Eingekapselte Suchverfahren für logisch-funktionale Programmiersprachen

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

ausgearbeitet von

Frank Steiner
Matrikelnummer 191708

Betreuer: Prof. Dr. Michael Hanus

Aachen, Oktober 1997

Ich bedanke mich bei Professor Hanus für die Betreuung, die zahlreichen Vorschläge und Anregungen und die geduldige Beantwortung meiner vielen Fragen, bei Herbert Kuchen und Frank Huch für Diskussionsbeiträge und Unterlagen zum Thema der Extravariablen, bei Markus Mohnen für TEXnische Hilfeleistung, bei meinen Eltern für das Korrekturlesen der Arbeit und für ihre Unterstützung während meines gesamten Studiums und bei meiner Freundin für ihre Geduld und ihr Verständnis während des Schreibens dieser Diplomarbeit.

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

Aachen, den 8. Oktober 1997

(Frank Steiner)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die deklarative Programmierung wird klassischerweise in zwei Hauptgebiete unterteilt: die funktionale und die logische Programmierung. Um die Vorteile beider Ansätze zu vereinigen, wird seit vielen Jahren die Entwicklung sogenannter funktional-logischer Sprachen betrieben [8].

Funktional-logische Sprachen ermöglichen den Umgang mit Funktionen höherer Ordnung, partiellen Funktionsapplikationen, geschachtelten Ausdrücken und unendlichen Datenstrukturen nach Art der funktionalen Sprachen und gestatten dadurch auf einfache Weise die Angabe von übersichtlichen und leicht verständlichen Programmen. Die Verwendung unsauberer Sprachelemente aus Prolog, beispielsweise der *Cut* zur Beschneidung des Suchraums oder das `call`-Prädikat zur Simulation von Funktionen höhererer Ordnung, kann auf diese Weise vermieden werden. Außerdem ermöglicht die deterministische Reduktion funktionaler Ausdrücke nach dem Prinzip der verzögerten Auswertung effizientere Berechnungen als in rein logischen Sprachen. Die Erweiterung der Logikprogrammierung um (deterministische) Funktionen ist aber nicht nur aus Effizienzgründen sinnvoll, sondern auch für die Anbindung externer Funktionen, beispielsweise zum Zugriff auf das Betriebssystem oder zur Integration in eine graphische Benutzeroberfläche.

Aus den logischen Programmiersprachen werden die Möglichkeiten zum Umgang mit partiellen Strukturen und zur Durchführung einer nichtdeterministischen Suche übernommen, die beispielsweise in Datenbankanwendungen oder bei Optimierungsproblemen vorteilhaft sind, von funktionalen Sprachen aber nicht zur Verfügung gestellt werden. Zwar besitzen auch funktionale Sprachen Suchmechanismen durch Verwendung von List-Comprehensions [34], aber dadurch kann beispielsweise kein Ausdruck der Art `s(s(X))`, mit `X` eine logische Variable, berechnet werden, der die Menge aller Peano-Zahlen größer oder gleich 2 repräsentiert. Eine funktionale Sprache müßte vielmehr eine unendliche Liste `[s(s(0)),s(s(s(0))),...]` erstellen, um eine solche Lösungsmenge darzustellen.

Mit Hilfe funktional-logischer Sprachen können also sowohl die Vorteile aus logischen wie aus funktionalen Programmiersprachen genutzt werden. Hauptsächlich zwei Ansätze wurden in der Vergangenheit zur Realisierung einer entsprechenden operationalen Semantik verfolgt: Residuation und Narrowing.

Residuation ist beispielsweise die Grundlage der funktional-logischen Sprachen Escher [21] und Oz [33]. Funktionsaufrufe werden so lange verzögert, bis alle freien Variablen in den Parametern an Terme gebunden sind, deren Gestalt eine deterministische Auswertung der Funktion gestattet. Auf diese Weise können nebenläufige Berechnungen über Variablenbindungen synchronisiert werden, nichtdeterministische Suche muß aber explizit durch Verwendung von Prädikaten oder wie in Oz durch Disjunktionen ermöglicht werden. Residuation ist jedoch keine vollständige Auswertungsstrategie, da Lösungen nicht berechnet werden können, wenn Variablen in den Parametern eines Funktionsaufrufs nicht entsprechend gebunden werden.

Beispielsweise kann für eine Funktion der Form

```
p(a) = 1.
p(b) = 2.
```

der Funktionsaufruf `p(X)` nicht ausgewertet werden, wenn `X` nicht an `a` oder `b` gebunden wird.

Möglich wird dies durch Verwendung von Narrowing, das die Reduktionsprinzipien funktionaler Sprachen um die Möglichkeit der Unifikation für die Parameterübergabe erweitert. Bei der Auswertung von `p(X)` durch Narrowing werden die verschiedenen Bindungen `X=a` und `X=b` erstellt und entsprechend die beiden Ergebnisse 1 und 2 berechnet. Narrowing wird beispielsweise in der funktional-logischen Sprache Babel [20] verwendet und ist im Gegensatz zu Residuation eine vollständige Strategie, d.h. Ergebnisse eines funktionalen Programms bzw. Lösungen eines logischen Programms werden berechnet, wenn sie existieren. Eine Übersicht über verschiedene Narrowingstrategien kann in [8] gefunden werden.

1995 haben Hanus, Kuchen und Moreno-Navarro mit einem ersten Entwurf der Programmiersprache Curry [13] den Versuch unternommen, einen Standard im Bereich der funktional-logischen Programmierung zu schaffen. Inzwischen ist die Entwicklung von Curry zu einem größeren Projekt geworden, an dem Mitarbeiter verschiedener Universitäten in Deutschland, Spanien und den USA beteiligt sind. Das Sprachdesign hat sich seit dem ersten Entwurf in mancher Hinsicht verändert, die grundlegenden Ideen sind jedoch dieselben geblieben. Eine aktuelle Beschreibung von Curry kann in [16] nachgelesen werden.

Curry vereint die Möglichkeiten der funktionalen, logischen und nebenläufigen Programmierung durch Verwendung eines Berechnungsmodells, das die Prinzipien von Residuation und Narrowing kombiniert [11]. Standardmäßig werden boolesche Funktionen durch Narrowing und alle anderen durch Residuation ausgewertet, der Benutzer kann dieses Verhalten jedoch nach Belieben verändern.

Curry stellt alle erwähnten Elemente aus funktionalen und logischen Sprachen zur Verfügung: logische Variablen, partielle Strukturen und nichtdeterministische Suche durch Narrowing werden kombiniert mit Funktionen höherer Ordnung, geschachtelten Ausdrücken, dem Prinzip der verzögerten Auswertung und unendlichen Datenstrukturen. Ein polymorphes Typsystem, wie es in vielen funktionalen Sprachen verwendet wird, ist in Curry ebenso vorhanden wie ein Modulsystem. Für die Ein-/Ausgabe wurde der monadische Ansatz gewählt [29], der beispielsweise in der funktionalen Sprache Haskell [18] realisiert wird. Dabei muß das Hauptprogramm aus einer Folge von monadischen Ein-/Ausgabeoperationen bestehen, die als Transformationen auf einer Außenwelt betrachtet werden. Da diese Außenwelt beispielsweise das gesamte Dateisystem umfaßt, darf sie niemals vervielfältig werden.

Bei Auftreten nichtdeterministischer Reduktionen werden in Curry automatisch alle möglichen Alternativen unabhängig voneinander in verschiedenen globalen Berechnungsräumen untersucht. Dieses Vorgehen hat vor allem zwei Nachteile:

- Ein Vergleich zweier Lösungen oder die explizite Angabe einer Suchstrategie ist nicht möglich,
- nichtdeterministische Auswertungen können nicht im Zusammenhang mit Ein-/Ausgabeoperationen verwendet werden, da das Erstellen mehrerer globaler Berechnungsräume einer Vervielfältigung der Außenwelt entspricht.

Um diese beiden Nachteile auszuräumen, müssen nichtdeterministische Berechnungen eingekapselt werden. In der Sprache Oz wird dies durch Verwendung eines Suchoperators erreicht, der eine Suche in einem lokalen Raum durchführt und nach Auftreten eines nichtdeterministischen Schritts die verschiedenen Alternativen als Datenstrukturen zurückliefert [32]. Auf diese

Weise wird eine Vervielfältigung des globalen Berechnungsraums und damit der Außenwelt verhindert. Zudem kann der Benutzer darüber entscheiden, ob und in welcher Reihenfolge die Alternativen weiteruntersucht werden sollen. Suchstrategien müssen nicht im Kern der Sprache verankert werden, sondern können durch Verwendung des Suchoperators als Funktionen definiert werden, denen man eine Anfrage übergibt, und die die berechneten Lösungen als Datenstrukturen zurückliefern.

In Curry sind zwei Mechanismen zur Realisierung einer eingekapselten Suche vorgesehen: In Anlehnung an Oz gibt es einen Suchoperator, dem man Anfragen in Form von Lambda-Abstraktionen übergeben kann und der entweder eine Lösung berechnet, wenn dies auf deterministische Weise möglich ist, oder im Fall nichtdeterministischer Reduktionen die verschiedenen Suchrichtungen wiederum als Lambda-Abstraktionen zurückliefert. Ähnlich wie in Oz können dann Suchalgorithmen einfach als Funktionen definiert werden.

Eine zweite Form der Einkapselung wird durch ein Committed-Choice-Konstrukt zur Verfügung gestellt, das beispielsweise auch in der Sprache AKL [19] verwendet wird. Es erlaubt die Angabe alternativer Reduktionswege, die sich jeweils aus einer Bedingung und einem Rumpf zusammensetzen. Sämtliche Bedingungen werden geprüft, bis eine erfüllbare gefunden wird. Die Suche wird dann nur in einer Richtung weitergeführt, indem der Rumpf der erfüllten Bedingung ausgewertet wird und alle anderen Reduktionswege verworfen werden, selbst wenn noch weitere erfüllbare Bedingungen existieren sollten. Die Prüfung der verschiedenen Bedingungen muß unabhängig voneinander erfolgen, soll aber nicht zu einer Vervielfältigung des globalen Berechnungsraums führen, da ja letztendlich immer nur eine Richtung weiter untersucht wird. Daher wird die Auswertung der Bedingungen in unabhängige lokale Räume eingekapselt, die alle innerhalb eines globalen Raums verwaltet werden.

In dieser Arbeit werden wir die operationale Semantik von Curry um die Definition der eingekapselten Suche erweitern. Dazu geben wir zunächst in Kapitel 2 einen Überblick über die Sprache Curry und ihre operationale Semantik.

In Kapitel 3 fassen wir die Anforderungen an die eingekapselte Suche zusammen und ziehen einen kurzen Vergleich mit den Suchmechanismen in Prolog und in funktionalen Sprachen. Nach Vorstellung des Konzepts der Einkapselung durch lokale Räume werden die Form und die Arbeitsweise des Suchoperators und des Committed-Choice-Konstrukts definiert und anhand einiger Beispiele erläutert. Dabei werden wir sehen, daß durch Verwendung lokaler Räume eine Unterscheidung zwischen lokalen und globalen Variablen erforderlich ist.

Die dazu notwendigen Änderungen werden in Kapitel 4 vorgenommen. Neben der Einführung eines Existenzquantors in Curry muß auch die in Kapitel 2 definierte operationale Semantik um einen Mechanismus zur Protokollierung aller Variablen, die in einem Berechnungsraums deklariert sind, erweitert werden.

In Kapitel 5 geben wir eine operationale Semantik lokaler Räume an und betrachten zwei spezielle Formen: den Suchraum und den Choice-Raum. Mit ihrer Hilfe können wir operationale Semantiken für den Suchoperator und die Committed-Choice definieren, die die in Kapitel 3 erläuterten Ideen und Konzepte verwirklichen.

Möglichkeiten zur Verwendung der eingekapselten Suche werden in Kapitel 6 vorgestellt. Zunächst programmieren wir einige Suchstrategien und erklären ihre Arbeitsweisen ausführlich anhand eines Beispiel. Danach zeigen wir, wie das `findall`-Prädikat, mit dem in Prolog alle Lösungen einer Anfrage in einer Liste gesammelt werden können, in Curry simuliert wird, und betrachten anschließend Konzepte, um den Fehlschlag einer Berechnung abzufangen und nichtdeterministische Reduktionen einzukapseln, beispielsweise für die Verwendung in Programmen, die Ein-/Ausgabe durchführen. Die praktische Anwendung dieser Konzepte demonstrieren wir anhand zweier interaktiver Programme, in denen die Suche durch Eingaben des Benutzers gesteuert wird. Abschließend erläutern wir, wieso durch Verwendung der eingekapselten Suche die Simulation von List-Comprehensions, die in funktionalen Sprachen verwendet werden können, nicht möglich ist.

Die Umsetzung der Theorie aus den Kapiteln 3, 4 und 5 haben wir in TasteCurry vorgenommen, einer ersten Prototyp-Implementierung von Curry, die in Prolog programmiert wurde. Bei ihrer Erweiterung um die eingekapselte Suche sind wir aus Effizienzgründen in einigen Punkten von der vorgestellten Theorie abgewichen. Da für andere Implementierungen diese Abweichungen ebenfalls interessant sein könnten, stellen wir die wichtigsten in Kapitel 7 vor.

Abschließend geben wir in Kapitel 8 einen kurzen Ausblick auf mögliche Erweiterungen der eingekapselten Suche und erläutern Ideen zur Realisierung des vorgestellten Konzepts in einer Implementierung von Curry, die momentan in der Sprache Java entwickelt wird [15].

Die Benutzung der auf der beiliegenden Diskette enthaltenen TasteCurry-Version wird im Anhang erklärt, in dem auch die kontextfreie Syntax der Sprache Curry aufgeführt ist.

Kapitel 2

Die Sprache Curry

Curry [16] ist eine deklarative Programmiersprache, die die Vorzüge funktionaler und logischer Sprachen vereint und auf diese Weise versucht, Programmierern aus beiden Gebieten eine gemeinsame Entwicklungsplattform zur Verfügung zu stellen. Sie wurde auch in der Lehre bereits erfolgreich eingesetzt, um sowohl funktionale als auch logische Programmiertechniken sowie deren Kombination anhand einer einzigen Sprache zu unterrichten [12].

Curry ist eine konstruktorbasierte Sprache, d.h. sie unterscheidet zwischen Datenkonstruktoren und definierten Funktionen, die auf diesen Datenstrukturen operieren. Ein Curry-Programm besteht aus Datentypdeklarationen, die die Menge der Konstruktoren bestimmen, und definierenden Gleichungen (oder Regeln), die die Operationen der definierten Funktionen auf der Menge der Konstruktorterme festlegen. Bei der Ausführung eines Programms versuchen wir, eine Anfrage, d.h. einen beliebigen Ausdruck, mit Hilfe der Regeln in einen Konstruktorterm zu überführen, der keine definierten Funktionen mehr enthält. Dazu stehen in Curry *beide* der meistverwendeten Reduktionsstrategien funktional-logischer Sprachen zur Verfügung: Residuation und Narrowing.

In diesem Kapitel werden wir zunächst die grundlegenden verwendeten Begriffe definieren, uns dann mit den wichtigsten Sprachelementen von Curry beschäftigen und schließlich die operationale Semantik von Curry vorstellen.

2.1 Grundlegende Begriffe

Wir führen die Definitionen einiger Begriffe aus der Termersetzung auf, die im folgenden zum Verständnis der Syntax und operationalen Semantik von Curry notwendig sind. Wir halten uns dabei an die Notationen aus [6]. Eine umfassende Einführung in die Termersetzung kann in [4] nachgelesen werden.

Definition 2.1 (konstruktorbasierte Signatur)
Eine Signatur Σ ist eine Menge von Operationssymbolen, denen eine Stelligkeit zugeordnet ist. Wir schreiben $\varphi \in \Sigma^{(n)}$, wenn φ ein n-stelliges Operationssymbol ist. Σ heißt konstruktorbasiert, wenn $\Sigma = \mathcal{F} \mathbin{\dot{\cup}} \mathcal{C}$ gilt und $\mathcal{C}$ eine abzählbar unendliche Menge von *(Daten-) Konstruktorsymbolen* und $\mathcal{F}$ eine abzählbar undendliche Menge von *definierten Funktionssymbolen* bezeichnen. $\square$

Wir beschränken uns hier der Einfachheit halber auf den einsortigen Fall, obwohl Curry mehrsortige Programme erlaubt, wie wir in Abschnitt 2.2 sehen werden. Dies ist jedoch für die Erklärung der grundlegenden Strukturen, auf die wir zur Erklärung der operationalen Semantik in Abschnitt 2.3 aufbauen werden, nicht von Bedeutung. Im folgenden bezeichnen wir mit f immer ein Funktionssymbol und mit c ein Konstruktorsymbol.

Definition 2.2 (Terme)
Sei Σ eine konstruktorbasierte Signatur[1], $\mathcal{X}$ eine abzählbar unendliche Menge von Variablen mit $\mathcal{X} \cap \Sigma = \emptyset$.

1. Die Menge der *Terme* oder *Ausdrücke* wird mit $\mathcal{T}(\Sigma, \mathcal{X})$ bezeichnet und ist definiert als die kleinste Menge T, für die gilt:
   (a) $\mathcal{X} \subseteq T$
   (b) $\varphi(e_1, \ldots, e_n) \in T$, wenn $\varphi \in \Sigma^{(n)}$ und $e_i \in T$ für $i = 1, \ldots, n$.
2. Die Menge der *Konstruktorterme* oder *Datenterme* wird mit $\mathcal{T}(\mathcal{C}, \mathcal{X})$ bezeichnet und ist definiert als die kleinste Menge T, für die gilt:
   (a) $\mathcal{X} \subseteq T$
   (b) $\varphi(t_1, \ldots, t_n) \in T$, wenn $\varphi \in \mathcal{C}^{(n)}$ und $t_i \in T$ für $i = 1, \ldots, n$
3. Ein Term $\varphi(e_1, \ldots, e_k)$ heißt *partielle Applikation*, wenn $\varphi \in \Sigma^{(n)}$ und $k < n$.
4. Mit $var(t)$ wird die Menge der Variablen bezeichnet, die in einem Term t auftreten. Ein Term (Konstruktorterm) t heißt *Grundterm* (*Konstruktorgrundterm*), wenn $var(t) = \emptyset$. Die Menge dieser Terme wird mit $\mathcal{T}(\Sigma, \emptyset)$ $(\mathcal{T}(\mathcal{C}, \emptyset))$ bezeichnet.
5. Ein Term $c(e_1, \ldots, e_n)$ ist in *Kopfnormalform*, wenn $c \in \mathcal{C}$ und $e_i \in \mathcal{T}(\Sigma, \mathcal{X})$ für $i = 1, \ldots, n$.
6. Ein Term $f(t_1, \ldots, t_n)$ heißt *Muster*, wenn $f \in \mathcal{F}$ und $t_i \in \mathcal{T}(\mathcal{C}, X)$ für $i = 1, \ldots, n$.
7. Ein Term t heißt eine *Variante* eines Terms t', wenn er aus t' durch Variablenumbenennung entsteht.

□

Definition 2.3 (Substitution)

- Eine Substitution ist eine Abbildung $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C}, \mathcal{X})$ mit endlichem Träger $dom(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$. Sie läßt sich durch $\sigma(\varphi(t_1, \ldots, t_n)) := \varphi(\sigma(t_1), \ldots, \sigma(t_n))$ zu einer Abbildung auf der Menge der Terme $\mathcal{T}(\Sigma, \mathcal{X})$ fortsetzen. Wir beschreiben eine Substitution im folgenden durch die Menge $\{x = \sigma(x) \mid x \in dom(\sigma)\}$. Die *leere* oder *identische* Substitution mit $dom(\sigma) = \emptyset$ wird mit id bezeichnet.
- Die Verkettung zweier Substitutionen σ und σ' wird mit $\sigma \circ \sigma'$ bezeichnet.
- Zwei Terme t_1 und t_2 heißen unifizierbar, wenn eine Substitution σ existiert mit $\sigma(t_1) = \sigma(t_2)$

□

Definition 2.4 (Stelle,Teilterm)

- Eine *Stelle* p in einem Term t wird durch ε oder eine Folge von natürlichen Zahlen dargestellt.
- Mit $t|_p$ wird der *Teilterm* von t an der Stelle p bezeichnet, mit $t[s]_p$ der Term, der entsteht, wenn man den Teilterm $t|_p$ durch den Term s ersetzt.

□

Beispiel 2.1
Sei $t = f(x, g(i, x))$ mit f, g zweistellige Funktionssymbole, x eine Variable, i ein nullstelliges Konstruktorsymbol. Dann gilt:

[1] Im folgenden werden wir Σ immer als konstruktorbasierte Signatur der Form $\mathcal{F} \,\dot\cup\, \mathcal{C}$ annehmen.

$t_{|\varepsilon} = t,\ t_{|1} = x,\ t_{|2} = g(i,x),\ t_{|2.1} = i,$
$t[y]_2 = f(x,y)$ ◇

2.2 Programme in Curry

Eine Beschreibung der Syntax von Curry, die aus [16] entnommen wurde, wird in Anhang C gegeben. Sie ist mit einigen Ausnahmen an die Syntax der Sprache Haskell [18] angelehnt, erlaubt für Bezeichner aber die Wahl verschiedener Darstellungen. Die Syntax ist noch Gegenstand der Diskussion und kann sich in zukünftigen Spezifikationen verändern. Wir verwenden im folgenden die Syntax aus dem Tastecurry-System, der ersten Prototyp-Implementierung von Curry, die wir in Kapitel 7 vorstellen werden. Sie ist in Prolog implementiert und verlangt eine Prolog-ähnliche Notation der Programme. Insbesondere müssen Variablen mit einem Großbuchstaben oder dem Unterstrich `"_"` beginnen, alle anderen Bezeichner mit einem Kleinbuchstaben. Listen werden wie in Prolog durch `[]` für eine leere Liste und `[X|Xs]` für eine Liste mit erstem Element `X` und Restliste `Xs` bezeichnet. Jede nicht leere Programmzeile muß mit einem Punkt beendet werden. Dies gilt auch für die Anfragen, die der Benutzer stellen kann, wir lassen jedoch in den Beispielreduktionen den Punkt nach einer Anfrage weg. Standardmäßig bezeichnen wir im folgenden Funktionssymbole mit f und g, Konstruktorsymbole mit c und d und Variablen mit beliebigen Großbuchstaben.

2.2.1 Datentyp- und Funktionsdeklarationen

Curry ist eine streng getypte Sprache mit einem Hindley/Milner-ähnlichen polymorphen Typsystem[2] [5] und erlaubt daher, wie schon erwähnt, mehrsortige Programme, so daß Konstruktoren und Funktionen bestimmte Typen besitzen.

Eine Datentypdeklarationen der Form

$$\text{data } t(A_1,\ldots,A_n) \;=\; c_1(\tau_{11},\ldots,\tau_{1n_1})\ ;\ \ldots;\ c_k(\tau_{k1},\ldots,\tau_{kn_k})$$

definiert einen n-stelligen Datentyp $t(A_1,\ldots,A_n)$ durch Angabe der zugehörigen Datenkonstruktoren c_i verschiedener Stelligkeit. Die Parameter der c_i sind Typausdrücke τ_{ij}, die aus den Typvariablen A_i und anderen Konstruktoren aufgebaut sind. Datenterme setzen sich gemäß Definition 2.2 aus Variablen und der Menge der definierten Datenkonstruktoren zusammen.

Beispiel 2.2
In Curry sind unter anderem der boolesche, der Listen- und der Paartyp wie folgt vordefiniert:

```
data bool      = true ; false.
data list(A)   = [] ; [A|list(A)].
data pair(A,B) = (A,B).
```

Die Konstruktoren `|` und `,` werden dabei in Infixnotation verwendet. Rekursive Datentypen sind wie im Beispiel des Listentyps erlaubt. Zusätzlich sind noch der funktionale Datentyp `->` und der Integertyp `int` vordefiniert. ◇

Die Definition einer Funktion erfolgt durch Angabe einer Menge von *definierenden Gleichungen* (oder *Regeln*) der Form

$$f(t_1,\ldots,t_n) \;=\; e$$

[2] Die Erweiterung auf Haskell-ähnliche Typklassen ist für die Zukunft geplant.

für eine n-stellige Funktion f $(n \geq 0)$, wobei die t_i Konstruktorterme und e ein Ausdruck sind und jede Variable in $f(t_1, \ldots, t_n)$ höchstens einmal vorkommen darf. f entspricht einem Funktionssymbol aus Abschnitt 2.1, also sind Ausdrücke entsprechend Definition 2.2 mit Hilfe der im Programm definierten Datenkonstruktoren und Funktionen aufgebaut. Durch eine Typdeklaration der Form

f `::` τ_1 `->` ... `->` τ_n `->` τ

in ge**Curry**ter Schreibweise, werden die Typen der Argumente t_i sowie der Zieltyp der Funktion f festgelegt, der mit dem Typ von e übereinstimmen muß.

Beispiel 2.3
Das folgende Curry-Programm definiert die Addition und Kleiner-Gleich-Relation auf Peano-Zahlen sowie die Konkatenation zweier Listen durch die Funktion `(++)`, die in Infix-Schreibweise angegeben wird. Die Null wird mit `z` bezeichnet, da 0 den Typ `int` hat.

```
data nat =  z ; s(nat).

add :: nat -> nat -> nat.
add(z,N)    = N.
add(s(M),N) = s(add(M,N)).

leq :: nat -> nat -> bool.
leq(z,N)       = true.
leq(s(M),z)    = false.
leq(s(M),s(N)) = leq(M,N).

(++) :: list(A) -> list(A) -> list(A).
[]     ++ Ys = Ys.
[X|Xs] ++ Ys = [X|Xs++Ys].
```

Der Zieltyp von `add` sowie der Argumente von `add` und `leq` ist also `nat`, der Zieltyp von `leq` ist `bool`. Ziel- und Argumenttypen von `(++)` sind `list`. Auf die korrekte Einhaltung der Typvorschriften bei Verwendung von Funktionen und Konstruktoren achtet der Typchecker von Curry, der zudem in der Lage ist, den Typ einer Funktion selbständig zu ermitteln, so daß eine Typangabe optional ist. Wir gehen auf das Typsystem nicht näher ein, da es für unsere Arbeit nicht von Bedeutung ist. $\diamond$

Funktionen können auch durch *bedingte Regeln* der Form

$f(t_1, \ldots, t_n)$ `if` $\{c\}$ `=` e
$f(t_1, \ldots, t_n)$ `if` b `=` e

definiert werden. Die Bedingung ist entweder ein Constraint c (siehe Abschnitt 2.2.3), der in geschweifte Klammern eingefaßt werden muß, oder ein boolescher Ausdruck b. Bevor ein bedingte Regel angewendet werden kann, muß die Bedingung erfolgreich reduziert werden, d.h. zum leeren Constraint oder im Falle einer booleschen Bedingung zu `true`.

Mit Hilfe von `where`-Klauseln kann man ähnlich wie in Haskell lokale Funktionen oder durch Pattern Matching auch Konstanten definieren, die nur in der Bedingung und rechten Seite der Regel sichtbar sind. Die folgende `reverse`-Funktion zum Umkehren einer Liste verwendet die lokal definierte Hilfsfunktion `reverse2` und erreicht dadurch lineare Laufzeit.

Beispiel 2.4

```
reverse(L) = reverse2(L,[])
           where
             reverse2([],L) = L;
             reverse2([X|Xs],L) = reverse2(Xs,[X|L]).
```

◇

Ein Beispiel für die lokale Konstantendefinition durch Pattern Matching zeigt die folgende Implementierung von Quicksort.

Beispiel 2.5
Die Funktionen `>=` und `<` haben in Curry den Zieltyp `bool` und können daher in der Bedingung verwendet werden.

```
split(E,[]) = ([],[]).
split(E,[X|Xs]) if E>=X = ([X|L],R)
                if E<X  = (L,[X|R])
                        where (L,R) = split(E,Xs).

quicksort([])     = [].
quicksort([X|Xs]) = quicksort(L) ++ [X|quicksort(R)]
                  where (L,R) = split(X,Xs).
```

Die Funktion `split` teilt eine Liste in zwei Teillisten auf: Die erste enthält alle Elemente, die kleiner gleich dem Pivotelement `E` sind, die zweite Liste die restlichen Elemente. Durch `(L,R) = split(...)` wird die erste Liste an `L` gebunden, die zweite an `R`. `quicksort` sortiert beide Teillisten und fügt sie anschließend durch die Funktion `(++)` wieder zu einer Liste zusammen. ◇

Nach den bisherigen Erklärungen können wir nun allgemein den Begriff eines *Curry-Programms* definieren.

Definition 2.5 (Curry-Programm)
Ein *Curry-Programm* besteht aus einer Menge von Datentypdeklarationen und einer Menge von (bedingten) Regeln, deren linke Seiten Muster sind, wobei in jedem Muster keine Variable mehr als einmal vorkommen darf. □

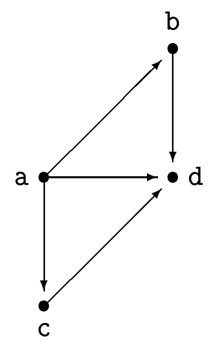
2.2.2 Nichtdeterministische Funktionen

Beispiel 2.6
Betrachten wir das folgende Programm zur Wegesuche im abgebildeten Graph.

```
data knoten = a;b;c;d.

kante(a)=b.
kante(a)=d.
kante(a)=c.
kante(b)=c.
kante(c)=d.

weg(X,Y)  if {X=Y}        = [X]
          if {kante(X)=Z} = [X] ++ weg(Z,Y).
```



◇

An diesem Beispiel können wir sehen, daß Curry anders als z.B. Haskell die Deklaration nichtdeterministischer Funktionen erlaubt und zwar auf dreierlei Art:

Durch überlappende linke Regelseiten: Es gibt drei Regeln mit der linken Seite `kante(a)`, daher kann der Ausdruck `kante(a)` zu `b`, `c` oder `d` reduzieren.

Durch Bedingungen, die sich nicht wechselseitig ausschließen: `weg(a,Y)` erfüllt beide Bedingungen der `weg`-Regel und kann deshalb zu beiden rechten Regelseiten reduzieren.

Durch Extravariablen in Bedingung und rechter Seite: Wir bezeichnen eine Variable in einer Bedingung oder rechten Seite als *Extravariable*, wenn sie nicht in der linken Regelseite auftaucht, so wie die Variable `Z` in der zweiten Bedingung. `weg(a,d)` berechnet aufgrund dieser Extravariable als Endergebnis einen disjunktiven Ausdruck mit drei verschiedene Lösungen:

```
[a,b,c,d] | [a,d] | [a,c,d]
```

Die Reduktion dieses Aufrufs werden wir in Abschnitt 6.2 näher betrachten.

Auf die Auswirkungen solcher Funktionen auf die Vollständigkeit der von Curry verwendeten Reduktionssemantik gehen wir in Abschnitt 2.3.4 ein, mit Extravariablen beschäftigen wir uns in Kapitel 4 ausführlicher. Man beachte, daß in lokalen Deklarationen, die innerhalb einer Regel durch `where` vorgenommen werden, grundsätzlich nur die Variablen der linken Regelseite sichtbar sind, nicht aber die Extravariablen.

Die Behandlung nichtdeterministischer Funktionen stellt für Curry kein Problem dar, aber wenn man sie trotzdem verhindern will, kann man folgende Einschränkungen vornehmen, die beispielsweise in der funktional-logischen Sprache Babel gelten [20]:

1. Für jede Regel l `if {`c`} =` r muß $var(r) \subseteq var(l)$ gelten. Extravariablen sind also in der Bedingung, nicht aber in der rechten Regelseite erlaubt.

2. Die *schwache Eindeutigkeitsbedingung* muß erfüllt sein, d.h. für jedes Paar von (bedingten) Regeln l_1 `if` $\{c_1\}$ `=` r_1 und l_2 `if` $\{c_2\}$ `=` r_2 gilt: Wenn eine Substitution σ existiert, so daß $\sigma(l_1) = \sigma(l_2)$, dann gilt einer der beiden folgenden Bedingungen:

   (a) $\sigma(r_1) = \sigma(r_2)$

   (b) $\sigma(c_1)$ und $\sigma(c_2)$ schließen sich gegenseitig aus, d.h. ihre Konjunktion ist unerfüllbar.

2.2.3 Strikte Gleichheit

Curry verwendet die *strikte Gleichheit*, d.h. zwei Ausdrücke sind gleich, wenn sie sich zum selben Konstruktorgrundterm reduzieren lassen, wobei möglicherweise Variablenbindungen vorgenommen werden. Ob zwei Funktionen gleich sind, wird also durch die Funktionswerte bestimmt, die sie berechnen. Daher sind Vergleiche zwischen nichtterminierenden Funktionen nur bedingt möglich, wie das folgende Beispiel zeigt

Beispiel 2.7

```
list1        = [1|list1].
anotherlist1 = [1|anotherlist1].
list2        = [2|list2].

f = f.
g = g.
```

Der Vergleich `list1=anotherlist1` ist unentscheidbar, da beide Funktionen zwar dieselbe Liste erzeugen, aber nicht terminieren. Der Vergleich der berechneten Listen wird daher unendlich fortgesetzt. `f` und `g` berechnen gar keinen Konstruktorterm als Ergebnis und daher kann auch nie entschieden werden, ob `f=g` gilt oder nicht. Der Vergleich `list1=list2` schlägt hingegen fehl, nachdem beide Seiten einen Reduktionsschritt ausgeführt haben, denn aufgrund der verzögerten Auswertung von Curry wird sofort entschieden, daß die Terme `[1|list1]` und `[2|list2]` nicht gleich sein können, auch wenn sie noch nicht vollständig berechnet wurden. ◇

Curry verwendet zwei verschiedene Formen der strikten Gleichheit. Die Testgleichheit `==` hat den Typ `A -> A -> bool` und wird in funktionalen Sprachen wie Haskell oder Miranda verwendet. Sie arbeitet nach den folgenden Vorschriften (wobei `==` als Infix-Operator notiert ist und `&&` die Konjunktion boolescher Funktionen bezeichnet, die sequentiell von links nach rechts ausgewertet wird):

$$\begin{array}{llllll}
c & \texttt{==} & c & \rightarrow & \texttt{true} & \forall c \in \mathcal{C}^{(0)} \\
c(X_1,\ldots,X_n) & \texttt{==} & c(Y_1,\ldots,Y_n) & \rightarrow & X_1\texttt{==}Y_1 \;\texttt{\&\&}\ldots\texttt{\&\&}\; Y_n\texttt{==}Y_n & \forall c \in \mathcal{C}^{(n)},\ n>0 \\
c(X_1,\ldots,X_n) & \texttt{==} & d(Y_1,\ldots,Y_k) & \rightarrow & \texttt{false} & \forall c \in \mathcal{C}^{(n)},\ d \in \mathcal{C}^{(k)} \\
 & & & & & \text{mit } c \neq d \text{ oder } n \neq k
\end{array}$$

Funktionsapplikationen auf einer der Seiten müssen zuerst zu Kopfnormalform reduziert werden, Variablen führen hingegen zur Verzögerung des Vergleichs.

Beispiel 2.8
`s(z)==s(z)` reduziert zu `true`, `1==s(z)` zu `false` und `X==s(z)` verzögert so lange, bis `X` an einen Ausdruck, der keine Variable ist, gebunden wird. ◇

Die Testgleichheit kann wie eine normale boolesche Funktion verwendet werden, beispielsweise in der Bedindung der `if-then-else`-Funktion, die Curry standardmäßig zur Verfügung stellt.

