanfrage lautet dabei: 'Welchen Grad an Auswertung müssen die Teilterme s, t, u des Funktionsaufrufs $f(s, t, u)$ erfahren um eine Auswertung von f zu erzielen, die dem Pattern π genügt'. Dazu werden zunächst die Argumente des Funktionsaufrufs durch unterschiedliche Variablen ersetzt ($f(x_1, x_2, x_3)$) und eine Analyse dieses vereinfachten Terms durchgeführt. Das Ergebnis dieser Analyse gibt an, zu welchem Grad π_i die einzelnen Argumente x_i von f ausgewertet werden müssen, um für f den Auswertungsgrad π zu erzielen. Nun muß dieses Ergebnis lediglich noch auf die tatsächlichen Terme s, t und u angewendet werden. Daraus erhält man drei abstrakte Substitutionen, welche für je einen Teilterm die nötigen Ersetzungen angeben. Diese drei Substitutionen enthalten nun möglicherweise unterschiedliche Informationen über Ersetzungen ein und derselben Variablen. Kommt z.B. die Variable y sowohl im Teilterm s als auch im Teilterm t vor, so kann es sein, daß y im Analyseergebnis von s an eine endliche Liste gebunden wird, während das Ergebnis bei Teilterm t eine potentiell unendliche Liste angibt. Diese unterschiedlichen Ergebnisse werden nun noch mit der Operation $\sqcup_A$ zu einer Substitution zusammengefaßt, welche alle Teilergebnisse subsumiert, d.h. im obigen Beispiel würde für y eine Auswertung hin zu einer endlichen Liste berechnet, da dies beiden Teilergebnissen gerecht wird.

Anhand eines konkreten Beispiels soll dieser Ablauf genauer beschrieben werden. Nehmen wir dazu an, daß für die schon genannte Beispielfunktion $f/3$ folgende drei Regeln zur Verfügung stehen (s.Abbildung 6.1 auf Seite 41):

$$f(t_{11}, t_{12}, t_{13}) \rightarrow r_1$$

$$f(t_{21}, t_{22}, t_{23}) \rightarrow r_2$$

$$f(t_{31}, t_{32}, t_{33}) \rightarrow r_3$$

Nun soll folgende Analyse durchgeführt werden: $[\![f(s, t, u)]\!]\ \pi$, daß heißt: 'Wie müssen s, t und u ausgewertet werden, damit das Ergebnis von $f(s, t, u)$ dem Pattern π genügt ?'. Dazu werden, wie schon angedeutet, die Parameter s, t und u durch drei frische, unterschiedliche Variablen x_1, x_2 und x_3 ersetzt und eine Analyse dieses neuen Terms bzgl. des gleichen Demandpattern π durchgeführt. Diese Analyse liefert dann drei Demandpattern π_1, π_2 und π_3, welche jeweils den Auswertungsgrad der entsprechenden Argumentposition wiederspiegeln. Zur Analyse von $f(x_1, x_2, x_3)$ werden nun aus dem, durch die Transformation Π berechneten, Termersetzungssystem alle Regeln der Funktion f herausgesucht. Da das Ergebnis der Funktion f dem Pattern π genügen muß führt man nun eine Analyse der rechten Regelseiten aller Regeln zu f bzgl. des Demandpatterns π durch. Man erhält also in unserem Fall drei neue Analysen $[\![r_1]\!]\ \pi, [\![r_2]\!]\ \pi$ und $[\![r_3]\!]\ \pi$. Diese Aufrufe werden analog zum Aufruf von $[\![f(s, t, u)]\!]\ \pi$ berechnet. Dieser Ablauf wird solange iteriert, bis die rechten Regelseiten einfache Terme sind, deren Analyseergebnis sich direkt ablesen läßt. Einfache Terme sind dabei Variablen, Konstruktoren angewendet auf Variabeln, nullstellige Konstruktoren und atomare Terme.

Nehmen wir an, die Analyse der rechten Regelseiten liefere die Antwortsubstitutionen σ_1, σ_2 und σ_3. Diese Substitutionen müssen nun jeweils auf die Parameter

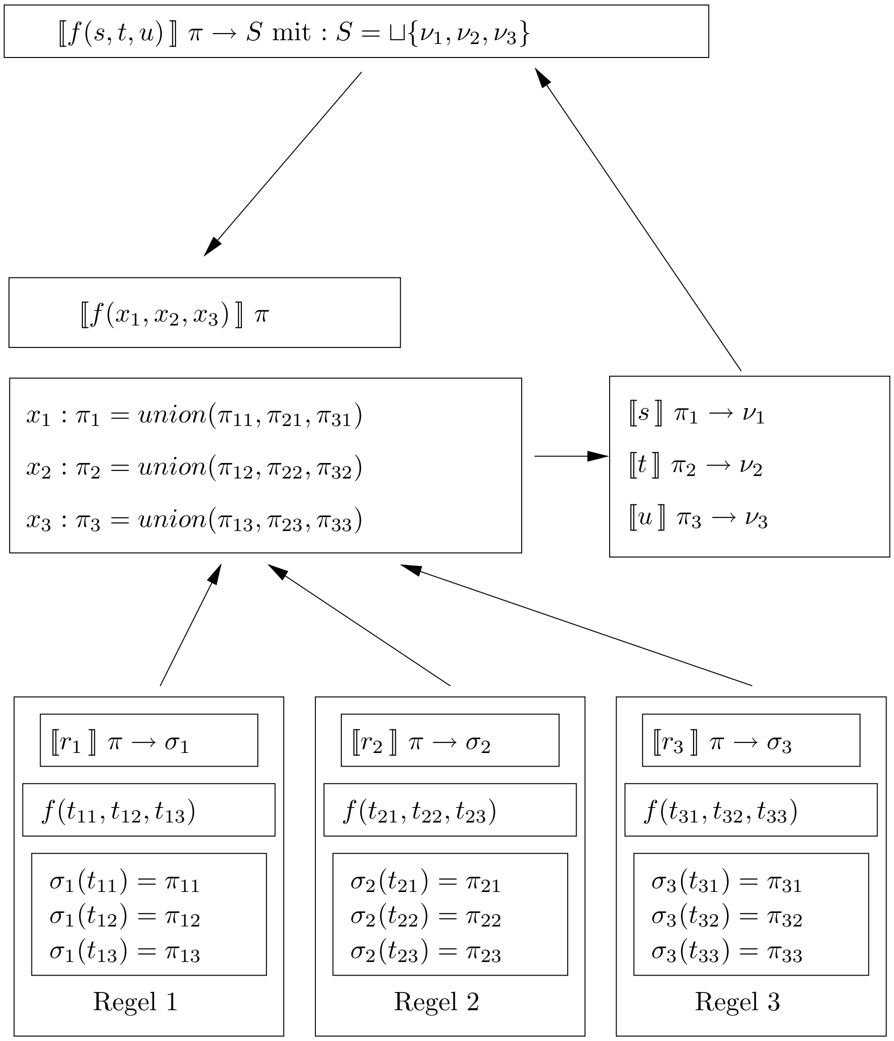


Abbildung 6.1: Die denotationelle Analyse von $f/3$

der linken Regelseiten angewendet werden. Diese Aufgabe übernimmt die Funktion *match* in der Analyse. Da für f drei Regeln definiert wurden ergeben sich daraus für jede der drei Argumentpositionen jeweils drei neue Demandpattern $(\pi_{1i}, \pi_{2i}, \pi_{3i}, i = 1, \ldots, 3)$. Da sich bei der abstrakten Analyse nicht entscheiden läßt, welche der drei Regeln im konkreten Fall zur Anwendung kommt, berechnet man den *lub* dieser drei Werte und erhält ein Pattern π_i, welches die minimale Anforderung an die Auswertung der Argumentposition i wiedergibt. Das bisher berechnete Ergebnis gibt also an, daß Argument x_1 insoweit ausgewertet werden muß, daß es dem Pattern π_1 gerecht wird. Analoges gilt für die beiden Argumente x_2 und x_3. Nun analysiert man die tatsächlichen Argumente von f bzgl. der ausgerechneten Pattern, also führt man $[\![s]\!]\ \pi_1, [\![t]\!]\ \pi_2$ und $[\![u]\!]\ \pi_3$ durch. Als Ergebnis erhält man drei abstrakte Anwortsubstitutionen ν_1, ν_2 und ν_3. Jede dieser Antwortsubstitutionen nimmt nun Ersetzungen auf den Variablen von $f(s, t, u)$ vor. Da diese Ersetzungen im allgemeinen nicht verträglich sind, berechnet man nun mittels $S_A = \sqcup_A\{\nu_1, \nu_2, \nu_3\}$ die endgültige Antwortsubstitution S_A, welche die verträgliche Approximation aller Eigenschaften von ν_1, ν_2 und ν_3 darstellt. Das Ergebnis der Analyse lautet damit: $[\![f(s, t, u)]\!]\ \pi = S_A$.

6.3.2 Die Probleme der Fixpunktberechnung

Der im letzten Abschnitt beschriebene Ablauf der Analyse gestaltet sich nur dann so einfach, wenn die behandelte Funktion nicht rekursiv ist. Im konkreten Fall terminiert eine rekursive Funktion durch stetigen Abbau der behandelten Datenstruktur und einmaliger Ausführung einer Terminierungsregel. Nehmen wir als Beispiel die bekannte Funktion *append*/2 zur Konkatenierung von Listen. Die Funktion *append* wird definiert durch folgende zwei Regeln:

```
append([ ],L) → L.

append([H|T],L) → [H|append(T,L)].
```

Bei der abstrakten Analyse dieser Funktion ergibt sich nun das Problem, daß, nicht wie im konkreten Fall, die Rekursion in der zweiten Regeln durch Abbarbeitung der ersten terminiert wird. Bei der Analyse des Aufrufs $[\![append(x_1, x_2)]\!]\ \pi$ trifft man bei der rechten Seite der zweiten Regel auf den Aufruf $[\![[H|append(T, L]]\!]\ \pi$. Dieser Aufruf wird nach obiger Erklärung so abgearbeitet, daß zunächst $[\![[x_1|x_2]]\!]\ \pi$ berechnet wird zu π_1 und π_2. Danach folgen für die ersetzten Teilterme die Aufrufe $[\![H]\!]\ \pi_1$ und $[\![append(T, L)]\!]\ \pi_2$. Der erste dieser beiden Aufrufe ist unproblematisch, da H Variable und damit ein einfacher Term ist. Der zweite Aufruf allerdings entspricht für den Fall, daß $\pi_2 = \pi$ (und nach einer Umbenennung der Variablen) genau dem Aufruf $[\![append(x_1, x_2)]\!]\ \pi$. Dieser Aufruf produziert zwangsweise wieder eine Anfrage $[\![append(T, L)]\!]\ \pi_2$ und so weiter. Die Analyse läuft in diesem Fall also in eine endlose Rekursion.

Zur Vermeidung dieser Tatsache speichert man vor jeder ersten Berechnung einer Funktion einen Initialwert $S_A^{\top}$. Trifft man auf eine Rekursion in einer rechten Regelseite, so nimmt man als Analyseergebnis diesen Initialwert anstatt eines rekursiven Aufrufs. Hat man das Ergebnis für die behandelte Funktion dann vollständig berechnet speichert man dieses als neuen Initialwert für eine weitere Iteration der Analyse.

Daraus ergibt sich zum einen das Problem der Speicherung und Verwaltung der aktuellen Zwischenergebnisse und zum anderen das Problem der Erkennung einer rekursiven Funktion. Zur Erkennung speichert man lediglich welche Funktion gerade berechnet wird und überprüft vor jedem neuen Aufruf der Analyse, ob es sich wieder um den gleichen Term handelt. Desweiteren ergibt sich das Problem, wann denn die Analyse abgeschlossen ist, das heißt wann keine nochmalige Iteration der Analyse mehr nötig, der Fixpunkt also erreicht ist. Durch die Rückgabe der Initialwerte werden bei rekursiven Funktionen in jedem Durchlauf der Analyse nur Approximationen von Φ berechnet. Die Frage die sich stellt ist, wie man feststellt, daß keine weitere Iteration mehr nötig ist, da sich das Ergebnis von Φ nicht mehr ändert wird.

6.3.3 Implementierung der Fixpunktberechnung

Während der Berechnung der Demandanalyse benötigt man Angaben darüber, welche Terme unter welchen Demandpattern zu welchen Ergebnissen geführt haben. Da jeder Term währen der Analyse mit potentiell jedem Demandpattern vorkommen kann ergibt sich zwingend die Speicherung einer Tabelle von Tripeln der Form (Term,Pattern,Ergebnis). Diese Tabelle soll im weiteren als LUT (Look-Up-Table) bezeichnet werden. Da die Tabelleneinträge zu unterschiedlichen Zeitpunkten der Analyse verfügbar sein müssen bietet sich eine globale Speicherung über die Database an. Nun lassen sich zu jedem Zeitpunkt alle Ergebnisse der bisherigen Analyse aus der Tabelle ablesen. Als nächstes stellt sich die Frage, wie festgestellt werden soll, ob ein Tabelleneintrag einem vorläufigen Ergebnis oder einem Fixpunkt entspricht. Daher erweitert man die Einträge der LUT um ein weiteres Element, welches den Status der bisherigen Berechnung wiedergibt. Für diese Statusangabe unterscheidet man zwischen drei Fällen:

> 1.Fall: Der Tabelleneintrag entspricht einem vorübergehenden Zwischenergebnis, einer Approximation der Analyse.
>
> 2.Fall: Der Tabelleneintrag entspricht einem endgültigen Berechnungsergebnis, dem Fixpunkt der Analyse.
>
> 3.Fall: Der Tabelleneintrag entspricht einem vorläufigen Wert und zur Zeit wird eine neue Approximation berechnet.