Die strikte Gleichheit mit Unifikation erlaubt Variablenbindungen und arbeitet auf Constraints:

```
data constraint = {}.

(=) :: A -> A -> constraint.
```

Der Gleichheitsconstraint e_1=e_2 ist erfüllbar, wenn die beiden Ausdrücke e_1 und e_2 zu unifizierbaren Konstruktortermen reduzierbar sind. Als Ergebnis der Reduktion erhält man σ▯`{}` mit $\sigma(e_1)$=$\sigma(e_2)$. `{}` bezeichnet den leeren, immer erfüllbaren Constraint. Können die e_i nicht zu unifizierbaren Konstruktortermen reduziert werden, schlägt der Constraint fehl, ohne einen Funktionswert zurückzuliefern, denn die Negation eines Gleichheitsconstraints ist i.a. ohne Ungleichheitsconstraints nicht ausdrückbar [3].

Beispiel 2.9
`s(X)=s(z)` reduziert zu {`X=z`}▯`{}`, `[X]=[]` schlägt hingegen fehl. ◇

Die Reduktion mit einer bedingten Regel wird daher einfach abgebrochen, wenn die Bedingung, die ein Constraint sein muß, unerfüllbar ist, während eine erfüllbare Bedingung wie im Beispiel der `weg`-Funktion auch Variablenbindungen vornehmen kann. Man beachte, daß Constraints in geschweifte Klammern eingefaßt werden, damit die Verwendung desselben Symbols `=` für Gleichheitsconstraints und Regeldefinitionen nicht zu mehrdeutig lesbaren Regeln führt.

Beispiel 2.10

```
weg(X,Y)  if {X=Y}        = [X]
          if {kante(X)=Z} = [X] ++ weg(Z,Y).
```

Der Aufruf `weg(a,Y)` nimmt in der ersten Bedingung die Bindung von `Y` an `a` vor, in der zweiten Bedingung von `Z` an das Ergebnis von `kante(a)`, beide Constraints sind in diesem Fall erfüllbar. Bei `weg(a,b)` ist hingegen der Constraint `a=b` in der ersten Bedingung nicht erfüllbar, was zum sofortigen Abbruch dieses Reduktionswegs führt. ◇

Die Konjunktion zweier Gleichheitsconstraints wird durch die Funktion `/\` mit dem Typ

```
/\ :: constraint -> constraint -> constraint.
```

vorgenommen. $\{c_1/\backslash c_2\}$ ist erfüllbar, wenn beide c_i erfüllbar sind, wobei die c_i im Gegensatz zur Konjunktion boolescher Funktionen nebenläufig ausgewertet werden können und vorgenommene Variablenbindung an das jeweils andere c_i weiterreichen, wie wir in Beispiel 2.16 in Abschnitt 2.3 sehen werden. Syntaktisch kann die Konjunktion auch durch ein Komma ausgedrückt werden. In Abschnitt 2.3 definieren wir eine operationale Semantik der strikten Gleichheit und einer sequentiellen Implementierung der Konjunktion, bei der immer nur eines der c_i einen Schritt ausführen darf. Eine nicht sequentielle nebenläufige Implementierung wird in [15] vorgestellt.

Constraints können als Parameter übergeben und auch als Funktionsergebnisse zurückgeliefert werden, wie das folgende Beispiel zeigt, dürfen jedoch nicht in der linken Seite einer Regel auftreten.

Beispiel 2.11

Eine sequentielle Auswertung zweier Constraints kann zum Beispiel durch die Einbettung in boolesche Funktionen und deren Konjunktion oder durch die Funktion `seq` erreicht werden.

```
seq :: constraint -> constraint -> constraint.
seq(C1,C2) if {C1} = {C2}.
```

`seq` besitzt den Zieltyp `constraint` und kann daher wieder als ein Constraint benutzt werden, beispielsweise als Bedingung einer anderen Regel:

```
test(C1,C2) if {seq(C1,C2)} = true.
```

◇

In Curry sind also sogenannte „deep guards“ erlaubt, d.h. es dürfen beliebige Ausdrücke und insbesondere Funktionsaufrufe in einer Bedingung verwendet werden, solange die Bedingung den Typ `constraint` behält. Auch in Beispiel 2.10 bei Angabe der `weg`-Funktion wurde ein Funktionsaufruf in einem Gleichheitsconstraint verwendet. Welche Form ein Constraint und damit auch eine Bedingung exakt annehmen darf, zeigt die EBNF, die in Anhang C.2 angegeben ist. Man beachte zusätzlich, daß die beiden Seiten eines Gleichheitsconstraints selbst keine Constraints sein dürfen, also beispielsweise ein Ausdruck der Form

```
X = {Y=1}
```

nicht erlaubt ist.

Der Anschluß von externen Constraint-Solvern und damit die Verwendung komplexerer Constraint-Strukturen ist in [16] beschrieben, ebenso der Zusammenhang mit parallelen Berechnungsmodellen und weiteren Möglichkeiten des Umgangs mit Constraints.

2.2.4 Funktionale und logische Programme

Für den funktionalen Programmierer stellt Curry viele der gewohnten Funktionen zur Verfügung, beispielsweise die Listenoperationen `(++)`, `member`, `foldr`, `map` u.a. (siehe Anhang B). Außerdem erlaubt Curry unendliche Datenstrukturen (Beispiel 2.7), Funktionen höherer Ordnung, partielle Funktionsapplikationen und Lambda-Abstraktionen der Form[3]

$\backslash X_1 \ldots X_n$ `->` e .

Die Syntax lehnt sich ebenfalls an Haskell an und ermöglicht, wie erwähnt, die Angabe von bedingten Regeln in der Form $f(t_1, \ldots, t_n)$ `if` b `=` e , wobei b ein boolescher Ausdruck ist. Intern wird dies in die Form $f(t_1, \ldots, t_n)$ `if` $\{b$`=true`$\}$ `=` e umgewandelt. In dieser Arbeit verwenden wir aus den erwähnten Gründen eine Prolog-ähnliche Notation der Programme, aber eigentlich erlaubt Curry die Verwendung der Haskell-Notation. Um den Unterschied deutlich zu machen, geben wir die Definition der `append`-Funktion, deren Arbeitsweise der von `(++)` entspricht, in beiden Darstellungen an, links die Haskell-Notation, rechts die Prolog-ähnliche.

```
data List a = [] | a : List a        data list(A) = [] ; [A|list(A)].

append :: [a] -> [a] -> [a]          append :: list(A) -> list(A) -> list(A).
append []     ys = ys                append([],Ys)     = Ys.
append (x:xs) ys = x:(append xs ys)  append([X|Xs],Ys) = [X|append(Xs,Ys)].
```

Wir werden im folgenden in kleineren Beispielreduktionen für eine übersichtlichere Darstellung `append` statt `(++)` verwenden. Vordefiniert ist in Curry jedoch nur die `(++)`-Funktion, und daher benutzen wir sie für die Angabe von Programmen, die sich auf der beigelegten Diskette befinden und im TasteCurry-System (vgl. Kapitel 7) verwendet werden können.

Logikprogramme werden in Curry durch die Darstellung von Prädikaten als boolesche Funktionen realisiert, indem man eine Klausel der Form $L \Leftarrow L_1, \ldots, L_n$ in eine Regel

L `if {`L_1`=true`,...,L_n`=true} = true`

umformt. Fakten werden als L`=true` notiert.

Beispiel 2.12
Die bekannte Großvaterrelation (väterlicherseits) läßt sich in Curry durch

```
vater(hans,klaus)  = true.
vater(hans,karl)   = true.
vater(klaus,peter) = true.
vater(klaus,egon)  = true.
vater(karl,stefan) = true.

grossvater(X,Y) if {vater(X,Z)=true, vater(Z,Y)=true} = true.
```

darstellen und erlaubt beispielsweise Anfragen wie `grossvater(hans,X)` zur Berechnung der drei Enkel Peter, Egon und Stefan, oder `grossvater(X,Y)`, um alle Großväter und deren Enkel zu ermitteln. Ein ausführlicheres Verwandtschaftsprogramm mit Beispielberechnungen findet sich in [16]. ◇

Beispiel 2.13
Eine alternative Schreibweise von Klauseln ist mit Hilfe von Constraints möglich:

[3] In der aktuellen TasteCurry-Implementierung sind wir jedoch auf einstellige Lambda-Abstraktionen beschränkt und verwenden daher auch nur solche in dieser Arbeit.

```
vater(hans,klaus)  = {}.
vater(hans,karl)   = {}.
⋮

grossvater(X,Y) = {vater(X,Z), vater(Z,Y)}.
```

Diese Schreibweise ist der aus Prolog bekannten Notation näher, führt jedoch zu nichtdeterministischen Berechnungen, wenn zwei linke Regelseiten überlappen, deren rechte Seiten sich wechselseitig ausschließen. In der Notation mit bedingten Regeln kann in einer entsprechenden Implementierung durch eine Prüfung der Bedingungen eine eindeutige Regel zur Anwendung bestimmt werden, während die Constraint-Notation beide Regeln anwenden muß, von denen später eine zum Fehlschlag führt. $\diamond$

Mit Hilfe von Narrowing (Abschnitt 2.3) ist der relationale Gebrauch von Funktionen möglich, beispielsweise durch `append(X,[3,4])=[1,2,3,4]` oder `append(X,Y)=[1,2,3,4]`. Sogar die `map`-Funktion kann relational verwendet werden. Die Anfrage `map(F,[1,2])=[3,4]` besitzt beispielsweise die Lösung `{F=inc}` mit `inc(X)=X+1`. Solche Lösungen können mit higher-order-Unifikation berechnet werden [14], das in Curry jedoch aufgrund des hohen Berechnungsaufwands nicht implementiert ist. Stattdessen wird die Applikation als Funktion `@` (in Infix-Schreibweise) betrachtet, die einen Funktionsaufruf um einen Parameter erweitert. Die Definition der `map`-Funktion sieht damit wie folgt aus:

```
map :: (A -> B) -> list(A) -> list(B).
map(F,[])     = [].
map(F,[X|Xs]) = [F@X|map(F,Xs)].
```

Ein Aufruf `F@`e, mit `F` eine Variable, wird so lange verzögert, bis `F` an eine bekannte Funktion gebunden wird, was zur Reduktion

```
{map(F,[1,2])=[3,4]} ⇝ F@1=3 /\ F@2=4
```

führt und damit eine Darstellung einer partiellen Definition der Funktion `F` durch `F(1)=3, F(2)=4` liefert. Die operationale Semantik der `@`-Funktion wird in Abschnitt 2.3.4 angegeben.

2.3 Operationale Semantik von Curry

Das Ziel einer Berechnung ist es, einen Ausdruck mit Hilfe der Regeln eines gegebenen Curry-Programms zu einem gelösten Antwort-Ausdruck zu reduzieren. Da die operationale Semantik von Curry versucht, die Prinzipien aus funktionalen und logischen Sprachen zu vereinigen, muß sie zweiteilige Antworten berechnen, sogenannte *Antwortausdrücke*.

Definition 2.6

- Ein *Antwortausdruck* ist ein Paar $\sigma\,[]\,e$, wobei σ eine Substitution und e einen Ausdruck bezeichnen. Ein Antwortausdruck heißt *gelöst*, wenn e ein Datenterm ist.
- Ein disjunktiver Ausdruck ist eine (Multi-)Menge $\{\sigma_1\,[]\,e_1, \ldots, \sigma_n\,[]\,e_n\}$ von Antwortausdrücken. Die Menge aller Antwortausdrücke bezeichnen wir mit $\mathcal{D}$.

$\square$

e entspricht einem Funktionswert, wie er durch ein funktionales Programm berechnet wird, während durch die Substitution σ eine Bindung der möglicherweise in der Anfrage enthaltenen freien Variablen angegeben wird, mit denen die Anfrage erfüllbar bzw. reduzierbar ist. Durch freie Variablen oder nichtdeterministische Funktionen kann es mehr als eine Lösung geben, so daß das Ergebnis ein disjunktiver Ausdruck sein kann.

Der besseren Lesbarkeit wegen notieren wir in den Beispielen einen disjunktiven Ausdruck $\{\sigma_1 [] e_1, \ldots, \sigma_n [] e_n\}$ in der Form $\sigma_1 [] e_1 \;|\ldots|\; \sigma_n [] e_n$ und führen die identische Substitution sowie den leeren Constraint nicht auf. Die geschweiften Klammern, in die Constraints eingefaßt werden müssen, lösen wir während des ersten Reduktionsschritt des Constraints auf.

Beispiel 2.14
Wir betrachten die Großvater-Relation aus Beispiel 2.12. `grossvater(hans,egon)` wird in den gelösten Ausdruck `true` überführt, `grossvater(hans,X)` reduziert zu einem disjunktiven Ausdruck

```
{X=peter}[]true | {X=egon}[]true | {X=stefan}[]true ,
```

und `grossvater(hans,X)=true` liefert

```
{X=peter} | {X=egon} | {X=stefan}
```

als Ergebnis. ◇

Vor einer präzisen Definition der Reduktionssemantik wollen wir an einigen Beispiel die Ideen verdeutlichen, die hinter dieser Semantik stecken.

2.3.1 Reduktionsstrategien

Um einen Ausdruck in einen gelösten Antwortausdruck zu überführen, müssen die enthaltenen Funktionsapplikationen zu Konstruktortermen reduziert werden. Eine Applikation $f(e_1, \ldots, e_n)$, deren Parameter Grundterme sind, wird durch einen Termersetzungsschritt [4] reduziert, indem man zuerst eine anwendbare Regel durch *Pattern Matching* bestimmt. Eine Regel $l = r$ ist anwendbar, wenn für die linke Seite l (das *Muster* oder *Pattern*) eine Substitution σ mit $\sigma(l) = f(e_1, \ldots, e_n)$ existiert. Die Applikation wird dann durch $\sigma(r)$ ersetzt.

Beispiel 2.15
Die `append`Funktion hatten wir durch die beiden Regeln

```
append([],Ys)     = Ys.
append([X|Xs],Ys) = [X|append(Xs,Ys)].
```

definiert. Für die Reduktion der Applikation `append([1,2],[3,4])` kann nur die zweite Regel mit Hilfe der Substitution {`X=1, Xs=[2], Ys=[3,4]`} angewendet werden und erzeugt den Antwortausdruck `[1|append([2],[3,4])]`. ◇

Bei der `append`-Funktion benötigen wir den Wert des ersten Parameters, um die anwendbare Regel zu bestimmen: Bei einer leeren Liste reduzieren wir mit der ersten Regel, ansonsten mit der zweiten. Wenn nun statt eines Grundterms eine freie Variable als Parameter eines `append`-Aufrufs verwendet wird, gibt es zwei Möglichkeiten, um eine Entscheidung über die anzuwendende Regel zu treffen: Residuation und Narrowing.

Beispiel 2.16
Wenn man nach dem Residuation-Prinzip vorgeht, das beispielsweise in den Sprachen Escher [21] und Oz [33] verwendet wird, wird die Entscheidung über die anwendbare Regel so lange verzögert, bis die Variable an einen Wert gebunden wird, der eine eindeutige Entscheidung zuläßt. Aus diesem Grund kann in der nachfolgenden Reduktion der linke Constraint zunächst nicht reduzieren, da `append(X,[3,4])` suspendiert. Erst nachdem der rechte Zweig `X` an die leere Liste gebunden hat, ist eine Reduktion möglich.

```
{append(X,[3,4]) =  L /\  X = []}
⇝ {X=[]}▯append([],[3,4]) = L /\ {}
⇝ {X=[]}▯[3,4] = L /\ {}
⇝ ...
```

Diese Beispielreduktion zeigt, wie durch Residuation die nebenläufige Auswertung zweier Constraints über Variablenbindungen synchronisiert werden kann. ◇

Beispiel 2.17
Bei Anwendung des Narrowing-Prinzips, das beispielsweise in Babel [20] verwendet wird, bindet Curry die Variable an die verschiedenen Muster aller linken Regelseiten und liefert einen disjunktiven Antwortausdruck zurück.

```
{append(X,Z) = [1]}
⇝ {X=[]}▯append([],Z) = [1]  |  {X=[Y|Ys]}▯append([Y|Ys],Z) = [1] .
```

Für beide Antwortausdrücke kann nun im nächsten Schritt eine anwendbare Regel bestimmt werden. ◇

Narrowing ermöglicht den relationalen Gebrauch von Funktionen, den wir in Abschnitt 2.2.4 vorgestellt haben, und kann daher zur Erstellung von Logikprogrammen verwendet werden, während durch Residuation, sofern keine nichtdeterministischen Funktionen verwendet werden, ein deterministisches Reduktionsverhalten erreicht wird, wie es aus funktionalen Sprachen bekannt ist. Da Curry versucht, logische und funktionale Programmieransätze zu vereinigen, werden boolesche Funktionen standardmäßig durch Narrowing, alle anderen durch Residuation ausgewertet. Dieses Verhalten kann durch Auswertungsvorschriften (Abschnitt 2.3.2) verändert werden.

Bei geschachtelten Funktionsaufrufen verwendet Curry das Prinzip der verzögerten Auswertung (*lazy evaluation*), um unnötige Berechnungen zu vermeiden. Dabei reduziert der äußerste Funktionsaufruf, sofern seine Argumente die Bestimmung einer anwendbaren Regel zulassen.

$\texttt{append}_1\texttt{([],append}_2\texttt{([3],[4]))} \rightsquigarrow \texttt{append}_2\texttt{([3],[4])}$

Andernfalls werden die benötigten Argumente reduziert, aber auch nur soweit, wie es zur Regelauswahl erforderlich ist.

$\texttt{append}_1\texttt{(append}_2\texttt{([3],[4]),[])}$
$\rightsquigarrow \texttt{append}_1\texttt{([3|append}_2\texttt{([],[4])],[])}$
$\rightsquigarrow \texttt{[3|append}_1\texttt{(append}_2\texttt{([],[4]),[])]}$
$\rightsquigarrow \texttt{...}$

Wenn mehrere Argumente zur Entscheidung ausgewertet werden müssen, kann die richtige Reihenfolge unnötige Reduktionen ersparen.

Beispiel 2.18
In Beispiel 2.3 hatten wir die Funktionen `add` und `leq` durch folgende Regeln definiert:

```
leq(z,N)         = true.          add(z,N)    = N.
leq(s(M),z)      = false.         add(s(M),N) = s(add(M,N)).
leq(s(M),s(N)) = leq(M,N).
```

Die Anfrage `leq(add(z,z),add(z,z))` könnte zuerst das zweite und dann das erste Argument von `leq` auswerten und würde dann wie folgt reduzieren:

$\texttt{leq(add}_1\texttt{(z,z),add}_2\texttt{(z,z))} \rightsquigarrow \texttt{leq(add}_1\texttt{(z,z),z)} \rightsquigarrow \texttt{leq(z,z)} \rightsquigarrow \texttt{true}$

Wertet man hingegen das erste Argument vor dem zweiten aus, erhält man mit Hilfe der ersten Regel von `leq` dasselbe Ergebnis, ohne daß der zweite Aufruf der `add`-Funktion ausgewertet werden muß. ◇

Um solche überflüssigen Schritte zu vermeiden, versucht Curry aus den Mustern der linken Regelseiten eine Strategie herzuleiten, in welcher Reihenfolge und wieweit die Argumente zur Bestimmung einer anwendbaren Regel ausgewertet werden müssen. Für die `leq`-Funktion lautet sie:

> *Werte das erste Argument zu Kopfnormalform aus. Wenn es die Gestalt* `s(X)` *hat, werte auch das zweite Argument zu Kopfnormalform aus und wende die 2. Regel an, ansonsten wende die 1. Regel an.*

Solche Vorschriften werden automatisch erzeugt, können aber auch vom Benutzer mit Hilfe der Auswertungsvorschriften definiert werden, die zudem die Auswahl der Reduktionsart (Narrowing oder Residuation) erlauben.

2.3.2 Auswertungsvorschriften

Durch Auswertungsvorschriften kann in Curry eine Strategie angegeben werden, bis zu welchem Grad und in welcher Reihenfolge die Parameter einer Funktion zur Bestimmung der Regelanwendung ausgewertet werden müssen.

Wird eine Funktion f nur durch *eine* Regel mit linker Seite $f(X_1, X_2, \ldots, X_n)$ definiert, ist für jede Applikation von f sofort ein Termersetzungsschritt möglich, was durch eine `rule`-Vorschrift angegeben wird.

Beispiel 2.19

```
square eval rule.
square(X) = X*X.
```
◇

Wenn die Werte von Argumenten zur Entscheidung über die Regelanwendung benötigt werden, muß für jedes dieser Argumente die Reduktionsstrategie angegeben werden. Die Vorschrift

```
append eval 1:rigid.
```

bedeutet, daß das erste Argument in Kopfnormalform ausgewertet werden muß, um eine anwendbare Regel bestimmen zu können. `rigid` erzwingt eine Auswertung nach dem Residuation-Prinzip, so daß `append(X,[3,4])` wie in Beispiel 2.16 suspendiert. Die Reduktion mittels Narrowing wie in Beispiel 2.17 wird durch die Vorschrift

```
append eval 1:flex.
```

erreicht. Für die `leq`-Funktion können wir die im vorigen Abschnitt ermittelte Strategie durch folgende Vorschrift angeben (wir haben hierbei Narrowing zur Auswertung gewählt):

```
leq eval 1:flex(s => 2:flex).
```

Wenn der erste Parameter zu einer Kopfnormalform mit dem Konstruktor `s` reduziert, muß der zweite Parameter ebenfalls ausgewertet werden, andernfalls kann sofort eine Reduktion durchgeführt werden. Wir können im Allgemeinfall mehrere Konstruktoren aufführen, wenn verschiedene Kopfnormalformen möglich sind, und tiefere Position in Argumenten angeben (vgl. Anhang C.2). Auch die Verwendung einer Variable statt eines Konstruktors ist möglich und sorgt dafür, daß vor Anwendung der Regel der Parameter auf jeden Fall in Kopfnormalform gebracht wird, gleichgültig mit welchem Konstruktor an der Spitze.

Für die Auswertung von Funktionen mit überlappenden linken Regelseiten benötigt man die *or*-Vorschrift, durch die zwei alternative Strategien für die Auswertung der Argumente angegeben werden. Das folgende Beispiel stammt aus [11].

Beispiel 2.20

```
mult eval 1:rigid or 2:rigid.
mult(0,X) = 0.
mult(X,0) = 0.
```

Wenn eines der beiden Argumente zu 0 ausgewertet werden kann, kann ein `mult`-Aufruf reduzieren, und daher wird die Anwendung beider Regeln versucht. ◇

Wenn keine Auswertungsvorschrift zu einer Funktion angegeben wird, verwendet Curry eine Standardstrategie, bei der das Pattern Matching der Argumente wie in funktionalen Sprachen von links nach rechts vorgenommen wird. Man kann aber durch explizite Angaben manchmal eine geschicktere Strategie wählen, wie das folgende Beispiel zeigt [11]:

Beispiel 2.21

```
f(z,z)    = z.
f(X,s(N)) = z.
```

Mit der Vorschrift `f eval 2:flex(z => 1:flex)` reduziert der Aufruf `f(g,z)` in einem Schritt sogar dann zu `z`, wenn `g` eine nichtterminierende Funktion ist. Bei der links-nach-rechts-Auswertung, wie sie beispielsweise in Haskell [18] verwendet wird, wäre dies nicht möglich. ◇

Da wie erwähnt boolesche Funktionen durch Narrowing ausgewertet werden sollen, werden für sie standardmäßig *flex*-Vorschriften erzeugt, für alle anderen Funktionen *rigid*-Vorschriften. Eine teilweise automatische Erzeugung kann durch Auswertungsvorschriften

```
f eval flex.
f eval rigid.
```

erreicht werden. Sie geben an, daß alle Argumente, soweit erforderlich, bei Angabe von `flex` durch Narrowing und bei Angabe von `rigid` durch Residuation reduziert werden sollen. Bis zu welchem Grad und in welcher Reihenfolge die Parameter überhaupt ausgewertet werden müssen, wird von Curry selbständig festgestellt.

Für größere Programmteile läßt sich die automatische Generierung auch durch Verwendung von Pragmas steuern [16], die wir an dieser Stelle nicht erläutern, da sie für unsere Arbeit nicht von Interesse sind. Die kontextfreie Syntax der Auswertungsnotationen wird in Anhang C.2 durch die Regeln für das Nichtterminalsymbol *EvalAnnot* angegeben.

2.3.3 Definierende Bäume

Die Auswertungsvorschriften für Funktionen werden in Curry beim Einlesen eines Programms in definierende Bäume übersetzt, die von Antoy [1] eingeführt wurden. Ein definierender Baum enthält alle Regeln zu einer Funktion und gibt an, in welcher Reihenfolge und zu welcher Form die Argumente ausgewertet werden müssen, um die Regeln anzuwenden. In [11] zeigt Hanus, wie durch eine einfache Erweiterung die Wahl zwischen Narrowing und Residuation in diese Bäume integriert werden kann. Die folgende Definition ist aus [11] entnommen, allerdings verzichten wir auf die dort zur Implementierung der nebenläufigen Konjunktion verwendeten *and*-Bäume und nehmen stattdessen eine Erweiterung der Reduktionsfunktion für Ausdrücke vor (Abschnitt 2.3.4).

Definition 2.7
$\mathcal{T}$ ist ein *definierender Baum mit Muster* π gdw. die Tiefe von $\mathcal{T}$ endlich ist und einer der folgenden Fälle gilt:

- $\mathcal{T} = rule(l{=}r)$, wobei $l{=}r$ eine Variante einer Regel ist und $l = \pi$ gilt.

- $\mathcal{T} = branch(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_n)$, wobei $\pi|_p$ eine Variable ist, $r \in \{flex, rigid\}$ gilt, $c_1, \ldots, c_k$ verschiedene Konstruktoren der Sorte[4] von $\pi|_p$ mit $k > 0$ sind, und $\mathcal{T}_i$ ein definierender Baum mit Muster $\pi[c_i(X_1, .., X_n)]|_p$ für $i = 1, \ldots, k$ ist, wobei n die Stelligkeit von c_i ist und $X_1, \ldots, X_n$ neue Variablen sind.

- $\mathcal{T} = or(\mathcal{T}_1, \mathcal{T}_2)$, wobei $\mathcal{T}_1$ und $\mathcal{T}_2$ definierende Bäume mit Muster π sind.

Ein *definierender Baum einer n-stelligen Funktion f* ist ein definierender Baum $\mathcal{T}$ mit Muster $f(X_1, \ldots, X_n)$, wobei die X_i paarweise verschiedene Variablen sind und es für jede Regel l=r mit $l = f(t_1, \ldots, t_n)$ einen Knoten $rule(l'$=$r')$ in $\mathcal{T}$ gibt, wobei l eine Variante von l' ist. Die Menge der definierenden Bäume bezeichnen wir im folgenden mit DT, und das Muster eines definierenden Baumes $\mathcal{T}$ mit $pat(\mathcal{T})$. □

Die Umsetzung von `rule` und `or`-Auswertungsvorschriften in die entsprechenden Bäume dürfte offensichtlich sein. Die Struktur des *branch*-Baums wollen wir uns am Beispiel der `leq`-Funktion verdeutlichen.

Beispiel 2.22
Die Auswertungsvorschrift für die `leq`-Funktion lautete:

```
leq eval 1:flex(s => 2:flex).
```

Entsprechend wird ein *branch*-Baum der Form $branch(\texttt{leq}(X_1, X_2), 1, flex, \mathcal{T}_1, \mathcal{T}_2)$ erzeugt. Er besagt, daß das aktuelle Muster `leq` nicht zur Regelanwendung ausreicht, sondern das erste Argument mit Narrowing ($flex$) zur Kopfnormalform reduziert werden soll. Wenn das Ergebnis `z` ist, können wir eine Regel anwenden, so daß $\mathcal{T}_1$ die Form $rule(\texttt{leq(z},Y\texttt{) = true})$ hat. $\mathcal{T}_2$ ist hingegen wieder ein *branch*-Baum der Form $branch(\texttt{leq(s(}M\texttt{)},X_2), 2, flex, \mathcal{T}_3, \mathcal{T}_4)$. Das Muster hat sich geändert, da nun die Reduktion für den Fall beschrieben wird, daß das erste Argument die Form $\texttt{s}(M)$ angenommen hat. Wir müssen dann den zweiten Parameter auswerten, und auch da gibt es wieder zwei Möglichkeiten der Kopfnormalform. Da aber in jedem Fall eine Regelanwendung möglich ist, sind $\mathcal{T}_3$ und $\mathcal{T}_4$ einfache *rule*-Bäume. Der komplette Baum für die `leq`-Funktion lautet:

$$
\begin{array}{l}
branch(\texttt{leq}(X_1, X_2), 1, flex, rule(\texttt{leq(z},Y\texttt{) = true}) \\
\qquad\qquad\qquad\qquad branch(\texttt{leq(s(}M\texttt{)},X_2), 2, flex, rule(\texttt{leq(s(}M\texttt{),z) = false}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad rule(\texttt{s(}M\texttt{),s(}N\texttt{)) = leq(}M,N\texttt{)))))}
\end{array}
$$

◇

Beim Compilieren oder Einlesen eines Curry-Programms durch einen Interpreter wird zu jeder Funktion ein entsprechender definierender Baum erstellt. Ohne Angabe einer Auswertungsvorschrift zu einer Funktion durch den Benutzer erfolgt eine automatische Generierung des definierenden Baums [16]. Im nächsten Abschnitt werden wir sehen, wie in Curry mit Hilfe dieser Bäume die in Abschnitt 2.3.1 beschriebenen Ideen der Reduktionssemantik verwirklicht werden.

2.3.4 Die Reduktionsfunktionen

Wenn wir in Curry eine Anfrage e stellen, wird sie zuerst in einen besonderen disjunktiven Ausdruck umgewandelt, den *initialen Ausdruck* $\{id[\!]e\}$. Die Reduktion auf der Menge der disjunktiven Ausdrücke erfolgt durch die Funktion

$$\stackrel{RN}{\longrightarrow} \;:\; \mathcal{D} \;\rightarrow\; \mathcal{D}$$

[4] Wir hatten uns bei den Definitionen in Abschnitt 2.1 auf den einsortigen Fall beschränkt. Da wir in der Regel aber mit mehreren Sorten arbeiten, dürfen hier nur die Konstruktoren verwendet werden, die dieselbe Sorte haben wie $\pi|_p$.

nach der Vorschrift aus Abbildung 2.1, die ebenso wie die Abbildungen 2.2 und 2.4, deren Darstellungen in bezug auf die nebenläufige Konjunktion geändert wurden, aus [16] entnommen ist. Solange ein nicht gelöster Antwortausdruck existiert, der durch die Funktion *cse* reduziert

$$
\boxed{\begin{array}{lll} \{\sigma [] e\} \cup D & \stackrel{RN}{\longrightarrow} & \{\sigma_1 \circ \sigma [] e_1, \ldots \sigma_n \circ \sigma [] e_n\} \cup D \\ & & \text{wenn } \sigma [] e \text{ ungelöst ist und } cse(e) = \{\sigma_1 [] e_1, \ldots, \sigma_n [] e_n\} \end{array}}
$$

Abbildung 2.1: Reduktionsschritt für einen disjunktiven Ausdruck

werden kann, wird er durch das Ergebnis der Reduktion, das wiederum ein disjunktiver Ausdruck sein kann, ersetzt. Antwortausdrücke, deren Reduktion suspendiert, verbleiben in der ungelösten Form.

Die Reduktion eines einzelnen Ausdrucks wird durch die Funktionen

$$
\begin{array}{lcllll} cse & : & \mathcal{T}(\Sigma, \mathcal{X}) & \to & \mathcal{D} \cup \{\perp\} \\ equal & : & \mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X}) & \to & \mathcal{D} \cup \{\perp\} \\ cs & : & \mathcal{T}(\Sigma, \mathcal{X}) \times DT & \to & \mathcal{D} \cup \{\perp\} \end{array}
$$

vorgenommen, wobei Σ die bekannte konstruktorbasierte Signatur ist. Das Symbol $\perp$ als Ergebnis bedeutet, daß die Reduktion des Ausdrucks suspendiert ist.

Die Arbeitsweise von *cse* ist in Abbildung 2.2 dargestellt. Darin werden, wie auch in allen folgenden Abbildungen, durch X und Y Variablen, durch c und d Konstruktoren und durch f und g Funktionen bezeichnet.

$$
\begin{array}{l}
cse(X) \quad = \quad \perp \\
cse(c(e_1, \ldots, e_n)) \\
= \left\{ \begin{array}{ll} replace(c(e_1, \ldots, e_n), k, cse(e_k)) & \text{wenn } cse(e_1) = \cdots = cse(e_{k-1}) = \perp \neq cse(e_k) \\ \perp & \text{wenn } cse(e_i) = \perp, \text{ für } i = 1, \ldots, n \end{array} \right. \\
\\
cse(X \texttt{@} e) \quad = \quad \perp \\
cse(\varphi(e_1, \ldots, e_n) \texttt{@} e) \quad = \quad \varphi(e_1, \ldots, e_n, e) \quad \text{wenn } \varphi \in \Sigma^{(m)} \text{ mit } n < m \\
cse(c_1 \,\texttt{/\textbackslash}\, c_2) \\
= \left\{ \begin{array}{ll} \{id [] \texttt{\{\}}\} & \text{wenn } c_1 = \texttt{\{\}} \text{ und } c_2 = \texttt{\{\}} \\ \emptyset & \text{wenn } cse(c_1) = \emptyset \text{ oder } cse(c_2) = \emptyset \\ \perp & \text{wenn } cse(c_1) = \perp \text{ und } cse(c_2) = \perp \\ replace(c_1 \,\texttt{/\textbackslash}\, c_2, 1, cse(c_1)) & \text{wenn } \mid cse(c_1) \mid \leq \mid cse(c_2) \mid \text{ oder } cse(c_2) = \perp \\ replace(c_1 \,\texttt{/\textbackslash}\, c_2, 2, cse(c_2)) & \text{wenn } \mid cse(c_1) \mid > \mid cse(c_2) \mid \text{ oder } cse(c_1) = \perp \end{array} \right. \\
\\
cse(e_1 \texttt{=} e_2) \quad = \quad equal(e_1, e_2) \\
cse(f(e_1, \ldots, e_n)) \quad = \quad cs(f(e_1, \ldots, e_n), \mathcal{T}) \quad \text{mit } \mathcal{T} \text{ der definierende Baum für } f \text{ mit} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{neuen Variablen und } f \notin \{\texttt{@}, \texttt{=}, \texttt{==}, \texttt{/\textbackslash}\}
\end{array}
$$

Abbildung 2.2: Reduktionsschritt für einen einzelnen (ungelösten) Ausdruck

Eine Variable kann durch *cse* nicht weiter reduziert werden, während bei einem Term in Kopfnormalform das erste Argument von links reduziert wird, dessen Berechnung nicht suspendiert. Entsteht dabei ein disjunktiver Ausdruck, muß für jeden Antwortausdruck eine

Kopie des Terms angelegt werden, in dem der Parameter durch den entsprechenden Wert ersetzt wird.

Beispiel 2.23

$$\begin{array}{rcl} cse(\texttt{[1,2,append(X,3),4]}) & = & \{\{\texttt{X=[]}\}\,\Box\,\texttt{[1,2,append([],3),4]}, \\ & & \{\texttt{X=[Y|Ys]}\}\,\Box\,\texttt{[1,2,append([Y|Ys],3),4]}\} \end{array}$$

◇

Die Funktion *replace* nimmt diese Anpassung vor:

$$replace(e,p,d) \quad = \quad \begin{cases} \{\sigma_1\,\Box\,\sigma_1(e)[e_1]_p, \ldots, \sigma_n\,\Box\,\sigma_n(e)[e_n]_p\} & \text{if } d = \{\sigma_1\,\Box\,e_1, \ldots, \sigma_n\,\Box\,e_n\} \\ \bot & \text{if } d = \bot \end{cases}$$

Die Applikation eines Arguments auf eine Funktion durch den Operator `@`, den wir in Abschnitt 2.2.4 vorgestellt haben, suspendiert, wenn die Funktion unbekannt ist, d.h. durch eine Variable repräsentiert wird. Andernfalls wird die partielle Funktionsapplikation um ein Argument erweitert.

Die nebenläufige Konjunktion muß gesondert behandelt werden, da wir im Unterschied zu der Semantik aus [16] keine definierenden Bäume für die Unterstützung der Nebenläufigkeit definiert haben. Sie reduziert zum leeren Constraint, wenn beide Zweige erfüllt worden sind, schlägt fehl, wenn einer der Zweige fehlschlägt, und suspendiert, wenn beide Zweige suspendieren. Andernfalls kann wenigtens ein Zweig eine erfolgreiche Reduktion durchführen, und um den disjunktiven Antwortausdruck so klein wie möglich zu halten, darf derjenige reduzieren, der eine kleinere Antwortmenge erzeugt. Dadurch kann in einigen Fällen das Einschlagen von unnötigen Berechnungswegen vermieden werden, z.B. bei dem Ausdruck `leg(X,Y) /\ X=z`, der deterministisch reduziert, da zuerst der rechte Zweig ausgewertet wird.

Wir werden in manchen Beispielen aus Bequemlichkeit eine Konjunktion e_1`/\`e_2 in einem Schritt zu $\sigma\,\Box\,e_2$ reduzieren, wenn $cse(e_1) = \sigma\,\Box\,$`{}` gilt (und ebenso zu $\sigma\,\Box\,e_1$ im umgekehrten Fall).

Die Reduktionsfunktion für die strikte Gleichheit mit Unifikation in Abbildung 2.3 basiert auf [23] und realisiert die in Abschnitt 2.2.3 beschriebenen Ideen. Die operationale Semantik der Testgleichheit führen wir nicht auf, da sie ähnlich lautet. Sie suspendiert, anstatt Variablen zu binden, liefert `true` statt `{}` und `false` statt einer leeren Menge zurück. Die ersten beiden Regeln sorgen dafür, daß in beiden Argumenten Funktionsapplikationen[5] zu Kopfnormalform ausgewertet werden. Der Vergleich zweier Terme in Kopfnormalform reduziert zu einem Vergleich der Argumente, wenn die Konstruktorsymbole beider Terme dieselben sind. Wir führen 0-stellige Konstruktoren nicht explizit auf, sondern interpretieren die Konjunktion $e_1 = e'_1$ `/\` $\ldots$ `/\` $e_n = e'_n$ für $n = 0$ als leeren Constraint.

Wenn einer der Terme eine Variable ist, wird eine Bindung an den anderen Term vorgenommen, wobei zwei Dinge zu beachten sind:

1. Eine Bindung darf nur an Konstruktorterme erfolgen.

2. Eine Variable darf nicht an einen Konstruktorterm gebunden werden, der diese Variable enthält (*occur check*).

Die Argumente eines Terms in Kopfnormalform können Funktionsapplikationen enthalten, und daher bindet die zweite *bind*-Regel nicht einfach die Variable an den Term $c(e_1, \ldots, e_n)$, sondern an $c(X_1, \ldots, X_n)$ und stellt die Auswertung der e_i zu Konstruktortermen durch die Gleichheitsconstraints $X_i = e_i$ sicher. Man könnte hier eine Verbesserung vornehmen, indem man nur für diejenigen e_i, die keine Konstruktorterme sind, eine neue Variable erzeugt. Damit

[5] Wir verwenden aus Platzgründen $f(\cdots)$ und $g(\cdots)$ als Abkürzungen für die Applikationen $f(e_1, \ldots, e_n)$ und $g(e'_1, \ldots, e'_k)$.

$$equal(f(\cdots),e) \;=\; \begin{cases} replace(f(\cdots){=}e, 1, cse(f(\cdots))) & \text{wenn } cse(f(\cdots)) \neq \bot \\ & \text{oder } e \neq g(\cdots) \\ replace(f(\cdots){=}e, 2, cse(e)) & \text{sonst} \end{cases}$$

$$\begin{array}{lcll} equal(e,f(\cdots)) & = & replace(e{=}f(\cdots), 2, cse(f(\cdots))) & \text{wenn } e \neq g(\cdots) \\ equal(X,e) & = & bind(X,e) & \text{wenn } e \neq f(\cdots) \\ equal(e,X) & = & bind(X,e) & \text{wenn } e \text{ in Kopfnormalform} \end{array}$$

$$equal(c(e_1,\ldots,e_n), d(e'_1,\ldots,e'_k)) \;=\; \begin{cases} \{id \,\Box\, e_1{=}e'_1 \,/\!\backslash \ldots /\!\backslash\, e_n{=}e'_n\} & \text{wenn } c = d,\; n = k \\ \emptyset & \text{sonst} \end{cases}$$

$$bind(X,Y) \;=\; \{\{X{=}Y\} \,\Box\, \{\}\}$$

$$bind(X, c(e_1,\ldots,e_n)) \;=\; \begin{cases} \{\{X{=}c(Y_1,\ldots,Y_n)\} \,\Box\, Y_1{=}e_1 \,/\!\backslash \ldots /\!\backslash\, Y_n{=}e_n\} \\ \qquad \text{wenn } X \notin cv(c(e_1,\ldots,e_n)),\; Y_i \text{ neue Variablen} \\ \emptyset \quad \text{sonst} \end{cases}$$

$$\begin{array}{lcl} cv(X) & = & \{X\} \\ cv(c(e_1,\ldots,e_n)) & = & \bigcup_{i=1}^{n} cv(e_i) \\ cv(f(e_1,\ldots,e_n)) & = & \emptyset \end{array}$$

Abbildung 2.3: Strikte Gleichheit mit Unifikation

würde der Constraint `X=[Y,add(z,z),s(z)]` zu

```
{X=[Y,B,s(z)]}□ B=add(z,z)
```

reduzieren und nicht zu

```
{X=[A,B,C]}□ A=Y /\B=add(z,z) /\ C=s(z),
```

wodurch man überflüssige Reduktionsschritte für die Auswertung des ersten und dritten Gleichheitsconstraints vermeiden könnte. In unseren Beispielen werden wir immer die verbesserte Form benutzen.

Für den *occur check* dürfen nur die Variablen von $c(e_1,\ldots,e_n)$ betrachtet werden, die nicht in einer Funktionsapplikation auftreten, da diese während der Reduktion der Funktion verschwinden können, wie das folgende Beispiel zeigt.

Beispiel 2.24

```
f(z,X)    = z.
f(s(Y),X) = s(X).

{X=s(f(z,X))}
⤳ {X=s(X1)}□ X1=f(z,s(X1))
⤳ {X=s(X1)}□ X1=z
⤳ {X=s(z),X1=z}□ {}
```

Hingegen wird beispielsweise der Constraint `X=[z,s(X),s(z)]` aufgrund des *occur check* fehlschlagen. ◇

Wir berechnen daher durch *cv* (*critical variables*) die Menge der Variablen, die nicht in einer Funktionsapplikation auftreten, und prüfen, ob X darin enthalten ist.

Für alle Funktionen außer der Applikation, der Konjunktion und der strikten Gleichheit wird der definierende Baum ermittelt und an die *cs*-Funktion übergeben, die die Reduktion nach den Ideen der vorigen Abschnitte vornimmt. Der definierende Baum wird mit frischen Variablen versehen, damit durch eine zweifache Anwendung einer Funktion nicht zweimal dieselben Variablennamen in der Berechnung auftreten.

$$
\begin{aligned}
cs(e, rule(l{=}r)) &= \{id\,[]\,\sigma(r)\} \quad \text{wenn } \sigma \text{ eine Substitution mit } \sigma(l) = e \text{ ist} \\
cs(e, or(\mathcal{T}_1, \mathcal{T}_2)) &= \begin{cases} cs(e,\mathcal{T}_1) \cup cs(e,\mathcal{T}_2) & \text{wenn } cs(e,\mathcal{T}_1) \neq \bot \neq cs(e,\mathcal{T}_2) \\ \bot & \text{sonst} \end{cases}
\end{aligned}
$$

$$
cs(e, branch(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_k))
$$

$$
= \begin{cases}
cs(e, \mathcal{T}_i) & \text{wenn } e|_p = c(e_1,\ldots,e_n) \text{ und } pat(\mathcal{T}_i)|_p = c(X_1,\ldots,X_n) \\
\emptyset & \text{wenn } e|_p = c(\cdots) \text{ und } pat(\mathcal{T}_i)|_p \neq c(\cdots),\ \text{für } i = 1,\ldots,k \\
\bot & \text{wenn } e|_p = X \text{ und } r = rigid \\
\bigcup_{i=1}^{k} \{\sigma_i\,[]\,\sigma_i(e)\} & \text{wenn } e|_p = X,\ r = flex,\ \text{und } \sigma_i = \{X{=}pat(\mathcal{T}_i)|_p\} \\
replace(e, p, cse(e|_p)) & \text{wenn } e|_p = f(e_1,\ldots,e_n)
\end{cases}
$$

Abbildung 2.4: Reduktionsschritt für eine Funktionsapplikation

Ein *rule*-Baum kann einen sofortigen Termersetzungsschritt durchführen (Beispiel 2.15). Ein *or*-Tree wie im Beispiel der `mult`-Funktion (Beispiel 2.20) ermöglicht zwei verschiedene Reduktionswege und liefert beide Ergebnisse als Menge zurück. Wenn einer der beiden Wege keine Reduktion durchführen kann, muß man aus Gründen der Vollständigkeit den gesamten Ausdruck suspendieren [11]. Ein *branch*-Baum signalisiert, daß der Wert des Arguments $e_{|p}$ zur Regelauswahl benötigt wird. Ist $e_{|p}$ bereits in Kopfnormalform, wird die Reduktion mit dem Baum fortgesetzt, der ein passendes Muster enthält. Existiert kein solcher Baum, schlägt die Berechnung fehl. Wenn $e|_p$ eine Variable ist, wird die Reduktion bei einem *rigid*-Baum durch Residuation vorgenommen, was zur Suspension (Beispiel 2.16) führt, während ein *flex*-Baum die Auswertung durch Narrowing (Beispiel 2.17) bewirkt und die Variable an die verschiedenen Muster aller Bäume bindet. Im letzten Fall ist $e|_p$ eine Funktionsapplikation, die wir zuerst zu Kopfnormalform auswerten müssen um eine Entscheidung über eine Regelanwendung zu ermöglichen.

Wir haben keine spezielle Semantik für Lambda-Abstraktionen angegeben, da sie in den bisherigen Implementierungen von Tastecurry durch Lambda-Lifting in normale Funktionen umgewandelt werden (vgl. Abschnitt 7.3). `\`X `->` e wird durch die partielle Funktionsapplikation `lambda(`$X_1,\ldots,X_n$`)` ersetzt, wobei `lambda` ein neuer Name ist und $\{X_1,\ldots,X_n\} = var(e)\backslash\{X\}$ gilt. Für `lambda` wird ein *rule*-Baum für die Funktionsregel

`lambda(`$X_1,\ldots,X_n,X$`) =` e

erstellt. Die Variablen von e können mit Ausnahme von X außerhalb der Lambda-Abstraktion auftauchen und müssen deshalb als Parameter übergeben werden.

Beispiel 2.25
Die Applikation

```
(\X -> {add(Y,X)=s(z)})@L
```

wird in den Ausdruck

```
lambda(Y)@L
```

umgewandelt, und für `lambda` wird der definierende Baum

$rule$(`lambda(Y,X) = {add(Y,X)=s(z)}`)

erstellt. ◇

Bedingte Regeln

Wir sind bei der Beschreibung der definierenden Bäume und der Einzelschrittsemantik von Regeln der Form `l=r` ausgegangen und haben somit den Fall der bedingten Regeln unberücksichtigt gelassen. Dies ist jedoch keine Einschränkung, da Curry in Anlehnung an Babel eine bedingte Regel der Form l `if {`c`} =` r als unbedingte Regel l `=` $(c \Rightarrow r)$. interpretiert. Die rechte Seite ist ein sogenannter *bewachter* Ausdruck mit Wächter (*guard*) c und Rumpf r. Die Semantik eines bewachten Ausdrucks kann durch eine Regel für die Funktion $\Rightarrow$ erklärt werden:

$$(\texttt{\{\}} \Rightarrow e) \texttt{ = } e$$

Wenn der Constraint `c` gelöst, d.h. zu `{}` reduziert werden konnte, wird zur rechten Regelseite e reduziert, andernfalls schlägt die Berechnung fehl. Dieses Vorgehen entspricht der Semantik bedingter Regeln, die in Abschnitt 2.2.3 erklärt wurde.

Man beachte, daß diese Regel nicht in einem Programm angegeben werden darf, sondern vordefiniert sein muß, da die Verwendung des leeren Constraints in einer linken Regelseite nicht zulässig ist.

Korrektheit und Vollständigkeit

In [11] stellt Hanus die Korrektheits- und Vollständigkeitsergebnisse der Reduktionsrelation $\xrightarrow{RN}$ im Vergleich mit der Standard-Termersetzungsrelation für funktional-logische Programme vor, deren Regeln $l = r$ die Bedingung $var(r) \subseteq var(l)$ erfüllen müssen und sich somit von Curry-Programmen aus Definition 2.5 in Abschnitt 2.2.1 unterscheiden. Dabei wird gezeigt, daß $\xrightarrow{RN}$ äquivalent zum Residuation-Prinzip arbeitet, wenn man ausschließlich *rigid*-Deklarationen in *branch*-Bäumen verwendet.

Beschränkt man sich auf die Klasse der induktiv sequentiellen Programme, die nur Funktionen ohne *or*-Bäume und ohne Verwendung des `/\`-Operators enthalten dürfen, dann entspricht die Reduktionsrelation bei ausschließlicher Verwendung von *flex*-Knoten dem Prinzip des *Needed Narrowing*, das für die Klasse dieser Programme eine vollständige und in bezug auf die Reduktionslänge optimale Strategie ist[2].

Allerdings haben wir, wie gesehen, in Curry die Möglichkeit, auch nicht-induktiv-sequentielle Programme zu schreiben, indem wir die nebenläufige Konjunktion verwenden oder durch Extravariablen auf der rechten Seite und überlappende linke Regelseiten nichtdeterministische Funktionen definieren. Auf das Problem der Extravariablen und ihren Einfluß auf die Vollständigkeitsergebnisse geht Hanus ausführlich in [10] ein, und in [7] wird gezeigt, daß auch nichtdeterministischen Funktionen eine deklarative Semantik gegeben werden kann. Vermutlich können die dort erzielten Vollständigkeitsergebnisse für Curry übernommen werden, dies ist aber noch zu zeigen. Bis dahin kann man sich, um die Vollständigkeit der Reduktion sicherzustellen, auf induktiv-sequentielle Programme beschränken, indem man die nebenläufige Konjunktion vermeidet und die Regeln aus Abschnitt 2.2.2 anwendet (keine Extravariablen in rechten Regelseiten, Einhaltung der schwachen Eindeutigkeitsbedingung).

Kapitel 3

Die Idee der eingekapselten Suche

Wir werden in diesem Kapitel anhand von Beispielen die Notwendigkeit für die Einkapselung bestimmter Suchvorgänge in Curry aufzeigen sowie einen Ansatz zur Realisierung der Einkapselung durch Verwendung lokaler Räume vorstellen. Die dabei auftretenden Probleme erfordern eine Erweiterung der bisherigen operationalen Semantik, die wir in Kapitel 4 vornehmen. Danach stellen wir die Reduktionssemantik für die eingekapselte Suche und einige Möglichkeiten ihrer Verwendung in den Kapiteln 5 und 6 vor.

3.1 Motivation für die eingekapselte Suche

Wie wir in den vorigen Abschnitten gesehen haben, können durch freie Variablen und die Verwendung von Narrowing sowie durch nichtdeterministische Funktionen verschiedene Reduktionswege eingeschlagen werden.

Während beispielsweise Prolog die verschiedenen Wege nacheinander durch Backtracking untersucht, generiert Curry bei einem nichtdeterministischen Schritt einen disjunktiven Ausdruck, der für jede mögliche Reduktionsrichtung einen eigenen Antwortausdruck enthält. Dieses Vorgehen bezeichnet man als „don't know“-Nichtdeterminismus. Durch die Reduktionsrelation $\xrightarrow{RN}$ werden nun in einer beliebigen Reihenfolge die Antwortausdrücke einzeln und unabhängig voneinander reduziert. Aus verschiedenen Gründen wäre jedoch manchmal eine Kontrolle über nichtdeterministische Reduktionsschritte wünschenswert.

3.1.1 Kontrolle der Suchstrategie

Wenn man zum Beispiel nur an einer Lösung interessiert ist, kann das Verfolgen aller Reduktionswege unnötige Berechnungen zur Folge haben oder gar eine Terminierung und damit die Ausgabe einer gefundenen Lösung verhindern. Wenn wir die `add`-Funktion durch Narrowing auswerten, kann man beispielsweise für die Anfrage `add(Y,z)` durch Anwendung der Regel `add(z,X)=z` die Lösung

$\{$`Y=z`$\}\,[]\,$`z.`

berechnen. Wir können diese Lösung jedoch niemals ausgeben, da die Anwendung der zweiten Regel eine nicht terminierende Reduktion nach sich zieht. An dieser Stelle wäre eine Depth-First-Suchstrategie wünschenswert, bei der nach Berechnung einer Lösung alle anderen Wege abgebrochen werden.

3.1.2 Antwortausdrücke als Datenobjekte

Das Programm zur Wegesuche aus Beispiel 2.6 liefert für die Anfrage `weg(a,d)` den gelösten disjunktiven Ausdruck

```
[a,b,c,d] | [a,d] | [a,c,d] .
```

Um die gefundenen Wege vergleichen zu können, beispielsweise um den kürzesten Weg herauszufinden, müßten alle drei innerhalb *einer* Berechnung als Datenobjekte vorliegen, denn zwischen den verschiedenen Antwortausdrücken eines disjunktiven Ausdrucks ist nach Definition von $\xrightarrow{RN}$ keine Interaktion möglich.

3.1.3 Kontrolle des Fehlschlags

Wenn ein Berechnungsweg fehlschlägt, wird der zugehörige Antwortausdruck einfach aus dem disjunktiven Ausdruck gelöscht. Es gibt beispielsweise keine Möglichkeit, bei Unerfüllbarkeit einer Bedingung weiterzuarbeiten und eine entsprechende Fehlerroutine aufzurufen.

```
sumOne(X,Y) if {add(X,Y)=s(z)} = true.
```

Diese boolesche Funktion liefert `true`, wenn ihre beiden Parameter in der Summe `s(z)` ergeben, kann andernfalls aber nicht zu `false` reduzieren.

```
sumOne(s(z),z) ⇝ ... ⇝ true
sumOne(s(z),s(z)) ⇝ ... ⇝ ∅
```

Manchmal wäre es wünschenswert, auch im Falle eines Fehlschlags mit der Programmausführung fortfahren zu können, wenn beispielsweise in einem interaktiven Programm ein Benutzer eine Anfrage stellt, die nicht erfüllbar ist. In diesem Fall sollte das Programm eine entsprechende Meldung ausgeben und eine neue Eingabe verlangen statt fehlzuschlagen. Ein Beispiel für ein solches Programm werden wir in Abschnitt 6.7 kennenlernen.

3.1.4 Nichtdeterminismus und Ein-/Ausgabe

Curry verwendet für die Ein-/Ausgabe den monadischen Ansatz [29] aus Haskell. Dabei betrachtet man die Gesamtheit der Elemente, auf denen Ein-/Ausgabe durchgeführt werden kann, beispielsweise das Dateisystem, die Tastatur etc., als eine Außenwelt, die nur durch vordefinierte Operationen verändert werden darf. Wir verwenden im folgenden die Funktionen `getChar` für das Einlesen eines Zeichens von der Tastatur sowie `putChar`, `putStr` und `putStrLn` für die Ausgabe von Zeichen und Zeichenketten (mit Zeilenumbruch) auf dem Bildschirm.

```
getChar  :: io(char).
putChar  :: char -> io().
putStr   :: string -> io().
putStrLn :: string -> io().
```

Weitere Ein-/Ausgabeoperationen, beispielsweise für die Behandlung von Dateien, werden in [16] vorgestellt. Alle Operationen haben den Zieltyp `io(T)`, wobei `T` den Typ eines eventuellen Rückgabewertes bezeichnet. In der aktuellen Tastecurry-Version verwenden wir zusätzliche eine noch nicht standardisierte Funktion `show :: A -> io()`, der man statt einer Zeichenkette einen beliebigen Term übergeben kann und die diesen Term in unveränderter Form ausgibt.

Die vordefinierte Funktion `(>>=)` verknüpft zwei Operationen und leitet das Ergebnis der ersten als Eingabe an die zweite weiter, während `(>>)` das Resultat der ersten Operation ignoriert.

```
getChar>>=putChar.
getChar>>putStr("Die Eingabe ist mir egal!").
```

Beide Befehlsfolgen lesen ein Zeichen von der Tastatur ein. Im ersten Fall wird es an `putChar` weitergeleitet und somit auf dem Bildschirm ausgegeben, während im zweiten Fall unabhängig von der Eingabe ein Text ausgegeben wird.

Wenn Curry einen nichtdeterministischen Reduktionsschritt durchführt, müßte man für jeden berechneten Antwortausdruck eine eigene Kopie der Außenwelt angelegen, damit nicht zwei Reduktionen auf derselben Welt Ein-/Ausgabeoperationen durchführen, die untereinander Konflikte verursachen. Aber natürlich können zum Beispiel das Dateisystem oder ein Bildschirm nicht einfach verdoppelt werden. In Programmen, die Ein-/Ausgabeoperationen verwenden, führt deshalb das Generieren eines disjunktiven Ausdrucks mit mehr als einem Element zu einem Laufzeitfehler.

Beispiel 3.1
Bei der Reduktion des Wächters der folgenden Funktion wird ein nichtdeterministischer Schritt durchgeführt, wodurch ein disjunktiver Ausdruck mit zwei Elementen entsteht. Daher führt der Aufruf zu einem Laufzeitfehler.

```
p(1) = true.
p(2) = true.
f if {p(X)=true} = putStr("Anfrage erfüllbar!").
```

```
f ⇝ ... ⇝ *** Error: Cannot duplicate the world!
```
$\diamond$

Wir benötigen also eine Möglichkeit, den Nichtdeterminismus in Curry einzukapseln, so daß ein nichtdeterministischer Reduktionsschritt keinen disjunktiven Ausdruck produziert, sondern die verschiedenen Wege in anderer Form zurückliefert. Wir fassen abschließend noch einmal die Forderung zusammen, die wir an die eingekapselte Suche stellen:

- Verhinderung von disjunktiven Ausdrücken bei nichtdeterministischer Reduktion.
- Antwortausdrücke als Datenobjekte.
- Kontrolle über die Suchstrategie, Möglichkeit der Programmierung von Suchalgorithmen, beispielsweise für das Berechnen einer, aller oder der besten der Lösungen.
- Kontrolle über den möglichen Fehlschlag einer Berechnung.

3.2 Kontrollmechanismen in Prolog

Auch in Prolog können normalerweise zwei durch Backtracking gefundene Lösungen nicht miteinander verglichen werden, und die Verwendung von Ein-/Ausgabeoperationen kann zu Problemen führen, da ihre Auswirkungen nicht mehr rückgängig gemacht werden können, wenn durch Backtracking ein alternativer Reduktionsweg eingeschlagen wird. Aber es gibt Möglichkeiten, eine Suche bis zu einem gewissen Grad zu steuern und zu kontrollieren, beispielsweise mit Hilfe eines Meta-Interpreters oder durch Anfragen zweiter Ordnung, bei denen nach einer Menge von Elementen gefragt wird, die eine bestimmte Eigenschaft erfüllen (wir beziehen uns im folgenden auf Befehlsnamen und -syntax aus SICStus Prolog [28]). Betrachten wir erneut das Programm zur Wegesuche in einem Graph aus Beispiel 2.6, diesmal in einer Prolog-Version:

```
kante(a,b).
kante(a,d).
kante(a,c).
kante(b,c).
kante(c,d).
```

```
weg(X,X,[X]).
weg(X,Y,[X|Weg]):-kante(X,Z),weg(Z,Y,Weg).
```

Die Anfrage

```
bagof(X,weg(a,d,X),L).
```

kann bewiesen werden, wenn `L` die Liste aller Instanzen von `X` enthält, für die `weg(a,d,X)` erfüllbar ist, und daher wird `L = [[a,b,c,d],[a,d],[a,c,d]]` zurückgeliefert. Man fragt also nach der Menge aller Lösungen für die *Suchvariable* `X` bezüglich eines booleschen Ausdrucks.

Will man die Menge aller Wege berechnen, die bei `b` beginnen und irgendwo enden, stellt man die Anfrage

```
bagof(X,weg(b,Y,X),L).
```

Aufgrund der freien Variable `Y` liefert Prolog durch Backtracking mehrere Bindungen für `Y` und damit auch für `L` zurück. Wenn der Wert von `Y` aber gar nicht interessiert und man einfach nur alle Wege, die bei `a` beginnen, in einer Liste zurückgeliefert bekommen möchte, kann man `Y` durch

```
bagof(X,Y^weg(b,Y,X),L).
```

existenzquantifizieren. `Y^T` bedeutet „Es existiert ein `Y`, so daß `T` wahr ist.“ Die Bindung von `Y` ist damit uninteressant und so liefert die letzte Anfrage als Ergebnis **eine** Lösung

```
L = [[b],[b,c],[b,c,d]] .
```

Darüberhinaus existieren noch die Prädikate *setof*, das doppelte Lösungen entfernt und die Lösungen ordnet, und *findall*, das sämtliche freien Variablen existenzquantifiziert. `findall(X,weg(a,Y,X),L)` hätte also den gleichen Effekt wie die letzte bagof-Anfrage. Allerdings schlagen bagof und setof fehl, wenn gar keine Lösung existiert, während findall in diesem Fall eine leere Liste zurückliefert, was unserem Wunsch nach einer Fehlschlagkontrolle entspricht. Eines unserer Ziele bei der Realisierung der eingekapselten Suche ist es, eine dem findall- bzw. bagof-Prädikat ähnliche Funktion definieren zu können, und dazu wird es auch in Curry notwendig sein, existenzquantifizierte Variablen einzuführen (siehe 4).

3.3 List-Comprehensions in funktionalen Sprachen

Auch funktionale Sprachen wie Haskell und Miranda [17] besitzen eine Suchvorrichtung: Durch List-Comprehensions können Mengen von Werten, die bestimmte Eigenschaften erfüllen, in Form von Listen zurückgeliefert werden. Für eine ausführliche Erläuterung mit vielen Beispielen sei auf [17] verwiesen, wir wollen uns hier die grundsätzliche Arbeitsweise nur an einem Beispiel verdeutlichen.

Beispiel 3.2
Die folgende List-Comprehension liefert als Ergebnis die Liste aller Quadrate der geraden Zahlen zwischen 1 und 9.

```
[x | x <- [1..9]; x mod 2 = 0]
```

Den Teil links von `|` bezeichnet man als *Kopf*, den Teil rechts davon als *Rumpf*. Der Rumpf setzt sich zusammen aus *Generatoren*, die eine Menge von Elementen erzeugen, und *Filtern*, die bestimmte Elemente aus der Menge herausfiltern. In unserem Beispiel ist `x <- [1..9]` ein Generator, der nacheinander alle Elemente der Liste an `x` bindet. `x mod 2 = 0` ist ein Filter, der von den vorgenommenen Bindungen nur diejenigen als Ergebnis zuläßt, die durch 2 teilbar sind. ◇

Die Verwendung mehrerer Generatoren und Filter ist ebenso möglich wie die Schachtelung von List-Comprehensions und die Verwendung iterativer Generatoren, die keine Listen, sondern Berechnungsvorschriften zum Erzeugen der Elemente benutzen. Durch diagonalisierende List-Comprehensions wird zudem eine faire Behandlung mehrerer Generatoren ermöglicht.

Für Curry reichen die Möglichkeiten der List-Comprehensions nicht aus. Daß partielle Strukturen und nichtdeterministische Funktionen nicht verarbeitet werden können und somit List-Comprehensions wie

```
[X|X <- weg(Y,Z); not(Z==c)]
[X|X <- weg(a,d)]
```

nicht zulässig sind, ist mehr eine generelle Einschränkung funktionaler Sprachen. Die Suchstrategie der List-Comprehensions ist jedoch für manche Fälle nicht flexibel genug. Beispielsweise könnte man in Haskell alle Wege in einem Graph durch eine List-Comprehension suchen lassen:

```
graph = [(a,b),(a,d),(a,c),(b,c),(c,d)]

routes graph x y
  | x == y = [[x]]
  | otherwise = [x:r|z <- neighbours graph x,
                     r <- routes graph z y]
```

`neighbours graph x` generiert dabei die Liste aller Nachbarn von `x`. Der Ausdruck `routes graph a b` reduziert zu

```
[[a,b,c],[a,d],[a,b,c,d]].
```

Ist man aber am kürzesten Weg interessiert ist, muß man entweder das Programm selber ändern, oder einen entsprechenden Vergleichsoperator über das Ergebnis der List-Comprehension laufen lassen, wozu aber alle Lösungen komplett berechnet worden sein müssen.

Wir werden hingegen in Abschnitt 6.1 einen Suchalgorithmus vorstellen, mit dem in Curry die Wege nur soweit berechnet werden müssen, bis man erkennen kann, daß sie länger sein werden als der bisher kürzeste, wodurch möglicherweise eine Menge Reduktionsschritte erspart werden können. Dieses Vorgehen, mit dem wir während der Berechnung den Suchraum beschneiden, ist mit List-Comprehensions nicht formulierbar, ohne das Programm selber zu ändern. Der Vorteil unseres Ansatzes für die eingekapselte Suche wird aber unter anderem darin liegen, daß die Anwendung verschiedener Suchstrategien möglich ist, ohne das Programm anpassen zu müssen.

Es ist mir jedoch im Rahmen dieser Arbeit nicht gelungen, eine allgemeingültige Vorschrift zu finden, nach der List-Comprehensions in äquivalente Ausdrücke in Curry übersetzt werden können, die mit Hilfe der eingekapselten Suche dieselbe Liste als Ergebnis berechnen. In konkreten Einzelfällen ist dies oftmals möglich, aber bei der Verwendung von iterativen Generatoren, Mustern in der linken Seite von Generatoren und diagonalisierenden List-Comprehensions werden diese Ausdrücke sehr schnell ausgesprochen kompliziert und unübersichtlich. Es ist daher zu überlegen, ob zukünftige Versionen von Curry die direkte Formulierung von List-Comprehensions ermöglichen sollten. Wir werden in Abschnitt 6.8 näher auf dieses Problem eingehen.

3.4 Das Konzept der Einkapselung durch lokale Räume

Wir wollen zwei Formen der eingekapselten Suche in Curry verwirklichen: einen Suchoperator ähnlich wie in Oz [32] und ein Committed-Choice-Konstrukt ähnlich wie in AKL [19]. Beide Formen können wir realisieren, indem wir Berechnungen in lokale Räume einschließen.

In Anlehnung an Oz verwenden wir im folgenden den Begriff des *Berechnungsraums*, den man sich als ein Tuppel (σ, e) vorstellen kann, in Curry dargestellt durch einen Antwortausdruck $\sigma\,[]\,e$, wobei e den zu reduzierenden Ausdruck und σ die bei der Reduktion enstandene Substitution bezeichnen. Wir werden noch sehen, daß für das Konzept der eingekapselten Suche diese Form von Berechnungsräumen nicht ausreicht, weshalb wir ihre endgültige Definition erst in Abschnitt 4.2.1 vornehmen werden.

Das momentane Reduktionsverhalten von Curry zeigt Abbildung 3.1. Jeder Antwortausdruck entspricht einem *globalen* Raum, der bei Fehlschlag der Berechnung gelöscht wird. Die `add`-Funktion aus Beispiel 2.3 wird hier und in allen folgenden Beispielen durch Narrowing ausgewertet und reduziert nach demselben Prinzip wie die `append`-Funktion, der Wert des ersten Parameters ist also entscheidend für die Regelauswahl.

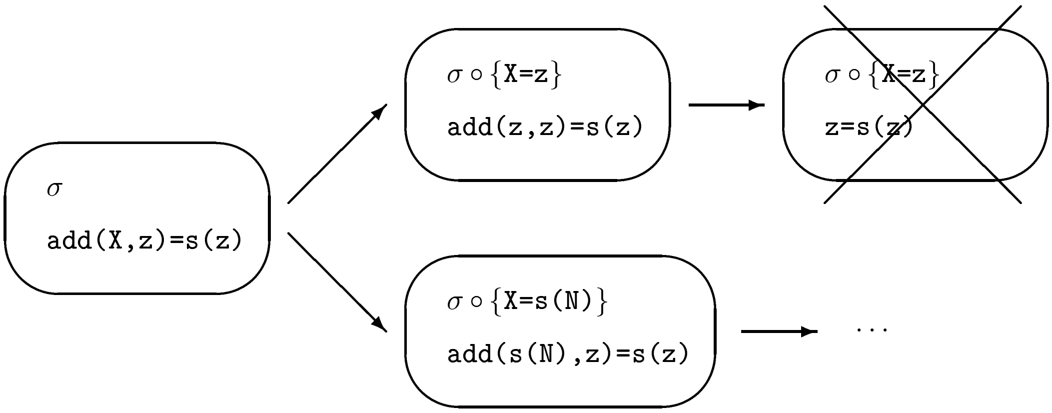


Abbildung 3.1: Verdoppelung des globalen Raumes durch einen nichtdeterministischen Schritt

Wie in Oz wird nun die Berechnung eines Ausdrucks in einen *lokalen* Raum **eingekapselt** (Abbildung 3.2). Nichtdeterministische Schritte sowie der Fehlschlag einer Berechnung haben

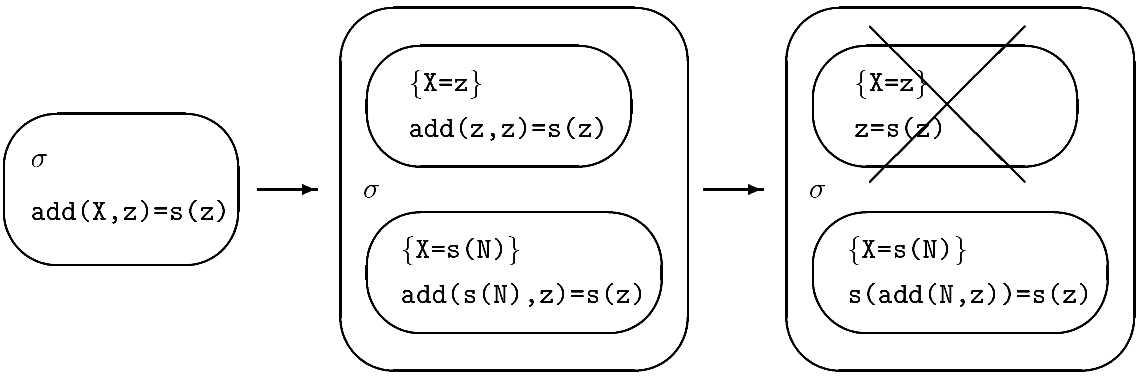


Abbildung 3.2: Einkapselung durch lokale Räume

daher keine Auswirkung auf den globalen Raum. Wir werden in Abschnitt 5.1 eine Reduktionssemantik für lokale Räume vorstellen, mit deren Hilfe wir sowohl den Suchoperator als auch die Commited Choice realisieren können. Dem Benutzer selbst wird die Existenz lokaler Räume verborgen bleiben, da sie nur von vordefinierten Sprachelementen wie dem Suchoperator oder der Committed-Choice verwendet werden dürfen.