Entsprechend diesen drei Fällen erhält jeder Eintrag der LUT als viertes Element einen Statuseintrag aus {approx, complete, fixpoint}. Durch das Hinzunehmen des Status *fixpoint* erhält man die oben angesprochene Speicherung der Anfrage die gerade in Berechnung ist. Soll also eine Anfrage $[\![t]\!]\ \pi$ berechnet werden und die LUT enthält einen Eintrag $(t, \pi, S_A, fixpoint)$, so wird anstatt einer neuen Berechnung das bisherige Ergebnis S_A zurückgegeben.

Die noch offene Frage der Erkennung eines Fixpunktes entpricht der Frage nach den Statusübergängen der LUT Einträge, da bei Erreichen des Fixpunkts der Statuseintrag auf *complete* gesetzt werden muß. Dazu definieren wir eine Transformation *upd* auf den Einträgen der LUT, welche aus einem gegebenen Eintrag und einem neuen Berechnungsergebnis den neuen Status des Eintrags ermittelt. Da der Status *fixpoint* lediglich während der Dauer einer Berechnung vorliegen kann und der Status *complete* keine erneute Berechnung mehr zuläst benötigt man lediglich Angaben darüber welche Transformation der Eintrag *approx* erfährt. Eine erste (naive) Definition dieser Transformation könnte folgendermaßen aussehen:

$upd : LUT \times \mathcal{AP}_A \rightarrow LUT$ mit
$upd((t, \pi, S_{alt}, approx), S_{neu}) = (t, \pi, S_{neu}, approx)$, falls $S_{neu} \neq S_{alt}$
$upd((t, \pi, S_{alt}, approx), S_{neu}) = (t, \pi, S_{neu}, complete)$, falls $S_{neu} = S_{alt}$

Hier wird also davon ausgegangen, daß der Fixpunkt einer Berechnung erreicht ist, sobald eine Iteration das gleiche Ergebnis liefert wie die letzte. Dies entspricht der mathematischen Vorstellung der Transformation Φ, wonach:

$$\Phi_A^n(t, \pi_A) = \Phi_A^{n+1}(t, \pi_A) \Longrightarrow \Phi_A^m(t, \pi_A) = \Phi_A^{m+1}(t, \pi_A) \; für \; m > n \; gilt.$$

Bei der Implementierung der abstrakten Transformation Φ ergibt sich aber, aufgrund der Benutzung approximierter Zwischenergebnisse, bei rekursiven Funktionen mit mehreren Regeln folgendes Problem. Neue Ergebnisse in der Berechnung von Untertermen können eventuell erst nach mehreren Iterationen eine Auswirkung auf das Ergebnis des Hauptterms ausüben, wenn diese durch Ergebniss in anderen Regeln überlagert werden. So liefert die erste Regel der append-Funktion zum Beispiel schon in der ersten Iteration das Ergebnis, daß der erste Parameter eine leere Liste ist. Die zweite Regel liefert in den ersten beiden Iterationen allerdings undefinierte Ergebnisse für beide Parameter. Damit entsprechen die Ergebnisse der ersten beiden Iterationen dem Ergebnis der ersten Regel, während das tatsächliche Analyseergebnis aus der zweiten Regel resultiert. Um also diese Möglichkeit auszuschließen reicht es nicht nur das vorläufige Analyseergebnis mit dem der vorherigen Iteration zu vergleichen, man muß zusätzlich darauf achten, wie das neue Zwischenergebnis entstanden ist. Erst, wenn die Analyse eines Unterterms für den vollständigen, ihm entsprechenden Teilbaum keinerlei neue Ergebnisse geliefert hat, kann man davon ausgehen, daß weitere Iteration ebenfalls keine Änderung mehr bewirken und der Fixpunkt somit erreicht ist.

Um diese zusätzliche Kontrolle zu gewährleisten erhält jeder Eintrag der LUT eine Information darüber, ob im Verlauf seiner Berechnung neue Erkenntnisse erzielt wurden. Diese Information soll als bool'scher Wert *noChanges* eingetragen werden. Dabei bedeutet noChanges=true, daß bisher keine neuen Ergebnisse auftraten, während noChanges=false signalisiert, daß von dem Erreichen des Fixpunktes frühestens nach der nächsten Iteration ausgegangen werden kann.

Aus Gründen der Effizienz wurde der *noChange*-Eintrag in der Implementierung nicht in die LUT geschrieben, sondern gesondert in die Database aufgenommen, da sich auf diese Weise eine Änderung vieler Einträge durch *retractall* realisieren läst. In der folgenden Definition der verbesserten Transformation *upd* wird dieser Eintrag der Einfachheit halber in der LUT gespeichert. Da während der Transformation Änderungen auf der gesamten LUT möglich sind erhält *upd* als ersten Parameter alle zur Zeit gespeicherten Einträge.

$$
\begin{array}{l}
\mathit{upd} : \wp(\mathit{LUT}) \times \mathit{LUT} \times \mathcal{AP}_A \rightarrow \wp(\mathit{LUT}) \times \mathit{LUT} \text{ mit} \\
\mathit{upd}(\mathcal{L}, (t, \pi, S_{alt}, \mathit{approx}, nC), S_{neu})) = (\mathcal{L}', (t, \pi, S_{neu}, \mathit{approx}, \mathit{false})), \\
\qquad \text{mit } (t_i, \pi_{ij}, S, \mathit{Status}, \mathit{false}) \in \mathcal{L}', \text{ für alle } (t_i, \pi_{ij}, S, \mathit{Status}, nC_i) \in \mathcal{L}, \\
\ \text{falls } S_{neu} \neq S_{alt} \\
\mathit{upd}(\mathcal{L}, (t, \pi, S_{alt}, \mathit{approx}, nC), S_{neu})) = (\mathcal{L}, (t, \pi, S_{neu}, \mathit{approx}, nC)), \\
\ \text{falls } nC = \mathit{false} \text{ und } S_{neu} = S_{alt} \\
\mathit{upd}(\mathcal{L}, (t, \pi, S_{alt}, \mathit{approx}, nC), S_{neu})) = (\mathcal{L}, (t, \pi, S_{neu}, \mathit{complete}, nC)), \\
\ \text{falls } nC = \mathit{true} \text{ und } S_{neu} = S_{alt}
\end{array}
$$

Zusätzlich muß gewährleistet sein, daß vor jedem Aufruf der Demandanalyse das *noChanges*-flag des behandelten Eintrags auf *true* gesetzt wird. Der Eintrag *complete* wird also nur noch dann erreicht, wenn das Ergebnis das gleiche wie in der letzten Iteration ist und *noChanges* auch nach der Berechnung noch auf *true* steht.

Diese Aufgabe der Aktualisierung der Look-up-Table übernimmt die Funktion *updLut*. Sie trägt die neu berechneten Ergebnisse in die Tabelle ein, setzt den durch *upd* angegebenen neuen Stauseintrag und verwaltet die *noChanges* Informationen. Die Kontrolle darüber, ob für eine gegebene Anfrage weitere Iterationen von Φ notwendig sind, übernimmt die Funktion *startIteration*. Sie bestimmt also, ob die angeforderte abstrakte Substitution aus der Look-up-Table entnommen wird oder durch Berechnung von Φ (bzw. in der ersten Iteration von Φ^0) gewonnen werden muß. Das Zusammenspiel dieser Funktionen ist im Anhang B genauer beschrieben. Dort findet sich auch eine Abbildung (Seite 65), die die Abhängigkeiten verdeutlichen soll.

6.4 Die Benutzershell

Dieser Abschnitt soll eine kurze Einführung in die Benutzung der Shell des Analysesystems dienen. Dabei sollen die unterstützten Befehle kurz beschrieben und ihre Funktionsweise erläutert werden. Da das Analysesystem außschließlich in Prolog implementiert wurde müssen alle Eingaben stets mit einem Punkt abgeschlossen werden. Eine Liste der unterstützten Befehle erhält man durch Aufruf des Befehls `help`. Zunächst sollen die Befehle vorgestellt werden, welche die Daten für die durchzuführende Analyse bereitstellen.

Vor dem Aufruf einer Analyse mittels `go` müssen die behandelte Datei mit den definierenden Bäumen, der zu analysierende Term, der behandelte Domain und das Demand Pattern eingegeben werden. Die Angabe der einzulesenden Datei geschieht mittels des Befehls `getFile`, der lediglich den Namen der Definitionsdatei abfragt. Wird der Dateiname ohne eine Ergänzung angegeben, so wird die Endung '.def' angefügt. Zur Bearbeitung der Datei `benchmrk.def`, welche die Informationen über die in Anhang E vorgestellten Benchmarkfunktionen enthält, genügt also die Eingabe `benchmrk`. Danach gibt man mittels des Befehls `getTerm` den zu analysierenden Term an. Man kann dabei nur Terme eingeben, die der Form $f/n(x_1, \dots, x_n)$ oder $f/0$ genügen. Dazu wird zunächst der Name f der behandelten Funktion, dann deren Stelligkeit abgefragt. Diese Eingaben werden mit den Informationen aus der Definitionsdatei verglichen und für den Fall das die Funktion bekannt ist werden danach noch die Indizes der Variablen x_i abgefragt. Zur Analyse des Terms $f(x_1, x_2, x_1)$ sind also nacheinander die Eingaben `f.;3.;1.;2.;1.;` nötig. Als nächstes muß der benutzte Domain bestimmt werden. Nach Eingabe von `getDomain` wird die Datei, welche die Definition des Domains enthält verlangt und zum System hinzu compiliert. Eine Beschreibung der Domainelemente und der durch sie abstrahierten Termmengen liefert der Befehl `showDomain`. Nun Bedarf es lediglich noch der Angabe des Demand Patterns mittels `getPattern`. Hier werden zunächst alle Elemente des Domains aufgezeigt und dann das gewünschte Demand Pattern abgefragt.

Damit sind alle Angaben gemacht und ein Aufruf von `go` startet die Analyse. Am Ende der Analyse wird das Ergebnis, also die berechnete abstrakte Substitution, ausgegeben. Zusätzlich gibt eine Liste der während der Analyse behandelten Parameter Aufschluß darüber, welchen Grad an Auswertung die behandelten Unterterme erfahren. Ist man an genaueren Informationen über den Ablauf der Analyse interessiert, so kann man mittels des Befehls `visible` eine detaillierte Kommentierung der einzelnen Schritte ausgeben lassen. Der Befehl `blind` schaltet den Kommentierungsmodus wieder aus und kehrt in den voreingestelten Modus zurück. Zusätzlich werden alle Analyseergebnisse in einer Datei mit dem Namen der Definitionsdatei und der Endung '.abs' gespeichert. Diese Datei ließe sich also von einem Compiler einlesen, um die gewonnene Information zu nutzen.

Mit `stop` oder `exit` verläßt man die Shell wieder. Dabei besteht der unterschied zwischen den beiden Befehlen in der Behandlung der, in der Database zwischengespeicherten Informationen. Bei Aufruf von `exit` werden alle Informationen gelöscht und stehen nach verlassen der Shell nicht mehr zur Verfügung. Verläßt man das System mit `stop`, so bleiben diese Informationen erhalten und lassen sich von der Prolog-Shell aus abfragen.

Damit sollte die Implementierung ausreichend beschrieben sein. Im Anschluß werden im Kapitel 7 die mit der Analyse erzielten Ergebnisse vorgestellt und interpretiert. Desweiteren folgt noch eine Erweiterung der im Kapitel 5 vorgestellten Demandanalyse. Diese Erweiterung um sogenannte 'disjunktive Bereiche' wird im abschließenden Kapitel 8 behandelt.

Kapitel 7

Ergebnisse

Dieses Kapitel soll zunächst die mit der beschriebenen Analyse erzielten Ergebnisse anhand einiger Beispiele darlegen und bewerten. Danach werden Maßnahmen diskutiert, welche die erzielten Ergebnisse unter bestimmten Vorraussetzungen weiter verbessern können. Diese Maßnahmen umfassen sowohl programmiertechnische Änderungen also auch Manipulationen am benutzten Domain. Am Ende werden die Ergebnisse mit denen von [SR96] verglichen und Unterschiede bewertet.