3.5 Der Suchoperator in Curry

In Logikprogrammen tritt eine Suche auf, wenn ein Ausdruck eine freie Variable enthält und man Lösungen, d.h. Bindungen, für die Variable sucht, mit denen der Ausdruck beweisbar ist. Dasselbe Prinzip verwenden wir in Curry bei der Reduktion von Constraints, und deshalb übergeben wir dem Suchoperator einen Constraint c und eine *Suchvariable* X und fragen nach den Lösungen von X bezüglich c. Etwas Ähnliches haben wir beim `bagof`-Prädikat in Prolog kennengelernt, wo man eine Suchvariable und einen booleschen Ausdruck übergibt. Die Erweiterung auf mehrere Suchvariablen ist mit Hilfe von Tuppeln und Listen möglich, wie wir in Beispiel 3.11 sehen werden.

Zwar können in Curry beliebige Ausdrücke freie Variablen enthalten, aber da wir jeden Ausdruck e in einen Gleichheitsconstraint $e = X$ umwandeln können, stellt die Verwendung von Constraints keine wirkliche Einschränkung dar. Sie ist zudem aus folgenden Gründen naheliegend:

1. Neben einem Narrowingschritt ist die Constraintgleichheit die einzige Möglichkeit, Variablenbindungen durchzuführen.

2. Normalerweise findet eine Suche in Logikprogrammen statt, und wie wir in Abschnitt 2.2.4 gesehen haben, werden logische Programme in Curry entweder direkt durch Constraints ausgedrückt oder mit Hilfe von Prädikaten L_i, die jedoch auch als Gleichheitsconstraints der Form L_i`=true` formuliert werden.

3. Extravariablen, d.h. Variablen, die nicht in einer linken Regelseite vorkommen, treten meistens in der Bedingung einer Regel auf (vgl. Abschnitt 4.1), so daß man bei der Erfüllbarkeitsprüfung eine Suche durchführen muß. Nach der Definition aus Abschnitt 2.2.1 werden Bedingungen durch Constraints ausgedrückt.

Wir definieren den Suchoperator in Curry als eine Funktion `try`[1] mit dem folgenden Typ:

```
try :: (A -> constraint) -> list(A -> constraint).
```

Da durch die Einkapselung verschiedene Lösungen für eine Variable innerhalb eines globalen Raums berechnet werden sollen, müssen wir die Suchvariable abstrahieren, um Bindungskonflikte zu vermeiden, und übergeben deshalb wie in Oz den Constraint und die Variable in Form einer Lambda-Abstraktion. `try(\`X`->`c`)` erzeugt einen lokalen Raum, in dem c durch deterministische Schritte so lange reduziert wird, bis einer der folgenden Fälle eintritt:

- Die Berechnung schlägt fehl, d.h. der Constraint ist unerfüllbar. Dieser Fehlschlag wird durch eine leere Liste als Ergebnis von `try` signalisiert.

- Der Constraint konnte zu `{}` reduziert werden, war also erfüllbar, und hat dabei für X einen Konstruktorterm e als Lösung berechnet. `try` liefert diese Lösung in einer einelementigen Liste der Form `[\`X`->`X`=`e`]` zurück.

- Es kann nur noch ein nichtdeterministischer Reduktionsschritt durchgeführt werden. In dem Fall werden die n verschiedenen möglichen Berechnungswege in einer Liste der Form `[\`X`->`$c_1, \ldots,$`\`X`->`c_n`]` zurückgegeben, wobei jeder Constraint c_i einen der Berechnungswege enthält.

[1] Der Name `try` wurde von Herbert Kuchen vorgeschlagen und signalisiert, daß der Versuch, eine Lösung zu finden, nur bis zum Auftreten eines nichtdeterministischen Schritts durchgeführt wird, also nicht immer eine vollständig berechnete Lösung zurückgeliefert wird.

- Der Constraint kann keinen Reduktionsschritt durchführen, ist aber auch noch nicht zu `{}` reduziert worden. In dem Fall suspendiert der Aufruf der `try`-Funktion.

Wir werden im folgenden die `append`-Funktion genau wie die `add`-Funktion *immer* durch Narrowing ausgewerten.

Beispiel 3.3 (Fehlschlag)
Der Constraint `{append([1,2],X)=[3,4,5,6]}` besitzt keine Lösung und führt daher zu folgender Reduktion:

try(\X -> {append([1,2],X)=[3,4,5,6]}) $\rightsquigarrow$ ... $\rightsquigarrow$ []

Statt eines Fehlschlags erhalten wir also ein Ergebnis in Form der leeren Liste und können nun eine entsprechende Fehlerbehandlung aufrufen, wie wir in Beispiel 3.8 sehen werden. $\diamond$

Beispiel 3.4 (Erfolgreiche Reduktion)
Der Constraint `{append([1,2],X)=[1,2,3,4]}` besitzt eine eindeutige Lösung {`X=[3,4]`}, und daher reduziert `try` wie folgt:

try(\X -> {append([1,2],X)=[1,2,3,4]}) $\rightsquigarrow$ ... $\rightsquigarrow$ [\X -> {X=[3,4]}] $\diamond$

Um eine Lösung weiterzuverarbeiten, beispielsweise auszugeben, muß man zuerst den Lambda-Ausdruck applizieren, beispielsweise durch eine Funktion der folgenden Form:

```
f if {try(\X -> {append([1,2],X)=[1,2,3,4]})=[L] /\ L@Y} = Y.
```

Der Aufruf von `f` liefert damit als Ergebnis `[3,4]`, indem zunächst die von `try` berechnete Lambda-Abstraktion an `L` gebunden wird und dann durch die Applikation `(\X -> {X=[3,4]})@Y` die Substitution `{Y=[3,4]}` erstellt wird. Dieses Vorgehen erscheint ziemlich umständlich, denn man könnte ja auch direkt den Wert `[3,4]` von `try` zurückliefern lassen. Bei partiellen Strukturen ist das Lambda-Format jedoch von Vorteil und wird aus diesem Grund auch in Oz verwendet (vgl. Abschnitt 5.2.3).

Beispiel 3.5
Die folgende Reduktion von `try` ist in dieser Form nicht korrekt, denn wir werden noch eine Änderung der operationalen Semantik im nächsten Kapitel vornehmen, wodurch `try` ein anderes Ergebnis berechnen wird (vgl. Beispiel 4.3).

try(\X -> {X=add(s(z),N)}) $\rightsquigarrow$ ...$\rightsquigarrow$ [\X -> {X=s(N)}]

Als Lösung für `X` erhalten wir die Menge aller natürlichen Zahlen ≥ 1. Bezeichnen wir die Lambda-Abstraktion durch `L`, so können wir nicht nur durch `L@Y` die Lösungsmenge an `Y` binden, sondern auch Elemente auf ihr Enthaltensein in der Menge testen. `L@s(s(z))` ist erfüllbar, also liegt die 2 in der Lösungsmenge, während `L@z` fehlschlägt.

Mehrere Elemente, beispielsweise 1 und 2, auf ihr Enthaltensein in der Lösungsmenge zu testen ist aber nicht möglich (zur Verdeutlichung werden beide Applikationen in einem Schritt ausgewertet):

L@s(s(z)) /\ L@s(s(s(z))) $\rightsquigarrow$ s(z)=s(N) /\ s(s(z))=s(N)

Dieser Constraint ist nicht erfüllbar, da er zwei verschiedene Bindungen für `N` vornehmen soll. In Beispiel 4.3 in Abschnitt 4.1 werden wir eine Lösung des Problems vorstellen. $\diamond$

Beispiel 3.6 (Nichtdeterministische Reduktion)
Die nichtdeterministische Reduktion von `add(X,z)=s(z)`, die wir in Abbildung 3.1 dargestellt haben, wird durch den Suchoperator eingekapselt (auch diese Reduktion ist noch nicht ganz korrekt, vgl. Beispiel 3.10).

```
try(\X -> {add(X,z)=s(z)}) ⇝ ... ⇝ [\X -> {X=z    /\ add(z,z)   =s(z)},
                                    \X -> {X=s(N) /\ add(s(N),z)=s(z)}]
```

Statt einer globalen Verdoppelung des Raumes erhalten wir nun also beide möglichen Berechnungswege als Datenobjekte und können nun selber entscheiden, wie wir die Berechnung fortsetzen wollen, denn obwohl `X` in beiden Fällen gebunden wurde, ist die Berechnung noch nicht beendet, solange ein Restconstraint zu überprüfen ist. Um sowohl diesen Constraint als auch die für `X` erstellte Substitution in *einem* Datenobjekt zurückliefern zu können, wird die Substitution wieder in einen Gleichheitsconstraint zurückverwandelt. Da der Wert der Suchvariable möglicherweise noch nicht endgültig feststeht, so wie hier im zweiten Fall, müssen wir wieder Lambda-Abstraktionen verwenden, um Bindungskonflikte zu vermeiden. ◇

Beispiel 3.7
Verschiedene Reduktionswege können durch nichtdeterministische Funktionen auch ohne freie Variablen verursacht werden. In Beispiel 2.6 hatten wir die `kante`-Funktion definiert und gesehen, daß der Aufruf `kante(a)` drei Ergebnisse `b`, `c` und `d` liefert. Die Berechnung einer nichtdeterministischen Funktion kann durch einen Gleichheitsconstraint eingekapselt werden.

```
try(\X -> {X=kante(a)}) ⇝ ... ⇝ [\X -> {X=b}, \X -> {X=c}, \X -> {X=d}]
```

◇

Die Verwendung von Lambda-Abstraktionen ermöglicht ein einheitliches Format für die ursprüngliche Anfrage, die Zwischenergebnisse und die Endergebnisse. Ein Zwischenergebnis kann daher erneut als Anfrage verwendet werden, was eine sehr flexible Verwendung des Suchoperators erlaubt, wie das folgende Beispiel zeigt. Obwohl wir die operationale Semantik von `try` noch nicht definiert haben, können wir bereits einen Suchalgorithmus angeben, der alle Lösungen für eine Anfrage berechnet.

Beispiel 3.8
Wir übergeben der Funktion `all` eine Lambda-Abstraktion und erhalten alle möglichen Lösungen in einer Liste zurückgeliefert.

```
all(G) = evalResult(try(G))
       where
         evalResult([])      = [];
         evalResult([S])     = [S];
         evalResult([A,B|C]) = concat(map(all,([A,B|C]))).
```

Wenn der Suchoperator für die Anfrage `G` keine oder eine Lösung findet, wird entsprechend eine leere Liste oder eine Liste mit der Lösung zurückgeliefert. Ansonsten werden für jeden möglichen Berechnungsweg alle Lösungen durch erneute Anwendung der `all`-Funktion berechnet und die Ergebnislisten durch die Funktion `concat` zu einer Liste vereinigt. Wir werden die Arbeitsweise in Abschnitt 6.1 noch genauer erklären und dabei auch die Regeln für `concat` angeben.

In Beispiel 2.6 haben wir die Funktion `weg(X,Y)` zur Wegesuche in einem Graph definiert. Die Menge aller Wege von `a` nach `d` können wir durch `all` in einer Liste zurückgeben lassen:

```
all(\X -> {X=weg(a,d)})
⇝ ... ⇝ [\X -> {X=[a,b,c,d]}, \X -> {X=[a,d]}, \X -> {X=[a,b,d]}]
```

◇

Die Arbeitsweise des Suchoperators verhindert also eine Vervielfältigung des globalen Raumes durch einen nichtdeterministischen Reduktionsschritt und stellt Zwischenergebnisse und

Lösungen als Datenobjekte dar, was sowohl die Programmierung von Suchalgorithmen (Abschnitt 6.1) als auch Operationen auf gefundenen Lösungen zuläßt, beispielsweise einen Vergleich zwischen zwei Lösungen. Fehlschläge werden ebenfalls angefangen und durch eine leere Liste signalisiert. Damit erfüllt unser Suchoperator alle Forderungen, die wir in Abschnitt 3.1 an die eingekapselte Suche gestellt haben.

Einen kurze Gegenüberstellung mit dem Suchoperator in Oz nehmen wir nach der Definition der operationalen Semantik von `try` in Abschnitt 5.2 vor.

3.6 Globale und lokale Variablen

Da wir nur von der Suchvariable abstrahiert haben, kann die Bindung anderer Variablen zu Widersprüchen führen, da verschiedene Bindungen innerhalb eines globalen Raumes vorliegen können.

Beispiel 3.9

```
all(\X -> add(Y,X)=s(z)) ⤳ ... ⤳ [\X -> {Y=z /\ X=s(z)},
                                   \X -> {Y=s(z) /\ X=z}].
```

Wenn wir die beiden Lösungen mit verschiedenen Variablen applizieren, um wie in Beispiel 3.4 die gefundenen Bindungen für `X` zurückzuliefern, kommt es jedoch zu einem Konflikt (zur Verdeutlichung werden beide Applikationen in einem Schritt ausgewertet):

```
(\X -> {Y=z /\ X=s(z)})@A /\ (\X -> {Y=s(z) /\ X=z})@B
⤳      Y=z /\ A=s(z)     /\         Y=s(z) /\ B=z
```

Dieser Constraint ist natürlich nicht erfüllbar, da `Y` an zwei verschiedene Werte gebunden werden müßte. ◇

Die Lösung liegt in der Einführung eines Existenzquantors der Form `local Y in` e (siehe Abschnitt 4.1). In einem durch `try` erzeugten lokalen Raum dürfen nur *lokale Variablen* gebunden werden, d.h. solche, die innerhalb des Raumes durch `local` deklariert wurden. Alle anderen Variablen gelten als global, d.h. außerhalb des lokalen Raum deklariert, und dürfen deshalb nicht gebunden werden. Ein entsprechender Versuch führt zur Verzögerung der Berechnung, die erst fortgesetzt werden kann, wenn die Variable *außerhalb* des lokalen Raums gebunden wird. Um für jeden Raum die Menge der in ihm deklarierten Variablen protokollieren zu können, benötigen wir eine Erweiterung der operationalen Semantik, die wir zusammen mit der Einführung des `local`-Operators in Kapitel 4 vornehmen.

Beispiel 3.10

```
try(\X -> {add(Y,X)=s(z)}) ⤳ ⊥

try(\X -> {local Y in add(Y,X)=s(z)})
⤳ ... ⤳ [\X -> add(z,X)=s(z),
          \X -> local N in add(s(N),X)=s(z)]
```

Da der `local`-Operator ein Existenzquantor ist, interessiert uns nicht der Wert von `Y`, sondern nur, ob ein `Y` existiert, so daß der Constraint erfüllbar ist. Deshalb liefert `try` die Bindung lokaler Variablen nicht zurück. Die Variable `N` ist bei der Reduktion innerhalb des lokalen Raums aufgetreten und wird daher ebenfalls als lokale Variable betrachtet. Man beachte, daß es sich bei solchen Variablen immer um ganz neue handelt, da sie aus dem Muster eines definierenden Baums stammen und bei der Definition von *cse* in Abschnitt 2.3.4 dafür gesorgt wurde, daß eine definierender Baum mit neuen Variablen versehen wird, bevor er an *cs* übergeben wird. ◇

Beispiel 3.11
Die Verwendung mehrerer Suchvariablen ist mit Hilfe des `local`-Operators und Verwendung von Tuppeln ebenfalls möglich (mehrere lokale Variablen geben wir einfach in einer Liste hinter `local` an):

```
all(\X -> {local [Y,Z] in X=(Y,Z) /\ add(Y,Z)=s(z)}
⇝ ... ⇝ [\X -> {X=(z,s(z))},
          \X -> {X=(s(z),z)}]
```

Bezeichnen wir die erste Lösung mit `L`, dann ist das Extrahieren der berechneten Werte auf zwei Arten möglich. `L@A` führt zur Substitution `{A=(z,s(z))}` während `L@(A,B)` als Ergebnis `{A=z,B=s(z)}` liefert. Genauso könnte man statt Tuppeln auch Listen oder andere Konstrukte verwenden. ◇

Es erscheint zunächst naheliegender, mehrstellige Lambda-Abstraktionen zu verwenden, um mehrere Suchvariablen anzugeben:

```
all(\(X,Y) -> {add(X,Y)=s(z)} ⇝ ... ⇝ [\(X,Y) -> {X=z /\ Y=s(z)},
                                        \(X,Y) -> {X=s(z) /\ Y=z}]
```

Ein Existenzquantor wäre aber dennoch notwendig, da man ansonsten Variablen, die, wie in Beispiel 3.10, während der Berechnung auftreten, als Parameter zur Lambda-Abstraktion hinzufügen müßte. Abgesehen davon, daß dadurch die Stelligkeit der Ausgabefunktion ungewiß ist, würde man im globalen Raum den direkten Zugriff auf die lokalen Variablen des lokalen Raums ermöglichen, obwohl deren Wert weder verlangt wird noch von Interesse ist. Eine Existenzquantifizierung dieser Variablen ist daher sinnvoll, eine Deklaration im Kopf einer Funktion entspräche jedoch einer Allquantifizierung, wie wir in Abschnitt 4.1.1 sehen werden.

Außerdem müßte `try` beliebigstellige Funktionen als Parameter akzeptieren, so daß man den Typ von `try` in `A -> [A]` ändern und die Überprüfung auf korrekte Verwendung mit Funktionen mit Ergebnistyp `constraint` gesondert vornehmen müßte. Bei dem von uns definierten Typ kann diese Überprüfung einfach durch den Typchecker geschehen.

3.7 Committed-Choice

Die Einführung des Suchoperators dient der Kontrolle des „don't know"-Nichtdeterminismus, der nach einem nichtdeterministischen Schritt einen disjunktiven Ausdruck erzeugt und jede mögliche Lösung berechnet.

Es gibt jedoch auch Fälle, in denen man eine Menge von Reduktionswegen angeben will, um auf verschiedene Situationen reagieren zu können, das Verfolgen *eines* Weges jedoch ausreichend ist. Wenn beispielsweise in nebenläufigen Sprachen ein Prozeß mehrere Nachrichten von verschiedenen Quellen erhalten kann, benötigt er entsprechend mehrere Behandlungsroutinen. Sobald er eine Nachricht erhalten hat, führt er die entsprechende Routine aus und verwirft alle anderen. Dieses Vorgehen bezeichnet man als „dont't care"-Nichtdeterminismus, der in Curry durch ein Committed-Choice-Konstrukt zur Verfügung gestellt wird. Die Syntax ist jedoch in der folgenden Form noch nicht ganz vollständig, und wir werden sie erst in Abschnitt 5.3 endgültig definieren.

$$\begin{array}{ll} \texttt{choice} & \{c_1\}\ \texttt{->}\ e_1; \\ & \vdots \\ & \{c_n\}\ \texttt{->}\ e_n \end{array}$$

$\{c_i\}$ `->` e_i ist ein bewachter Ausdruck mit einem Constraints c_i als Wächter und einem beliebigen Ausdruck e_i als Rumpf, wobei für $k \in \{1, \ldots, n\}$ alle e_k denselben Typ besitzen müssen, der dem Zieltyp des `choice`-Ausdrucks entspricht. Wie bei bedingten Regeln kann statt eines Constraints auch ein boolescher Ausdruck als Wächter verwendet werden.

`choice` prüft die Erfüllbarkeit *aller* Wächter, indem es die c_i zu `{}` zu reduzieren versucht. Gelingt dies für ein c_j, so wird der gesamte `choice`-Ausdruck durch den zugehörigen Rumpf e_j des bewachten Ausdrucks ersetzt. Sind mehrere c_i erfüllbar, hängt es von der Implementierung der Committed-Choice ab, welcher Rumpf als Ergebnis zurückgeliefert wird. Wenn festgestellt wird, daß ein Wächter unerfüllbar ist, wird der bewachte Ausdruck gelöscht. Der `choice`-Ausdruck schlägt fehl, falls alle Ausdrücke gelöscht wurden, und suspendiert, wenn die Erfüllbarkeit der Wächter, die nicht gelöscht wurden, momentan nicht entscheidbar ist.

Beispiel 3.12
Die Funktion `merge` zur Vereinigung zweier Listen kann mit Hilfe von `choice` formuliert werden:

```
merge :: list(A) -> list(A) -> list(A).
merge(L1,L2) = choice {L1=[]}    -> L2;              1
                      {L2=[]}    -> L1;              2
                      {L1=[E|R]} -> [E|merge(R,L2)]; 3
                      {L2=[E|R]} -> [E|merge(L1,R)]. 4
```

`merge` ist eine nichtdeterministische Funktion, denn beispielsweise gibt es für den Aufruf `merge([1],[2])` insgesamt 6 verschiedene Reduktionswege, da zu jedem Zeitpunkt mehr als ein Wächter erfüllbar ist:

$\texttt{merge([1],[2])} \overset{3}{\leadsto} \texttt{[1|merge([],[2])]} \overset{1}{\leadsto} \texttt{[1,2]}$
$\texttt{merge([1],[2])} \overset{3}{\leadsto} \texttt{[1|merge([],[2])]} \overset{4}{\leadsto} \texttt{[1,2|merge([],[])]} \overset{1}{\leadsto} \texttt{[1,2]}$
$\texttt{merge([1],[2])} \overset{3}{\leadsto} \texttt{[1|merge([],[2])]} \overset{4}{\leadsto} \texttt{[1,2|merge([],[])]} \overset{2}{\leadsto} \texttt{[1,2]}$
$\texttt{merge([1],[2])} \overset{4}{\leadsto} \texttt{[2|merge([1],[])]} \overset{2}{\leadsto} \texttt{[2,1]}$
$\texttt{merge([1],[2])} \overset{4,3,1}{\leadsto} \texttt{[2,1]}$
$\texttt{merge([1],[2])} \overset{4,3,2}{\leadsto} \texttt{[2,1]}$

Welche Lösung `merge` berechnet, hängt davon ab, welcher der erfüllbaren Wächter von `choice` als erster vollständig reduziert wird. In einer sequentiellen Implementierung wird dies normalerweise bei gleichlautenden Aufrufen immer derselbe sein, aber bei einer parallelen Auswertung aller Wächter könnte `merge([1],[2])` tatsächlich beide Ergebnisse auf die jeweils drei verschiedenen Arten berechnen.

Eine parallele Auswertung der verschiedenen Wächter würde eine einfache Realisierung einer fairen Strategie ermöglichen, bei der ein in endlich vielen Schritten erfüllbarer Wächter seine Reduktion auch dann erfolgreich beenden kann, wenn die Auswertung anderer Wächter suspendiert oder eine unendliche Berechnung zur Folge hat. Aber auch in sequentiellen Implementierungen kann eine faire Strategie ohne allzu großen Aufwand verwirklicht werden, wie wir in Abschnitt 5.3 zeigen.

Funktionen, die mit Committed-Choice arbeiten, können also nichtdeterministisch sein, und daher gelten bei Verwendung von `choice` die Vollständigkeitsresultate aus [11] für unsere Reduktionsfunktion $\xrightarrow{RN}$ nicht mehr. In Abschnitt 2.3.4 haben wir aber bereits erwähnt, daß auch für nichtdeterministische Funktionen eine deklarative Semantik angegeben werden kann [7].

Wir werden bei der Definition der operationalen Semantik für `choice` die Reduktion der Wächter in lokale Räume einkapseln, um einerseits eine gegenseitige Beeinflussung der verschiedenen Auswertungen und andererseits eine Vervielfältigung des globalen Raums zu verhindern. Da `choice` nur ein Ergebnis berechnen soll, müssen, nachdem ein Wächter erfolgreich reduziert werden konnte, alle anderen bewachten Ausdrücke gelöscht werden. Eine solche Kontrolle über die bewachten Ausdrücke ist aber nur innerhalb eines Berechnungsraums möglich, da eine globale Vervielfältigung nicht rückgängig gemacht werden kann. Abbildung 3.3 zeigt das grundsätzliche Reduktionsprinzip, wobei wir den zu dem Wächter gehörenden Rumpf an den lokalen Raum angehängt haben. In Abschnitt 5.3.1 stellen wir eine Erweiterung der lokalen Räume vor, die dies gestattet. Wieso wir die Substitution des lokalen Raums auf den Rumpf anwenden, sie aber nicht in den globalen Raum zurückliefern, erklären wir in Abschnitt 5.3.

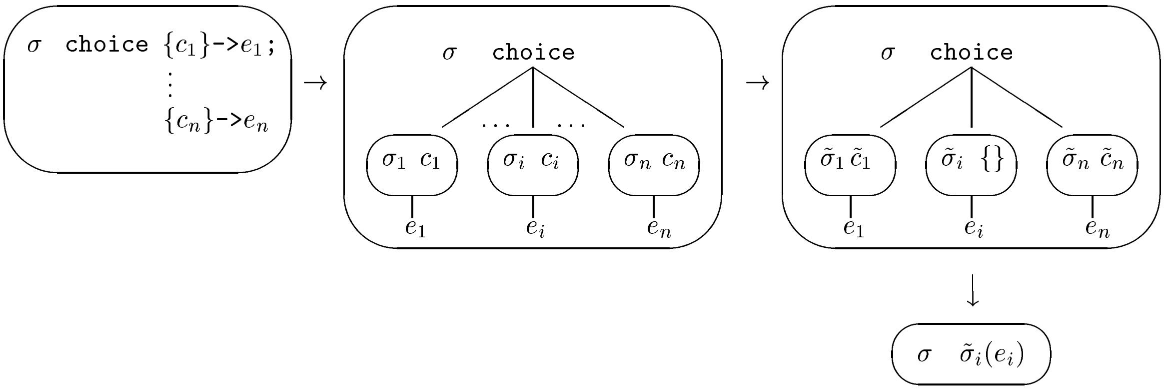


Abbildung 3.3: Eingekapselte Berechnung durch Committed-Choice

Durch die Verwendung lokaler Räume ist auch eine einfache Unterscheidung lokaler und globaler Variablen möglich, die wir genau wie beim Suchoperator auch für die Committed-Choice vornehmen müssen. Stellen wir uns die `merge`-Funktion als einen Prozeß vor, der zwei verschiedene Kommunikationskanäle `L1` und `L2` abhört. Sobald eine der Variablen gebunden wird, entspricht das dem Senden einer Nachricht, und `merge` reagiert darauf durch eine entsprechende Routine. Die Wächter des `choice`-Konstrukts müssen daher auf die Bindung von `L1` und `L2` durch andere Prozesse warten und dürfen sie nicht selbständig vornehmen.

Beispiel 3.13

```
merge(L1,L2)=L /\ L1=[1,2]
```

Der erste Wächter `{L1=[]}` (siehe Beispiel 3.12) könnte theoretisch `L1` an die leere Liste binden und wäre dadurch erfüllt, so daß `merge` zu `L2` reduzieren könnte. Die Nachricht, die anschließend gesendet wird, hat aber nicht die Form einer leeren Liste und führt daher zu einem Widerspruch der Form `[]=[1,2]`. Daher wird die Auswertung der `merge`-Funktion verzögert, bis eine Bindung einer der Variablen von außen erfolgt:

```
merge(L1,L2)=L /\ L1=[1,2]
⇝ {L1=[1,2]}▯merge([1,2],L2) /\ {} ⇝ ... ⇝ {L1=[1,2]}▯[1,2,L2]
```

◇

Im Gegensatz zu `L1` und `L2` sollen aber die Variablen `E` und `R` im dritten und vierten Wächter gebunden werden, um die übergebenen Listen aufzuspalten. Durch den Aufruf `merge([1,2],L2)` erhält der dritte Wächter die Form `{[1,2]=[E|R]}` und muß `E` an 1 und `R` an `[2]` binden, damit die Reduktion mit seinem Rumpf fortgesetzt werden kann.

Wir lassen daher auch bei der Committed-Choice die Deklaration lokaler Variablen zu und sorgen dafür, daß in den lokalen Räumen, in denen wir die Wächter reduzieren, nur diese Variablen gebunden werden dürfen. Wo genau Existenzquantoren verwendet werden dürfen und welche unterschiedlichen Wirkungen sie haben, erläutern wir erst in Abschnitt 5.3 bei

Angabe der korrekten Syntax und der operationalen Semantik des `choice`-Konstrukts. An dieser Stelle betrachten wir nur das Beispiel der `merge`-Funktion, deren korrekte Formulierung wie folgt lautet:

```
merge(L1,L2) = choice {L1=[]}                    -> L2;
                      {L2=[]}                    -> L1;
                      local [E,R] in {L1=[E|R]} -> [E|merge(R,L2)];
                      local [E,R] in {L2=[E|R]} -> [E|merge(L1,R)].
```

Also dürfen `L1` und `L2` im Gegensatz zu `E` und `R` bei der Reduktion der Wächter nicht gebunden werden. `merge` kann jedoch die Suspension aufgrund der Verwendung des `choice`-Konstrukts so lange hinauszögern, bis wirklich beide Argumente Variablen sind, da andernfalls immer ein Wächter gefunden wird, der reduzieren kann.

Beispiel 3.14
Wir betrachten bei den folgenden Reduktionen immer nur eine Möglichkeit, indem wir die Prüfung der Wächter von oben nach unten vornehmen.

```
merge(L1,L2) ⇝ ⊥

merge([1,2,L1],[3,4,L2])
⇝ ... ⇝ [1,2|merge(L1,[3,4,L2]] ⇝ ... ⇝ [1,2,3,4|merge(L1,L2)] ⇝ ⊥
```

Durch die nebenläufige Konjunktion kann mit Hilfe der `merge`-Funktion in der Tat eine Art Sender-Receiver-Beziehung darstellt werden, beispielsweise durch den Aufruf

```
merge(L1,L2)=L /\ L1=[1|L4] /\ L2=[2|L3] /\ L3=[3] /\ L4=[4] ,
```

bei dem `merge` nacheinander verschiedene Zahlenwerte erhält, diese sofort verarbeitet, dann wieder suspendiert und auf den nächsten Wert wartet. Auf ähnliche Weise modelliert AKL durch Verwendung von Variablen als Kanäle die Kommunikation zwischen Prozessen durch Datenströme [19]. ◇

Man beachte, daß nach den bisherigen Vorschriften im Rumpf eines bedingten Ausdrucks im Gegensatz zum Wächter die Bindung beliebiger Variablen erlaubt ist.

Beispiel 3.15

```
f(X,Y) = choice {X=1} -> {Y=X};
                {X=2} -> {Y=X*X}.
```

Der Aufruf `f(X,Y)` suspendiert, während `f(1,Y)` die Substitution {`Y=2`} und `f(2,Y)` die Substitution {`Y=4`} berechnet. Nach der Auswertung der Wächter wird die Reduktion des entsprechenden Rumpfs also wie die jedes anderen Ausdrucks ohne zusätzliche Einschränkungen durchgeführt. Eine Einkapselung oder Trennung zwischen lokalen und globalen Variablen ist nicht mehr notwendig, da wir uns nun für *einen* Reduktionsweg entschieden haben und diesen im globalen Raum weiterverfolgen. ◇

Die Struktur des `choice`-Konstrukts erinnert an bedingte Regeln und man könnte auf die Idee kommen, `merge` in der Form

```
merge_bed(L1,L2) if {L1=[]}    = L2
                 if {L2=[]}    = L1
                 if {L1=[E|R]} = [E|merge_bed(R,L2)]
                 if {L2=[E|R]} = [E|merge_bed(L1,R)].
```

zu formulieren. Im Unterschied zu `merge` reduziert `merge_bed` jedoch nach dem Prinzip des „don't know"-Nichtdeterminismus und liefert daher *alle* Lösungen in einem disjunktiven Ausdruck zurück:

$$\texttt{merge\_bed([1],[2])} \rightsquigarrow \ldots \rightsquigarrow \texttt{[1,2] | [1,2] | [1,2] | [2,1] | [2,1] | [2,1]}$$

Die Committed-Choice stellt also ein wirklich neues Reduktionskonzept zur Verfügung, daß nicht durch andere Sprachelemente simuliert werden kann. Es ermöglicht zudem, wie wir am Beispiel der beiden Merge-Funktionen gesehen haben, eine einfache Einkapselung nichtdeterministischer Reduktionen, denn während `merge_bed` eine Vervielfältigung des globalen Berechnungsraums bewirkt, läuft aufgrund der Verwendung lokaler Räume für die Auswertung der Wächter die gesamte Reduktion von `merge` innerhalb *eines* globalen Raums ab, so daß `merge` beispielsweise im Zusammenhang mit Ein-/Ausgabe-Operationen verwendet werden kann (vgl. Abschnitt 3.1.4).

Die operationale Semantik des `choice`-Konstrukts definieren wir in Abschnitt 5.3, und in Abschnitt 5.3.3 nehmen wir einen kurzen Vergleich mit der Realisierung von „don't-care"-Nichtdeterminismus durch Committed-Choice in den Sprachen AKL und Oz vor.

Kapitel 4

Die Einführung lokaler Variablen

Um lokale Variablen in Curry zu ermöglichen, muß zum einen ein Existenzquantor zur Deklaration solcher Variablen eingeführt werden und zum anderen die operationale Semantik aus Abschnitt 2.3 derart erweitert werden, daß sie in der Lage ist, zwischen lokalen und globalen Variablen zu unterscheiden.

4.1 Der Existenzquantor

4.1.1 Quantifizierung der Variablen in Regeln

In einer Regeldefinition sind die Variablen der linken Seite allquantifiziert. Die Regel

```
append([Y|Ys],Z) = [Y|append(Ys,Z)]. ,
```

die wir zur der Definition der `append`-Funktion in Abschnitt 2.2.4 angegeben haben, ist als

$\forall$ {Y,Ys,Z}: `append([Y|Ys],Z) = [Y|append(Ys,Z)].`

zu lesen, denn die Regel bedeutet, daß für *alle* Bindungen der Variablen `Y, Ys` und `Z` der Ausdruck `append([Y|Ys],Z)` zum Ausdruck `[Y|append(Ys,Z)]` reduziert werden kann.[1]

Anders als in funktionalen Sprachen können in logischen Sprachen jedoch auch freie Variablen in einer Anfrage auftreten und gelten dann als existenzquantifiziert, da man fragt, ob Bindungen für diese Variablen existieren, so daß die Anfrage erfüllbar ist. Beispielsweise ist die Anfrage

```
{append(X,Y)=[1,2,3,4]}
```

in der Form

$\exists$ {X,Y}: `{append(X,Y)=[1,2,3,4]}`

zu lesen. In Curry entspricht eine solche logische Anfrage einem Constraint, denn wir suchen bei der Reduktion eines Constraints nach Variablenbelegungen, mit denen er erfüllbar ist (vgl. Abschnitt 2.2.3).

Logische Sprachen erlauben auch die Verwendung von *Extravariablen*, die zwar in der Bedingung, aber nicht in der linken Regelseite auftreten. In Abschnitt 2.2.2 haben wir erwähnt, daß dies in Curry ebenfalls möglich ist. Betrachten wir beispielsweise die Definition des `member`-Prädikats als boolesche Funktion. Man beachte, daß `append` durch Narrowing ausgewertet werden muß, damit `member` in der folgenden Form überprüfen kann, ob ein Element in einer Liste auftritt.

[1] Natürlich müssen die Typvorschriften eingehalten werden, d.h. `Y, Ys` und `Z` müssen an Ausdrücke gleichen Typs gebunden werden.

```
member(E,L) if {append(Xs,[E|Ys])=L} = true.
```

Diese Regel bedeutet, daß für alle Variablen `E` und `L` der Ausdruck `member(E,L)` zu `true` reduzieren kann, wenn Variablen `Xs` und `Ys` existieren, so daß der Constraint `{append(Xs,[E|Ys])=L}` erfüllbar ist.

Die Bedingung einer Regel ist also nicht anderes als eine Anfrage, und wir müssen, um sie zu erfüllen, nach einer Bindung ihrer freien Variablen suchen. Somit sind Extravariablen in einer Bedingung existenzquantifiziert, und die Regel kann in der Form

```
∀{E,L}: member(E,L) if {∃{Xs,Ys}: append(Xs,[E|Ys])=L} = true.
```

gelesen werden.

In Curry dürfen also innerhalb eines Constraints existenzquantifizierte Variablen verwendet werden. Ihr Sichtbarkeitsbereich ist auf den Constraint beschränkt, d.h. diese Variablen sind außerhalb des Constraints nicht bekannt, und man bezeichnet sie daher auch als *lokale Variablen* des Constraints.[2] Um für die Realisierung der eingekapselten Suche eine Unterscheidung zwischen lokalen und globalen Variablen zu ermöglichen, führen wir als neues Sprachkonstrukt einen Existenzquantor ein, der die explizite Deklaration lokaler Variablen erlaubt und aufgrund der vorhergegangenen Erklärungen nur in Constraints verwendet werden darf.

Wir geben nun die vollständige Syntax von Constraints in Form einer EBNF an (vgl. Anhang C.2).

Definition 4.1 (Kontextfreie Syntax von Constraints)

$$
\begin{array}{rcl}
\textit{Constraint} & ::= & \{\ [\texttt{local}\ [\ \textit{VarId}_1\ ,\ \ldots\ \textit{VarId}_k\ ]\ \texttt{in}]\ \textit{ConstrExpr}_1\ ,\ \ldots\ \textit{ConstrExpr}_n\ \} \\
& & \hfill k > 0,\ n \geqslant 0 \\
\textit{ConstrExpr} & ::= & \textit{Expr} = \textit{Expr} \\
& | & \textit{Expr}
\end{array}
$$

□

Ein Constraint ist eine Konjunktion (dargestellt durch das Komma) von Constraintausdrücken mit einem optionalen Existenzquantor *am Anfang* der Deklaration. Ein Constraintausdruck ist entweder ein Gleichheitsconstraint oder ein Ausdruck (vgl. EBNF in Anhang C.2), beispielsweise eine Variable oder ein Funktionsaufruf, und muß den Zieltyp `constraint` besitzen, was durch den Typchecker überprüft wird. Natürlich kann der Ausdruck selbst auch wieder ein Constraint sein, wodurch die Schachtelung von Existenzquantoren möglich wird (Beispiel 4.2). Wenn nur *eine* Variable lokal deklariert werden soll, dürfen die Listenklammern auch weggelassen werden.

Beispiel 4.1
Die folgenden Verwendungen des Existenzquantors sind zulässig (wir benutzen in den Beispielen den `/\`-Operator statt des Kommas für die nebenläufige Konjunktion).

```
f1(Y)  = {local X in X=z /\ Y=add(X,z)}.
f2(C)  if {local X in p(X) /\ C} = {local X in X=1}.
f3(Y)  = {local X in X=Y /\ {local Y in X=Y}}.
```

Der Parameter `C` von `f2` muß vom Typ `constraint` sein, ebenso wie die Funktion `p`. Die Funktion `f3` zeigt eine geschachtelte Verwendung von Constraints. Man beachte, daß bei einer Schachtelung jeder Constraint wiederum in geschweifte Klammern eingefaßt werden muß! Die folgende Definition ist daher nicht zulässig:

[2] In Abschnitt 4.1.4 werden wir sehen, daß in manchen Fällen lokale Variablen auch außerhalb eines Constraints sichtbar sein sollten, aber zunächst beschränken wir uns auf die Verwendung innerhalb eines Constraints.

```
f4(Y) = {local X in X=Y /\ local Y in X=Y}.
```

◇

Die Auswirkungen von Extravariablen für die Vollständigkeit der Reduktion wird ausführlich in [10] untersucht. Wir beschäftigen uns daher im folgenden nur mit den Erweiterungen der operationalen Semantik, die durch die Einführung eines Existenzquantors und die Behandlung von Extravariablen notwendig werden.

4.1.2 Die operationale Semantik des Existenzquantors

Die operationale Semantik von `local` $[X_1, \ldots, X_n]$ `in` c, wobei c für eine Konjunktion von Constraint-Ausdrücken (*ConstrExpr*) steht, soll folgende Punkte verwirklichen:

- Die Sichtbarkeit der X_i wird auf c beschränkt und überdeckt dabei die Sichtbarkeit von Variablen gleichen Namens, die außerhalb dieser Deklaration verwendet werden. Die Verwendung des Variablennamens X_i außer- und innerhalb von c bezeichnet also zwei verschiedene Variablen (vgl. Beispiel 4.2).

- Die X_i werden zur Menge der im aktuellen Berechnungsraum deklarierten Variablen hinzugefügt. Die operationale Semantik für die Verwirklichung dieser Idee stellen wir in Abschnitt 4.2.1 vor.

Zur Angabe der operationalen Semantik benötigen wir den Begriff eines freien (oder sichtbaren) Vorkommens einer Variable.

Definition 4.2 (Freies Variablenvorkommen)

1. Ein Vorkommen einer Variable X in einem Ausdruck e heißt frei, wenn es nicht in einem Teilterm $e|_p$ der Form $e|_p = \{\texttt{local}\ [.., X, ..]\ \texttt{in}\ c\}$ auftritt. Die Menge aller Variablen, die frei in e vorkommen, bezeichnen wir mit $free(e)$.

2. Seien e, e' Ausdrücke. $e[X/e']$ bezeichnet den Ausdruck, der entsteht, wenn in e alle freien Vorkommen der Variablen X durch e' ersetzt werden.

□

Die operationale Semantik geben wir in Abbildung 4.1 durch eine zusätzliche Regel für die *cse*-Funktion aus Abschnitt 2.3.4 an. Wir erreichen die Einschränkung des Sichtbarkeitsbereichs

$$cse(\texttt{local}\ [X_1, \ldots, X_n]\ \texttt{in}\ c) \;=\; \{id [] c[X_i / X_{i_{fresh}}]\} \quad \text{für } i = 1, \ldots, n \text{ mit } X_{i_{fresh}} \text{ neue Variablen}$$

Abbildung 4.1: Operationale Semantik des Existenzquantors

durch eine Ersetzung der deklarierten durch ganz neue Variablen, die somit in der gesamten Berechung nur innerhalb des Constraints bekannt sind und angesprochen werden können. Die Überdeckung der Sichtbarkeit von Variablen, die außerhalb des Constraints unter demselben Namen verwendet werden, ist dadurch ebenfalls gegeben, da diese Namen nach der Umbenennung nicht mehr in dem Constraint vorkommen.

Beispiel 4.2

```
{local X in X=z  /\ Y=add(X,z)}
⤳  X₁=z /\ Y=add(X₁,z)
⤳ ... ⤳ {X₁=z,Y=z}
```

```
-----

             Y=3 /\ {local X in X=Y  /\ {local Y in Y=X}}
⇝{Y=3}          []{local X in X=3  /\ {local Y in Y=X}}
⇝{Y=3}          []           X₁=3 /\ {local Y in Y=X₁}
⇝{Y=3,X₁=3}     []                   {local Y in Y=3}
⇝{Y=3,X₁=3}     []                          Y₁=3
⇝{Y=3,X₁=3,Y₁=3}

-----

{local X in X=1 /\ {local X in X=2}}  ⇝ ... ⇝ {X₁=1,X₂=2}
```

◇

Beispiel 4.3
Wir können nun das Problem aus Beispiel 3.4 lösen. Dort hatten wir

```
try(\X -> {X=add(s(z),N)}) ⇝ ... ⇝ [\X -> {X=s(N)}]
```

berechnet, die Applikation der Lösung auf `s(z)` und `s(s(z))` hatte jedoch zu einem Fehlschlag geführt. Die Deklaration von `N` als lokale Variable ermöglicht die mehrfache Applikation.

```
try(\X -> {local N in X=add(s(z),N)}) ⇝...⇝ [\X -> {local N in X=s(N)}]
```

Bei jeder Applikation der Lambda-Abstraktion, die wir wieder mit `L` bezeichnen, schafft der Existenzquantor eine neue lokale Variable (zur Verdeutlichung werden beide Applikationen parallel ausgewertet):

```
{L@s(z) /\ L@s(s(z))}
⇝ {local N in s(z)=s(N)} /\ {local N in s(s(z))=s(N)}
⇝             s(z)=s(N₁)  /\            s(s(z))=s(N₂)
```

Der Constraint ist somit erfüllbar, und wir wissen, daß `s(z)` und `s(s(z))` in der Lösungsmenge liegen. ◇

Erforderliche Änderungen

Die Verwendung von Existenzquantoren erfordert die Anpassung der Definition 2.3 einer Substitution aus Abschnitt 2.1, denn es dürfen nur noch freie Vorkommen von Variablen in einem Term substituiert werden.

Beispiel 4.4
Die Anwendung der Substitution {`X=1`} auf den Constraint

```
{p(X) /\ {local X in q(X)}}
```

soll als Ergebnis

```
{p(1) /\ {local X in q(X)}}
```

liefern und nicht etwa

```
{p(1) /\ {local 1 in q(1)}}
```

◇

Damit die Anwendung von Substitutionen auf Ausdrücke durch die *replace*- und die *cse*-Funktion aus Abschnitt 2.3.4 nicht zu solchen Fehlern führt, müssen wir den ersten Teil von Definition 2.3 wie folgt ersetzen:

Definition 4.3 (Neue Definition einer Substitution)
Eine Substitution ist eine Abbildung $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C}, \mathcal{X})$ mit endlichem Träger $dom(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$. Sie läßt sich durch $\sigma(e) := e[x/\sigma(x)]$ mit $e \in \mathcal{T}(\Sigma, \mathcal{X})$ zu einer Abbildung auf der Menge der Terme (Ausdrücke) fortsetzen. □

Ebenso muß die Implementierung von Lambda-Abstraktionen, die wir in Abschnitt 2.3.4 kurz vorgestellt haben, geändert werden. `\`X `->` e wird durch `lambda`$(X_1, \ldots, X_n)$ ersetzt, wobei nun $\{X_1, \ldots, X_n\} = \mathbf{free}(e) \backslash \{X\}$ statt $\{X_1, \ldots, X_n\} = \mathbf{var}(e) \backslash \{X\}$ gelten muß, da existenzquantifizierte Variablen in e außerhalb der Lambda-Abstraktion nicht sichtbar sind.

Ein letzter Punkt ist zu beachten, den wir jedoch nur erwähnen und nicht explizit in den Reduktionsfunktionen aufführen. Wenn bei Verwendung eines *branch*-Baumes bei der *cs*-Funktion aus Abschnitt 2.3.4 der Fall $e|_p = f(e_1, \ldots, e_n)$ auftritt, müssen wir den Funktionsaufruf zuerst auswerten und rufen daher erneut die *cse*-Funktion auf. Durch Einführung des Existenzquantors kann es jedoch auch vorkommen, daß

$e|_p$ = `local [`X_1`,...,`X_n`] in` c

gilt. Der Einfachheit halber betrachten wir daher den Existenzquantor auch als eine Funktion, beispielsweise der Form `localVars([`X_1`,...,`X_n`],`c`)`, so daß seine Reduktion durch den Fall $e|_p = f(e_1, \ldots, e_n)$ abgedeckt wird. Dasselbe gilt auch für die Konstrukte `searchSpace` und `choiceSpaces`, die wir in Kapitel 5 einführen werden.

4.1.3 Bindung von Extravariablen durch eine Normalform

Da der Existenzquantor optional ist, darf die Bedingung einer Regel immer noch Extravariablen enthalten, die nicht explizit als lokal deklariert wurden.

Beispiel 4.5
```
f1(Z) if {local X in append(X,Y)=Z} = true.
f2(Z) if {local [X,Y] in append(X,Y)=Z} = true.
```

Aus Benutzersicht verhalten sich beide Funktionen gleich, allerdings weisen sie ein unterschiedliches Reduktionsverhalten auf, da nur in der zweiten Regel `Y` durch eine frische Variable ersetzt wird. Dies ist jedoch nicht weiter schlimm, solange wir keine Extravariablen in der rechten Seite berücksichtigen müssen (siehe Abschnitt 4.1.4). ◇

Für die Protokollierung der Variablen, die in einem Berechnungsraum deklariert sind, müßte man jedoch bei Verwendung von `f1` eine Unterscheidung zwischen `X` und `Y` vornehmen. `X` könnte durch den Existenzquantor zur Menge der deklarierten Variablen hinzugefügt werden. Für `Y` müßte man hingegen einen zusätzlichen Mechanismus schaffen, der jedoch die operationale Semantik verkomplizieren würde und durch die zusätzliche Arbeit auch einen negativen Einfluß auf das Laufzeitverhalten einer Implementierung haben könnte.

Nach Abschnitt 4.1.1 wissen wir jedoch, daß auch `Y` in `f1` durch einen impliziten Existenzquantor gebunden ist, und so sorgen wir für eine Gleichbehandlung aller Extravariablen, indem wir durch Erstellung einer Normalform den impliziten Existenzquantor in einen expliziten umwandeln.

Definition 4.4 (Vorläufige Normalform von bedingten Regeln)
Wir definieren die vorläufige Normalform einer bedingten Regel durch die Funktion nf_{bed}, die wir in Abbildung 4.2 angeben. Aus Platzgründen verwenden wir dort die Abkürzung $\overline{X}_n$ für eine Folge von Variablen der Form $X_1, \ldots, X_n$. Alle ungebundenen Extravariablen werden explizit deklariert. Man beachte, daß bei der Erweiterung des vorhandenen Existenzquantors im dritten Fall keine gleichen Variablennamen in $\overline{X}_n$ und $\overline{Y}_k$ auftreten können, da kein Y_i

$$nf_{bed}(l\ \texttt{if \{}c\texttt{\} =}\ r) \;=\; l\ \texttt{if \{}declare(free(c)\backslash var(l),\, c)\texttt{\} =}\ r$$

$$declare(V,c) \;=\; \begin{cases} c & \text{wenn } V=\emptyset \\ \texttt{local [}\overline{X}_n\texttt{] in } c & \text{wenn } c \neq \texttt{local [...] in } \tilde{c}, \\ & V=\{X_1,\ldots,X_n\} \\ \texttt{local [}\overline{X}_n,\overline{Y}_k\texttt{] in } \tilde{c} & \text{wenn } c = \texttt{local [}\overline{Y}_k\texttt{] in } \tilde{c} \\ & V=\{X_1,\ldots,X_n\} \end{cases}$$

Abbildung 4.2: Vorläufige Normalform einer bedingten Regel

frei in c vorkommt. Diese Normalform ist noch nicht endgültig und wird in Definition 4.7 im nächsten Abschnitt erweitert. □

Alle bedingten Regeln müssen durch einen Compiler oder Interpreter vor dem Erstellen der definierenden Bäume in ihre Normalform umgewandelt werden, damit während der Reduktion nur explizit gebundene Extravariablen auftreten.

Eine alternative Vorgehensweise wäre das Verbot von nicht deklarierten Extravariablen. Da jedoch nur bei Verwendung lokaler Räume durch den Suchoperator und die Committed-Choice die explizite Angabe lokaler Variablen notwendig ist (vgl. Abschnitte 3.6 und 3.7), erlauben wir aus Gründen der Benutzerfreundlichkeit auch weiterhin Extravariablen, die im Programm nicht durch ein `local` aufgeführt werden, und sorgen durch unsere Normalform für eine einfache Behandlung durch die operationale Semantik.

Beispiel 4.6

nf(`f1(Z) if {append(X,Y)=Z} = true`) =
`f1(Z) if {local [X,Y] in append(X,Y)=Z} = true`

nf(`f2(Z) if {local X in append(X,Y)=Z} = true`) =
`f2(Z) if {local [X,Y] in append(X,Y)=Z} = true`

nf(`f3 if {try(\X -> {add(Y,X)=s(z)})=L} = true`) =
`f3 if {local [Y,L] in try(\X -> {add(Y,X)=s(z)})=L} = true`

◇

Der `try`-Aufruf, den wir in Beispiel 3.9 zur Motivation für die Einführung des Existenzquantors vorgestellt haben, kann also ohne die explizite Deklaration von `Y` nicht reduzieren, da `Y` außerhalb der Lambda-Abstraktion deklariert wird und somit als globale Variable betrachtet wird. Da `Y` aber an anderer Stelle gar nicht auftritt, könnte man sich auch einen Algorithmus zur Erstellung der Normalform vorstellen, der Extravariablen in dem kleinstmöglichen Sichtbarkeitsbereich deklariert. Die Normalform von `f3` würde dann

```
f3 if {local [L] in try(\X -> {local [Y] in add(Y,X)=s(z)})=L} = true
```

lauten. Durch Verwendung eines solchen Algorithmus bräuchte der Benutzer keine Existenzquantoren anzugeben, so daß man auf ihre Einführung auf Sprachebene verzichten könnte und nur eine interne Verwendung notwendig wäre. Wir haben uns aus 2 Gründen gegen diese Lösung entschieden:

1. Der Algorithmus müßte bei geschachtelter Verwendung von Constraints, des Suchoperators und des `choice`-Konstrukts einen enormen Aufwand betreiben, um den kleinstmöglichen Sichtbarkeitsbereich festzustellen.

2. Durch einen Existenzquantor kann der Benutzer explizit darauf hinweisen, daß er nur nach der Existenz einer Variable fragt, an ihrem Substitutionswert also nicht interessiert ist, oder daß eine Variable nur innerhalb eines bestimmten lokalen Bereichs verwendet werden soll. Dies trägt zur Strukturierung und besseren Übersicht gerade bei geschachtelten Programmen bei, da auf eine regelweite Namenseindeutigkeit der Variablen nicht geachtet werden muß.

Wir verwenden also den einfachen Standard-Algorithmus, der jede ungebundene Extravariable in einer Bedingung am Anfang der Bedingung deklariert. Ist dies nicht erwünscht, muß eine explizite lokale Deklaration an anderer Stelle vorgenommen werden.

Eine Umbenennung der ursprünglich ungebundenen Extravariablen zur Laufzeit wäre eigentlich nicht erforderlich, da auf die Namenseindeutigkeit durch den Benutzer geachtet wurde. Eine entsprechende Unterscheidung kann deshalb in einer effizienten Implementierung vorgenommen werden, für die Definition einer möglichst einfachen operationalen Semantik verzichten wir jedoch darauf.

4.1.4 Extravariablen in rechten Regelseiten

In der Praxis ist die Verwendung von Extravariablen nicht nur in Bedingungen, sondern auch in rechten Regelseiten von Vorteil.

Beispiel 4.7
Man beachte, daß wir die `weg`-Funktion in Abschnitt 2.2.2 mit Hilfe der Funktion `(++)` zur Konkatenation zweier Listen anstelle von `append` definiert haben. Nur `(++)` ist in Curry vordefiniert und wird daher in allen größeren Beispielprogrammen verwendet, die sich auf der beigefügten Diskette befinden (vgl. Anhang A).

```
weg(X,Y)  if {X=Y}        = [X]
          if {kante(X)=Z} = [X] ++ weg(Z,Y).

last(L) if {append(Xs,[E])=L} = E.
```

Damit die Berechnung des letzten Elements einer Liste durch `last` möglich ist, muß wie schon bei der Definition von `member` in Abschnitt 4.1.1 die `append`-Funktion durch Narrowing ausgewertet werden. Hingegen wird `(++)` nach dem Residuation-Prinzip reduziert. ◇

Die Reduktion eines Ausdruck e durch eine bedingte Regel l `if` $\{c\}$ `=` r mit $\sigma(l) = e$ soll also nach dem Prinzip

$$e \;\rightsquigarrow\; \begin{cases} \{\tilde{\sigma} [\!] \tilde{\sigma} \circ \sigma(r)\} & \text{wenn } \sigma(c) \rightsquigarrow \ldots \rightsquigarrow \tilde{\sigma} [\!] \texttt{\{\}} \\ \emptyset & \text{sonst} \end{cases}$$

vorgenommen werden, die Variablenbindungen, die bei der Reduktion des Constraints erstellt wurden, sollen also auch weiterhin zur Verfügung stehen. Vor der Einführung von Existenzquantoren war dies mit unserer operationalen Semantik bei Realisierung der bedingten Regeln durch bewachte Ausdrücke ($c \Rightarrow r$) (siehe Abschnitt 2.3.4) möglich. Da Bedingung und rechte Seite beide Parameter der Funktion $\Rightarrow$ sind, können Variablenbindungen aus c einfach durch die *replace*-Funktion (Abschnitt 2.3.4) nach r gereicht werden.

Durch die Verwendung des `local`-Operators zur Quantifizierung von Extravariablen ist jedoch das Weiterreichen von Bindungen unabhängig von der Implementierung bedingter Regeln zunächst nicht mehr möglich.

Beispiel 4.8
Wenn wir `last([1,2])` aufrufen, sollte die Bedingung wie folgt reduzieren:

$$\texttt{\{append(Xs,[E])=[1,2]\}} \leadsto \texttt{...} \leadsto \{\texttt{Xs=[1],E=2}\}[]\texttt{\{\}}$$

Das Ergebnis von `last([1,2])` wäre also 2. Durch unsere Normalformerstellung kann diese Reduktion jedoch nicht in gewünschter Weise vorgenommen werden, denn die Normalform der `last`-Funktion lautet:

```
last(L) if {local [Xs,E] in append(Xs,[E])=L} = E
```

Die Sichtbarkeit der Variable `E` wird dadurch auf die Bedingung beschränkt und auch nur dort umbenannt. Wir verwenden nachfolgend die Darstellung bedingter Regeln durch bewachte Ausdrücke gemäß Abschnitt 2.3.4:

$$\begin{array}{ll}
\texttt{last([1,2])} & \leadsto \texttt{(\{local [E,Xs] in append(Xs,[E])=[1,2]\}} \;\Rightarrow\; \texttt{E)} \\
& \leadsto \texttt{(append(Xs}_1\texttt{,[E}_1\texttt{])=[1,2]} \;\Rightarrow\; \texttt{E)} \\
& \vdots \\
& \leadsto \{\texttt{Xs}_1\texttt{=[1],E}_1\texttt{=2}\}[]\texttt{(\{\}} \;\Rightarrow\; \texttt{E)} \\
& \leadsto \{\texttt{Xs}_1\texttt{=[1],E}_1\texttt{=2}\}[]\texttt{E}
\end{array}$$

Die Bindung für `E` kann also nicht aus der Bedingung in die rechte Seite gereicht werden. ◇

Man könnte das Problem lösen, indem man solche Variablen nicht durch ein `local` bindet, wodurch sie nicht umbenannt werden. Intern müßten wir dann aber einen zusätzlichen Mechanismus schaffen, um solche Variablen zur Menge der Variablen eines Berechnungsraums hinzufügen zu können. Außerdem sollte es der Vollständigkeit halber möglich sein, alle Extravariablen explizit deklarieren zu können, und deshalb verwenden wir eine Lösung, die es in ähnlicher Weise auch in den Sprachen Escher und Oz gibt.

Beispiel 4.9
Escher stellt ein `if-then-else`-Konstrukt zur Verfügung, bei dem lokale Variablen, die in der Bedingung deklariert werden, auch nach deren Prüfung sichtbar bleiben.

$$\texttt{if exists [}x_1,\ldots,x_n\texttt{]}\ \mathit{Cond}\ \texttt{then}\ F1\ \texttt{else}\ F2$$

Nach Prüfung der Bedingung wird der Sichtbarkeitsbereich der x_i entweder auf $F1$ oder $F2$ erweitert, je nachdem, ob die Bedingung erfüllt werden konnte oder nicht. Allerdings müssen dazu $F1$ und $F2$ genau wie die Bedingung vom Typ Boolean sein, denn nur dann kann man die Bedeutung des Konstrukts durch den disjunktiven Ausdruck

$$\texttt{(exists [}x_1,\ldots,x_n\texttt{] (}\mathit{Cond}\ \texttt{\&\&}\ F1\texttt{)) || (not exists [}x_1,\ldots,x_n\texttt{]}\ \mathit{Cond}\texttt{) \&\&}\ F1\texttt{))}$$

erklären. `&&` bezeichnet die nebenläufige Konjunktion, d.h. der rechte Ausdruck kann auch vor dem linken ausgewertet werden, `||` die Disjunktion.

In Oz können sowohl in disjunktiven als auch in bedingten Ausdrücken, die wir in den Abschnitten 5.2.3 und 5.3.3 vorstellen werden, mehrere bewachte Ausdrücke der Form

$$x_1 \ldots x_n\ \textsf{\textbf{in}}\ E\ \textsf{\textbf{then}}\ D$$

mit Wächter E, Rumpf D und den lokalen Variablen $x_1,\ldots,x_n$ verwendet werden. Die x_i sind sowohl in E als auch in D sichtbar. Dies könnte man durch eine Umformung des Rumpfs erklären, indem man D durch eine Konjunktion von E und D ersetzt, vor der die lokalen Variablen erneut deklariert werden:

$$\textsf{\textbf{local}}\ x_1 \ldots x_n\ \textsf{\textbf{in}}\ E\ D$$

Eine solche Konjunktion ist möglich, da E und D denselben Typ (Constraint) besitzen. Auf diese Weise wäre der Sichtbarkeitsbereich der x_i korrekt für den Wächter und den Rumpf definiert. Der Wächter E müßte dann natürlich zweimal ausgewertet werden, und daher wird man in der Praxis ein solches Vorgehen vermeiden. Es dient nur zur Erklärung, wieso die lokalen Variablen im Wächter und Rumpf sichtbar sind.

Man beachte, daß eine direkte Ersetzung des bewachten Ausdrucks durch die Konjunktion von E und D nicht möglich ist, da zunächst die Erfüllbarkeit aller Wächter, die in einem disjunktiven oder bedingten Ausdruck auftreten, untersucht werden muß, bevor der Ausdruck durch höchstens *einen* Rumpf ersetzt werden kann (vgl. Abschnitte 5.2.3 und 5.3.3). $\diamond$

In Curry wäre durch Verwendung der `seq`-Funktion aus Beispiel 2.11 zur sequentiellen Ausführung zweier Constraints eine ähnliches Vorgehen möglich, wenn die rechte Seite ein Constraint ist. Eine Regel

`f if {local X in` C_1`} =` C_2`.`

könnte in

`f = {local X in seq(`C_1, C_2`)}.`

umgeformt werden, wodurch `X` auch in der rechten Seite C_2 sichtbar wäre. Die rechte Seite einer Curry-Regel kann aber ein beliebiger Ausdruck sein, und daher erweitern wir die Semantik von bedingten Regeln durch Einführung eines Existenzquantors, der Variablen in der Bedingung und einer rechten Seite beliebigen Typs quantifiziert. Die neue Sytnax für bedingte Regeln geben wir in Form einer EBNF an (vgl.Anhang C.2).

Definition 4.5 (Kontextfreie Syntax für bedingte Regeln)

$$\begin{array}{rcl} \mathit{Equat} & ::= & \mathit{FunctionName}\ \mathit{Pattern}\ \mathit{CondExpr}\ [\texttt{where}\ \mathit{LocalDefs}] \\ \mathit{CondExpr} & ::= & \texttt{if}\ [\texttt{local}\ \texttt{[}\mathit{VarId}_1\ \texttt{,}\ \ldots\ \mathit{VarId}_k\ \texttt{]}\ \texttt{in}]\ \mathit{Constraint}\ \texttt{=}\ \mathit{Expr}\ [\mathit{CondExpr}] \\ & & \hfill k > 0 \end{array}$$

$\square$

Die lokalen Variablen $VarID_i$ sind im Bedingungsconstraint und nach dessen Reduktion auch in der rechten Seite sichtbar. Um den Unterschied der Wirkungsweise zu den bisherigen lokalen Deklarationen zu verdeutlichen, wird der Existenzquantor *außerhalb* der geschweiften Klammern angegeben, in die die Bedingung eingefaßt ist. Damit gilt auch weiterhin, daß eine lokale Variablendeklaration, die in geschweiften Klammern auftritt, auch nur innerhalb des geklammerten Bereichs sichtbar ist. Man beachte, daß in lokalen Definitionen einer Regel, die hinter `where` angegeben werden, im Gegensatz zu den Variablen der linken Regelseite grundsätzlich keine Extravariablen der Regel sichtbar sind, weder explizit noch durch die Erstellung der Normalform implizit deklarierte.

Formal korrekt ist eine solche Verwendung des Existenzquantors nicht, wenn die rechte Seite kein Constraint ist, aber wir lassen sie aus Gründen der Benutzerfreundlichkeit zu. Man beachte, daß auf diese Weise wie im Beispiel der `weg`-Funktion nichtdeterministische Funktionen entstehen können, wenn für die Extravariablen in der rechten Seite einer Regel verschiedene Bindungen berechnet werden können. In der funktional-logischen Sprache Babel sind daher eigentlich Extravariablen nur in der Bedingung einer Regel erlaubt [20], aber da sie wie gesehen in der Praxis auch in der rechten Seite vorteilhaft sind, können sie durch ein Compiler-Flag ermöglicht werden. Der Benutzer muß aber darauf achten, daß nur *ein* Wert für jede Extravariable in der rechten Seite berechnet werden kann. Die Deklaration der `weg`-Funktion wäre daher in Babel nicht auf dieselbe Weise möglich wie in Curry.

Beispiel 4.10
Die Funktionen aus Beispiel 4.7 können unter Verwendung des zusätzlichen Existenzquantors wie folgt definiert werden:

```
weg(X,Y)  if {X=Y}               = [X]
          if local Z in {kante(X)=Z} = [X] ++ weg(Z,Y).

last(L) if local E in {local Xs in append(Xs,[E])=L} = E.
```

Beim Aufruf von `last[1,2]` wird bei Reduktion der Bedingung die Variable E auch in der rechten Seite umbenannt. Dadurch wird eine Bindung von `E` in der Bedingung durch *replace* in den zweiten Parameter von $\Rightarrow$, die rechte Seite, weitergereicht:

```
last([1,2]) ⇝ (local E in {local Xs in append(Xs,[E])=[1,2]}  ⇒  E)
            ⇝ ({local Xs in append(Xs,[E1])=[1,2]}             ⇒  E1)
            ⋮
            ⇝ {Xs1=[1]}      ▯(E1=2  ⇒  E1)
            ⇝ {Xs1=[1],E1=2}▯({}     ⇒  2)
            ⇝ {Xs1=[1],E1=2}▯2
```

◇

Da ein `local`-Operator vor der Bedingung auch für die rechte Seite gilt, überdeckt er folgerichtig auch dort die Sichtbarkeit von Variablen aus der linken Regelseite. Beispielsweise soll bei Verwendung der Regel

```
f(X) if local X in {X=1} = [X].
```

der Aufruf `f(2)` zu `[1]` und nicht zu `[2]` reduzieren. Wir müssen daher Definition 4.2 eines freien Vorkommens einer Variable aus Abschnitt 4.1.2 erweitern.

Definition 4.6 (Freies Vorkommen)
Ein Vorkommen einer Variable X in einem Ausdruck e heißt frei, wenn es nicht in einem Teilterm $e|_p$ der Form $e|_p =$ `{local` $[.., X, ..]$ `in` c`}` auftritt und e nicht Teilterm des Constraints `{c}` oder der rechten Seite `r` einer bedingten Regel der Form

l `if local` $[.., X, ..]$ `in {`c`} =` r

ist. Die Menge aller Variablen, die frei in e vorkommen, bezeichnen wir mit $free(e)$. □

Die Verwendung des zusätzlichen Existenzquantors durch Benutzer ist nicht notwendig, denn wir werden nicht deklarierte Extravariablen, die in der Bedingung und der rechten Seite auftauchen, durch eine erweiterte Normalform sammeln. In Abschnitt 5.3 werden wir ein ähnliches Problem mit Extravariablen bei bestimmten Committed-Choice-Ausdrücken kennenlernen, für dessen Lösung die Verwendung eines zusätzlichen Existenzquantors zwingend erforderlich ist, weshalb wir ihn aus Gründen der Vollständigkeit auch bei bedingten Regeln ermöglichen. In der Praxis kann der Benutzer jedoch auf ihn verzichten.

Wir haben in der bisherigen Betrachtung zwei Fälle ausgelassen:

1. Extravariablen in rechten Regelseiten von bedingten Regeln, die nicht in der Bedingung auftreten,
2. Extravariablen in rechten Regelseiten von unbedingten Regeln.

Um keine zusätzliche Behandlung für diese Variablen einführen zu müssen, werden wir sie einfach vor der Bedingung der Regel deklarieren, wozu wir im zweiten Fall unbedingte in bedingte Regeln transformieren. Wenn die rechte Regelseite ein Constraint ist, könnten die Variablen auch dort aufgeführt werden, aber wir berücksichtigen diesen Fall bei der Definition der Normalform nicht, um eine möglichst einfache Formulierung zu erreichen.

Definition 4.7 (Normalform von Regeln)
Wir definieren die endgültige Normalform eine Regel durch die Funktion nf, die wir in Abbildung 4.3 angeben. Die Funktion *declare* wurde gegenüber Definition 4.4 aus dem vorigen Abschnitt nicht verändert. Aus Platzgründen verwenden wir wieder die Abkürzung $\overline{X}_n$ für eine Folge von Variablen der Form $X_1, \ldots, X_n$.

$$nf(l\ \texttt{if}\ bed\ \texttt{=}\ r) = \begin{cases} l\ \texttt{if}\ declare(V, \texttt{\{}c_{new}\texttt{\}})\ \texttt{=}\ r \\ \quad \text{wenn } bed = \texttt{\{}c\texttt{\}} \\ l\ \texttt{if}\ declare(V, \texttt{local [}\overline{X}_n\texttt{] in \{}c_{new}\texttt{\}})\ \texttt{=}\ r \\ \quad \text{wenn } bed = \texttt{local [}\overline{X}_n\texttt{] in \{}c\texttt{\}} \end{cases}$$

$$\text{mit} \quad \begin{array}{lcl} c_{new} & := & declare(free(bed) \backslash (var(l) \cup free(r)), c) \\ V & := & free(r) \backslash free(l) \end{array}$$

$$nf(l\ \texttt{=}\ r) = \begin{cases} l\ \texttt{=}\ r & \text{wenn } V = \emptyset \\ l\ \texttt{if}\ declare(V, \texttt{\{\}})\ \texttt{=}\ r & \text{sonst} \end{cases}$$

$$\text{mit} \quad V := free(r) \backslash free(l)$$

$$declare(V, c) = \begin{cases} c & \text{wenn } V = \emptyset \\ \texttt{local [}\overline{X}_n\texttt{] in}\ c & \text{wenn } c \neq \texttt{local [...] in}\ \tilde{c}, \\ & V = \{X_1, \ldots, X_n\} \\ \texttt{local [}\overline{X}_n, \overline{Y}_k\texttt{] in}\ \tilde{c} & \text{wenn } c = \texttt{local [}\overline{Y}_k\texttt{] in}\ \tilde{c} \\ & V = \{X_1, \ldots, X_n\} \end{cases}$$

Abbildung 4.3: Normalform von Regeln

Bei bedingten Regeln werden vor der Bedingung alle Extravariablen der rechten Seite deklariert, die nicht in der linken Seite auftreten, wodurch die Extravariablen, die in Bedingung *und* rechter Seite auftauchen, mit erfaßt werden. Ein eventuell schon vorhanderer Existenzquantor vor der Bedingung wird entsprechend erweitert.

Innerhalb der geschweiften Klammern der Bedingung werden alle freien Variablen deklariert, die nur in der Bedingung auftreten und deren Sichtbarkeit somit auf den Constraint c beschränkt werden kann. Man beachte, daß nach der neuen Definition eines freien Vorkommens weder in $free(bed)$ noch in $free(r)$ eine Variable aus $\overline{X}_n$ auftauchen kann.

Bei unbedingten Regeln werden eine immer erfüllbare Bedingung `{}` geschaffen und vor ihr die Extravariablen der rechten Seite deklariert. □

Beispiel 4.11

nf(`last(L) if {append(Xs,[E])=L} = E`) =
 `last(L) if local E in {local Xs in append(Xs,[E])=L} = E`

nf(`f(X) if local Y in {add(X,Y)=Z} = Z`) =
 `f(X) if local [Y,Z] in {add(X,Y)=Z} = Z`

nf(f(X) = add(X,Y)) =
  f(X) if local Y in {} = add(X,Y)

◇

Auf diese Weise werden durch die Normalform alle Extravariablen durch Existenzquantor deklariert, so daß wir in Abschnitt 4.2.1 nur eine sehr geringe Änderung der operationalen Semantik vornehmen müssen, um die lokalen Variablen eines Berechnungsraums zu protokollieren.