7.1 Benchmarkanalyse

Zunächst sollen an einigen Beispielen durch die Analyse erzielte Ergebnisse vorgestellt werden. Dabei wurden einige Benchmark-Funktionen in Curry implementiert, die sich auch in [SR96] wiederfinden. Als erstes wurde das 8-Damenproblem, bei dem man versucht 8 Damen eines Schachspiels derart auf dem Schachbrett zu positionieren, daß keine Damen von einer anderen bedroht wird, umgesetzt. Desweiteren wurden eine Primzahlberechnung nach dem Verfahren von Eratosthenes, ein Quicksort Algorithmus, sowie eine Matrixmultiplikation als Beispiel herangezogen. Im weiteren Verlauf sollen die oben genannten Funktionen wie in den Listings im Anhang E angegeben bezeichnet werden:

damen8 : Lösungen des 8-Damen-Problems ausgeben

primzahlen(N) : Liste der ersten N Primzahlen berechnen

qs(L) : sortieren der Liste L mittels Quicksort

matmult(A,B) : berechnen des Matrixproduktes von A und B

Zur Analyse wurde zunächst der in Anhang D auf Seite 76 dargestellte Domain 5 herangezogen und alle Funktionen des Benchmark Files auf eine Auswertung hin zu einer vollständig ausgewerteten Liste untersucht. Im genannten Domain entspricht dies dem Demand Pattern *liste*. Die weiteren Domainelemente sind neben *top* und *bot* zunächst *e*, die Menge aller vollständig instanziierten Terme und *hn* die Menge aller Terme in Kopfnormalform. Die Menge *list* aller Listen, also aller Terme deren Topkonstruktor ein *cons* ist, wurde weiter differenziert danach, ob endliche oder potentiell unendliche Listen vorliegen. Dabei bezeichnet *listhn* die Menge aller endlichen Listen deren erstes Element in Kopfnormalform vorliegt und *listf* die Menge aller endlichen Listen. Desweiteren ist *heade* die Menge aller potentiell unendlichen Listen, deren Kopfelement vollständig instanziiert ist und *headhn* die Menge aller potentiell unendlichen Listen, deren Kopfelement in Kopfnormalform vorliegt. Die Ordnung auf diesem Domain wird durch die in Anhang D verdeutlichte Struktur definiert. Dort sind auch die genauen Definitionen der hier intuitiv beschriebenen Domainelemente aufgeführt. Bei der Analyse ergaben sich die folgenden Gesamtergebnisse:

$$[\![damen8]\!] : liste \longrightarrow true$$

$$[\![primzahlen(x_1)]\!] : liste \longrightarrow [(x_1 : e)]$$

$$[\![qs(x_1)]\!] : liste \longrightarrow [(x_1 : heade)]$$

$$[\![matmult(x_1, x_2)]\!] : liste \longrightarrow [(x_1 : listf), (x_2 : top)]$$

Dabei steht die abstrakte Substitution *true* für die identische Substitution, also für eine erfolgreiche Auswertung der Anfrage. Bei der Analyse dieser 4 Anfragen werden insgesamt 67 Parameter übergeben. Dabei verteilen sich die Bilder der berechneten abstrakten Substitutionen wie folgt auf die Elemente des Domains:

22%	strikte Auswertung (e,liste)
3%	endliche Listen (listhn,listf)
24%	potentiell unendliche Listen (list,heade,headhn)
51%	keine genaueren Angaben möglich (top)

Dabei waren zur Berechnung des Fixpunktes durchschnittlich 4 Iterationen notwendig.

7.2 Probleme und Verbesserungen

Aufgrund der großen Anzahl von Parametern, über die durch die Analyse keine Erkenntnisse gewonnen wurden stellt sich die Frage, wodurch dies verursacht wird, und welche Möglichkeiten der Verbesserung sich bieten.

7.2.1 Vermeidung von if-then-else

Als problematisch erweisen sich bei der Analyse die if-then-else Konstrukte (ite), welche außer bei der Funktion *matmult* stets benutzt werden. Der Grund liegt darin, daß die Funktion *ite* nicht strikt ist, wodurch die Analyse folgerichtig auch keine Striktheitsaussagen über die Parameter von *ite* berechnet. In der Analyse schlägt sich dies bei der Vereinigung der Ergebnisse einzelner Regeln zu einem Gesamtergebnis mittels $union_A$ nieder. Die Funktion if-then-else (ite) ist z.B. wie folgt definiert:

Regel 1: $ite(x_1, x_2, x_3)|\{x_1 == true\} = x_2.$
Regel 2: $ite(x_1, x_2, x_3)|\{x_1 == false\} = x_3.$

Die Analyse von $[\![ite(x_1, x_2, x_3)]\!]\ e$, d.h. der Auswertung der Frage wann *ite* einen vollständig ausgewerteten Ausdruck liefert, ergibt dabei folgende Konstellation:

$$\begin{array}{rl} \text{Regel 1:} & \rightarrow \{(x_1, e), (x_2, e), (x_3, top)\} \\ \text{Regel 2:} & \rightarrow \{(x_1, e), (x_2, top), (x_3, e)\} \\ \hline union_A: & \rightarrow \{(x_1, e), (x_2, top), (x_3, top)\} \end{array}$$

Man sieht, daß durch die Vereinigung der Teilergebnisse mittels $union_A$ alle Informationen über die beiden hinteren Parameter verloren gehen. Dies ergibt sich durch die Berechnung des least-upper-bound (lub) der Werte e und *top* zu *top*. Durch die Berechnung des Demand Patterns *top* erzielt die Analyse dann auch keine Erkenntnisse mehr über eventuelle Unterterme des *ite*-Aufrufs, da die Anfrage $[\![t]\!]\ top$ stets die Antwortsubstitution $[(x_1 : top), \ldots, (x_n : top)]$ mit $\{x_1, \ldots, x_n\} = \mathcal{V}ar(t)$ liefert.

Wie kann man nun dieses Problem umgehen oder zumindest einschränken. Die erste Möglichkeit besteht darin, alle *ite*-Aufrufe im Programm zu vermeiden, indem man die jeweilige Funktion so definiert, daß zwei Regeln mit je einem if-then-Aufruf (it) entstehen. Dies bedingt folgende Änderungen.

Anstelle von:

$$ite(x_1, x_2, x_3)|\{x_1 == true\} = x_2.$$

$$ite(x_1, x_2, x_3)|\{x_1 == false\} = x_3.$$

definert man:

$$it(true, x) = x.$$

und ändert Funktionsdefinitionen mit ite-Aufrufen wie folgt ab.

$$\begin{array}{ll} \text{statt:} & f(\ldots) = \ldots ite(t_1, t_2, t_3) \ldots \\ \text{setze:} & f(\ldots) = \ldots it(t_1, t_2) \ldots \\ & f(\ldots) = \ldots it(t_1 == false, t_3) \ldots \end{array}$$

Dabei ergibt sich aber das Problem, daß die entstehenden definierenden Bäume dieser Funktionen allesamt or-Knoten enthalten. Dies geht deutlich zu Lasten der Effizienz in der Ausführung dieser Funktionen [Sad97]. Dennoch soll hier untersucht werden, welche Vorteile eine solche Änderung für die Güte der Analyseergebnisse birgt. Dazu wurden die gleichen Analysen, wie im Abschnitt 7.1 auf den gänderten Funktionen ausgeführt und es ergab sich folgendes Endergebnis:

$$[\![damen8]\!] : liste \longrightarrow true$$

$$[\![primzahlen(x_1)]\!] : liste \longrightarrow [(x_1 : e)]$$

$$[\![qs(x_1)]\!] : liste \longrightarrow [(x_1 : liste)]$$

$$[\![matmult(x_1, x_2)]\!] : liste \longrightarrow [(x_1 : listf), (x_2 : top)]$$

Die einzige Änderung scheint in der besseren Analyse des Parameters von qs zu liegen, der nun nicht mehr als potentiell unendliche Liste sondern als endliche vollständig ausgewertete Liste berechnet wird. Schaut man sich allerdings analog zur ersten Analyse die Verteilung der Parameter an dann stellt man fest, daß zur Analyse nur noch 61 Parameter berechnet werden mußten, die sich wie folgt verteilen:

43%	strikte Auswertung (e,liste)
18%	endliche Listen (listhn,listf)
10%	potentiell unendliche Listen (list,heade,headhn)
29%	keine genaueren Angaben möglich

Hierbei kommt es teilweise vor, daß im Vergleich zum ersten Benchmark eine zusätzliche Iteration notwendig wird. Dies ergibt sich daraus, daß nun auch die Unterterme der if-then-Aufrufe analysiert werden, während dies bei Verwendung des if-then-else durch den angesprochenen Informationsverlust überflüssig ist. Desweiteren zeigt das Ergebnis eine Verdopplung der ausfindig gemachten Striktheiten, eine Reduzierung der Parameter ohne Anaylseergebnis um ca.20% und ein aussagekräftigeres Ergebnis für Parameter mit Listenkonstruktoren. Damit zeigt sich, daß durch Verwendung von nicht-strikten Funktionen (wie if-then-else) eine deutliche Verschlechterung der gesamten Analyse einhergeht. Diese Probleme gestanden wohl auch die Autoren von [SR96] bei Veröffentlichung ihrer Analyse ein.

7.2.2 Vermeidung des Informationsverlustes

Die zweite Möglichkeit, die sich bietet, besteht darin den Informationsverlust durch die Vereinigung der Teilergebnisse zu vermeiden. Teilergebnisse einzelner Regeln werden nicht mehr zu einer abstrakten Substitution vereinigt sondern über sogenannte disjunktive Bereiche zusammengefaßt. Dabei bildet man Disjunktionen aus den Ergebnissen der einzelnen Regeln. Am Beispiel der *ite*-Funktion sei dies erläutert. Die Analyse von $[\![ite(x_1, x_2, x_3)]\!]\ e$ liefert:

$$\begin{array}{rcl}
\text{Regel 1:} & \rightarrow & \{(x_1, e), (x_2, e), (x_3, top)\} \\
\text{Regel 2:} & \rightarrow & \{(x_1, e), (x_2, top), (x_3, e)\} \\
\hline
union' : & \rightarrow & \{(x_1, e), (x_2, e), (x_3, top)\} \vee \\
 & & \{(x_1, e), (x_2, top), (x_3, e)\}
\end{array}$$

Hier läßt sich also weiterhin ablesen, daß einer der hinteren Parameter strikt sein muß. Bei anderen Funktionen kann sich dabei eine Fallunterscheidung über den ersten Parameter ergeben, die sich hier aufgrund der Tatsache, daß das Ergebnis für den ersten Parameter beide Male gleich ist, nicht einstellt. Eine genaue Betrachtung der notwendigen Änderungen in der Analyse findet sich in Kapitel 8. Dort sollen dann auch Analyseergebnisse, die unter Einsatz von disjunktiven Bereichen erzielt wurden mit denen der bisherigen Analyseform verglichen und bewertet werden.

7.2.3 Verbesserungen des Domains

Nun soll untersucht werden, in wie weit nicht zufriedenstellende Analyseergebnisse durch Änderungen am benutzten Domain kompensiert werden können. Dazu betrachte man die auch im Benchmark eingesetzte Programmiertechnik, daß Funktionen so implementiert werden, daß eine Regel Transformationen auf allgemeinen Daten durchführt, während eine weitere für die Terminierung der Funktion durch z.B. Behandlung der leeren Liste sorgt. Die Terminierungsregel ist dabei in der abstrakten Analyse weniger von Interesse als die eigentlich 'ergebnisproduzierenden' weiteren Regeln. Es ergibt sich also wiederum das Problem der Vereinigung von Ergebnissen einzelner Regeln zu einem Gesamtergebnis, wie schon oben angedeutet.

Hier soll speziell auf Funktionen obigen Shemas eingegangen werden, welche auf Listen arbeiten. Da Listen die dominierende Datenstruktur der deklarativen Programmierung darstellen, lassen sich durch diese Untersuchung sehr viele der genannten Fälle abdecken. Dies wird besonders deutlich bei Funktionsdefinitionen die analog zu denen von *matmult* und *rowmult* aufgebaut sind, also bei Funktionen, deren Terminierungsregel nicht benutzte, anonyme Variablen aufweisen und als Ergebnis die leere Liste liefern. Betrachten wir als Beispiel die Funktion *matmult*:

$$\begin{array}{ll} matmult([\,], Y) & = [\,]. \\ matmult([X|Xs],[Y|Ys]) & = [rowmult(X,Y)|matmult(Xs,Y)]. \end{array}$$

Hier wird also über das erste der beiden Argument rekursiert, währen das zweite Argument unverändert übergeben wird. Analysiert man nun unter Domain 5 oder Domain 6 [s. Anhang D] eine Anfrage wie $[\![matmult(X,Y)]\!]$ *liste*, d.h. man möchte als Ergebnis der Funktion *matmult* eine vollständig ausgewertete Liste erhalten, so ergibt sich folgendes Problem. Die Analyse der ersten Regel produziert die Anfrage $[\![[\,]]\!]$ *liste* und diese liefert sicherlich das Ergebnis *true*. Damit liefert die Analyse der ersten Regel insgesamt die abstrakte Substitution $[(X : nil), (Y : top)]$ (bzw. im Fall von Domain 5 $[(X : liste), (Y : top)]$. Dadurch wird für das Gesamtergebnis $[(X : listf), (Y : top)]$ der Analyse die Ersetzung $(Y : top)$ durch die erste Regel bestimmt, ungeachtet des Ergebnisses der zweiten, eigentlich interessanten Regel.

Der Grund dafür, daß dieses Ergebnis nicht zufriedenstellt liegt darin, daß man sich bei der abstrakten Analyse der Funktion *matmult* nicht dafür interessiert, wann diese eine leere Liste zurückliefert. Viel mehr will man wissen, wie die beiden Parameter von *matmult* beschaffen sein müssen, damit das Ergebnisse eine vollständig ausgewertete Liste ungleich der leeren Liste ist. Dies bedeutet, die Analyse der ersten Regel ist hier äusserst hinderlich.

Durch eine Änderung des Domains läßt sich aber erreichen, daß genau diese erste Regel nicht mehr in die Analyse einbezogen wird, wenn dies nicht erwünscht ist. Dazu definieren wir den Domain 7 [s. Anhang D]. Hier bezeichnet das neu eingefügte Element *listne* genau die Menge der nichtleeeren, vollständig ausgewerteten Listen. Nun hat man die Möglichkeit die oben angestrebte Analyse auch zu formulieren. Stellt man also die Anfrage $[\![matmult(X, Y)]\!]\ listne$, so produziert jetzt die erste Regel die Anfrage $[\![[\,]]\!]\ listne$. Diese liefert als Ergebnis die Substitution $false$, da $nil \notin \gamma(listne)$. Dadurch ergibt sich als Gesamtergebnis der ersten Regel die abstrakte Substitution $[(X : bot), (Y : bot)]$. Hiermit tritt der umgekehrte Effekt wie vorher ein. Das Gesamtergebnis der Analyse von *matmult* wird nun ausschließlich durch die zweite Regel bestimmt und man erhält als Ergebnis $[(X : listf), (Y : liste)]$. Dies bedeutet, daß nicht wie vorher keinerlei Information über den zweiten Parameter gewonnen werden, sondern man erkennt sogar die Striktheit dieses Paramters für den Fall, daß das Ergebnis vollständig instanziiert sein soll.

Es soll an dieser Stelle nicht verschwiegen werden, daß es auch Fälle gibt, in denen obige Änderung des Domains zu einem Nachteil wird. Dies tritt ein, wenn das Ergebnis der ersten Regel benötigt wird, um der Analyse der zweiten Regel einen Initialwert zu liefern. Es gibt also nicht *den* Domain, der alle Probleme optimal behandelt. Sattdessen muß man eventuell auf mehrere Ergebnisse zurückgreifen und im jeweiligen Fall entscheiden, welcher Domain für die Anfrage am besten geeignet ist.

Bei der Analyse grösserer Programme dürften sich aber die Vorteile des Domains 7 durchsetzen, wie die Analyseergebnisse des Benchmarks belegen, die insgesamt ein leicht verbessertes Resultat aufzeigen.

47%	strikte Auswertung (e,liste)
16%	endliche Listen (listhn,listf)
12%	potentiell unendliche Listen (list,heade,headhn)
25%	keine genaueren Angaben möglich

Hier stellt sich endgültig heraus, daß die Größe des benutzten Domains keine Einflüsse auf die benötigte Zahl von Iterationen hat. In wie weit sich aber der höhere Aufwand (81 statt 61 bearbeitete Parameter) im Vergleich zum einfacheren Domain lohnt müssen zukünftige Untersuchungen zeigen. Aufgrund der Tatsache, daß in [SR96] ein Domain mit 30 Elementen verwendet wurde kann man aber schließen, daß der zusätzliche Aufwand sich bei weiterer Optimierung auszahlt. Zählt man die tatsächlich strikten Argumente im Benchmark, so kommt man auf ca. 80% strikter Parameter. Damit erkennt die letzte vorgestellte Analyse schon immerhin ca. 60% aller Striktheiten.

Nach dieser Diskussion über die erzielten Ergebnisse mit der in Kapitel 5 definierten denotationellen Analyse sollen die wichtigsten Erkenntnisse kurz zusammengefaßt werden, bevor die Erweiterung um disjunktive Bereiche im Kapitel 8 vorgestellt wird.

7.3 Zusammenfassung der Ergebnisse

Man hat gesehen, daß die denotationelle Analyse für Funktionen, die nicht für die Analyse optimiert wurden, Aussagen über ca. 50% der bearbeiteten Parameter treffen konnte und ca. 22% als strikt identifiziert wurden. Im Vergleich zu den in [SR96] angegebenen Werten von 90-100% erscheint dies sehr ungenau. Bei der Suche nach den Gründen für diesen Unterschied trat das extreme Verbesserungspotential zu Tage, welches in der Vermeidung von nicht-strikten Funktionen (z.B. *ite*) liegt. Ersetzt man die Funktion *ite* wo möglich durch das strikte if-then, so erreicht man eine Verdopplung der aufgedeckten Striktheiten und eine Reduzierung der Parameter ohne Analyseaussage um über 20%. Dadurch erreicht man im Falle der Funktion *Quicksort* bereits das in [SR96] angegebenen Ergebnis, daß lediglich zwei der bearbeiteten Parameter nicht strikt sind. Sicherlich kann man durch eine weitere 'Anpassung' der Funktionsdefinitionen an die abstrakte Analyse weitere Verbesserungen erzielen. Dann stellt sich aber die Frage, in wie Weit diese Ergebnisse dann noch praxisrelevante Aussagen ermöglichen.

Ein weiteres Verbesserungspotential liegt in der Anpassung des Domains an die Analysierten Funktionen, was durch eine weitere Verbesserung der Ergebnisse mit Einführung des Domains 7 belegt wird. Dabei ist sicherlich zu beachten, daß mit der Erweiterung des Domains eine Verschlechterung der Laufzeit einhergeht. Dies liegt an einer steigenden Anzahl zu berechnender Parameter, da potentiell jede Funktion mit jedem Demand Pattern analysiert werden kann. Es zeigt sich jedoch, daß nur selten eine deutlich höhere Zahl an Berechnungen auftritt und die Zahl der benötigten Iterationen gar konstant ist. Damit dürfte sich der Vorteil einer genaueren Analyse gegenüber dem Nachteil der Laufzeitverschlechterung behaupten. Dies stellt auch [SR96] fest, wo zur Berechnung der dargestellten Ergebnisse ein Domain mit 30 Elementen benutzt wurde, um für jede analysierte Funktion einen bestmöglichen Domain zu bieten. Vergleicht man dies mit dem hier behandelten Domain 7 mit lediglich 12 Elementen, so eröffnet sich noch einiges an weiterem Verbesserungspotential. Dafür müssten lediglich die Elemente des Domains weiter aufgegliedert werden, welche einer Menge strikter Demand Pattern entsprechen (besonders e). Auch kann man weitere Konstruktoren, außer den Listenkonstruktoren, behandeln. Bringt man weiterhin genauere Informationen über die Parameter vordefinierter Funktionen in die Analyse ein, anstatt wie hier geschehen lediglich deren Striktheit zu deklarieren, so läßt sich die Struktur der strikten Parameter detaillierter analysieren.

Ein erster Schritt hin zu besseren Ergebnissen muß nun sicherlich die Bestimmung eines Domains sein, welcher für die meißten logisch-funktionalen Programme gute Ergebnisse erzielt. Um ein Gefühl dafür zu entwickeln, wie sich Änderungen am Domain auf die Analyse auswirken wurden 8 verschiedene Domains implementiert, mit denen sich für unterschiedliche Aufgaben akzeptable Resultate erzielen lassen. In wie weit sich die Ausnutzung der Analyseergebnisse durch einen Compiler auf die effiziente Auswertung von Curry Programmen auswirkt, soll nicht Bestandteil dieser Arbeit sein.

Im Anschluß folgt noch die Erweiterung der Analyse um disjunktive Bereiche und eine Diskussion über mögliche Vor- und Nachteile einer solchen Ergänzung.

Kapitel 8

Disjunktive Bereiche

Wie schon erwähnt lassen sich Probleme bei der Analyse durch die Einführung von disjunktiven Bereichen beheben. Hier sollen nun die notwendigen Änderungen in der denotationellen Analyse sowie die Integration in das Analysesystem beschrieben werden. Desweiteren folgt eine Diskussion der Ergebnisse, die durch Hinzunahme der disjunktiven Bereiche zur Analyse erzielt werden konnten.

8.1 Semantik mit disjunktiven Bereichen

Die disjunktiven Bereiche wurden motiviert durch einen Verlust an Information, welcher sich bei der Vereinigung von Teilergebnissen ergab. Die durch $union_A$ berechnete abstrakte Substitution vereinigt dabei die Informationen der Analyseergebnisse der einzelnen Regeln einer Funktion. Dabei kann es vorkommen, daß Informationen über einzelne Parameter der Funktion vollständig verloren gehen. Dies soll an einem Beispiel noch einmal erläutert werden. Betrachten wir dazu die Funktion *ite/3*, welche ein if-then-else Konstrukt definiert (s. Kapitel 7). Bei der Analyse wie wir sie in Kapitel 5 definiert haben ergibt sich bei der Berechnung von $[\![ite(X_1, X_2, X_3)]\!]\ e$ folgende Konstellation (Eine Beschreibung der Domainelemente findet sich im Anhang D).

$$\begin{array}{rl} \text{Regel 1:} & \rightarrow \{(x_1, e), (x_2, e), (x_3, top)\} \\ \text{Regel 2:} & \rightarrow \{(x_1, e), (x_2, top), (x_3, e)\} \\ \hline union_A: & \rightarrow \{(x_1, e), (x_2, top), (x_3, top)\} \end{array}$$

Wie man sieht erhält man lediglich Informationen über einen der drei Parameter, obwohl die beiden Teilergebnisse jeweils genauer sind. Dieser Informationsverlust entsteht durch die Berechnung des *lub* (least-upper-bound) auf dem abstrakten Domain, da hier gilt $lub(e, top) = top$. Um dies zu Vermeiden bietet sich die Möglichkeit, die einzelnen Teilergebnisse nicht zu einem Gesamtergebnis zu vereinigen, sondern disjunktiv zu verbinden.

$$\begin{array}{rcl} \text{Regel 1:} & \rightarrow & \{(x_1,e),(x_2,e),(x_3,top)\} \\ \text{Regel 2:} & \rightarrow & \{(x_1,e),(x_2,top),(x_3,e)\} \\ \hline \mathit{union}': & \rightarrow & \{(x_1,e),(x_2,e),(x_3,top)\}\vee \\ & & \{(x_1,e),(x_2,top),(x_3,e)\} \end{array}$$

In diesem Fall bleibt die gesamte Information beider Regeln erhalten. Da sich solche abstrakten Substitutionen mit den bisherigen Möglichkeiten aber nicht ausdrücken lassen muß zunächst die Definition der abstrakten Substitution geändert werden.

Eine abstrakte Substitution φ_A auf einer Menge $\{X_1,\ldots,X_n\}$ von Variablen sei nun folgendermaßen definiert:

$$\varphi_A = \bigvee_{i=1}^{n}\bigwedge_{j=1}^{m} X_j^{a_{ij}}$$

wobei $a_{ij} \in \mathcal{DP}_A$. Auf variablenfreien Termen verwenden wir weiterhin die Substitutionen $true$ und $false$. Dabei steht $true$ für die Menge, die nur die identische Substitution enthählt ($\gamma(true) = \{\{\}\}$) und $false$ für die leere Substitutionsmenge ($\gamma(false) = \{\}$). Andernfalls approximiert eine Disjunktion die Vereinigung der durch die Konjunktionen approximierten Mengen.

$$\gamma(\varphi_A) = \bigcup_{i=1}^{n}\gamma(\bigwedge_{j=1}^{m} X_j^{a_{ij}})$$

wobei weiterhin gilt:

$$\gamma(\bigwedge_{j=1}^{m} X_j^{a_j}) = \{\alpha \in Sub \mid dom(\alpha) \subseteq \{X_1,\ldots,X_n\}, \alpha(X_j) \in \gamma(a_j), \forall j\}$$

Mit dieser Definition lassen sich nun die im obigen Beispiel gezeigten abstrakten Substitutionen darstellen. Aufgrund der geänderten abstrakten Substitutionen müssen nun auch alle Operationen auf diesen Substitutionen geändert werden. Es sind dies die Operationen $\sqcup_A$, $union_A$ und $match_A$, die im folgenden neu definiert werden sollen.

Die Funktion $match_A$ wendet eine abstrakte Substitution auf die Teilterme einer linken Regelseite an. Dies muß nun für jedes Disjunktionsglied gemacht werden. Daher liefert das neue $match'_A$ eine Disjunktion der Ergebnisse des alten $match_A$ (vergl. Kapitel 5).

$$match'_A(\{X_1,\ldots,X_l\}, \bigvee_{i=1}^{n}\bigwedge_{j=1}^{m} Y_j^{a_{ij}}, \{t_1,\ldots,t_l\}) = \bigvee_{i=1}^{n}\bigwedge_{k=1}^{l} X_k^{apply_A(t_k, \bigwedge_{j=1}^{m} Y_j^{a_{ij}})}$$

Als nächtes betrachten wir die Operation $union_A$. Diese vereinigt die Ergebnisse der einzelnen Regeln zu einem Gesamtergebnis. Wie im Beispiel gesehen sollen die Teilergebnisse nun lediglich durch Disjunktionen verknüpft werden. Daraus ergibt sich, daß $union'_A$ lediglich die übergebenen Substitutionen zu einer Disjunktion verbinden muß.

$$union'_A(\varphi^j_A)_j = \bigvee_j \varphi^j_A$$

Damit verbleibt nur noch die Definition der Operation $\sqcup_A$. $\sqcup_A$ vereinigt Substitutionen mit überlappenden Domains zu einer Gesamtsubstitution, welche alle Eigenschaften der Ausgangssubstitutionen erhält. Beim Auftreten disjunktiver Glieder müssen nun alle Kombinationen von Disjunktionsgliedern der einzelnen Substitutionen beachtet werden. Jede dieser Kombinationen kann dann wieder mit der alten $\sqcup_A$-Operation behandelt werden.

$$\sqcup'_A(\bigvee_{i_1=1}^{n_1} \psi_1, \dots, \bigvee_{i_k=1}^{n_k} \psi_k) = \bigvee_{\substack{m\in\{1,\dots,k\}\\ n\in\{1,\dots,k\}\\ m\neq n}} \sqcup_A(\psi_{k_m}, \psi_{k_n})$$

Damit sind alle Operationen, die auf abstrakten Substitutionen arbeiten neu definiert. Besonders an der Definition von $\sqcup'_A$ wird deutlich, daß die Größe der abstrakten Substitutionen explosionsartig ansteigen kann, wenn Funktionen mit vielen Argumenten analysiert werden.