Die Berechnung einer Normalform ist ein einmaliger Aufwand beim Erstellen der definierenden Bäume und hat keinen negativen Auswirkungen auf die Laufzeiteffizienz eines Systems. Dem Benutzer ermöglicht sie auch weiterhin die Verwendung nichtdeklarierter Extravariablen, denn zwingend erforderlich ist die Angabe lokaler Variablen nur bei bestimmten Anwendungen der eingekapselten Suche. Eine Benutzung des Existenzquantors auch an anderer Stelle kann jedoch für eine bessere Übersicht und Strukturierung von Regeln mit stark geschachtelten Ausdrücken hilfreich sein.

4.1.5 Die Implementierung bedingter Regeln

Die operationale Semantik für die Reduktion bedingter Regeln muß sicherstellen, daß die gemeinsamen Extravariablen bei Reduktion des Existenzquantors der Bedingung auch in der rechten Seite umbenannt werden. Wir geben eine Möglichkeit für die mehrfach erwähnte Darstellung bedingter Regeln durch bewachte Ausdrücke an, die beispielsweise auch in der TasteCurry-Implementierung verwendet wird.

Definition 4.8 (Transformation in bewachte Ausdrücke)
Wir formalisieren die Transformation von Regeln in bewachte Ausdrücke in Verbindung mit der Erstellung der Normalform aus Definition 4.7 durch eine Funktion $trans$.

$$trans(eq) \;=\; \begin{cases} l \texttt{ = } (bed \;\Rightarrow\; e) & \text{wenn } eq \text{ eine Regel ist und } nf(eq) \;=\; l \texttt{ if } bed \texttt{ = } r \\ l \texttt{ = r} & \text{wenn } eq \text{ eine Regel ist und } nf(eq) \;=\; l \texttt{ = } r \end{cases}$$

Diese Definition gilt sowohl für $bed = \texttt{\{}c\texttt{\}}$ als auch für $bed = \texttt{local [}X_1, \ldots, X_n\texttt{] in \{}c\texttt{\}}$. □

Wenn jede Regel vor dem Eintrag in einen definierenden Baum durch $trans$ umgewandelt wird, kann die korrekte Behandlung von Extravariablen durch Hinzufügen einer einzigen Regel für die Reduktionsfunktion cse ermöglicht werden, die in Abbildung 4.4 dargestellt ist. Der erste

$$cse(\texttt{local [}X_1, \ldots, X_n\texttt{] in \{}c\texttt{\}} \;\Rightarrow\; e) \;=\; \{id [] c[X_i/X_{i_{fresh}}] \Rightarrow e[X_i/X_{i_{fresh}}]\}$$
$$\text{für } i = 1, \ldots, n \text{ mit } X_{i_{fresh}} \text{ neue Variablen}$$

Abbildung 4.4: Operationale Semantik bewachter Ausdrücke

Parameter eines bewachten Ausdrucks kann nach Definition von $trans$ nur die beiden Formen $\texttt{\{}c\texttt{\}}$ oder $\texttt{local [}X_1, \ldots, X_n\texttt{] in \{}c\texttt{\}}$ haben. Im ersten Fall wird $\Rightarrow$ als normale Funktion betrachtet und mit Hilfe des definierenden Baums für die Regel

$(\texttt{\{\}} \;\Rightarrow\; e) \texttt{ = } e$

ausgewertet. Im zweiten Fall erreichen wir durch die zusätzliche *cse*-Regel die geforderte Umbenennung der X_i im Constraint `{`c`}` *und* dem Ausdruck e der rechten Seite.

Diese einfache Sonderbehandlung des Existenzquantors vor der Bedingung ist möglich, weil die geschweiften Klammern, in die Constraints eingefaßt sind, bei der Transformation durch nf und $trans$ erhalten bleiben und man daher unterscheiden kann, ob der Existenzquantor *vor* oder *in* dem Bedingungsconstraint steht. Da es sich bei den Klammern aber eigentlich um ein syntaktisches Mittel handelt, kann man fordern, daß sie für die operationale Semantik nicht von Bedeutung sein dürfen. In diesem Fall muß man eine andere Kennzeichnung des `local`-Operators vor der Bedingung erreichen. Da er sowieso durch eine eigene *cse*-Regel reduziert wird, könnte man ihm beispielsweise einen anderen Namen geben, z.B. `localIn`, und die *cse*-Regel aus Abbildung 4.4 auf diesen Namen umändern.

In Tastecurry verwenden wir eine ganz andere Möglichkeit, die jedoch zu stark auf die Implementierung des Systems in Prolog zugeschnitten ist, um bei der Formulierung einer möglichst allgemeingültigen Theorie in diesem Kapitel von Interesse sein zu können. Wir werden sie in Abschnitt 7 vorstellen.

4.2 Erweiterung der Reduktionssemantik

In Abschnitt 3.6 haben wir die Notwendigkeit der Unterscheidung zwischen Variablen, die innerhalb und außerhalb eines lokalen Raumes deklariert wurden, kennengelernt und gefordert, daß eine Variable nur in dem Raum gebunden werden darf, in dem sie deklariert wurde. Eine Deklaration kann auf drei Art erfolgen:

1. Die Reduktion eines Existenzquantors schafft neue Variablen.
2. Bei einem Narrowingschritt wird wie in Beispiel 3.10 eine Variable an ein Muster aus einer linken Regelseite gebunden, das neue Variablen enthält.
3. Ein Gleichheitsconstraint bindet eine Variable an einen Konstruktorterm, wobei gemäß der Definition der Funktion *equal* in Abschnitt 2.3.4 die Argumente des Konstruktors durch neue Variablen ersetzt werden.

Diese neuen Variablen sollen in die Menge der im aktuellen Berechnungsraum deklarierten hinzugefügt werden. An anderer Stelle können keine neuen Variablen auftreten, da durch die Normalform, die wir in den vorigen Abschnitten entwickelt haben, jede Extravariable durch einen Existenzquantor deklariert wird. Die einzige zusätzliche Überprüfung muß für den zweiten Fall erfolgen, so daß die Erweiterung der operationalen Semantik aus Abschnitt 2.3.4 sehr gering bleiben kann. Folgende Änderungen müssen wir vornehmen:

1. Der globale Berechnungsraum muß um eine Komponente erweitert werden, nämlich um die Menge der deklarierten Variablen, die wir im folgenden mit V bezeichnen.
2. Die Reduktionsfunktionen *cse*, *cs* und *equal* müssen erweitert werden, so daß sie
   (a) neu deklarierte Variablen zu V hinzufügen,
   (b) Variablen, die aus dem zu reduzierenden Ausdruck verschwinden, aus V entfernen und
   (c) die Bindung von nicht in V enthaltenen Variablen verzögern.

   Die Entfernung von Variablen aus V ist nicht unbedingt notwendig, aber da nur die Variablen, die in dem noch zu reduzierenden Ausdruck auftreten, von Interesse sind, sollten auch nur diese in V gesammelt werden, um die Enthaltenseinsprüfung nicht unnötig zu verlangsamen.

Da wir lokale Räume erst in Abschnitt 5.1 einführen, wird jede Variable, die in der Berechnung auftaucht, in V liegen und es daher zu keiner Verzögerung kommen, so daß die Änderungen eigentlich nicht notwendig wären, solange wir eine Reduktion ohne lokale Räume durchführen[3]. Die Erweiterung der Semantik für den allgemeinen Fall ermöglicht uns jedoch die Verwendung derselben Reduktionsfunktionen für jeden globalen und lokalen Raum, auch bei beliebig tiefer Schachtelung lokaler Räume, so daß wir bei der Implementierung der lokalen Räume die *cse*-Funktion als eine „black box" betrachten können. Eine eventuelle Änderung von *cse*, beispielsweise durch eine echte parallele Verarbeitung der nebenläufigen Konjunktion, läßt also die operationale Semantik für lokale Räume unberührt.

4.2.1 Die neue Einzelschrittreduktion

Wir definieren zunächst den Begriff des globalen Raums.

Definition 4.9
Ein globaler Berechnungsraum ist ein Dreituppel

$$(V, \sigma, e) \ ,$$

wobei V eine Menge von Variablen, σ eine Substitution und e einen Ausdruck bezeichnen. □

In V sollen alle Variablen protokolliert werden, die während der Reduktion von e auftreten. Wir stellen den Berechnungsraum jedoch nicht explizit als Dreituppel, sondern in Form eines Antwortausdrucks dar, den wir daher ebenfalls neu definieren müssen.

Definition 4.10 (Antwortausdruck)
Ein *Antwort-Ausdruck* hat die Form $\exists V : \sigma [] e$, wobei V eine Menge von Variablen, σ eine Substitution und e einen Ausdruck bezeichnen. Ein Antwort-Ausdruck heißt *gelöst*, wenn e ein Datenterm ist. □

Zu Beginn der Berechnung müssen wir für die korrekte Initialisierung der Variablenmenge sorgen.

Definition 4.11 (Initialer Ausdruck)
Ein Anfrage e wird vor Beginn der Reduktion in den *initialen Ausdruck* $\{\exists free(e) : id [] e\}$ umgewandelt. □

Alle freien Variablen der Anfrage gelten also im globalen Raum als deklariert. Um die Menge V während der Reduktion anzupassen und die Prüfung bei Bindungen vornehmen zu können, übergeben wir V an *cs*, *cse* und *equal*.

Die Teile der Definition von *cse*, die in Abbildung 4.5 nicht aufgeführt sind, unterscheiden sich von der ursprünglichen Definition aus Abschnitt 2.3.4 nur dadurch, daß überall $cse(V, \ldots)$, $cs(V, \ldots)$ und $equal(V, \ldots)$ anstelle von $cse(\ldots)$, $cs(\ldots)$ und $equal(\ldots)$ stehen muß. Die einzige entscheidene Änderung betrifft den Existenzquantor, der nun nicht nur die Variablen umbenennen, sondern sie auch zur Menge der deklarierten Variablen des aktuellen Berechnungsraums hinzufügen muß. Die Regel für die Reduktion bewachter Ausdrücke führen wir der Vollständigkeit halber auch auf, es sei aber daran erinnert, daß bedingte Regeln in Curry auf beliebige Weise implementiert werden können und die Darstellung durch bewachte Ausdrücke nur eine Möglichkeit ist.

Parallel zu Abschnitt 2.3.4 ersetzt *replace* einen Teilterm $e|_p$ in e durch das Ergebnis der Reduktion $cse(V, e|_p)$, wobei die Menge V alle deklarierten Variablen des aktuellen Raums

[3] Eine effiziente Vorgehensweise, bei der die Variablenmenge nur bei Auftreten lokaler Räume protokolliert wird, verwenden wir in der Tastecurry-Implementierung (vgl. Abschnitt 7.1).

$$cse(V, c_1 \mathtt{/\backslash} c_2) = \begin{cases} \{\exists V : id \,[]\, \mathtt{\{\}}\} & \text{wenn } c_1 = \mathtt{\{\}} \text{ und } c_2 = \mathtt{\{\}} \\ \vdots & \end{cases}$$

$$\begin{aligned} cse(V, \mathtt{local}\ [X_1, \ldots, X_n]\ \mathtt{in}\ c) &= \{\exists V \cup \textstyle\bigcup_{i=1}^{n} \{X_{i_{fresh}}\} : id \,[]\, c[X_i / X_{i_{fresh}}]\} \\ &\quad \text{für } i = 1, \ldots, n,\ X_{i_{fresh}} \text{ neue Variablen} \\ cse(V, \mathtt{local}\ [X_1, \ldots, X_n]\ \mathtt{in}\ \mathtt{\{}c\mathtt{\}} \Rightarrow e) &= \{\exists V \cup \textstyle\bigcup_{i=1}^{n} \{X_{i_{fresh}}\} : id \,[]\, \\ &\quad c[X_i / X_{i_{fresh}}] \Rightarrow e[X_i / X_{i_{fresh}}]\} \\ &\quad \text{für } i = 1, \ldots, n,\ X_{i_{fresh}} \text{ neue Variablen} \end{aligned}$$

Abbildung 4.5: Reduktionsschritt für einen einzelnen (ungelösten) Ausdruck

enthalten muß. Sie wird während der Reduktion entsprechend angepaßt und kann daher als neue Variablenmenge zurückgeliefert werden, nachdem der Teilterm ersetzt wurde.

$$replace(e, p, d) = \begin{cases} \bigcup_{i=1}^{n} \{\exists V_i : \sigma_i \,[]\, \sigma_i(e)[e_i]_p\} & \text{wenn } d = \bigcup_{i=1}^{n} \{\exists V_i : \sigma_i \,[]\, e_i\} \\ \perp & \text{wenn } d = \perp \end{cases}$$

Bei der strikten Gleichheit führen wir die Regeln für *equal* nicht auf, da sie sich von der ursprünglichen Definition nur durch das Weiterreichen von V oder dadurch unterscheiden, daß vor dem Ergebnisausdruck eine unveränderte Variablenmenge in der Form $\exists V$ angegeben wird. Die wirklichen Änderungen betreffen *bind*.

Wenn beide Seiten eines Gleichheitsconstraints Variablen sind, hängt das Ergebnis von V ab. Wenn X nicht in V liegt, so kann trotzdem eine Bindung erfolgen, wenn Y in V liegt. Tritt keine der Variablen in der Menge auf, wird die Berechnung verzögert. Dasselbe gilt für den zweiten Fall, wenn X nicht in V liegt. Ansonsten wird X gebunden und aus V entfernt, und die neu eingeführten Variablen werden zur Menge der deklarierten hinzugefügt.

$$bind(V, X, Y) = \begin{cases} \{\exists V \setminus \{X\} : \{X\mathtt{=}Y\} \,[]\, \mathtt{\{\}}\} & \text{wenn } X \in V \\ \{\exists V \setminus \{Y\} : \{Y\mathtt{=}X\} \,[]\, \mathtt{\{\}}\} & \text{wenn } X \notin V,\ Y \in V \\ \perp & \text{sonst} \end{cases}$$

$$bind(V, X, c(e_1, \ldots, e_n)) = \begin{cases} \{\exists V \cup \{Y_1, \ldots, Y_n\} : \\ \quad \{X\mathtt{=}c(Y_1, \ldots, Y_n)\} \,[]\, Y_1\mathtt{=}e_1 \mathtt{/\backslash} \ldots \mathtt{/\backslash}\, Y_n\mathtt{=}e_n\} \\ \quad \text{wenn } X \in V,\ X \notin cv(c(e_1, \ldots, e_n)), \\ \qquad Y_i \text{ neue Variablen} \\ \emptyset \quad \text{sonst} \end{cases}$$

$$\begin{aligned} cv(X) &= \{X\} \\ cv(c(e_1, \ldots, e_n)) &= \textstyle\bigcup_{i=1}^{n} cv(e_i) \\ cv(f(e_1, \ldots, e_n)) &= \emptyset \\ cv(\mathtt{local}\ [X_1, \ldots, X_n]\ \mathtt{in}\ c) &= cv(c) \setminus \{X_1, \ldots, X_n\} \end{aligned}$$

Abbildung 4.6: Strikte Gleichheit

Man beachte, daß eine zusätzliche Regel für cv zur Berechnung der kritischen Variablen notwendig ist, da die Variable X nur in den *freien* Variablen von $c(e_1, \ldots, e_n)$ nicht auftreten darf, denn die lokal deklarierte Variablen unterscheiden sich auf jeden Fall von X. Diese cv-Regel kann übrigens nur zum Einsatz kommen, wenn in $c(e_1, \ldots, e_n)$ ein Konstruktor verwendet wird, der einen Constraint als Parameter haben kann[4]. Andernfalls können Existenzquantoren nur in Parametern von Funktionssymbolen auftreten, da nach Abschnitt 2.2.3 keine Seite eines Gleichheitsconstraints selbst wieder ein Constraint sein darf, und werden dann nach Definition von cv gar nicht untersucht.

$$cs(V, e, rule(l{=}r)) \quad = \quad \{\exists\, V : id \,[]\, \sigma(r)\} \quad \text{wenn } \sigma \text{ eine Substitution mit } \sigma(l) = e \text{ ist}$$

$$cs(V, e, or(\mathcal{T}_1, \mathcal{T}_2)) \quad = \quad \begin{cases} cs(V, e, \mathcal{T}_1) \cup cs(V, e, \mathcal{T}_2) & \text{wenn } cs(V, e, \mathcal{T}_1) \neq \bot \neq cs(V, e, \mathcal{T}_2) \\ \bot & \text{sonst} \end{cases}$$

$$cs(V, e, branch(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_k))$$

$$= \begin{cases} cs(V, e, \mathcal{T}_i) & \text{wenn } e|_p = c(e_1, \ldots, e_n) \text{ und } pat(\mathcal{T}_i)|_p = c(X_1, \ldots, X_n) \\ \emptyset & \text{wenn } e|_p = c(\cdots) \text{ und } pat(\mathcal{T}_i)|_p \neq c(\cdots),\ \text{für } i = 1, \ldots, k \\ \bot & \text{wenn } e|_p = X \text{ und } r = rigid \\ \bot & \text{wenn } e|_p = X,\ r = flex \text{ und } X \notin V \\ \bigcup_{i=1}^{k} \{\exists\, V_{new} :\ \sigma_i \,[]\, \sigma_i(e)\} & \text{wenn } e|_p = X,\ r = flex,\ X \in V,\ \sigma_i = \{X{=}pat(\mathcal{T}_i)|_p\} \\ & \text{und } V_{new} = V \setminus \{X\} \cup free(pat(\mathcal{T}_i)|_p) \\ replace(e, p, cse(V, e|_p)) & \text{wenn } e|_p = f(e_1, \ldots, e_n) \end{cases}$$

Abbildung 4.7: Reduktionsschritt für eine Funktionsapplikation

Die cs-Funktion in Abbildung 4.7 wurde gegenüber der ursprünglichen Definition nur für den Fall eines $branch$-Baumes grundlegend verändert. Wenn $e|_p$ einer Variablen X entspricht und zudem $r = flex$ gilt, darf eine Bindung nur erfolgen, wenn X in V enthalten ist, ansonsten wird die Berechnung suspendiert. Wenn die Bindung vorgenommen wird, ersetzt man X in e durch $pat(T_i)|_p$, so daß X entfernt werden kann und die Variablen, die in $pat(T_i)|_p$ enthalten sind, zu V hinzugefügt werden müssen.

Zur Verdeutlichung der Arbeitsweise betrachten wir die neuen Reduktionsabläufe einiger Anfragen. Sie werden zuerst in initiale Ausdrücke umgewandelt und dann durch cse reduziert. Wir lösen die geschweiften Klammern um einen Constraint auf, sobald wir die Anfrage in einen Antwortausdruck umgewandelt haben.

Beispiel 4.12
Bei der ersten Reduktion treten keine Variablen auf, V bleibt also leer.

$$\texttt{add(z,z)} \quad \leadsto \quad \exists\,\emptyset : id \,[]\, \texttt{add(z,z)} \quad \leadsto \quad \exists\,\emptyset : id \,[]\, \texttt{z}$$

Im folgenden Beispiel wird `X` protokolliert und nach der Bindung wieder entfernt.

$$\texttt{\{z=X\}} \quad \leadsto \quad \exists\,\{\texttt{X}\} : id \,[]\, \texttt{z=X} \quad \leadsto \quad \exists\,\emptyset : \{\texttt{X=z}\} \,[]\, \texttt{\{\}}$$

Lokal deklarierte Variablen werden erst zum Zeitpunkt der Reduktion des Existenzquantors zur Variablenmenge hinzugefügt.

[4] Ob ein solcher Konstruktor sinnvoll ist, sei dahingestellt. Theoretisch ist er jedoch erlaubt und muß daher berücksichtigt werden.

$$\begin{array}{l}\texttt{\{X=1 /\textbackslash\ \{local Y in Y=2\}\}}\\ \rightsquigarrow \exists\{\texttt{X}\} : id \;\,[]\texttt{X=1 /\textbackslash\ \{local Y in Y=2\}}\\ \rightsquigarrow \exists\,\emptyset \;\; : \{\texttt{X=1}\}[]\texttt{\{local Y in Y=2\}}\\ \rightsquigarrow \exists\{\texttt{Y}_1\} : \{\texttt{X=1}\}[]\texttt{Y}_1\texttt{=2}\\ \rightsquigarrow \ldots\end{array}$$

Die Variable `N` wird bei der folgenden Reduktion durch einen Narrowingschritt zur Variablenmenge hinzugefügt.

$$\begin{array}{l}\texttt{add(X,Y)}\\ \rightsquigarrow \exists\{\texttt{X,Y}\} : id \;\,[]\texttt{add(X,Y)}\\ \rightsquigarrow \exists\{\texttt{Y}\} \; : \{\texttt{X=z}\}[]\texttt{add(z,Y)} \quad | \quad \exists\{\texttt{Y,N}\} : \{\texttt{X=s(N)}\}[]\texttt{add(s(N),X)}\\ \rightsquigarrow \ldots\end{array}$$

Ebenso treten neue Variablen bei der Bindung an einen Konstruktorterm auf.

$$\begin{array}{l}\texttt{\{X=s(add(z,z))\}}\\ \rightsquigarrow \exists\{\texttt{X}\} : id \;\,[]\texttt{X=s(add(z,z))}\\ \rightsquigarrow \exists\{\texttt{Y}\} : \{\texttt{X=s(Y)}\}[]\texttt{Y=add(z,z)}\\ \rightsquigarrow \ldots\end{array}$$

◇

Variablen können aus einem Ausdruck verschwinden, ohne gebunden zu werden, und verbleiben in diesem Fall trotzdem in V.

Beispiel 4.13
Betrachten wir den Constraint aus Beispiel 4.3.

$$\begin{array}{l}\texttt{\{local N in X=add(s(z),N)\}}\\ \rightsquigarrow \exists\{\texttt{X}\} : id[]\texttt{local N in X=add(s(z),N)}\\ \vdots\\ \rightsquigarrow \exists\{\texttt{X,N}\} : id \;\,[]\texttt{X=s(N)}\\ \rightsquigarrow \exists\{\texttt{N}\} \; : \{\texttt{X=s(N)}\}[]\texttt{\{\}}\end{array}$$

In Kapitel 5 werden wir sehen, daß dies für die korrekte Arbeitsweise des Suchoperators und der Committed-Choice notwendig ist. ◇

In seltenen Fällen kann aber auch eine Variable in V verbleiben, die weder im Ausdruck noch in der Substitution auftaucht.

Beispiel 4.14
Die `leq`-Funktion haben wir in Beispiel 2.3 definiert. 0 (`z`) ist kleiner als alle natürlichen Zahlen, daher reduziert der folgende Aufruf zu `true`, ohne den zweiten Parameter zu beachten.

$$\texttt{leq(z,X)} \;\rightsquigarrow\; \exists\{\texttt{X}\} : id[]\texttt{leq(z,X)} \;\rightsquigarrow\; \exists\{\texttt{X}\} : id[]\texttt{true}$$

Ein solcher Fall tritt in der Praxis jedoch eher selten auf und beeinflußt auch nicht die Korrektheit der Berechnung. Wir belassen daher `X` in V, denn der kleine Effizienzverlust, der dadurch entsteht, rechtfertigt nicht den Aufwand, den wir für die Entfernung von `X` betreiben müßten.

◇

Im folgenden werden wir in den Beispielen die Menge der deklarierten Variablen nur aufführen, wenn sie für die Reduktion von Bedeutung ist.

Im nächsten Kapitel zeigen wir, wie mit Hilfe der in diesem Kapitel eingeführten Erweiterungen die operationale Semantik für die eingekapselte Suche definiert werden kann. Dabei werden wir auch Beispiele betrachten, bei denen eine Berechnung suspendiert, weil eine Variable gebunden werden soll, die nicht in der Menge der deklarierten Variablen liegt.

Kapitel 5

Reduktionssemantik für die eingekapselte Suche

Der Suchoperator und die Commited Choice benötigen dasselbe Reduktionverhalten lokaler Räume, müssen die Reduktionsergebnisse jedoch unterschiedlich verarbeiten. Es ist daher sinnvoll, die Berechnung in lokalen Räumen als „black box“ zu betrachten und durch eine eigene Reduktionsfunktion zu definieren. Der Suchoperator beispielsweise kann nun für den Rumpf der Lambda-Abstraktion einen lokalen Raum erzeugen und nach Ende der Reduktion das Ergebnis wieder in einen oder, bei einer nichtdeterministischen Reduktion, mehrere Lambda-Abstraktionen verpacken, ohne daß die eigentliche Auswertung des lokalen Raums für ihn sichtbar ist.

Dies hat den Vorteil, daß eine mögliche Änderung der Reduktionssemantik lokaler Räume, beispielsweise durch eine andere Definition des Stabilitätskriteriums, das wir in Abschnitt 5.1 einführen werden, die Arbeitsweise des Suchoperators und der Committed-Choice nicht berührt, solange die Schnittstelle unverändert bleibt. Zudem wird die weitere Verwendung lokaler Räume erleichtert, die möglicherweise für zukünftige Konzepte sinnvoll sein könnte.

5.1 Lokale Räume

Ein lokaler Berechnungsraum enthält im Prinzip dieselben Komponenten wie ein globaler, allerdings beschränken wir die Reduktion aus den in Abschnitt 3.5 angegebenen Gründen auf Constraints.

Definition 5.1 (Lokaler Raum)

- Ein nicht gelöster lokaler Berechnungsraum ist ein Dreituppel

  $$(V, \sigma, c)\ ,$$

  wobei V eine Menge von Variablen, σ eine Substitution und c einen Constraint bezeichnen.

- Ein gelöster lokaler Berechnungsraum ist ein Zweituppel

  $$(V, \sigma)\ ,$$

  wobei V eine Menge von Variablen und σ eine Substitution bezeichnen.

□

In V sollen nur die lokalen Variablen des Raums protokolliert werden, also diejenigen, die innerhalb des Raums deklariert werden. σ bezeichnet die Substitution, die während der Reduktions des Constraints entsteht.

Während wir globale Berechnungsräume nach Abschnitt 4.2.1 durch Antwortausdrücke darstellen, treten lokale Räume tatsächlich als Datenobjekte in Form eines Dreituppels in Berechnungen auf. Der Benutzer hat jedoch keinen direkten Zugriff auf diese Datenobjekte, denn die Erzeugung und Verwendung lokaler Räume sind nur durch den Suchoperator und das `choice`-Konstrukt möglich. Die Unterscheidung zwischen gelösten und nicht gelösten Räumen ist nicht unbedingt notwendig, aber für die Definition und Verwendung der Reduktionsfunktion *cslocal* in Abschnitt 5.1.1 hilfreich.

Ein lokaler Raum muß bestimmte Bedingungen erfüllen, die wir teilweise bereits in Kapitel 3 kennengelernt haben[1].

1. Ein Fehlschlag der Berechnung führt zu einem Fehlschlag des lokalen und nicht des globalen Raums.
2. Ein nichtdeterministischer Schritt führt zur Vervielfältigung des lokalen und nicht des globalen Raums.
3. Die lokalen Variablen sind außerhalb des lokalen Raums nicht sichtbar.
4. Der Versuch, globale Variablen zu binden, führt zur Suspension der Auswertung des lokalen Raums.

Diese Punkte müssen wir bei der Definition der operationalen Semantik sicherstellen. Prinzipiell soll die Reduktion mit Hilfe der erweiterten *cse*-Semantik aus Abschnitt 4.2.1 erfolgen, die die Realisierung der letzten beiden Punkte auf einfache Art und Weise ermöglicht. Zusätzlich müssen wir nur die Auflösung und den Fehlschlag eines lokalen Raumes überwachen.

Ein lokaler Raum schlägt fehl, wenn die Reduktion des enthaltenen Constraints c fehlschlägt, d.h. $cse(V, c)$ zu $\emptyset$ reduziert, wobei V die Menge der lokalen Variablen des Raums bezeichnet. Die Auflösung des lokalen Raums muß und darf nur in zwei Fällen erfolgen:

1. Wenn der enthaltene Constraint erfüllbar ist und zu `{}` reduziert werden konnte. In diesem Fall wird $(V, \sigma, \texttt{\{\}})$ durch den gelösten Raum (V, σ) ersetzt.
2. Wenn ein nichtdeterministischer Schritt erforderlich ist. Für jede mögliche Reduktionsrichtung wird ein eigener lokaler Raum erzeugt, und der bisherige Raum wird durch die Menge der neuen Räume ersetzt.

Die Vervielfältigung eines lokalen Raums kann je nach Implementierung einen sehr hohen Aufwand bedeuten, beispielsweise wenn für jeden lokalen Raum ein eigener Betriebssystem-Thread gestartet wird, und sollte daher vermieden werden, solange die Möglichkeit einer deterministischen Reduktion besteht. Zum Teil wird dies durch die operationale Semantik der nebenläufigen Konjunktion aus Abschnitt 2.3.4 sichergestellt, aber wie das folgende Beispiel zeigt, kann es trotzdem zu unnötigen Vervielfältigungen kommen.

Beispiel 5.1
Der Constraint `{Y=s(X) /\ add(X,z)=z}` besitzt die eindeutige Lösung $\{$`X=z, Y=s(z)`$\}$. Beim Aufruf

```
L=try(\X -> {Y=s(X) /\ add(X,z)=z}) /\ Y=s(z)
```

kann in dem von `try` erzeugten lokalen Raum nur der rechte Zweig der Konjunktion ausgeführt werden, denn für die Reduktion des linken Zweigs müßte `Y` gebunden werden, das aber nicht zu den lokalen Variablen des Raums gehört. Die Auswertung der `add`-Funktion verdoppelt jedoch den lokalen Raum, so daß der Suchoperator das Ergebnis

[1] Im folgenden unterscheiden wir immer nur zwischen einem lokalen und einem globalen Raum, ebenso nur zwischen lokalen und globalen Variablen. Natürlich ist eine Schachtelung lokaler Räume erlaubt und korrekterweise müßten wir daher von „lokalen Variablen und denen des übergeordneten Raums“ sprechen.

```
L=[\X -> {X=z /\ Y=s(z) /\ add(z,z)=z},
   \X -> {local N in X=s(N) /\ Y=s(s(N)) /\ add(s(N),z)=z}] /\ Y=s(z)
```

zurückliefert. Nur die erste Alternative kann jedoch erfolgreich sein, da wir im globalen Raum noch den Constraint `Y=s(z)` reduzieren werden. Die Verdoppelung des lokalen Raums und die Rücklieferung beider Alternativen hätte also vermieden werden können, wenn man die Bindung von `Y` im globalen Raum abgewartet hätte. ◇

Wir verhindern solche überflüßigen Vervielfältigungen durch Einführung eines *Stabilitätskriteriums*, das ein lokaler Raum erfüllen muß, damit er aufgelöst werden darf. Die Idee stammt aus den Sprachen Oz und AKL, wo ein lokaler Raum bzw. Constraint-Speicher als stabil gilt, wenn kein anderer Reduktionsschritt als ein nichtdeterministischer möglich ist und auch durch Hinzufügen von Constraints außerhalb des lokalen Raums kein deterministischer möglich gemacht werden kann. Die genauen Definition kann man in [32] und [19] nachlesen.

In Curry können nur durch die Bindung globaler Variablen Informationen in den lokalen Raum hineingereicht werden, und daher definieren wir einen stabilen Raum in Curry wie folgt:

Definition 5.2 (Stabiler lokaler Raum)
Ein lokaler Berechnungsraum (V, σ, c) heißt stabil, wenn $c =$ `{}` gilt, oder wenn $cse(c) = \{d_1, \ldots, d_n\}$ mit $n > 1$ gilt und $free(c) \subseteq V$ erfüllt ist. □

In Anlehnung an Oz schreiben wir nun vor, daß ein lokaler Raum nur aufgelöst werden darf, wenn er stabil ist. Auf diese Weise wird ein nichtdeterministischer Schritt und damit die Vervielfältigung des lokalen Raums so lange verzögert, bis auf keinen Fall mehr ein deterministischer Schritt durch die Bindung einer globalen Variable möglich gemacht werden kann.

Beispiel 5.2
Wie wir in Abschnitt 5.2.1 noch genau definieren werden, erzeugt die Auswertung des Suchoperators im Ausdruck

```
try(\X -> {Y=s(X) /\ add(X,z)=z}) /\ Y=s(z)
```

einen lokalen Raum der Form

({X}, id, Y=s(X) /\ add(X,z)=z) .

Die Berechnung ist nicht suspendiert, ein deterministischer Reduktionsschritt ist auch nicht möglich, aber der Raum ist nicht stabil, da die Variable `Y` nicht zur Menge der lokalen Variablen des Raums gehört. Daher wird der lokale Raum suspendiert, wodurch in Beispiel zunächst der Constraint `Y=s(z)` außerhalb des lokalen Raum reduziert wird. Nachdem die Variablenbindung in den lokalen Raum hineingereicht wurde, besitzt dieser die Form

({X}, id, s(z)=s(X) /\ add(X,z)=z)

und kann nun die Berechnung deterministisch fortsetzen. Die Vervielfältigung, die in Beispiel 5.1 vorgenommen wurde, ist also vermieden worden. Der lokale Raum

({X}, id, add(X,z)=z)

beispielsweise ist hingegen stabil und darf daher zu den beiden lokalen Räumen

({X},   {X=z},   add(z,z)=z)
({X,N}, {X=s(N)}, add(s(N),z)=z)

reduziert werden. ◇

Wir haben nur gefordert, daß der Constraint c des lokalen Raums keine globalen Variablen enthalten darf. In der Substitution dürfen sie daher auftauchen.

Beispiel 5.3
Der lokale Raum $(\{\texttt{X,A}\}, id, \texttt{X=s(Y) /\textbackslash\ add(A,z)=s(z)})$ darf die Reduktion

$$(\{\texttt{X,A}\},\emptyset,\texttt{X=s(Y) /\textbackslash\ add(A,z)=s(z)}) \leadsto \ldots \leadsto (\{\texttt{A}\},\{\texttt{X=s(Y)}\},\texttt{add(A,z)=s(z)})$$

ausführen und ist danach stabil, da die globale Variable `Y` in der Substitution verschwunden ist. Die Verdoppelung des Raums, die der nächste Reduktionsschritt bewirken wird, könnte nicht durch eine Bindung von `Y` außerhalb des lokalen Raums verhindert werden, da sie keinen Einfluß auf den Constraint `add(A,z)=s(z)` hätte. $\diamond$

5.1.1 Die operationale Semantik lokaler Räume

Die Reduktion lokaler Räume erfolgt durch die Funktion

$$cslocal \;:\; \mathcal{L} \;\to\; \mathcal{L} \cup \{\bot\}\,,$$

deren Regeln wir in Abbildung 5.1 angeben. Im Unterschied zu den bisherigen Reduktionsfunktionen liefert sie keinen disjunktiven Ausdruck zurück, da wir ihr Ergebnis nicht direkt als Antwortausdruck verwenden werden, sondern durch die Reduktionsfunktionen für den Suchoperator und die Committed-Choice, die wir in den Abschnitten 5.2.2 und 5.3.2 vorstellen, in jeweils andere Formate verpacken lassen.

$$cslocal((V,\sigma,c)) \;=\; \begin{cases} \{(V,\sigma)\} & \text{wenn } c = \texttt{\{\}} \\ \bot & \text{wenn } c \neq \texttt{\{\}} \text{ und } cse(V,c) = \bot \\ \emptyset & \text{wenn } c \neq \texttt{\{\}} \text{ und } cse(V,c) = \emptyset \\ \{(\tilde{V},\tilde{\sigma}\circ\sigma,\tilde{c})\} & \text{wenn } c \neq \texttt{\{\}} \text{ und } cse(V,c) = \{\exists\tilde{V} : \tilde{\sigma} [] \tilde{c}\} \\ \bot & \text{wenn } c \neq \texttt{\{\}},\ cse(V,c) = \{e_1,\ldots,e_n\} \text{ mit } n > 1 \\ & \text{und } free(c) \not\subseteq V \\ \{(V_1,\sigma_1\circ\sigma,c_1),\ldots,(V_n,\sigma_n\circ\sigma,c_n)\} & \\ & \text{wenn } c \neq \texttt{\{\}},\ cse(V,c) = \{e_1,\ldots,e_n\} \text{ mit } n > 1, \\ & e_i = \exists V_i : \sigma_i [] c_i \text{ und } free(c) \subseteq V \end{cases}$$

Abbildung 5.1: Reduktionsschritt für einen lokalen Raum

cslocal liefert einen gelösten Raum zurück, wenn der Constraint erfolgreich reduziert werden konnte. Die Verwendung gelöster Räume ist an dieser Stelle für eine möglichst allgemeingültige Definition hilfreich, da wir andernfalls die Übergabe von Räumen mit dem leeren Constraint `{}` verhindern müßten, denn ein solcher Raum kann keinen Reduktionsschritt mehr durchführen und führt daher zur Suspension. Um einen Raum nicht vor der Übergabe an *cslocal* prüfen zu müssen, verwenden wir daher eine besonderes Format zur Kennzeichnung von Räumen, die ihre Reduktion beendet haben.