Nun können wir die abtrakte Semantik definieren (s. Abbildung 8.1), welche im Wesentlichen nicht von der in Kapitel 5 abweicht, wenn man die obigen Operationen durch die neu definierten ersetzt. Der einzige Unterschied besteht darin, daß im dritten Fall der Komposition die Ergebnisse der $\sqcup'_A$-Aufrufe mittels $union'_A$ vereinigt werden müssen.

8.2 Anlyseergebnisse mit disjunktiven Bereichen

Die im letzten Abschnitt definierte Analyse mit disjunktiven Bereichen soll nun ebenfalls an den Benchmark-Funktionen aus Kapitel 7 getestet werden. Dazu wurde das in Kapitel 6 implementierte Analysesystem um disjunktive Bereiche erweitert. Damit dem Benutzer auch hier die Wahl bleibt, ob er die Erweiterung nutzen möchte, wurden der Shell zwei weitere Befehle hinzugefügt. Durch die Eingabe *disjunct* wird die Analyse auf die Benutzung von disjunktiven Bereichen umgestellt. Dieser Zustand bleibt solange erhalten, bis mittels *nodisjunct* wieder auf die ursprüngliche Analyse zurückgeschaltet wird.

Die in der Analyse erzielten Ergebnisse sollen nun mit denen aus Kapitel 7 verglichen werden. Dazu wurde zunächst die Analyse durchgeführt mit der das Kapitel 7 beendet wurde, d.h. die Analyse der Funktionen im zweiten Benchmark. Wir erinnern uns, daß im zweiten Benchmark so weit wie möglich auf die

Abstrakte Semantik für disjunktive Bereiche
$[\![t]\!]\ \pi_A = (\bigsqcup_{n\geq 0})\ (t,\pi_A)$wobei $\Psi^0_A(t,\pi_A) = false$, für $\mathcal{V}ar(t) = \{\}$ oder
$\Psi^0_A(t,\pi_A) = X_1^\perp \wedge \ldots \wedge X_n^\perp$, für $\mathcal{V}ar(t) = \{X_1,\ldots,X_n\}$
Einfache Terme
$\Psi^{n+1}_A(X,\pi_A) \quad = X^{\pi_A} \qquad X \in \mathcal{X}$
$\Psi^{n+1}_A(c(\overline{X},\pi_A) \quad = \bigwedge_i X_i^{proj^i_A(c,\pi_A)} \qquad c \in \mathcal{C}$
$\Psi^{n+1}_A(f(\overline{X}),\pi_A) \quad = union'_A(match'_A(\Psi^n_A(r_i,\pi_A),\{\overline{t_i}\}))_{f(\overline{t}_i)\to r_i\in\mathcal{R}_\epsilon} \qquad f \in \mathcal{D}_\epsilon$
Komposition
$\Psi^{n+1}_A(t,\pi_A) \quad = true \qquad \gamma(\pi_A)\cap\mathcal{X} \neq \{\}, \mathcal{V}ar(t) = \{\}$
$\Psi^{n+1}_A(t,\pi_A) \quad = X_1^\top \wedge \ldots \wedge X_n^\top \qquad \gamma(\pi_A)\cap\mathcal{X} \neq \{\}$, $\mathcal{V}ar(t) = \{X_1,\ldots,X_n\}$
$\Psi^{n+1}_A(f(\overline{t}),\pi_A) \quad = union'_A(\sqcup_A(\Psi^{n+1}_A(t_j,a_ij)_i)_j \qquad \Psi^{n+1}_A(f(\overline{Z}),\pi_a) = \bigvee_i\bigwedge_j Z_j^{a_ij}$, $f \in \mathcal{F}_\epsilon$,

Abbildung 8.1: Die Analyse mit disjunktiven Bereichen

Benutzung von *ite*-Aufrufen verzichtet wurde. Analysiert werden diese Funktionen ebenfalls wieder mit dem vordefinierten Domain 7 (s. Anhang D). Zum Vergleich nun zunächst die Ergebnisse der Analyse aus Kapitel 7:

81 Parameter bearbeitet, davon	
47%	strikte Auswertung (e,liste)
16%	endliche Listen (listhn,listf)
12%	potentiell unendliche Listen (list,heade,headhn)
25%	keine genaueren Angaben möglich

Schaut man sich nun die Ergebnisse an, die mittels der disjunktiven Bereiche erzielt wurden, so fällt zunächst der extrem höhere Aufwand auf. Anstatt der bisher 81 Paramter mussten nun vielmehr 197 Parameter bearbeitet werden und zusätzlich wird in den meißten Fällen eine weitere Iteration nötig.

197 Parameter bearbeitet, davon	
70%	strikte Auswertung (e,liste)
8%	endliche Listen (listhn,listf)
6%	potentiell unendliche Listen (list,heade,headhn)
16%	keine genaueren Angaben möglich

Man sieht aber deutlich, daß durch die disjunktiven Bereiche eine erhebliche Steigerung der Güte der Analyse erreicht wurde. So wurde ein Anteil von strikten Parametern von 70% erreicht. Vergleich man diesen Wert mit dem theoretisch möglichen von ca. 80%, so werden in etwa 85% der tatsächlichen Striktheiten auch aufgedeckt.

Da der zweite Benchmark definiert wurde, um die Analyse von *ite*-Konstrukten zu umgehen, können wir nun, wo der Informationsverlust durch $union_A$ beseitigt ist, wieder auf den alten Benchmark zurückgreifen. Wie wir im Kapitel 7 gesehen haben, hängt die Zahl der zu berechnenden Parameter auch von der

Größe des benutzten Domains ab. Um den Analyseaufwand in etwa wieder an den der alten Analyse anzugleichen analysieren wir nun mit einem weniger umfangreichen Domain. Daher sollen jetzt die Analyseergebnisse aufgezeigt werden, welche sich mit dem vordefinierten Domain 3 (s. Anhang D) ergaben.

116 Parameter bearbeitet, davon	
60%	strikte Auswertung (e,liste)
11%	Listen allgemein (list)
12%	potentiell unendliche Listen (list,heade,headhn)
29%	keine genaueren Angaben möglich

Vergleichen wir diese Werte nun mit denen der ersten Anaylse auf diesem Benchmark, so stieg die Zahl der strikten Parameter von 22 auf 60%, die Zahl derer ohne Analyseerfolg fiel von 51 auf 29% (vergl. Seite 48).

Eine Überprüfung in wie weit sich die besseren Ergebnisse auch in die Optimierung eines Compilers umsetzen lassen müssen spätere Untersuchungen zeigen. Was die Aussagekraft der Analyseergebnisse angeht ist die Benutzung von disjunktiven Bereichen auf jeden Fall lohnenswert.

Kapitel 9

Fazit und Ausblick

Über die Definition einer denotationellen Semantik wurde die Basis zur Implementierung einer Demand Analyse für Curry geschaffen. Es hat sich gezeigt, daß es möglich ist die Semantik eines Terms aus der Semantik seiner Teilterme zu erhalten und damit die Semantik eines Programms zu definieren. Durch den erbrachten Beweis der Korrektheit der Semantik unter bestimmten Annahmen ließen sich die Eigenschaften der Semantik auf eine Abstraktion übertragen. Diese Abstrakte Semantik mußste lediglich einige leicht zu überprüfende Eigenschaften besitzen und die Aussagen über Korrektheit und Existenz des Fixpunktes konnten von der denotationellen Analyse übernommen werden.

Diese abstrakte Analyse wurde so definiert, daß sie über einige Operationen parametrisiert war. Dadurch läßt sie sich als Rahmen für verschiedene Aufgaben, wie eine Demandanalyse oder auch eine Striktheitsanalyse, nutzen. Aufbauend auf diesem Rahmen wurde eine Demand Analyse entwickelt, welche für die Analyse von Curry-Programmen implementiert wurde. Das Analysesystem wurde so modular gehalten, daß jederzeit die zu analysierenden Programme und die benutzten Domains ausgetauscht werden können. Dadurch bietet es die Möglichkeit auch über den Rahmen dieser Arbeit hinaus als Experimentierwerkzeug zur Optimierung der Domains zu dienen.

Um die erzielbaren Resultate zu untersuchen wurde ein Benchmark-Programm benutzt und mit verschiedenen Parametern analysiert. Die Ergebnisse dieser Analysen haben gezeigt, daß auch einfache Domains durchaus ausreichen, um erste brauchbare Resultate zu erzielen. Weiterhin stellte sich heraus, daß durch Manipulationen des Domains ein großes Verbesserungspotential erschlossen werden kann. Eine weitere Möglichkeit die Genauigkeit der Analyseergebnisse zu steigern bot sich durch die Einführung von disjunktiven Bereichen. Hier konnte man feststellen, daß auf Kosten eines erheblich höheren Rechenaufwandes nocheinmal eine deutliche Steigerung der Genauigkeit möglich war.

Aufgabe für die Zukunft ist es nun durch ausreichend viele Analysen zu einem für die Compileroptimierung geeigneten Domain zu finden. Danach gilt es einen Compiler zu entwickeln, der in der Lage ist, die vom Analysesystem berechneten Informationen zu verwerten. Dann erst wird sich am Laufzeitverhalten der optimierten Programme zeigen, welches Verbesserungspotential wirklich in der Demand Analyse steckt. Auf jeden Fall haben die Untersuchungen der Benchmark-Funktionen gezeigt, daß es möglich ist zur Compilezeit ausreichend verwertbare Demand Informationen zu berechnen und das es sich lohnen wird Aufbauend auf dieser Implementierung nach effizienten Analysesystemen zu suchen.

Anhang A

Programmdateien

Zur Benutzung des Systems unter Sicstus-Prolog (oder auch SWI-Prolog für MS-DOS Rechner) müssen zunächst die einzelnen Programmteile compiliert werden. Dazu compiliert man zunächst das Modul *getall.pl*. Dieses definiert ein Prädikat mit dem Namen *getAll/0*. Durch anschließenden Aufruf von *getAll/0* werden die einzelnen Teile des Systems compiliert und danach das System mit einem Aufruf des Prädikats *start/0* aufgerufen.

Nun soll kurz beschrieben werden, welche Sourcecode Dateien das Analysesystem bilden und welche Funktion sie jeweils erfüllen.

Neben dem Modul *getall.pl* mit der oben beschriebenen Funktion umfaßt das System folgende weitere Dateien:

- `helpfunk.pl`
- `shell.pl`
- `readtree.pl`
- `transf.pl`
- `abstract.pl`
- `domainX.pl`

Die Datei *shell.pl* enthält alle nötigen Prädikate zur Steuerung der Benutzershell. Dazu gehören die Eingabe der Parameter für die Analyse, der Start der Analyse, sowie die Auswertung und Ausgabe des Analyseergebnisses.

helpfunk.pl enthält Hilfsfunktionen, die in allen Programmteilen benutzt werden. Dies sind größtenteils allgemeine Listenfunktionen und ähnliche universelle Hilfmittel.

Die Einbindung der Dateien, die Informationen über die definierenden Bäume der zu analysierenden Funktionen enthalten, geschieht durch Aufruf der Prädikate in *readtree.pl*. Die Informationen aus diesen Dateien werden eingelesen und in eine interne Darstellung umgewandelt.

Bevor die Analyse gestartet werden kann müssen die definierenden Bäume in äquivalente Termersetzungssystem umgewandelt werden. Diese der Transformation Π aus Abschnitt 4.4 entsprechnde Aufgabe wird durch *transf.pl* bereitgestellt.

Die Steuerung der Analyse, also insbesondere die Abwicklung der Fixpunktberechnung, werden durch die Prädikate in *abstract.pl* realisiert.

Alle diese Module werden durch den Aufruf von *getAll* compiliert und bilden den Hauptteil des Analysesystems.

Schließlich enthalten die Dateien *domain1*, ... ,*domain8* die vordefinierten Domains, diese werden erst bei Bedarf dem System hinzugefügt. Dies geschieht durch den Befehl *getDomain* an der Eingabeaufforderung der Shell. Eine detailierte Beschreibung der Domains findet sich im Anhang D. Da die Domains erst zur Laufzeit in das System eingebunden werden besteht die Möglichkeit eigene Domains zu definieren. Dazu wird der Aufbau einer Domaindatei detailliert beschrieben im Anhang C. Eine genaue Übersicht über die Befehle der Shell steht im Abschnitt 6.4.

Anhang B

Die Funktionen der Fixpunktberechnung

Der zentrale Aspekt der Implementierung einer abstrakten Analyse ist die Umsetzung der Fixpunktberechnung. Der prinzipielle Ablauf wurde bereits in Abschnitt 6.3 auf Seite 39 erläutert. Hier sollen nun die Funktionen, die der Abbildung B.1 zu entnehmen sind detailliert beschrieben werden.

Im Einzelnen sind dies:

- startIteration: Starten einer neuen Iteration der Berechnung
- phi0: Berechnen eines Startwertes Φ_A^0 für die Fixpunktberechnung
- phi: Berechnet für gegebenen Term und Pattern Φ_A^{n+1}
- updLut: Trägt neue Berechnungsergebnisse in die Look-up-Table ein

Zunächst soll das Verhalten von *startIteration(T,DP)* beschrieben werden. Diese Funktion erhält den zu Analysierenden Term T und das zu erreichende Demand Pattern DP und liefert eine abstrakte Substitution zurück. Dabei werden folgende vier Fälle überprüft:

1. In der Look-up-Table steht der Eintrag (T,DP,S_A,complete,_)

   Dies bedeutet, daß die Berechnung für den Term T und das Pattern DP bereits abgeschlossen ist und keine neue Iteration durchgeführt werden muß. Daher liefert *startIteration* als Ergebnis das in der Tabelle eingetragene S_A zurück.

2. In der Look-up-Table steht der Eintrag (T,DP,S_A,fixpoint,_)

   Dies bedeutet, daß die Berechnung für T und DP im Gange ist, sich also bei dem Aufruf von *startIteration* um eine rekursive Berechnung handelt. Da ein erneuter Aufruf von *phi* in eine endlose Rekursion führen würde wird auch hier der in der Tabelle eingetragene Wert S_A zurückgeliefert.

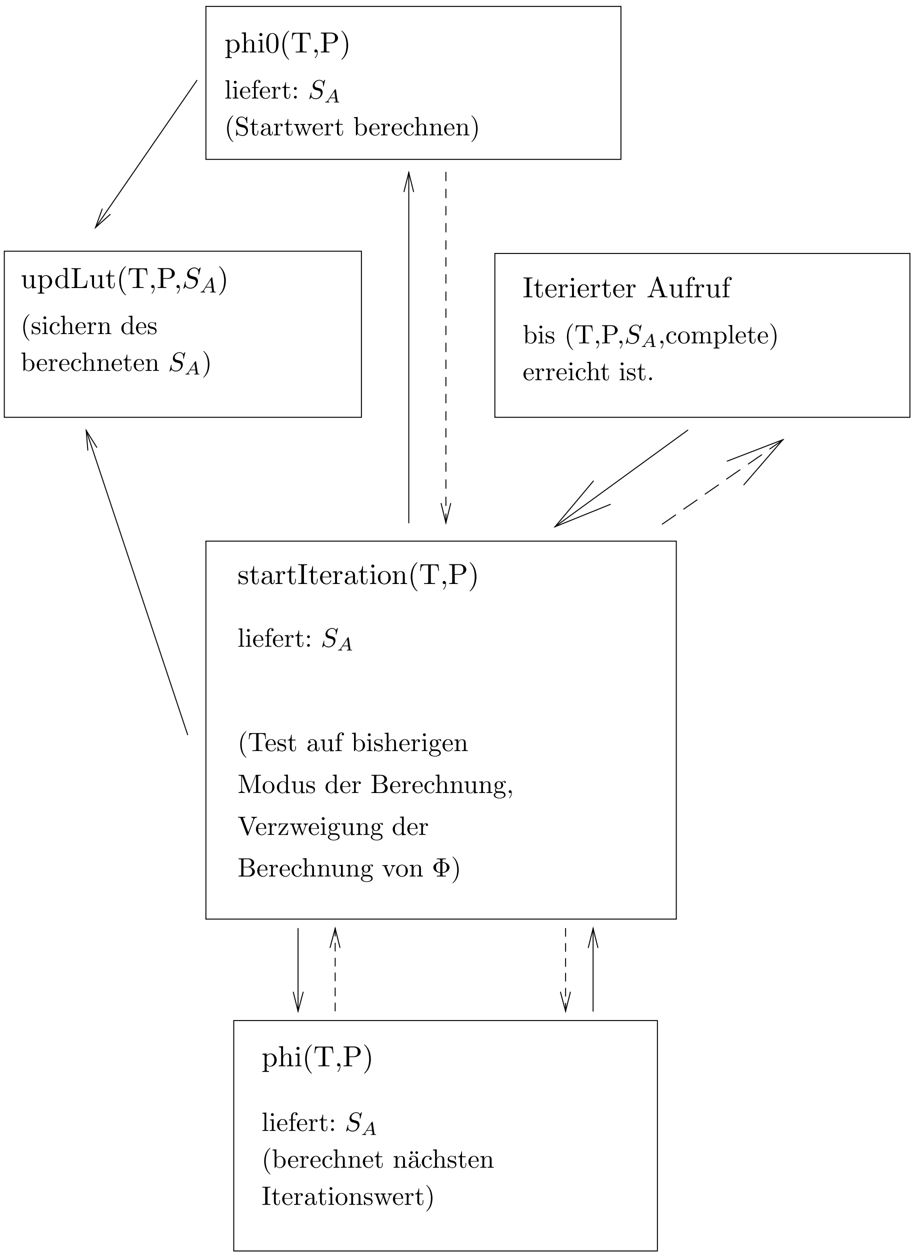


Abbildung B.1: Die Funktionen der Fixpunktberechnung

3. In der Look-up-Table steht kein Eintrag für T und DP.

   Dies bedeutet, daß für das Paar (T,DP) zum erstenmal eine Berechnung durchgeführt wird. In diesem Fall wird mittels *phi0* ein Startwert berechnet, dieser durch *updLut* in die LUT eingetragen und dann durch einen Aufruf von *Phi* eine neue Iteration für T und DP gestartet. Das Ergebnis dieser Iteration wird wieder in die LUT eingetragen und als Ergebnis von *startIteration* zurückgeliefert.

4. In der Look-up-Table steht der Eintrag (T,DP,S_A,approx,_)

   Das heißt, daß der Eintrag der Tabelle einen vorläufigen Wert für T und DP darstellt. Es wurden noch nicht vollständig berechnetet Zwischenergebnisse verwendet oder es ist noch nicht sicher, ob der Eintrag dem Fixpunkt entspricht. Bevor nun eine neue Iteration mittels *phi* angestoßen wird muß der LUT Eintrag auf den Modus *fixpoint* gesetzt werden, um im Falle einer Rekursion analog zum 3.Fall handeln zu können. Das Ergebnis von *phi* wird mittels *updLut* in die Tabelle eingetragen und als Ergebnis von *startIteration* zurückgegeben.

Als nächstes soll kurz die Funktion von *phi0(T,DP)* zur Berechnung von Initialwerten beschrieben werden. Hier wird analog zur Definition von Φ_A^0 aus Kapitel 5 in zwei Fälle unterschieden.

1. Falls $\mathcal{V}ar(T) = \{\ \}$, dann liefere als Startwert die Substitution $S^\perp = false$ und sichere diese mittels *updLut* in der Look-up-Table.
2. Falls $\mathcal{V}ar(T) = \{X_1, \ldots, X_n\}$, dann liefere die Substitution $S^\perp = \{X_1^\perp, \ldots, X_n^\perp\}$ als Initialwert zurück. Auch dieser Wert wird mittels *updLut* in die LUT übertragen.

Diese Definition folgt also direkt aus der Definition von Φ_A^0 und bedarf keiner weiteren Erläuterungen. Die Implementierung der Funktion *phi* setzt exakt die Definition der Transformation Φ_A^{n+1} aus Kapitel 5 um. Eine genauere Betrachtung der definierten Prädikate, sowie der benutzten Techniken würde den Rahmen dieser Arbeit sprengen und zu sehr in programmtechnische Details gehen. Daher soll es hier ausreichen, das ein Aufruf von *phi(T,DP)* eine neue abstrakte Substitution S_A für Term T und Demand Pattern DP berechnet.

Als letztes fehlt noch die Beschreibung der Funktion *updLut(T,DP,SA)* welche die berechneten Ergebnissubstitutionen S_A in die Look-up-Table schreibt. Wie im Kapitel 6 beschrieben bedingt der Eintrag eines neuen Ergebnisses in die LUT eine Transformation des gespeicherten Modus. Das bedeutet, daß die Funktion *updLut* nicht nur das neue Zwischenergebnis ablegt, sondern auch die durch *upd* in Kapitel 6 definierte Transformation auf dem restlichen Tupel des Eintrags vornimmt. Damit übernimmt diese Funktion die Kontrolle über die Steuerung der Fixpunktberechnung. Die Berechnung der Anfrage wird als vollständig betrachtet, sobald der Eintrag, welcher der Anfrage entspricht den Modus *fixpoint* erreicht hat. Dann wird der aktuelle Wert für die berechnete Substitution aus der Look-up-Table abgelesen und zurückgeliefert.

Damit sind die wichtigsten Funktionen der Fixpunktberechnung beschrieben. Neben diesen treten lediglich noch einzelne Hilfsprädikate auf, deren Funktion durch die vorangegangenen Erläuterung klar werden dürfte.

Anhang C

Aufbau eines Domains

Das System umfaßt 8 verschiedene Domains, welche zur Laufzeit wahlweise in das System eingebunden werden können. Dadurch besteht die Möglichkeit neue Domains zu definieren und auszutesten. Hier soll daher beschrieben werden, welche Prädikate durch eine Domaindatei definiert werden müssen, und welche Aufgaben diese erfüllen. Auf Seite 70 ist beispielhaft die Datei für den vordefinierten Domain 3 (siehe Seite 75) aufgeführt. Diese soll als Verdeutlichung der folgenden Ausführungen dienen.

C.1 Definition der Domainstruktur

Zunächst muß die Struktur des Domains definiert werden, d.h. es wird definiert, welche Elemente der Domain umfaßt und wie die Vergleichsoperation auf dem Domain definiert ist. Die Elemente des Domains werden mit dem Prädikat *dom/1* eingeführt, welches als Parameter eine Liste der Domainelemente erhält. Im Beispiel des Domains3 geschieht dies durch:

`dom([bot,liste,e,list,top]).`

Nun müssen mehrere dieser Domainelemente besonders ausgezeichnet werden. Zunächst müssen das kleinste und das größte Element des Domains kenntlich gemacht werden. Dies geschieht durch die Prädikate *getBottom* und *getTop*. Im Domain 3 bedeutet dies, daß:

`getBottom(bot)` und
`getTop(top)`

aufgeführt werden müssen.

Das weitere ausgezeichnete Element (D_{predef}) steht für die Abstrahierung aller Demand Pattern, welche Ergebnis der vordefinierten Operationen von Curry beschreiben könnten. Diese Operationen (z.B. +,*,/,- usw.) werden in der Analyse gesondert behandelt, da sie keinen zugeordneten definierenden Baum besitzen. Trifft die Analyse also auf eine dieser Funktionen, so geht sie davon aus, daß die

Abstraktion ihres Ergebnisses in D_{predef} liegt und daß alle Parameter diesem ausgewiesenen Demand Pattern entsprechen müssen. Dies bedeutet, daß z.B. die Anfrage $[\![x + y]\!]\ D_{predef}$ zwei neue Anfragen $[\![x]\!]\ D_{predef}$ und $[\![y]\!]\ D_{predef}$ auslöst. Das Element D_{predef} wird durch das Prädikat *getPreDefPat(D_{predef})* ausgewiesen. In unserem Fall also durch:

```
getPreDefPat(e).
```

Die Benutzershell stellt den Befehl *showDomain* zur Verfügung, welcher die Elemente des Domains sowie eine kurze Beschreibung der von ihnen abstrahierten Mengen ausgibt. Der beschreibende Text wird dabei jeweils durch das Prädikat *whatIs(D)* für alle D $\in DP_A$, also für alle Elemente des Domains, ausgegeben. Damit hat das Prädikat *whatIs* z.B. folgendes aussehen:

```
whatIs(list):-write('Contains all cons-headed terms.').
```

Nun muß noch die Ordnung auf dem Domain definiert werden. Dies geschieht durch das zweistellige Prädikat *ord*. Damit bei großen Domains keine zu große Zahl von Fakten aufgeführt werden muß, genügt es direkt benachbarte Paare aufzuführen. Dabei steht *ord(D_1,D_2)* synonym für $D_1 \leq D_2$. Der reflexive transitive Abschluß der Vergleichsoperation ist in der Datei *abstract.pl* definiert. Die Vergleichsoperation im Domain 3 wird also durch die folgenden 4 Fakten definiert:

```
ord(bot,_).
ord(liste,e).
ord(liste,list).
ord(_,top).
```

Damit ist die Struktur des Domains vollständig definiert und es fehlt nur noch die Angabe, welche der Domainelemente abstrahierte Variablen enthalten. Diese Information wird in der Analyse in den Regeln zur Komposition benötigt und durch das Prädikat *is_Variable* zur Verfügung gestellt. Da hier lediglich das Element *top* Variablen abstrahiert, genügt folgendes Faktum:

```
is_Variable(top).
```

Desweiteren müssen noch die domainabhängigen Operationen $proj_A$ und $apply_A$ definiert werden.

C.2 $proj_A$ und $apply_A$

Während sich $union_A$, $match_A$ und $\sqcup_A$ unabhängig vom gewählten Domain definieren lassen, müssen $apply_A$ und $proj_A$ im Zusammenhang mit dem Domain stehen, da sie direkt von den konkreten Elementen des Domains Gebrauch machen. Diese zwei Operationen werden durch die Prädikate *proj/4* und *apply/3* realisiert.

Dabei gibt $proj(p, c, d_1, d_2)$ an, daß der Konstruktor c einen Term liefert, der durch das Demand Pattern d_1 abstrahiert wird, wenn das Argument an Position

p einem Pattern aus $\gamma(d_2)$ genügt. Diese Angabe wird für jeden Konstruktor, jede Argumentposition und jedes Domainelement benötigt, auch für null-stellige Konstruktoren, wie *nil*. Da die häufigsten Konstruktoren in logisch-funktionalen Programmen die Listenkonstruktoren *nil* und *cons* sind, wurden diese hier besonders behandelt, während alle anderen Konstruktoren durch anonyme Variablen an der zweiten Argumentposition von *proj* aufgefangen werden. Die zur vollständigen Definition der Operation $proj/4$ notwendigen Regeln und Fakten sind dem unten aufgeführten Listing zu entnehmen.