Jeder andere Raum wird zur Reduktion zusammen mit der Menge seiner lokalen Variablen an *cse* übergeben. Die korrekte Behandlung der Variablen wird durch die in Abschnitt 4.2.1 neu definierten Reduktionsfunktionen erreicht: Explizit deklarierte oder während der Berechnung neu auftretende Variablen werden zu V hinzugefügt, die Bindung von Variablen, die nicht in V liegen und daher außerhalb des lokalen Raums deklariert worden sein müssen, wird verhindert.

Beim Erzeugen eines lokalen Raums muß also darauf geachtet werden, daß V als leere Menge initialisiert wird.

Wenn die Reduktion des Constraints suspendiert oder fehlschlägt, suspendiert oder schlägt auch der lokale Raum fehl. Konnte ein deterministischer Reduktionsschritt durchgeführt werden, liefert *cslocal* das Ergebnis in einem neuen lokalen Raum zurück. Wenn nur ein nichtdeterministischer Schritt möglich war, wird die Stabilität des Raums geprüft. Wenn er nur lokale Variablen enthält, ist er stabil und wird durch eine Menge von neuen Räumen ersetzt, indem man jeden Antwortausdruck in einen eigenen lokalen Raum verpackt. Wenn der Raum nicht stabil ist, suspendieren wir ihn.

Die Arbeitsweise der Reduktionsfunktion *cslocal* haben wir bereits in den Beispielen 5.2 und 5.3 betrachtet. Weitere Beispiele werden wir im nächsten Abschnitt vorstellen.

Für eine effiziente Implementierung könnte man eine Verbesserung vornehmen, indem man Räume, die entweder suspendieren, weil sie eine nichtdeterministische Reduktion durchführen wollen aber nicht stabil sind, oder weil sie eine nicht lokale Variable binden wollen, unter Angabe der für die Suspension verantwortlichen Variablen kennzeichnet. Die Auswertung eines solchen Raums bräuchte erst wieder „aufgeweckt“ zu werden, wenn mindestens eine der Variablen gebunden worden ist. Für die Angabe einer möglichst knappen und verständlichen operationalen Semantik verzichten wir an dieser Stelle auf eine solche Kennzeichnung, verwenden sie aber in der TasteCurry-Implementierung (vgl. Abschnitt 7.2).

Es wäre außerdem sinnvoller, die Variablenkennzeichnung nicht nur für lokale Räume, sondern für jeden Teil einer Konjunktion vorzunehmen, der aufgrund einer nicht gebundenden Variable suspendiert, beispielsweise wenn dort eine Funktion, die nach dem Residuation-Prinzip ausgewertet werden soll, mit einer Variable als Parameter aufgerufen wird. Dazu wäre jedoch eine grundlegende Änderung der Reduktionsfunktionen erforderlich, die nicht Gegenstand dieser Arbeit sein soll.

Zwei Punkte wollen wir abschließend kurz erwähnen:

- Die Informationen, die ein gelöster lokaler Raum (V, σ) berechnet hat, können niemals im Widerspruch zum globalen Raum stehen, da die Bindung von globalen Variablen im lokalen Raum verhindert wird. Die berechnete Substitution σ kann daher nur Bindungen lokaler Variablen enthalten, die im globalen Raum ebensowenig sichtbar sind wie die Elemente aus V. Wir können also σ und V zur Substitution bzw. Variablenmenge des globalen Raums hinzufügen, ohne einen Widerspruch zu verursachen.

  Die Informationen zweier gelöster lokaler Räume mit dem globalen zu vereinigen, könnte hingegen Konflikte verursachen, wenn die beiden Räume verschiedene Bindungen für eine Variable berechnet haben. Wenn wir beispielsweise nach einem nichtdeterministischen Schritt aus dem Raum

  ({X,Y},*id*,add(X,Y)=s(z))

  die beiden neuen Räume

  ({Y},{X=z},add(z,Y)=s(z))   ({N,Y},{X=s(N)},add(s(N),Y)=s(z))

  erzeugen, können diese in die gelöste Formen

  (∅,{X=z,Y=s(z)})   (∅,{X=s(z),Y=z})

  überführt werden. Die berechneten Informationen sind widersprüchlich und dürfen daher in dieser Form nicht global sichtbar gemacht werden.

- Die Einführung lokaler Räume dient der Einkapselung von Berechnungen, und daher dürfen Informationen, die während der Berechnung entstehen, nicht außerhalb des Raums sichtbar werden. Dies ist insbesondere dann wichtig, wenn verschiedene Reduktionswege in verschiedenen lokalen Räumen verfolgt werden. Damit die Berechnungen unabhängig voneinander erfolgen können, dürfen beispielsweise keine Variablensubstitutionen zwischen den Räumen ausgetauscht werden, da diese widersprüchlich sein können, wie wir im vorigen Punkt gesehen haben. Würden wir dort beispielsweise die Bindung `X=z` aus dem ersten in den zweiten lokalen Raum weiterreichen, gäbe es einen Widerspruch mit der Substitution `X=s(N)`, was zum sofortigen Fehlschlag führen würde.

Bei der Definition der operationalen Semantik für den Suchoperator und die Committed-Choice müssen wir also darauf achten, daß keine Informationen aus einem lokalen Raum nach außen gelangen können, bevor dieser seine Reduktion beendet hat. Wenn zwei lokale Räume verschiedene Lösungen berechnen, müssen wir zudem einen Mechanismus schaffen, um beide Lösungen verarbeiten zu können, ohne dadurch einen Widerspruch zu verursachen.

5.2 Die operationale Semantik des Suchoperators

5.2.1 Suchräume

Für die Realisierung des Suchoperator verwenden wir eine Erweiterung des lokalen Raums: den Suchraum.

Definition 5.3 (Suchraum)
Ein Suchraum hat die Form

$$\texttt{searchSpace}(X, l)$$

mit X eine Variable und $l \in \mathcal{L}$. □

Hinter der Erweiterung um eine Komponente verbirgt sich die Idee, für den Suchoperator einen lokalen Raum zu schaffen, in dem die Suche durchgeführt wird, und sich zusätzlich die Suchvariable zu merken, um den lokalen Raum nach seiner Auflösung wieder korrekt in eine oder mehrere Lambda-Abstraktionen umwandeln zu können. In Abschnitt 5.2.2 werden wir dies näher erläutern.

Wir geben die operationale Semantik für den Suchoperator `try` durch eine weitere Regel für die Funktion *cse* in Abbildung 5.2 an. Der Suchoperator erhält eine Funktion als Parame-

$$\boxed{\begin{array}{rcl} cse(V, \texttt{try}(f(e_1,\ldots,e_n))) & = & \{\exists V : id[]\texttt{searchSpace}(X, (X, id, f(e_1,\ldots,e_n)@X))\} \\ & & \text{mit } X \text{ eine neue Variable, } f \in \Sigma^{n+1},\ n \geq 0 \end{array}}$$

Abbildung 5.2: Operationale Semantik des Suchoperators

ter und erzeugt einen Suchraum für die Reduktion dieser Funktion. Wir haben bisher immer Lambda-Abstraktionen benutzt, aber das Typsystem erlaubt beliebige Funktionen des Typs `A -> constraint`. Jede Funktion, die den Zieltyp `constraint` hat, kann daher als partielle Applikation mit *einem* fehlenden Parameter an `try` übergeben werden. Das Format des Suchraums erläutern wir an einem Beispiel.

Beispiel 5.4
Betrachten wir den Suchoperator aus Beispiel 4.3:

```
try(\X -> {local N in X=add(s(z),N)})
```
$\rightsquigarrow \exists\emptyset$: id[]`searchSpace(X`$_1$`, ({X`$_1$`},`id`,(\X -> {local N in X=add(s(z),N)})@X`$_1$`) )`

◇

Da durch die Lambda-Abstraktion von der Suchvariable abstrahiert worden ist, muß zunächst für die Reduktion eine konkrete Suchvariable erstellt werden. Dazu wird eine neue Variable geschaffen und in der ersten Komponente des Suchraums gespeichert.

Als zweite Komponente des Suchraums wird ein lokaler Raum erzeugt, in dem die Reduktion des Constraints durchgeführt werden soll. Dazu wird die Suchvariable auf die Lambda-Abstraktion appliziert und in die Menge der lokalen Variablen des Raums eingetragen, damit sie während der Berechnung gebunden werden kann. Wir sehen hier die deutliche Trennung zwischen den globalen und lokalen Variablen: $cse(V,$`try(...)`$)$ liefert die globalen Variablen in V als Teil des Antwortausdrucks zurück, der den globalen Raum repräsentiert. Die lokalen Variablen des Suchraums sind hingegen in dem lokalen Raum versteckt, der kein Element aus V übergeben bekommt und daher während seiner Reduktion die Bindung globaler Variablen verhindern kann. Auf diese Weise kann nach Abschnitt 5.1.1 kein Widerspruch zwischen den Informationen des Suchraums und des globalen Raums auftreten.

Beispiel 5.5 (Fortsetzung von Beispiel 5.4)
Der erste Reduktionsschritt des Suchraums sieht wie folgt aus:

`searchSpace(X`$_1$`,({X`$_1$`},`id`, (\X -> {local N in X=add(s(z),N)})@X`$_1$` ))`

$\rightsquigarrow \exists\emptyset$: id[]`searchSpace(X`$_1$`,({X`$_1$`},`id`,local N in X`$_1$`=add(s(z),N)))`

Die Berechnung des Constraints wird also eine Substitution für unsere Suchvariable `X`$_1$ erstellen. Wir haben hier schon einen kleinen Vorgriff auf die operationale Semantik für Suchräume vorgenommen, allerdings ist dabei nichts Besonderes geschehen, da der lokale Raum einen normalen Reduktionsschritt durchführen konnte. ◇

Anstelle einer Lambda-Abstraktion können wir auch eine beliebige partielle Funktionsapplikation des Typs `A -> constraint` übergeben. In diesem Fall übernimmt der fehlende Parameter die Rolle der Suchvariable.

Beispiel 5.6
Wenn wir die Großvater-Relation wie in Beispiel 2.12 als Funktion mit dem Ergebnistyp `constraint` definieren, können wir eine eingekapselte Suche nach den Enkeln von `hans` durch folgenden `try`-Aufruf durchführen:

```
vater(hans,klaus) = {}.
⋮
grossvater(X,Y) = {local Z in vater(X,Z), vater(Z,Y)}.

try(grossvater(hans))
```
$\rightsquigarrow \exists\emptyset$: id[]`searchspace(X`$_1$`,({X`$_1$`},`id`,grossvater(hans)@X`$_1$`)`

$\rightsquigarrow \exists\emptyset$: id[]`searchspace(X`$_1$`,({X`$_1$`},`id`,grossvater(hans,X`$_1$`))`

Der fehlende Parameter, der Enkel, wird also automatisch als Suchvariable verwendet. ◇

Daß wir die Applikation auf die Variable explizit als Ausdruck in den lokalen Raum eintragen müssen und nicht gleich den Rumpf der Funktion extrahieren und dort den letzten Parameter durch die Suchvariable ersetzen können, liegt an der Möglichkeit der Verwendung beliebiger

Funktionen. Wären wir auf Lambda-Abstraktionen beschränkt, könnten wir den Rumpf durch die Vorschrift

$$cse(V, \texttt{try}(\backslash X \texttt{ -> } \{c\})) \quad = \quad \{\exists V : id[]\texttt{searchSpace}(X_1, (\{X_1\}, id, c[X/X_1]))\}$$

bei der Erstellung des Suchraums sofort extrahieren. Hätten wir hingegen die beiden Regeln

```
vater(hans,karl)   = {}.
vater(klaus,peter) = {}.
```

definiert und würden `try(vater(X))` aufrufen, so gibt es zwei mögliche Rümpfe, und diesen Nichtdeterminismus wollen wir ja in die Berechnung in einem lokalen Raum einkapseln. Daher wird die Applikation und somit der nichtdeterministische Schritt erst nach Erzeugung des Suchraums ausgeführt. In Abschnitt 5.2.3 werden wir sehen, daß ein ganz ähnliches Vorgehen auch in Oz verwendet wird.

5.2.2 Die operationale Semantik von Suchräumen

Da wir nach Abbildung 5.2 aus dem letzten Abschnitt Suchräume als Antwortausdrücke zurückgeben, benötigen wir eine entsprechende Erweiterung von *cse* für die Reduktion eines `searchSpace`-Ausdrucks, die wir in Abbildung 5.3 angeben. Wir führen zunächst durch

$$\boxed{cse(V, \texttt{searchSpace}(X, l)) \quad = \quad \begin{cases} \{\exists V : id[]returnSearch(X, cslocal(l))\} \\ \qquad \text{wenn } l \in \mathcal{L} \text{ und } cslocal(l) \neq \perp \\ \perp \quad \text{wenn } l \in \mathcal{L} \text{ und } cslocal(l) = \perp \end{cases}}$$

Abbildung 5.3: Operationale Semantik für einen Suchraum

cslocal (vgl. Abschnitt 5.1.1) einen Reduktionsschritt auf dem lokalen Raum durch. Wenn eine Reduktion möglich war, übergeben wir das Ergebnis zur Auswertung an die Funktion *returnSearch*, die in Abbildung 5.4 dargestellt ist. Ihre Aufgabe ist es, entweder einen neuen Suchraum zurückzuliefern, oder bei Beendigung der Reduktion das Ergebnis in eine oder mehrere Lambda-Abstraktionen zu verpacken. War keine Reduktion des lokalen Raums möglich, wird die Berechnung suspendiert.

Wir erkennen auch hier wieder die in Abschnitt 5.1.1 geforderte klare Trennung zwischen dem Suchraum und dem ihn umgebenden Raum. Unabhängig von dem Reduktionsergebnis des Suchraums erzeugen wir einen Antwortausdruck, der die identische Substitution und eine unveränderte Menge V der Variablen des globalen Raums zurückliefert. Ergebnisse aus dem lokalen Raum können also nicht nach außen dringen, sondern nur nach Beendigung der Reduktion als Datenobjekte behandelt werden, die die Funktion *returnSearch* gemäß den Ideen aus Abschnitt 3.5 erzeugt. Dort hatten wir gefordert, daß

- ein Fehlschlag durch eine leere Liste signalisiert wird,
- das Ergebnis eines vollständig reduzierten Suchraums in einer Lambda-Abstraktion zurückgeliefert wird und
- bei einem nichtdeterministischen Reduktionsschritt eine Liste von Lambda-Abstraktionen erstellt wird, die die verschiedenen Berechnungswege repräsentieren.

$$
returnSearch(X,d) \;=\; \begin{cases} \texttt{[]} & \text{wenn } d=\emptyset \\ \texttt{[}cl(X,(V,\sigma))\texttt{]} & \text{wenn } d=(V,\sigma) \\ \texttt{[}cl(X,l_1),\ldots,cl(X,l_n)\texttt{]} & \text{wenn } d=\{l_1,\ldots,l_n\} \text{ mit } n>1 \\ \texttt{searchSpace}(X,d) & \text{sonst} \end{cases}
$$

Abbildung 5.4: Rückgabe von Suchräumen

returnSearch nimmt eine Fallunterscheidung bezüglich des Ergebnisses von *cslocal* vor. Eine leere Menge zeigt den Fehlschlag des lokalen Raums an und bewirkt eine leere Liste als Ergebnis. Ein gelöster Raum wird durch die Funktion *cl*, die wir in Abbildung 5.5 angeben, in eine Lambda-Abstraktion verpackt, die das Ergebnis repräsentiert. Wir benötigen hier das Format des gelösten Raums zur Unterscheidung von einem lokalen Raum, der einen normalen Reduktionsschritt durchgeführt hat.

Ein nichtdeterministischer Reduktionsschritt bewirkt eine Menge von lokalen Räumen als Ergebnis, und wir wandeln diese Menge in eine Liste von Abstraktionen um, indem wir *cl* auf jeden Raum anwenden. Man beachte, daß dabei wie in Abschnitt 5.1.1 gefordert, keinerlei Informationen zwischen den lokalen Räumen ausgetauscht werden können, da jeder durch einen eigenen Aufruf von *cl* behandelt wird.

Ist keiner der drei Fälle eingetreten, dann wurde der lokale Raum durch einen normalen deterministischen Berechnungsschritt in einen neuen Raum überführt, und wir liefern diesen wieder in Form eines Suchraums zurück.

Die Erklärung der Funktion *cl* bedarf einiger Bemerkungen vorab. Das Umwandeln eines Suchraums in eine Lambda-Abstraktion ist nach dem folgendem Prinzip möglich:

$$
\texttt{searchSpace(}X,(\{X,X_1,\ldots,X_n\},\{Y_1\texttt{=}e_1,\ldots Y_k\texttt{=}e_k\},c)\texttt{)}
$$
$$
\updownarrow
$$
$$
\texttt{\textbackslash}X\ \texttt{->}\ \texttt{\{local [}X_1,\ldots,X_n,Y_1,\ldots,Y_k\ldots\texttt{] in } Y_1\texttt{=}e_1\ \texttt{/\textbackslash}\ldots\texttt{/\textbackslash}\ Y_k\texttt{=}e_k\ \texttt{/\textbackslash}\ c\texttt{\}}
$$

Die Umwandlung des Suchraums in die Lambda-Abstraktion wird durch Formulierung der erstellten Substitution als Konjunktion von Gleichheitsconstraints und die Deklaration der lokalen Variablen möglich. Die bereits gebundenen Variablen Y_i zählen natürlich auch zu den lokalen Variablen des Raums, und da sie nach der neuen Definition der Reduktionsfunktion in Abschnitt 4.2.1 bei ihrer Bindung aus der Menge der lokalen Variablen entfernt wurden, müssen wir sie erneut deklarieren.

Die Übergabe der Lambda-Abstraktion an den Suchoperator wird nach einigen Reduktionsschritten einen Suchraum erzeugen, der bis auf Umbenennung der Variablen dem ursprünglichen gleicht.

Beispiel 5.7
Der Suchraum

```
searchSpace(X,({N},{X=s(N)},add(s(N),z)=s(z)))
```

wird nach dem beschriebenen Prinzip in die Lambda-Abstraktion

```
\X -> {local N in X=s(N) /\ add(s(N),z)=s(z)}
```

umgewandelt. Der erneute Aufruf durch `try`, den beispielsweise unser `all`-Search-Algorithmus (Beispiel 3.8) vornehmen würde, reduziert wie folgt:

$\texttt{try(\textbackslash X -> \{local N in X=s(N) /\textbackslash\ add(s(N),z)\})}$

$\rightsquigarrow$ $\texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,(\textbackslash X -> \{local N in X=s(N) /\textbackslash\ add(s(N),z)=s(z)\})@X}_1\texttt{))}$

$\rightsquigarrow$ $\texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,\{local N in X}_1\texttt{=s(N) /\textbackslash\ add(s(N),z)=s(z)\}))}$

$\rightsquigarrow$ $\texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{,N}_1\texttt{\},}id\texttt{,X}_1\texttt{=s(N}_1\texttt{) /\textbackslash\ add(s(N}_1\texttt{),z)=s(z)))}$

$\rightsquigarrow$ $\texttt{searchSpace(X}_1\texttt{,(\{N}_1\texttt{\},\{X}_1\texttt{=s(N}_1\texttt{)\},add(s(N}_1\texttt{),z)=s(z)))}$

Damit haben wir bis auf die Variablennamen wieder den ursprünglichen Suchraum erzeugt, in dem nun die Berechnung fortgesetzt werden kann. $\diamond$

Da man bei der eingekapselten Suche aber nur an der Lösung der Suchvariable interessiert ist, muß man nicht die komplette Substitution des Suchraums wieder in einen Constraint umwandeln.

Beispiel 5.8
Setzen wir die Berechnung aus dem vorigen Beispiel fort, so werden wir schließlich einen nichtdeterministischen Reduktionsschritt für $\texttt{add(N}_1\texttt{,z)=z}$ ausführen müssen, der die beiden lokalen Räume

$\texttt{(}\emptyset\texttt{,\{X}_1\texttt{=s(z),N}_1\texttt{=z\},add(z,z)=z)}$

$\texttt{(\{M\},\{X}_1\texttt{=s(s(M)),N}_1\texttt{=s(M)\},add(s(M),z)=z)}$

zurückliefert, die wir durch *returnSearch* verarbeiten lassen. Die Bindung von $\texttt{N}_1$ ist für die Lösung unserer Suchvariable $\texttt{X}_1$ uninteressant und muß daher nicht zurückgeliefert werden. $\diamond$

Die Funktion *cl* in Abbildung 5.5 erhält die zwei Komponenten, aus denen ein Suchraum besteht, und erstellt daraus nach dem beschriebenen Prinzip eine Lambda-Abstraktion. Der

$$cl(X,(V,\sigma,c)) = \begin{cases} \texttt{\textbackslash X -> \{}declare_{cl}(V,c)\texttt{\}} & \text{wenn } \sigma(X)=X \\ \texttt{\textbackslash X -> \{}declare_{cl}(V,X\texttt{=}\sigma(X)\ \texttt{/\textbackslash}\ c)\texttt{\}} & \text{sonst} \end{cases}$$

$$cl(X,(V,\sigma)) = \begin{cases} \texttt{\textbackslash X -> \{\}} & \text{wenn } \sigma(X)=X \\ \texttt{\textbackslash X -> \{}declare_{cl}(V,X\texttt{=}\sigma(X))\texttt{\}} & \text{sonst} \end{cases}$$

$$declare_{cl}(V,c) = \begin{cases} c & \text{wenn } V=\emptyset \\ \texttt{local [}X_1,\ldots,X_n\texttt{] in } c & \text{wenn } V=\{X_1,\ldots,X_n\} \end{cases}$$

Abbildung 5.5: Umwandlung eines Suchraums in eine Lambda-Abstraktion

Rumpf besteht aus dem Constraint des lokalen Raums, der Bindung der Suchvariable und der Deklaration der lokalen Variablen. Die restliche Substitution, die der lokale Raum berechnet hat, wird also verworfen. Falls der lokale Raum gelöst war, tragen wir nur die Bindung der Suchvariable und die Deklaration der Variablen als Rumpf ein, falls eine Bindung existiert. Andernfalls war die Anfrage, die wir durch den Suchoperator gestellt haben, unabhängig vom Wert der Suchvariable erfüllbar, und daher liefern wir nur den leeren Constraint zurück.

Man beachte, daß eine lokale Variablendeklaration am Anfang eines Constraints bei der Reduktion sofort durch einen deterministischen Schritt abgebaut wird. Der Constraint c des lokalen Raums kann daher nur dann mit einer solchen Deklaration beginnen, wenn *cslocal* als letzten Schritt eine nichtdeterministische Funktion aufgerufen hat, deren rechte Seite die Form `{local [...] in ...}` hat. In diesem Fall ist der Constraint korrekt geklammert, so daß wir im ersten Fall von *cl* problemlos die Konjunktion erstellen können, ohne die korrekte Form von Constraints aus Definition 4.1 in Abschnitt 4.1.1 zu verletzen. Aus demselben Grund können wir eine vereinfachte Version der *declare*-Funktion aus Definition 4.7 in Abschnitt 4.1.4 verwenden.

Wir betrachten nun ausschnittsweise einige Beispielreduktionen, um uns die Arbeitsweise der Funktionen *returnSearch* und *cl* zu verdeutlichen. Wenn wir dabei mehrere Reduktionsschritte zusammenfassen, markieren wir den Reduktionspfeil mit einem Stern: $\overset{*}{\leadsto}$. Die Variablenmenge und die Substitution des Antwortausdrucks führen wir nicht auf, wenn es sich dabei um eine leere Menge und die identische Substitution handelt. Bei der Rückumwandlung eines Suchraums in Lambda-Abstraktionen verwenden wir wieder den ursprünglichen Namen für die Suchvariable. Zunächst greifen wir die Beispiele aus Abschnitt 3.5 auf, anhand derer wir das Format des Suchoperators erklärt hatten.

Beispiel 5.9 (Fehlschlag)

`try(\X -> {append([1,2],X)=[3,4,5,6]})`

$\overset{*}{\leadsto}$ `searchSpace(`X_1`,({`X_1`},`id`,1=3 /\ append([2],`X_1`)=[3,4,5,6]))`
$\overset{cslocal}{\leadsto}$ $\emptyset$
$\overset{returnSearch}{\leadsto}$ `[]`

◇

Beispiel 5.10 (Erfolgreiche Reduktion)

`try(\X -> append([1,2],X)=[1,2,3,4]})`

$\overset{*}{\leadsto}$ `searchSpace(`X_1`,({`X_1`},{`X_1`=[3,4]},{}))`
$\overset{returnSearch}{\leadsto}$ `[\X -> {X=[3,4]}]`

◇

Beispiel 5.11 (Nichtdeterministische Reduktion)

`try(\X -> {add(X,z)=s(z)})`

$\overset{*}{\leadsto}$ `searchSpace(`X_1`,({`X_1`},`id`,add(`X_1`,z)=s(z)))`
$\overset{cslocal}{\leadsto}$ `{(`$\emptyset$`,{`X_1`=z},add(z,z)=s(z)),`
`({N},{`X_1`=s(N)},add(s(N),z)=s(z))}`
$\overset{returnSearch}{\leadsto}$ `[\X -> {X=z /\ add(z,z)=s(z)},`
`\X -> {local N in X=s(N) /\ add(s(N),z)=s(z)}]`

Wir übergeben die zweite Lambda-Abstraktion erneut an den Suchoperator.

`try(\X -> {local N in X=s(N) /\ add(s(N),z)=s(z)})`

$\overset{*}{\leadsto}$ `searchSpace(`X_1`,({`N_1`},{`X_1`=s(`N_1`)},add(`N_1`,z)=z))`
$\overset{returnSearch}{\leadsto}$ `[\X -> {X=s(z) /\ add(z,z)=z},`
`\X -> {local M in X=s(s(M)) /\ add(s(M),z)=z}]`

Die Bindung für N_1 haben wir bei der letzten Transformation in Lambda-Abstraktionen aus den erwähnten Gründen verworfen (vgl. Beispiel 5.8). ◇

Wenn der Constraint unabhängig vom Wert der Suchvariable erfüllbar ist, erhalten wir nach der Definition von *cl* den leeren Constraint als Rumpf der Lambda-Abstraktion.

Beispiel 5.12

$$\begin{array}{l}
\texttt{try(\textbackslash X -> \{leq(z,X)=true\})}\\
\leadsto\ \texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,leq(z,X}_1\texttt{)=true))}\\
\leadsto\ \texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,true=true))}\\
\leadsto\ \texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,\{\}))}\\
\leadsto\ \texttt{[\textbackslash X -> \{\}]}
\end{array}$$

In einem solchen Fall kann $\texttt{X}_1$ nicht aus der Variablenmenge entfernt werden, da dies nur bei einer Bindung geschieht, aber eine Lösung unabhängig vom Wert der Suchvariable nur möglich ist, wenn diese nicht gebunden wird. ◇

Wie wir bei der Definition der Stabilität in Abschnitt 5.1 erläutert haben, darf ein nicht stabiler Raum nicht verdoppelt werden. Wir greifen das Beispiel 5.1 noch einmal auf und erläutern die korrekte Reduktion.

Beispiel 5.13
Der erste Teil des Constraints im folgenden Suchraum kann nicht reduzieren, da er die globale Variable `Y` nicht binden darf. Der zweite Teil suspendiert ebenfalls, denn er könnte zwar einen nichtdeterministischen Schritt ausführen, allerdings ist der lokale Raum nicht stabil, da er die globale Variable `Y` enthält.

$$\begin{array}{l}
\texttt{L=try(\textbackslash X -> \{Y=s(X) /\textbackslash{} add(X,z)=z\}) /\textbackslash{} Y=s(z)}\\[1ex]
\overset{*}{\leadsto}\ \exists\{\texttt{L,Y}\}:id\,[]\,\texttt{L=searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,}\underset{\substack{\downarrow cse\\ \perp}}{\texttt{Y=s(X}_1\texttt{)}}\texttt{ /\textbackslash{} }\underset{\substack{\downarrow \texttt{Stabilitätskriterium}\\ \perp}}{\texttt{add(X}_1\texttt{,z)=z}}\texttt{)) /\textbackslash{} Y=s(z)}\\[1ex]
\leadsto\ \exists\{\texttt{L}\}:\{\texttt{Y=s(z)}\}\,[]\,\texttt{L=searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,s(z)=s(X}_1\texttt{) /\textbackslash{} add(X}_1\texttt{,z)=z))}\\
\overset{*}{\leadsto}\ \exists\{\texttt{L}\}:\{\texttt{Y=s(z)}\}\,[]\,\texttt{L=[\textbackslash X -> \{X=z\}]}\\
\leadsto\ \texttt{...}
\end{array}$$

◇

In Beispiel 4.13 hatten wir festgestellt, daß Variablen bei einer Bindung zwar aus dem zu reduzierenden Ausdruck verschwinden können, aber trotzdem in der Menge der deklarierten Variablen verbleiben. Wieso dies notwendig ist, können wir nun aufgrund der Definition von *cl* verstehen.

Beispiel 5.14

$$\begin{array}{l}
\texttt{try(\textbackslash X -> \{local N in X=add(s(z),N)\})}\\[1ex]
\overset{*}{\leadsto}\ \texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{\},}id\texttt{,local N in X}_1\texttt{=add(s(z),N)))}\\
\overset{*}{\leadsto}\ \texttt{searchSpace(X}_1\texttt{,(\{X}_1\texttt{,N}_1\texttt{\},}id\texttt{,X}_1\texttt{=s(N}_1\texttt{)))}\\
\overset{*}{\leadsto}\ \texttt{searchSpace(X}_1\texttt{,(\{N}_1\texttt{\},\{X}_1\texttt{=s(N}_1\texttt{)\},\{\}))}\\
\overset{returnSearch}{\leadsto}\ \texttt{[\textbackslash X -> \{local N}_1\texttt{ in X=s(N}_1\texttt{)\}]}
\end{array}$$

Da die Variable $\texttt{N}_1$ im Rumpf der neuen Lambda-Abstraktion auftritt, muß sie auch wieder als lokale Variable deklariert werden, und dazu muß sie nach Definition von *cl* in der Menge der lokalen Variablen protokolliert sein. Wäre sie bei der Bindung von dort entfernt worden, würde *cl* die Abstraktion $\texttt{\textbackslash X -> \{X=s(N}_1\texttt{)\}}$ erzeugen. Da in dieser Form bei der Applikation keine neue Variable für `N`$_1$ geschaffen würde, könnten wir nicht wie in Beispiel 4.3 mehrere Elemente auf ihr Enthaltensein in der Lösungsmenge, der Menge aller natürlichen Zahlen größer 0, prüfen, ohne einen Konflikt wie in Beispiel 3.5 zu verursachen. ◇

Es natürlich auch passieren, daß die Berechnung eines Suchraums suspendiert und nie wieder angestoßen werden kann, beispielsweise wenn in Beispiel 5.13 der Zweig `Y=z` gefehlt hätte. Oder wenn, wie im folgenden Beispiel, eine Funktion, die durch Residuation ausgewertet werden soll, mit einer lokalen Variable als Parameter aufgerufen wird.

Beispiel 5.15

```
f eval 1:rigid.
f(X) = {}.
```

try(\X -> {f(X)}) $\rightsquigarrow$ ... $\rightsquigarrow$ searchSpace(X$_1$,({X$_1$},id,f(X$_1$))) $\rightsquigarrow$ $\perp$

Die Reduktion dieses Suchraums kann nie wieder angestoßen werden, da die `X`$_1$ im Suchraum selbst nicht gebunden wird und außerhalb des Raums nicht sichtbar ist. ◇

Anhand der betrachteten Beispiele sollte klar geworden sein, wie die von uns definierte Reduktionssemantik für Suchräume arbeitet, und daß sie alle Forderungen erfüllt, die wir im ersten Abschnitt dieses Kapitels sowie in Kapitel 3 an sie gestellt haben.

Die Ergebnisse des Suchoperators können entweder direkt weiterverarbeitet oder wieder an `try` übergeben werden. Im zweiten Fall wird wiederum ein Suchraum erzeugt, in dem die Suche forgesetzt wird (vgl. Beispiel 5.11). Die erneute Übergabe ist zur Programmierung von Suchalgorithmen sinnvoll, die nach einem nichtdeterministischen Schritt die verschiedenen Alternativen je nach verwendeter Strategie weiterverfolgen. Auf diese Weise können wir sicher sein, daß wir durch Anwendung eines Suchalgorithmus nur Lösungen und keine Zwischenergebnisse erhalten. In Beispiel 3.8 haben wir bereits einen Algorithmus vorgestellt, weitere werden wir in Abschnitt 6.1 kennenlernen.

Wollen wir eine berechnete Lösung direkt weiterverarbeiten, müssen wir die Lambda-Abstraktion zuerst applizieren und so die Informationen aus dem Suchraum, der durch cl in diese Lambda-Abstraktion umgewandelt wurde, im globalen Raum sichtbar machen. Neben einer berechneten Bindung der Suchvariable können auf diese Weise auch ursprünglich lokale Variablen zum globalen Raum hinzugefügt werden, da sie im globalen Raum erneut deklariert werden.

Beispiel 5.16
Die folgende Reduktion des Suchoperators haben wir in Beispiel 5.14 vorgestellt.

$\exists${L,E}:σ[]try(\X -> {local N in X=add(s(z),N)})=[L] /\ L@E}

$\overset{*}{\rightsquigarrow}$ $\exists${L,E}:σ[][\X -> {local N$_1$ in X=s(N$_1$)}]=[L] /\ L@E}

$\overset{*}{\rightsquigarrow}$ $\exists${E}: {L= \X ->...}$\circ\sigma$[](\X -> {local N$_1$ in X=s(N$_1$)})@E
$\rightsquigarrow$ $\exists${E}: {L= \X ->...}$\circ\sigma$[]local N$_1$ in E=s(N$_1$)
$\rightsquigarrow$ $\exists${E,N$_2$}: {L= \X ->...}$\circ\sigma$[]E=s(N$_2$)
$\rightsquigarrow$ $\exists${N$_2$}: {E=s(N$_2$),L= \X ->...}$\circ\sigma$[]{}

In den letzten beiden Schritten werden die Informationen des ehemals lokalen Raums, die lokale Variable `N`$_1$ und die berechnete Lösung `X=s(N`$_1$`)`, zum globalen Raum hinzugefügt. ◇

Bei der Definition der Reduktionsfunktionen in diesem und im vorigen Abschnitt haben wir auf eine strikte Trennung zwischen lokalen und globalen Variablen geachtet und dafür gesorgt, daß ein lokaler Raum keine globale Variable binden kann. Daher ist nach Abschnitt 5.1.1 ein

Widerspruch zwischen Suchraum und globalem Raum ausgeschlossen, und wir können jederzeit eine Applikation einer Lambda-Abstraktion, die die Informationen eines lokalen Raums enthält, auf eine Variable durchführen ohne dadurch einen Fehlschlag zu verursachen. Es ist aber auch eine Anwendung auf einen Grundterm möglich, wodurch die berechnete Lösung nicht einfach global sichtbar gemacht, sondern direkt auf eine bestimmte Form getestet wird. Dadurch kann es natürlich zu einem Fehlschlag kommen (vgl. Beispiel 4.3).