Als letztes folgt die Definition von $apply_A$. Wie der Name andeutet realisiert die Operation $apply(t, \sigma_A, d)$ die Anwendung einer abstrakten Substitution σ_A auf einen Term t. Dabei gibt das Domainelement d an, in welcher Konkretisierungsmenge sich das Demand Pattern befindet dem der entstehende Term genügt. Aufgrund der Struktur der denotationellen Analyse ist der Term t entweder eine Varibale oder ein Konstruktorterm. Wie schon bei $proj_A$ so wird auch hier der Listenkonstruktor *cons* speziell betrachtet, während bei anderen Konstruktoren in das Domainelement *top* abgebildet wird. Für den Fall des Domains 3 ist auch dieses Prädikat im folgenden Listing ersichtlich.

C.3 Ein Beispieldomain

```
/* implementiert Operationen, die direkt vom Domain abhaengen */
/*-----------------------------------------------------------*/
/* Domain ={bot,nil,liste,e,list,top} mit: */
/* bot:undef, nil:leere Liste, liste:vollst. ausgew. Listen */
/* e:vollst. ausgew. Terme, list:alle Listen, top:alle Terme */
/*-----------------------------------------------------------*/

:-dynamic dom/1, getBottom/1, getTop/1, whatIs/1, ord/2,
is_Variable/1, apply/3, proj/4, getPreDefPat/1.

/* Die Elemente des abstrakten Domains */
dom([bot,liste,e,list,top]).

/* liefert die Elemente Top und Bottom des Domains */
getBottom(bot).
getTop(top).

/* liefert Demand Pattern vordefinierter Funktionen */
getPreDefPat(e).

/* whatIs(DP) liefert zu jedem Pattern eine Erlaeuterung */
whatIs(bot):-write('The bottom element. Less than the rest.').
whatIs(liste):-write('Contains all fully instantiated lists.').
whatIs(list):-write('Contains all cons-headed terms.').
whatIs(e):-write('Contains all fully instantiated terms.').
whatIs(top):-write('The top element. Greater than the rest.').

/* ord gibt die Ordnung auf dem abstrakten Domain an, dabei */
/* ist ord(X,Y) erfuellt, wenn X direkt vor (unter) Y steht */
```

```
ord(bot,_).
ord(liste,e).
ord(liste,list).
ord(_,top).

/* is_Variable(Pa) gibt an, ob in Gamma(Pa) eventuell */
/* Variablen enthalten sind. */
is_Variable(top).

/* proj(I,C,Pa,Pa2) projeziert Pa auf i-tes Argument des */
/* Terms C, d.h. proj(I,C,Pa,Pa2)->Pa2 ist DP für */
/* Teilterm I des Terms C. Bei 0-st. Konstruktoren */
/* liefert proj(0,...) direkt abstr.Substit. */
proj(0,[],bot,false).
proj(0,[],_,true).


proj(0,_C,top,true).
proj(0,_C,e,true).
proj(0,_C,_,false).

proj(N,':',P1,P2):- proj(N,cons,P1,P2).

proj(_,cons,top,top).
proj(_,cons,list,top).
proj(1,cons,e,e).
proj(2,cons,e,liste).
proj(1,cons,liste,e).
proj(2,cons,liste,liste).
proj(_,cons,_,bot).
proj(_,_,liste,bot).
proj(_,_,list,bot).
proj(_,_,_,top). /* ansonsten keine Aussage moeglich -> top */

/* apply(Term,Sa,Pa) wendet abstrakte Subst.Sa auf Term */
/* an und liefert das abstrakte Ergebnispattern zurueck */
apply(_,false,bot). /* Substitution false liefert immer bot */

/* apply fuer alle 0-st. Konstruktoren */
apply(term([],0,[]),_,liste).
apply(term(_C,0,[]),_,e).

/* Falls erstes Argument eine Variable ist */
apply(var(_),true,top).
apply(var(X),[(var(X),Ai)|_],Ai). /* Ergebnispattern ablesen */
apply(var(X),[_|SR],A):- apply(var(X),SR,A).
apply(var(_),[],top). /* kein Eintrag fuer X -> top */

/* Testet Fall 1(bottom) fuer Konstruktoren */
/* apply(c(X1,...,Xn),S)=bot , wenn ex. i mit S(Xi)=bot */
apply(term(_C,_A,[var(X)|_XR]),Sa,bot):- apply(var(X),Sa,bot).
apply(term(C,A,[var(_)|XR]),Sa,bot):- apply(term(C,A,XR),Sa,bot).
```

```
/* Testet Fall 2(list) fuer Konstruktoren :,Cons */
/* Falls erstes Argument ein Listen-Construktorterm ist */
apply(term(':',2,AL),Sa,P):- apply(term(cons,2,AL),Sa,P).

apply(term(cons,2,[var(X1),var(X2)]),Sa,liste):-
           apply(var(X1),Sa,e),
           apply(var(X2),Sa,liste).
apply(term(cons,2,[var(X1),var(X2)]),Sa,liste):-
           apply(var(X1),Sa,liste),
           apply(var(X2),Sa,liste).



apply(term(cons,2,[var(_),var(_)]),_,list).

/* Trifft keiner der Teilfaelle zu wird 'top' gesetzt */
apply(term(_,_,_),_,top).
```

Alle weiteren implementierten Domains (Domain1 bis Domain8) sind nach genau dem gleichen Shema und aus genau den gleichen Prädikaten definiert. Daher ist es zur Definition eigener Domains ratsam einen der zur Verfügung stehenden als Vorlage zu benutzen, welche lediglich erweitert werden muß. Die Definition der einzelnen Prädikate ist , bei genauer Vorstellung über Struktur und Semantik der Elemente des Domains, eine schnell erledigte Aufgabe. Neu definierte Domains können dann mittels des Befhels *getDomain* in das Analysesystem eingebunden und ausgetestet werden.

Anhang D

Implementierte Domains

Man ist bei der Analyse bestrebt möglichst genaue Angaben über die Beschaffenheit von Parametern in möglichst kurzer Zeit zu ermitteln. Die Genauigkeit der Angaben hängt dabei größtenteils von der Mächtigkeit des definierten Domains ab. Je detaillierter der Domain ist, so aussagekräftiger sind auch die Ergebnisse. Allerdings bedingt ein detaillierterer und damit größerer Domain auch eine schlechtere Laufzeit der Analyse. Um hier den richtigen Kompromiß zu finden bedarf es ausgiebiger Tests der verschiedensten Beispielprogramme mit den unterschiedlichsten Domains. Die im Analysesystem implementierten Doamins 1 bis 8 dienen dabei als Grundlage. Je nach analysierten Programmen, aber auch je nach Verwendungszweck der Ergebnisse müssen diese Domains sicherlich angepaßt oder verfeinert werden.

D.1 Erläuterungen zu den Domains

Hier sollen nun die genannten acht Domains aufgezeigt und erläutert werden. Zur Verdeutlichung der Struktur der Domains dient jeweils eine Skizze, die sowohl die Elemente, als auch die Vergleichsrelation zwischen den Elementen beinhaltet. Die einzelnen Domains sind aufeinander aufgebaut, so daß gleiche Elementbezeichner jeweils für die gleichen abstrahierten Mengen stehen. Die einzelnen Domainelemente sollen im folgenden anhand ihrer konkreten Bedeutung kurz vorgestellt werden. Man beachte dabei, daß die Konkretisierungsmenge immer als Teilmenge der *linearen* Terme aus $\mathcal{T}(\mathcal{C}, \mathcal{X})$ zu verstehen ist. Daher müssen alle unten genannten Konkretisierungsmengen noch mit der Menge der linearen Terme aus $\mathcal{T}(\mathcal{C}, \mathcal{X})$ geschnitten werden. Die wird aus Gründen der einfacheren Definition hiermit vorrausgesetzt.

> bot, top: Bezeichnet jeweils das kleinste (bot) bzw. das größte Element eines Domains (top). Dabei entspricht bot der leeren Termmenge und top der Menge aller Demand Pattern. D.h.:
>
> $$\gamma(bot) = \{\ \}$$

$$\gamma(top) = \{\mathcal{T}(\mathcal{C}, \mathcal{X})\}$$

nil: Die Menge, die nur die leere Liste enthält. D.h.:

$$\gamma(nil) = \{\ [\,]\ \}$$

liste: Entspricht der Menge aller vollständig instanziierten Listen. D.h.:

$$\gamma(liste) = \{t \mid t = cons(X, Y), X \in \gamma(e), y \in \gamma(liste)\} \cup \{\ [\,]\ \}$$

e: Steht für die Menge aller Demand Pattern ohne Variablen D.h.:

$$\gamma(e) = \{t \mid \mathcal{V}ar(t) = \{\ \}\}$$

hn: Die Menge aller Demand Pattern in Kopfnormalform. D.h.:

$$\gamma(hn) = \mathcal{T}(\mathcal{C}, \mathcal{X}) \cap \mathcal{X}$$

list: Enthält alle Listenterme, jeden beliebigen Auswertungsgrades. D.h.:

$$\gamma(list) = \{t \mid t = cons(X, Y); X, Y \in \gamma(top)\} \cup \{\ [\,]\ \}$$

listf: Enthält alle endlichen Listen. D.h.:

$$\gamma(listf) = \{t \mid t = cons(X, Y); X \in \gamma(top), Y \in \gamma(listf)\} \cup \{\ [\,]\ \}$$

headhn: Steht für alle Listen, deren erstes Element in Kopfnormalform ist. D.h.:

$$\gamma(headhn) = \{t \mid t = cons(X, Y), X \in \gamma(hn)\} \cup \{\ [\,]\ \}$$

listhn: Entspricht der Menge aller endlichen Listen, deren erstes Element in Kopfnormalform ist. D.h.:

$$\gamma(listhn) = \{t \mid t = cons(X, Y), X \in \gamma(hn), Y \in \gamma(listf)\} \cup \{\ [\,]\ \}$$

heade: Bezeichnet alle Listen, deren Kopf vollständig ausgewertet ist. D.h.:

$$\gamma(heade) = \{t \mid t = cons(X, Y), X \in \gamma(e)\} \cup \{\ [\,]\ \}$$

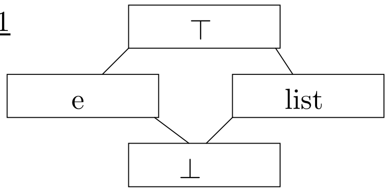
listne: Schließlich steht für die Menge aller endlichen vollständig ausgewerteten Listen, die nicht leer sind. D.h.:

$$\gamma(listne) = \gamma(liste) - \{\ [\,]\ \}$$

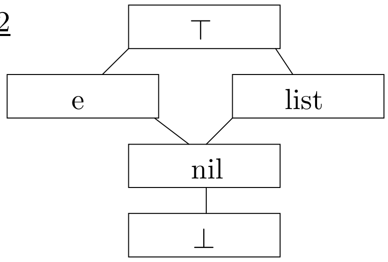
Dies sind alle Element, die in den folgenden acht Domains Verwendung finden. Die definierten Domains entstanden durch Anregung von [RAMAKXXXX] und wurden teilweise erweitert aufgrund der in Kapitel 7 gewonnenen Erkenntnisse.

D.2 Die Beispieldomains

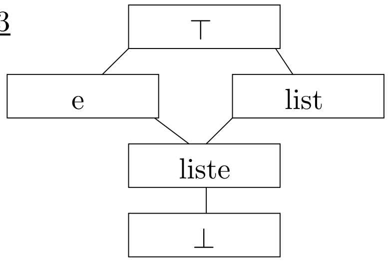
Domain 1



Domain 2



Domain 3



Domain 4

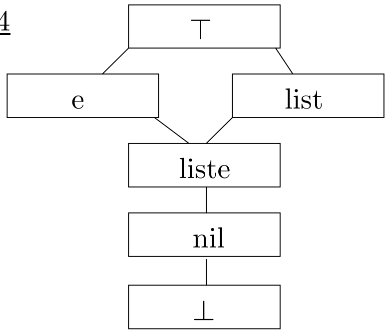
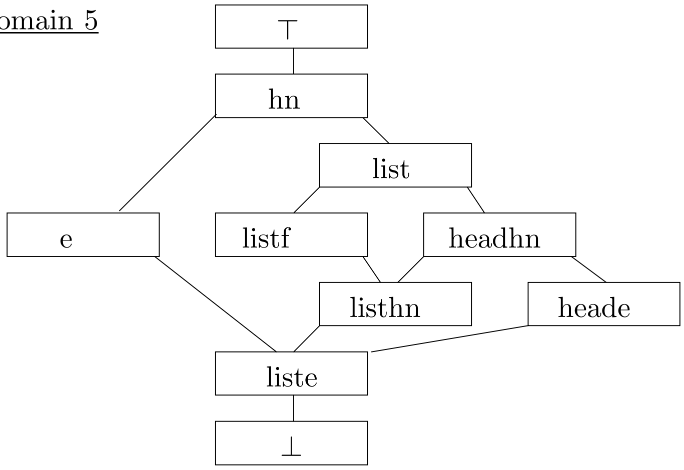
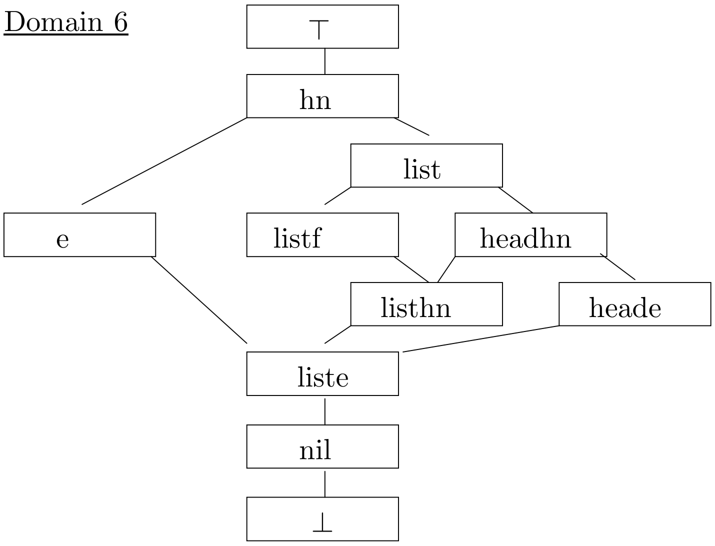


Abbildung D.1: Die vordefinierten Domains 1 - 4

Domain 5



Domain 6



Domain 7

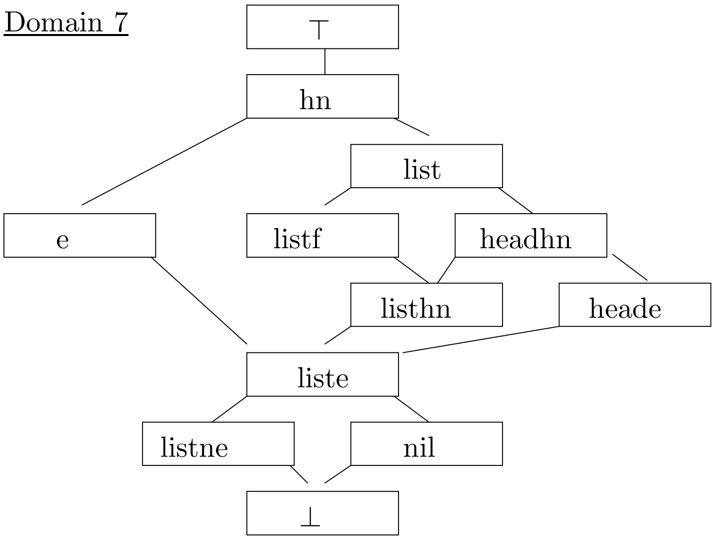


Abbildung D.2: Die vordefinierten Domains 5 - 7

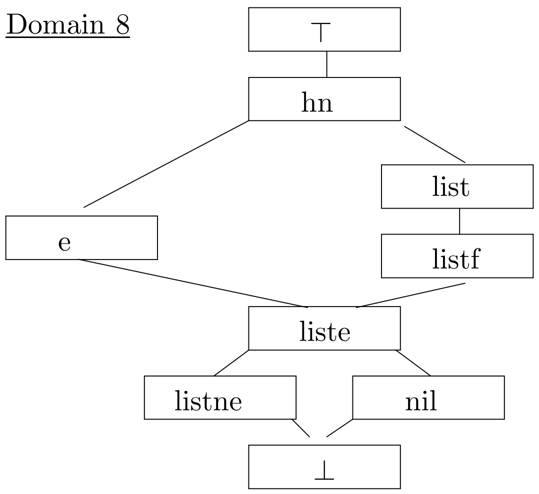


Abbildung D.3: Der vordefinierte Domain 8

Anhang E

Der Benchmark

Die im Kapitel 7 vorgestellten Ergebnisse basieren alle auf dem gleichen, aus 4 Programmen bestehenden, Benchmark. Wie aus den Ausführungen im oben genannten Kapitel zu sehen war brachte eine Abwandlung des Benchmarks, mit Vermeidung von if-then-else Konstruktionen, eine erhebliche Verbesserung der Ergebnisse mit sich. Daher sollen im Folgenden die einzelnen Funktionen der Benchmarks erläutert werden und soll auch ein Augenmerk darauf gelegt werden, welche Funktionen wie von obiger Änderung betroffen waren.

Die Benchmarks bestehen wie gesagt aus 4 verschiedenen Funktionen. Dies sind eine Lösung des 8 Damenproblems (*damen8/0*), ein Quicksortalgorithmus (*qs/1*), eine Matrizenmultiplikation (*matmult/2*) sowie das Primzahlverfahren von Eratosthenes (*primzahlen/1*). Alle Funktionen wurden zunächst straightforward implementiert ohne an Effizienzaspekte zu denken. Dabei ergaben sich folgende Algorithmen.

Die Funktion *damen8/0* entspricht einem 'generate and test' Ansatz. Die Liste der Zahlen von 1 bis 8 wird permutiert und überprüft, ob die berechnete Permutation eine Lösung des Problems ist. Alle Lösungen werden wieder in einer Listenstruktur abgespeichert und zurückgegeben. Das Sieb des Eratosthenes generiert durch die Funktion *primzahlen(N)* eine Liste von natürlichen Zahlen aus denen alle nicht Primzahlen sukkzessive ausgestrichen werden. Von dieser berechneten Liste werden die ersten N Zahlen abgetrennt. *matmult(A,B)* multipliziert die beiden als Liste über Listen realisierten Matrizen A und B. Dazu wird das Matrixprodukt auf eine Reihe von Vektormultiplikationen und diese wiederherum auf eine Reihe von skalaren Multiplikationen reduziert. Die Implementierung der Matrixmultiplikation kommt als einzige der vier Funktionen ohne die Benutzung eines if-then-else aus.

Aufgrund der in Kapitel 7 dargelegten Probleme mit der Analyse der Funktion *ite/3*, also der Umsetzung eines if-then-else Konstruktes, wurden diese Funktionen dahingehend abgewandelt, daß möglichst auf die Benutzung des *ite* verzichtet wurde. Dazu wurden 6 von 8 *ite*-Aufrufen durch die Verwendung eines if-then Konstruktes (*it*) eingespart. Neben der Funktion *matmult* war damit auch die Funktion *qs* frei von *ite*-Aufrufen, während die anderen beiden Funktionen auch weiterhin einmal von diesem Konstrukt Gebrauch machten. Der Einfluß dieser Änderung ist ausführlich in Kapitel 7 beschrieben.

Das Analysesystem gibt die Ergebnisse der Anfrage und eine Statistische Aussage über die restlichen Ergebnisse auf dem Bildschirm aus. Das vollständige Analyseergebnis wird in eine Datei mit dem Namen der Definitionsdatei und der Endung `*.abs` geschrieben. Diese dient als Eingabedatei für einen zu entwickelnden Compiler, welcher die Analyseergebnisse nutzt. Andererseits hat man durch diese Datei aber auch die Möglichkeit die genauen Resultate mit Hilfe eines Editors anzusehen. Damit man die einzelnen Funktionen einer der vier beschriebenen Benchmarkfunktionen zuordnen kann soll nun kurz, ohne Kommentare, der orginale Benchmark (mit *ite*-Aufrufen) vorgestellt werden. Zunächst also die Lösung zum 8-Damenproblem.

E.1 Das 8-Damenproblem

```
damen8 = filmap(perm([1,2,3,4,5,6,7,8])).

perm([]) = [[]].
perm([H | T]) = mapit(H,perm(T)).


insert(A,L,[]) = [append(L,[A])].
insert(A,L,[H | T]) = [append(L, [A | [H | T]])
                     | insert(A,append(L,[H]),T)].


mapit(A,[]) = [].
mapit(A,[H | T]) = append(insert(A,[],H),mapit(A,T)).

filmap([]) = [].
filmap([H | T]) = ite(filter2(H),[H | filmap(T)],filmap(T)).

filter2([]) = true.
filter2([H | T]) = ite(safe(H,T,1), filter2(T),false).

safe(X,[],N) = true.
safe(X,[H | T],N) = ite(dif(X,H)==N,false,safe(X,T,N+1)).

dif(X,H) = ite(X<=H, H-X,X-H).
```

Dabei berechnet *perm(L)* die Liste der Permutationen von L mittels der Funktionen *insert/2* und *mapit/2*. *filter2(L)* liefert `true`, falls L eine Lösung des Prioblems ist, sonst `false`. Schließlich filtert die Funktion *filmap(L)* alle Lösungen in der Liste L aus und liefert eine Liste der gültigen Lösungen als Rückgabewert. Dabei prüft das Prädikat *safe/3*, ob eine gesetzte Dame bedroht wird und das Prädikat *dif/2* berechnet lediglich den Betrag der Differenz der beiden Parameter. Mit dieser kurzen Erläuterung sollte das obige Listing verständlich sein.Weit weniger aufwendig sind die drei weiteren Benchmarkfunktionen.

E.2 Das Sieb des Eratosthenes

primzahlen(N) liefert eine Liste der ersten N Primzahlen größer als 1. Dazu generiert *from(2)* eine Liste aller Zahlen grösser als 1. Daraus filtert *pfilter/1* alle Zahlen, welche Vielfache eines Vorgängers in der Liste sind, raus. Dazu wird für jedes Listenelement die Funktion *fil/2* aufgerufen , um die Vielfachen der übergebenen Zahl zu eliminieren. Die Hilfsfunktion *take/2* nimmt ein festes Anfangsstück der übergebenen Liste und liefert es zurück, während *teiler/2* lediglich prüft, ob zwei Zahlen ohne Rest teilbar sind.

```
from(N) = [N | from(N+1)].

fil(X,[Y dim Ys]) = ite(teiler(X,Y),fil(X,Ys), [Y | fil(X,Ys)]).

pfilter([X | Xs]) = [X | pfilter(fil(X,Xs))].

primzahlen(N) = take(N,pfilter(from(2))).

take(N,[H | T]) = ite(0<N, [H | take(N-1,T)], [ ]).

teiler(X,Y) = (Y mod X) == 0.
```

E.3 Matrixmultiplikation

Die folgende Implementierung der Matrixmultiplikation sollte nach obigen Ausführungen leich zu verstehen sein.

```
matmult([],Y) = [].
matmult([X | Xs], Y) = [rowmult(X,Y) | matmult(Xs,Y) ].

rowmult(X,[]) = [].
rowmult(X,[Y | Ys]) = [dotmult(X,Y) | rowmult(X,Ys) ].

dotmult([],[]) = 0.
dotmult([X | Xs],[Y | Ys]) = X*Y + dotmult(Xs,Ys).
```

E.4 Der Quicksortalgorithmus

Als letztes folgt noch die Implementierung der Quicksortfunktion *qs/1.* Hier wird die übergebene Liste mittels *unten/2* und *oben/2* in eine Liste mit Zahlen kleiner dem ersten Element und eine Liste mit Zahlen nicht kleiner dem ersten Element aufgeteilt. Diese beiden Teillisten werden sortiert und mittels *append/2* zur Ergebnisliste zusammengefügt.

```
qs([]) = [].
qs([X | Xs]) = append(qs(unten(Xs,X)), [X | qs(oben(Xs,X))]).

append([],L) =L.
append([H | T],L) = [H | append(T,L)].
```

```
unten([],X) = [].
unten([H | T],X) = ite(H<X, [H | unten(T,X)], unten(T,X)).

oben([],X) = [].
oben([H | T],X) = ite(H>X, [H | oben(T,X)], oben(T,X)).
```

Die angesprochene Abwandlung dieser Funktionen, welche versucht *ite*-Aufrufe zu vermeiden läßt sich nun mit Hilfe der Ausführungen in Kapitel 7 leicht herleiten. Zur Verwendung im Analysesystem wurden die definierenden Bäume der beiden Benchmarks in den Dateien `bench.def` und `bench2.def` abgelegt. So können diese durch den Shellbefehl *getFile* jederzeit gewechselt werden. Natürlich ist es mit diesem Befehl auch möglich die Dateien eigener Programme zu laden.

Literaturverzeichnis

[AEH94] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Principles of Programming Languages*, pages 268–279. ACM, 1994.

[Bru91] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programms. In *Journal of Logic Programming*, pages 91–214, 1991.

[CH94] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for prolog. *ACM Trans. Prog. Lang. and Sys.*, 16(1):35–101, 1994.

[Han94] M. Hanus. The integration of functions into logic programming: a survey. Technical Report MPI-I-94-201, Max-Plank-Institut Fuer Informatik, 1994. J. Func. Prog. to appear.

[Han97] Michael Hanus. A unified computation model for functional and logic programming. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 80–93, Paris, France, 15–17 January 1997.

[Han98] Michael Hanus. *Curry: An Integrated Functional Logic Language*, Feb. 1998.

[HKMN95] Michael Hanus, Herbert Kuchen, and Juan Jos e Moreno-Navarro. Curry: A truly functional logic language. In *Workshop on Visions for the Future of Logic Programming*, pages 95–107. ILPS, 1995.

[KM92] K.Muthukumar and M.Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2-3):315–347, 1992.

[MNKMC+93] J. J. Moreno-Navarro, H. Kuchen, J. Marino-Carballo, S. Winkler, and W. Hans. Efficient lazy narrowing using demandedness analysis. *Lecture Notes in Computer Science*, 714:167–??, 1993.

[Sad97] Ramin Sadre. Nebenlaeufige implementierung der programmiersprache curry in java. Master's thesis, RWTH Aachen, 1997.

[SR96] R. Sekar and I. V. Ramakrishnan. Fast strictness analysis based on demand propagation. *ACM Trans. Prog. Lang. and Sys.*, 17(6):896–937, November 1996.

[Zar97] F. Zartmann. Denotational abstract interpretation of functional logic programs. *Lecture Notes in Computer Science*, 1302:141–??, 1997.