Aufgrund der Verwendung von Lambda-Abstraktionen können auch verschiedene, sich widersprechende Lösungen innerhalb eines Berechnungsraums verarbeitet werden, ohne einen Konflikt zu verursachen.

Beispiel 5.17
Der `all`-Algorithmus aus Beispiel 3.8 für die Berechnung aller Lösungen liefert uns für die folgende Abfrage zwei verschiedene Lösungen für `X`:

`all(\X -> {local Y in add(X,Y)=s(z)})` $\overset{*}{\rightsquigarrow}$ `[\X -> {X=z}, \X -> {X=s(z)}]`

Die Vereinigung beider Lösungen mit dem globalen Raum ist wegen der Abstraktion der Suchvariable möglich, wenn wir verschiedene Variablen auf die beiden Lambda-Abstraktionen applizieren.

`(\X -> {X=z})@A /\ (\X -> {X=s(z)})@B` $\rightsquigarrow$ `{A=z,B=s(z)}▯{}`

Das Applizieren derselben Variable würde hingegen zu einem Widerspruch führen, da wir auf diese Weise versuchen würden, die verschiedenen Lösungen gleichzusetzen. ◇

Theoretisch kann man natürlich auch die Lambda-Abstraktionen, die der Suchoperator nach einem nichtdeterministischen Schritt zur Darstellung der verschiedenen Reduktionswege zurückliefert, applizieren und damit die Informationen global sichtbar machen. Obwohl die nach dem nichtdeterministischen Schritt von *cslocal* erzeugten lokalen Räume widersprüchliche Bindungen für gleichlautende Variablen enthalten können, werden durch die Umwandlung in Lambda-Abstraktionen Konflikte vermieden.

Beispiel 5.18
Der Aufruf

`try(\X -> {local Y in add(X,Y)=s(z)})`

erzeugt einen Suchraum, in dem der lokale Raum in die Form

`({X,Y},`*id*`,add(X,Y)=s(z))`

überführt wird. In Abschnitt 5.1.1 haben wir gesehen, daß *cslocal* nun die beiden neuen Räume

`({Y},{X=z},add(z,Y)=s(z))    ({N,Y},{X=s(N)},add(s(N),Y)=s(z))`

erzeugt. Durch *returnSearch* und *cl* erhalten wir daher zwei Abstraktionen `L1` und `L2`:

```
[\X -> {local Y in X=z /\ add(z,Y)=s(z)},              L1
 \X -> {local [N,Y] in X=s(N) /\ add(s(N),Y)=s(z)}]    L2
```

Durch die Existenzquantoren in beiden Ausdrücken haben wir neben der Suchvariable auch die gleichlautenden lokalen Variablen abstrahiert. Bei einer Applikation der beiden Ausdrücke werden daher zwei verschiedene neue Variablen für die beiden `Y` geschaffen, so daß wir ohne Probleme die beiden unterschiedlichen Bindungen berechnen können:

`L1@A /\ L2@B` $\overset{*}{\rightsquigarrow}$ `{A=z,` Y_1`=s(z), B=s(z),` Y_2`=z}`

◇

Nachdem wir nun ausführlich die operationale Semantik des Suchoperators erläutert haben, stellen wir im nächsten Abschnitt einen kurzen Vergleich mit dem Vorbild aus Oz an, und beschäftigen uns danach mit der Realisierung der Committed-Choice. In Kapitel 6 werden wir dann Möglichkeiten zur Verwendung der eingekapselten Suche betrachten.

5.2.3 Der Suchoperator in Oz

In [32] beschreiben Schulte und Smolka zur Realisierung der eingekapselten Suche die Verwendung sogenannter *Solver*, die wir uns zum Vorbild für unseren Suchoperator genommen haben. Die Suche wird in Oz durch den Aufruf

Solve Q U

der vordefinierten Prozedur Solve gestartet, wobei Q eine partielle Funktionsapplikation mit einem fehlenden Parameter ist, beispielsweise eine einstellige Lambda-Abstraktion. Der Parameter U wird für die Rückgabe des Ergebnisses verwendet. Der Aufruf reduziert zu

solve[X:{Q X};U] ,

wobei X eine neue Variable ist und {Q X} die Applikation von X auf Q bezeichnet. Der Aufruf des Suchoperators **solve** erzeugt einen lokalen Berechnungsraum für den Ausdruck

local X **in** {Q X} .

Entsprechend unserer Vorgehensweise, die wir in Abschnitt 5.2.1 erläutert haben, wird die Applikation der Lambda-Abstraktion innerhalb des lokalen Raums durchgeführt. Allerdings wird in Oz die Suchvariable X durch einen expliziten Existenzquantor zur Menge der lokalen Variablen des Raums hinzugefügt. Bei der Erstellung des Suchraums, die wir in Abschnitt 5.2.1 definiert haben, ist diese Vorgehen nicht möglich, da die neu eingeführte Suchvariable bereits außerhalb des lokalen Raums, in der ersten Komponente des Suchraums, unter einem bestimmten Namen gespeichert wird. Ein expliziter Existenzquantor würde jedoch eine erneute Umbenennung innerhalb des lokalen Raums vornehmen. Wir fügen daher die Suchvariable bereits bei Erstellung des lokalen Raums zur Menge der in ihm deklarierten Variablen hinzu.

Die Reduktion in dem lokalen Raum erfolgt, bis der Constraint fehlschlägt, gelöst wird oder ein nichtdeterministischer Schritt durchgeführt werden muß. Die Rücklieferung erfolgt jedoch nicht in einer Liste, sondern durch

U=failed ,
U=solved(E) ,
U=distributed(E_1, E_2) ,

wobei es sich bei E, E_1 und E_2 um einstellige Lambda-Abstraktionen handelt. E_1 enthält die erste Alternative, E_2 eine Disjunktion der restlichen. Das grundsätzliche Format ist unserem also ähnlich, allerdings müssen Funktionen in Oz explizite Disjunktionen der Form

or $C_1 [] \ldots [] C_n$ **ro**

verwenden, um verschiedene Reduktionsrichtungen mit Hilfe der eingekapselten Suche verfolgen zu können. Jedes C_i ist ein bewachter Ausdruck der Form

$x_1 \ldots x_n$ **in** E **then** D

mit (optionalen) lokalen Variablen $x_1, \ldots x_n$ und beliebigen Ausdrücken E als Wächter und D als Rumpf (vgl. Abschnitt 4.1.4). Da die Vervielfältigung des globalen Berechnungsraums in Oz nicht zulässig ist, darf ein disjunktiver Ausdruck nur reduzieren, wenn entweder genau einer oder keiner der Wächter erfüllbar ist. Sind mehrere Wächter erfüllbar, wird die Berechnung suspendiert.

Nur durch die Verwendung des Suchoperators ist es möglich, die verschiedenen Alternativen zu untersuchen. Wenn der lokale Raum, in dem die Suche durchgeführt wird, stabil geworden ist und einen disjunktiven Ausdruck **or** $C_1 [] \ldots [] C_n$ **ro** enthält, liefert der Suchoperator die erste Möglichkeit C_1 und die restliche Disjunktion **or** $C_2 [] \ldots [] C_n$ **ro** in zwei

Lambda-Abstrationen E_1 und E_2, die zusätzlich die bisher berechneten Ergebnisse enthalten, als Parameter von distributed zurück. Die verschiedenen Reduktionswege können nun parallel weiterverfolgt werden.

In Curry benötigt man keine expliziten Disjunktionen für die Anwendung der eingekapselten Suche. Jede Funktion, die durch einen nichtdeterministischen Schritt ausgewertet werden kann, ist zur Durchführung einer eingekapselten Suche verwendbar. Wenn wie im folgenden Beispiel der nichtdeterministische Schritt nur durch Variablen in den Parametern hervorgerufen wird, kann dieselbe Funktion bei Übergabe von Parametern ohne diese Variablen auch für deterministische Reduktionen benutzt werden.

Beispiel 5.19
Da wir `add` wie bisher auch durch Narrowing auswerten, kann durch Übergabe von Variablen eine Suche gestartet werden, während ohne Verwendung von Variablen eine deterministische Auswertung möglich ist.

$$\texttt{add(add(s(z),z),s(z))} \rightsquigarrow \texttt{...} \rightsquigarrow \texttt{s(s(z))}$$

$$\texttt{add(X,Y)=s(z)} \rightsquigarrow \texttt{...} \rightsquigarrow \{\texttt{X=z,Y=s(z)}\} \texttt{ | } \{\texttt{X=s(z),Y=z}\}$$

$$\texttt{all(\textbackslash X -> \{local Y in add(X,Y)=s(z))\}} \rightsquigarrow\texttt{...}\rightsquigarrow \texttt{[\textbackslash X -> \{X=z\}, \textbackslash X -> \{X=s(z)\}]}$$

Ob wir eine Funktion für deterministische Reduktionen oder für die Durchführung einer (eingekapselten) Suche benutzen, hängt in Curry also nicht wie in Oz von ihrer Definition, sondern von der Art ihrer Verwendung ab. $\diamond$

In der jüngsten Version von Oz ist das Konzept der eingekapselten Suche überarbeitet worden. Statt der beschriebenen Form der Solver wird ein direkterer und flexiblerer Zugriff auf lokale Räume ermöglicht, wodurch Suchmethoden zur Verfügung gestellt werden können, die bisher in der Constraint-Programmierung nicht möglich waren. Eine genaue Beschreibung des neuen Konzept gibt Schulte in [31].

5.3 Die operationale Semantik der Committed-Choice

In Abschnitt 3.7 haben wir das `choice`-Konstrukt in der Form

$$\begin{array}{ll}\texttt{choice} & \{c_1\} \texttt{ -> } e_1\texttt{;} \\ & \vdots \\ & \{c_n\} \texttt{ -> } e_n\end{array}$$

mit bewachten Ausdrücken $\{c_i\}$ `->` e_i vorgestellt. Die Idee der Reduktion besteht darin, alle Wächter c_i nach einer fairen Strategie auszuwerten, bis ein c_i zum leeren Constraint `{}` reduziert worden ist, und dann den gesamten `choice`-Ausdruck durch den zugehörigen Rumpf zu ersetzen:

$$\texttt{choice }\{c_1\}\texttt{->}e_1;\ldots;\{c_n\}\texttt{->}e_n \quad\rightsquigarrow\quad \begin{cases}\{\exists V:\ id[]\sigma(e_i) & \text{wenn } \exists i\in\{1,\ldots,n\} \\ & \text{mit } c_i \rightsquigarrow \ldots \rightsquigarrow \exists V:\sigma[]\texttt{\{\}} \\ \emptyset & \text{sonst}\end{cases}$$

Wenn ein Wächter erfüllt werden konnte, liefern wir die dabei aufgetretenen Variablen zurück und wenden zudem die berechnete Substitution auf den Rumpf an. Wieso wir die Variablen, aber nicht die Substitution im Antwortausdruck zurückgeben müssen, erläutern wir in Abschnitt 5.3.2. Die Anwendung der Substitution auf den Rumpf ist notwendig, um beispielsweise die `last`-Funktion aus Abschnitt 4.1.4 durch Committed-Choice formulieren zu können:

```
last(L) if {append(Xs,[E])=L} = E.
last_c(L) = choice {append(Xs,[E])=L} -> E.
```

Die Bindung der Variable E, die im Wächter berechnet wird, soll in den Rumpf weitergereicht werden. Anhand der merge-Funktion haben wir jedoch in Abschnitt 3.7 erläutert, daß die Auswertung der Wächter in lokale Räume eingekapselt werden muß, um eine gegenseitige Beeinflussung sowie eine globale Vervielfältigung zu verhindern und die Unterscheidung zwischen lokalen und globalen Variablen zu ermöglichen. Damit E und Xs im lokalen Raum des Wächters gebunden werden können, müssen wir sie also explizit lokal deklarieren. In der Form

```
last_c(L) = choice {local [Xs,E] in append(Xs,[E])=L} -> E.
```

würden wir jedoch die Sichtbarkeit von E auf den Wächter beschränken, so daß die Bindung nicht in den Rumpf weitergereicht werden könnte. Ähnlich wie bei den bedingten Regeln in Abschnitt 4.1.4 erlauben wir daher die Verwendung von Existenzquantoren vor dem Wächter:

```
last(L) if local E in {local Xs in append(Xs,[E])=L} = E.
last_c(L) = choice local E in {local Xs in append(Xs,[E])=L} -> E.
```

E ist im Wächter (der Bedindung) und dem Rumpf (der rechten Seite) sichtbar, während Xs nur innerhalb des Wächters (der Bedingung) bekannt ist. Man beachte, daß die Verwendung der Existenzquantoren in der last-Regel optional, in der last_c-Regel hingegen zwingend erforderlich ist, da andernfalls eine Normalform von last_c gemäß Definition 4.7 aus Abschnitt 4.1.4 erstellt würde, in der E und Xs vor einer leeren Bedingung und damit außerhalb des choice-Konstrukts deklariert werden. Die Syntax des choice-Konstrukts geben wir nun in Form einer EBNF an (vgl. Anhang C.2).

Definition 5.4 (Kontextfreie Syntax eines choice-Ausdrucks)

ChoiceExpr	::=	choice { *ChoiceBranch*$_1$; ... *ChoiceBranch*$_n$ [;] }	$n \geqslant 0$
ChoiceBranch	::=	[local [*VarId*$_1$, ... *VarId*$_k$] in] *Constraint* -> *Expr*	$k > 0$

□

Beispiel 5.20
choice kann wie eine normale Funktion verwendet werden, beispielsweise als Parameter einer anderen Funktion oder eines Konstruktors. Der Zieltyp von choice wird durch die Typen der Rümpfe festgelegt.

```
s(choice local X in {add(z,X)=s(z)} -> X)
4 + choice(local X in {X=3} -> X*X) * 5
```

Aus Gründen der Übersichtlichkeit kann wie im letzten Beispiel die Folge der bewachten Ausdrücke hinter choice eingeklammert werden. Man beachte, daß nach der Semantik des Existenzquantors die Sichtbarkeit der lokalen Variablen auf den choice-Ausdruck beschränkt ist und dort äußere Vorkommen gleichnamiger Variablen überlagert:

```
f(X) = X + choice(local X in {X=1} -> X*X) * X.

f(2) ⤳ 2 + choice(local X in {X=1} -> X*X}) * 2
```

◇

Um die korrekte Arbeitsweise eines Existenzquantors, einer Substitutionsanwendung und der Funktion nf aus Abschnitt 4.1.4 zur Erstellung der Normalform von Regeln sicherzustellen, benötigen wir erneut eine erweiterte Definition eines freien Variablenvorkommens.

Definition 5.5 (Freies Vorkommen)
Ein Vorkommen einer Variable X in einem Ausdruck e heißt frei, wenn es nicht in einem Teilterm $e|_p$ der Form

1. $e|_p =$ `{local` $[..,X,..]$ `in` c`}` oder
2. $e|_p =$ `local` $[..,X,..]$ `in {`c`} ->` e

auftritt und e nicht Teilterm des Constraints `{c}` oder der rechten Seite `r` einer bedingten Regel der Form

l `if local` $[..,X,..]$ `in {`c`} =` r

ist. Die Menge aller Variablen, die in e frei vorkommen, bezeichnen wir mit $free(e)$. □

Da wir mit dem Suchoperator bereits ein umfangreiches Konzept zur Einkapselung von Berechnungen geschaffen haben, könnte man auf die Idee kommen, das `choice`-Konstrukt durch einen entsprechenden Algorithmus unter Anwendung des Suchoperators zu realisieren. Wir müßten dazu jeden Wächter durch einen `try`-Aufruf in einen eigenen lokalen Raum einkapseln und seine Reduktion überwachen, um festzustellen, ob er erfolgreich reduziert werden konnte und wir den `choice`-Ausdruck durch den zugehörigen Rumpf ersetzen können.

`choice` soll jedoch eine faire Strategie verwirklichen, bei der ein in endlich vielen Schritten erfüllbarer Wächter seine Reduktion auch dann erfolgreich beenden kann, wenn die Auswertung anderer Wächter suspendiert oder unendliche Berechnungen zur Folge hat. Mit dem bisherigen Sprachumfang haben wir aber weder die Möglichkeit, eine unendliche Reduktion, die einmal begonnen hat, wieder zu unterbrechen, noch können wir eine suspendierte von einer nicht suspendierten Reduktion unterscheiden. Wir benötigen daher einen zusätzlichen Überwachungsmechanismus, der jedem Wächter gleichermaßen die Reduktion ermöglicht, einen vollständig reduzierten Wächter erkennt und die übrigen verwirft.

5.3.1 Choice-Räume

Sowohl die Committed-Choice als auch der Suchoperator kapseln Berechnungen in lokale Räume ein, ihre Erzeugung und die Auswertung ihrer Reduktionsergebnisse unterscheiden sich jedoch in einigen Punkten:

1. Die Committed-Choice benötigt keine Lambda-Abstraktionen als Eingabe, da wir bei der Reduktion der Wächter keinen Wert für eine bestimmte Variable suchen, sondern allgemein die Erfüllbarkeit überprüfen wollen und zunächst einmal an allen dabei berechneten Bindungen interessiert sind.

2. Wenn ein Wächter vollständig reduziert wurde, wollen wir die berechnete Substitution auch nicht in Form einer Lambda-Abstraktion zurückgegeben, sondern direkt auf den zugehörigen Rumpf anwenden.

3. Es soll *eine beliebige* Lösung berechnet werden. Kann einer der Wächter einen nichtdeterministischen Schritt ausführen und ist der lokale Raum stabil, liefern wir daher die verschiedenen Reduktionsrichtungen nicht wie der Suchoperator zurück, sondern erzeugen für jeden möglichen Weg einen eigenen lokalen Raum, der gleichberechtigt mit allen bisherigen Räumen an der Berechnung teilnimmt. Dieses Prinzip wird auch in AKL [19] verwendet und dort als „choice splitting“ bezeichnet. Es sorgt dafür, daß automatisch alle möglichen Reduktionsrichtungen verfolgt werden, bis eine Lösung gefunden wurde.

Gerade im Fall eines „choice splitting“ müssen wir uns merken, welcher lokale Raum zu welchem Rumpf gehört, und daher verwenden wir wie beim Suchoperator eine spezielle Form des lokalen Raumes: den Choice-Raum.

Definition 5.6 (Choice-Raum)
Ein Choice-Raum ist ein Zweituppel

$$(l, e) \ ,$$

wobei l einen lokalen Raum und e einen Ausdruck bezeichnen. □

Bei der Auswertung des `choice`-Konstrukts durch eine zusätzliche Regel für die Reduktionsfunktion cse, die in Abbildung 5.6 dargestellt ist, wird jeder bewachte Ausdruck in einen Choice-Raum umgewandelt. Der Wächter wird in die erste Komponente, den lokalen Raum eingetragen, der Rumpf in die zweite. Ein Existenzquantor vor dem Wächter wird sofort reduziert, indem die deklarierten Variablen sowohl im Wächter als auch im Rumpf umbenannt und in die Menge der Variablen des lokalen Raums eingetragen werden. Dadurch können wir nach erfolgreicher Reduktion des Wächters die Bindung dieser Variablen einfach durch Anwendung der berechneten Substitution in den Rumpf weiterreichen (vgl. Abbildung 5.7). Die Menge

$$
\begin{array}{lcl}
cse(V, \texttt{choice}\ e_1; \ldots; e_n) & = & \{\exists V : id [] \texttt{cSpaces}([cr(e_1), \ldots, cr(e_n)])\} \qquad n > 0 \\[1em]
cr(\texttt{\{}c\texttt{\} -> }e) & = & (\ (\emptyset, id, c)\ , e) \\
cr(\texttt{local [}X_1, \ldots, X_n\texttt{] in \{}c\texttt{\} -> }e) & = & (\ (\{\tilde{X}_1, \ldots, \tilde{X}_n\}, id, c[X_i/\tilde{X}_i])\ , e[X_i/\tilde{X}_i]) \\
 & & \text{für } i = 1, \ldots, n, \text{ mit } \tilde{X}_i \text{ neue Variablen}
\end{array}
$$

Abbildung 5.6: Operationale Semantik des `choice`-Konstrukts

der Choice-Räume sammeln wir in einer Liste, die wir durch das Schlüsselwort `cSpaces` kennzeichnen. Die Verwendung einer Liste, einer Folge oder einer beliebigen anderen Darstellung, die eine Ordnung auf der Menge der Choice-Räume ermöglicht, ist notwendig, um eine einfache operationale Semantik für die Verwirklichung einer fairen Strategie angeben zu können. Für eine bessere Lesbarkeit stellen wir im folgenden eine Menge von Choice-Räumen nicht als `cSpaces([(l1,r1),...,(ln,rn)])`, sondern in der Form `cSpaces(l1->r1,...,ln->rn)` dar.

Beispiel 5.21
Die Definition der `merge`-Funktion mit korrekter Verwendung der Existenzquantoren haben wir bereits in Abschnitt 3.7 vorgestellt.

```
merge(L1,L2) = choice {L1=[]}                   -> L2;
                      {L2=[]}                   -> L1;
                      local [E,R] in {L1=[E|R]} -> [E|merge(R,L2)];
                      local [E,R] in {L2=[E|R]} -> [E|merge(L1,R)].
```

```
merge([1],[2])
```

$\overset{*}{\leadsto}$ `cSpaces(`$(\emptyset, id,$`[1]=[]) -> [2],`
$(\emptyset, id,$`[2]=[]) -> [1],`
$(\{E_1, R_1\}, id,$ `[1]=[`E_1`|`R_1`]) -> [`E_1`|merge(`R_1`,[2])],`
$(\{E_2, R_2\}, id,$ `[2]=[`E_2`|`R_2`]) -> [`E_2`|merge([1],`R_2`)])`

◇

Das grundsätzliche Reduktionsprinzip ist in Erweiterung von Abbildung 3.3 aus Abschnitt 3.7 mit Verwendung von Choice-Räumen in Abbildung 5.7 dargestellt. Der Einfachheit halber haben wir dort keine Existenzquantoren vor der Bedingung verwendet, so daß die V_k zu Beginn leere Mengen bezeichnen.

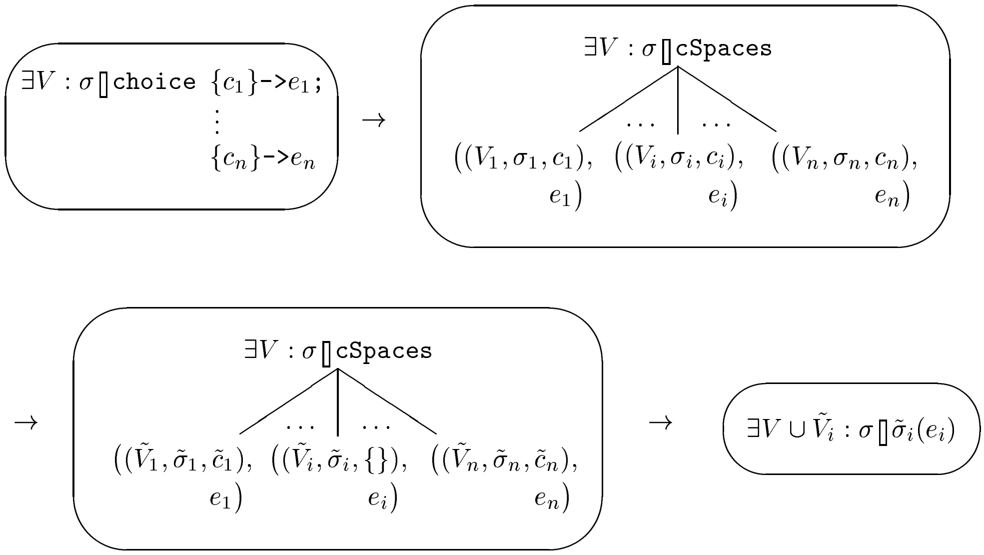


Abbildung 5.7: eingekapselte Berechnung durch Committed-Choice

In einem lokalen Raum dürfen nur die Variablen gebunden werden, die im oder vor dem Wächter deklariert wurden, und daher ist die Substitution des lokalen Raums nach seiner Auflösung nicht von Interesse und wird deshalb nicht zurückgeliefert. Auf den Rumpf müssen wir sie aber anwenden, da wie im Beispiel der `merge`- oder der `last_c`-Funktion lokale Variablen im Wächter *und* im Rumpf sichtbar sein können. In Beispiel 5.24 werden wir erklären, wieso wir im Gegensatz zur Substitution die Menge der Variablen des lokalen Raums zurückliefern müssen.

Im nächsten Abschnitt definieren wir eine operationale Semantik für die Liste der Choice-Räume, die alle bisher erläuterten Reduktionsprinzipien realisiert.

5.3.2 Die operationale Semantik von Choice-Räumen

Zunächst übergeben wir durch eine weitere Regel für *cse*, die in Abbildung 5.8 dargestellt ist, die Auswertung eines `cSpaces`-Konstrukts an eine eigene Reduktionsfunktion *cseChoice*. Die Menge der globalen Variablen V muß mit übergeben werden, damit *cseChoice* einen korrekten Antwortausdruck generieren kann.

$$\boxed{cse(V, \texttt{cSpaces}(S)) \quad = \quad cseChoice(V, S)}$$

Abbildung 5.8: Operationale Semantik des `choice`-Konstrukts

Da wir eine Einzelschrittsemantik angeben, darf immer nur einer der Choice-Räume einen Reduktionsschritt durchführen, und daher benötigen wir eine Ordnung auf der Menge der

Choice-Räume, um eine faire Strategie verwirklichen zu können. Wir haben uns hier für eine Darstellung durch eine Liste entschieden und verwenden bei der Definition der Funktion $cseChoice$, die in Abbildung 5.9 dargestellt ist, die Nonationen $[\,]$ für eine leere und $[S|Xs]$ für eine Liste mit mindestens einem Element, sowie $++$ für die Konkatenation zweier Listen.

Für die Realisierung einer fairen Strategie durchlaufen wir die Liste der Räume von Anfang an und verschieben einen Raum, der suspendiert ist oder einen Schritt durchführen konnte, ans Ende der Liste, so daß er erst dann erneut ausgewertet wird, nachdem alle anderen Räume die Chance zu einem Reduktionsschritt hatten. Man kann sich die Räume also in einer Art Ringstruktur vorstellen, in der ein Zeiger den aktuellen Raum markiert, nach dessen Reduktionsschritt eine Position weiter rückt und auf den nächsten Raum zeigt.

$$cseChoice(V, \texttt{cSpaces}(S)) \quad = \quad choiceEval(V, S, [\,])$$

$$choiceEval(V, [(l,e) \mid S], R)$$
$$= \begin{cases} \{\exists V \cup V_l : id[\!]\sigma(e)\} & \text{wenn } cslocal(l) = \{(V_l, \sigma)\} \\ choiceEval(V, S, R++[(l,e)]) & \text{wenn } cslocal(l) = \perp \\ \{\exists V : id[\!]\texttt{cSpaces}(S\texttt{++}R)\} & \text{wenn } cslocal(l) = \emptyset \\ & \text{und } S\texttt{++}R \neq [\,] \\ \emptyset & \text{wenn } cslocal(l) = \emptyset \\ & \text{und } S\texttt{++}R = [\,] \\ \{\exists V : id[\!]\texttt{cSpaces}(S\texttt{++}R\texttt{++}[((V_l, \sigma_l, c_l), e)])\} & \text{wenn } cslocal(l) = \{(V_l, \sigma_l, c_l)\} \\ \{\exists V : id[\!]\texttt{cSpaces}(S\texttt{++}R\texttt{++}[(l_1, e), \ldots, (l_n, e)])\} & \text{wenn } cslocal(l) = \{l_1, \ldots, l_n\} \\ & \text{mit } n > 1 \end{cases}$$

$$choiceEval(V, [\,], R) \quad = \quad \perp$$

Abbildung 5.9: Reduktionssemantik von Choice-Räumen

Wir können Choice-Räume, die keinen Reduktionsschritt durchführen können, nicht einfach ans Ende der Liste anhängen, da wir sonst in eine unendliche Schleife laufen würden, wenn alle Räume suspendieren. Wir verwenden daher einen zusätzlichen Listenparameter, in dem wir solche Räume sammeln, und rufen eine Hilfsfunktion $evalChoice$ auf, der wir zusätzlich eine leere Liste übergeben. Man könnte diesen Schritt einsparen, wenn man diese Liste bereits von cse an $cseChoice$ übergeben ließe, aber aus Modularisierungsgründen sollte für cse die konkrete Realisierung der Reduktion der Choice-Räume nicht sichtbar sein.

$choiceEval$ verwendet die Funktion $cslocal$ aus Abschnitt 5.1.1, um den lokalen Raum l des ersten Choice-Raums der Liste auszuwerten. Wenn ein gelöster Raum (V_l, σ) zurückgeliefert wird, ist ein erfüllbarer Wächter gefunden worden und das gesamte `choice`-Konstrukt kann durch den zugehörigen Rumpf ersetzt werden.

Sowohl bei der Erzeugung der Choice-Räume als auch bei ihrer Reduktion durch $choiceEval$ wird eine strikte Trennung zwischen globalen und lokalen Variablen vorgenommen, so daß σ nur lokale Variablenbindungen enthalten kann. Die Vereinigung mit der Substitution des globalen Raums würde daher nach Abschnitt 5.1.1 keinen Widerspruch erzeugen, aber da keine der Variablen aus $dom(\sigma)$ in $\sigma(e)$ oder an anderer Stelle des globalen Raums auftreten kann, wird die lokale Substitution nicht mehr benötigt. Ähnlich sind wir ja auch bei der Umwandlung eines Suchergebnisses in eine Lambda-Abstraktion in Abschnitt 5.2.2 vorgegangen, als wir nur die Bindung für die Suchvariable zurückgeliefert haben und nicht die gesamte Substitution. Wie wir in Beispiel 5.24 zeigen werden, können hingegen Variablen aus V_l in $\sigma(e)$ auftreten, so daß V_l zur Menge der Variablen des globalen Raums hinzugefügt werden muß. Man beachte, daß V und V_l aufgrund der erwähnten Trennung zwischen lokalen und globalen Variablen disjunkt sind.

Wenn die Auswertung des lokalen Raums l suspendiert, wird der Choice-Raum ans Ende der zweiten Liste verschoben, die alle bislang suspendierten Räume enthält, und *choiceEval* erneut aufgerufen, um einen reduzierbaren Raum zu finden.

Schlug die Berechnung fehl, werden der Choice-Raum entfernt und die restlichen Räume wieder in einer Liste zurückgegeben. Wenn jedoch außer dem fehlgeschlagenen kein weiterer Raum existiert, schlägt das gesamte `choice`-Konstrukt fehl, und daher wird die leere Menge zurückgeliefert.

Ein Choice-Raum, der einen deterministischen Reduktionsschritt durchführen konnte, wird in seiner neuen Form an das Ende der Liste verschoben. Bevor er erneut durch *cslocal* ausgewertet werden kann, erhält jeder andere Raum die Möglichkeit zur Reduktion, und dadurch erreichen wir die beschriebene faire Strategie.

Als letzte Möglichkeit kann der lokale Raum l einen nichtdeterministischen Reduktionsschritt durchgeführt haben, so daß *cslocal* eine Menge von neuen lokalen Räumen zurückliefert. Dies ist möglich, da Curry die Verwendung beliebiger Ausdrücke, inbesondere Funktionsapplikationen, in den Wächtern erlaubt (sog. „deep guards", vgl. Abschnitt 2.2.3), und daher nichtdeterministische Schritt möglich sind. Nach dem Prinzip des „choice splitting" (vgl. Abschnitt 5.3.1) wird der alte Choice-Raum durch einen Menge von neuen Räumen ersetzt, die alle denselben Rumpf besitzen, und aufgrund der fairen Strategie am Ende der Liste zurückgeliefert werden.

Wenn alle Choice-Räume suspendiert sind und in die zweite Liste verschoben wurden, wird durch die zweite Regel von *choiceEval* der gesamte `cSpaces`-Ausdruck suspendiert, da kein reduzierbarer Choice-Raum existiert.

Man beachte, daß die Reduktionsfunktionen die Forderungen aus Abschnitt 5.1.1 erfüllen: Informationen aus einem lokalen Raum können nicht nach außen gelangen, bevor er gelöst wurde, und ein Konflikt durch widersprüchliche Bindungen zweier Räume ist auch nicht möglich. Zwar erzeugt *evalChoice* nach einem nichtdeterministischen Schritt eines Wächters mehrere Choice-Räume mit gleichlautenden Variablennamen, aber sobald ein Raum gelöst wurde, werden alle anderen verworfen, so daß immer nur die Bindungen *einer* Substitution in den globalen Raum gelangen.

Die Arbeitsweise der Reduktionsfunktionen demonstrieren wir anhand einiger Beispiele. Aus Platzgründen kennzeichnen wir in manchen Fällen die Choice-Räume bei ihrer Erzeugung durch Buchstaben und verwenden diese während der Reduktion.

Beispiel 5.22
In Abschnitt 3.7 hatten wir gezeigt, daß der Aufruf `merge([1],[2])` zwei verschiedene Ergebnisse durch je drei unterschiedliche Reduktionswege berechnen kann. Durch unsere sequentielle operationale Semantik wird jedoch immer derselbe Weg beschritten: Reduktion durch den dritten und den ersten Wächter. Die Erzeugung der Choice-Spaces wird wie in Beispiel 5.21 vorgenommen.

$$
\begin{array}{lll}
 & \texttt{merge([1],[2])} & \\
\overset{*}{\leadsto} & \texttt{cSpaces((}\emptyset\texttt{,}id\texttt{,[1]=[]) -> [2],} & \texttt{A} \\
 & \quad\texttt{(}\emptyset\texttt{,}id\texttt{,[2]=[]) -> [1],} & \texttt{B} \\
 & \quad\texttt{(\{E}_1\texttt{,R}_1\texttt{,}id\texttt{,[1]=[E}_1\texttt{|R}_1\texttt{]) -> [E}_1\texttt{|merge(R}_1\texttt{,[2])],} & \texttt{C} \\
 & \quad\texttt{(\{E}_2\texttt{,R}_2\texttt{,}id\texttt{,[2]=[E}_2\texttt{|R}_2\texttt{]) -> [E}_2\texttt{|merge([1],R}_2\texttt{)])} & \texttt{D}
\end{array}
$$

$\leadsto$ `cSpaces([B,C,D])`
$\leadsto$ `cSpaces([C,D])`

Die ersten beiden Choice-Räume wurden sofort gelöscht, da ihre Wächter bereits beim ersten Reduktionsschritt einen Fehlschlag verursachten. Die beiden verbleibenden Wächter führen nun abwechselnd einen Reduktionsschritt durch und tauschen dabei fortwährend ihre Position, da ja ein Choice-Raum nach seiner Reduktion ans Ende der Liste verschoben wird. Der Einfachheit halber betrachten wir nur die lokalen Räume der Wächter und lassen die Rümpfe und `cSpaces` weg.

$\leadsto (\{E_1,R_1\},id,[1]=[E_1 \mid R_1])$	$(\{E_2,R_2\},id,[2]=[E_2 \mid R_2])$	C D
$\leadsto (\{E_2,R_2\},id,[2]=[E_2 \mid R_2])$	$(\{E_1,R_1\},id,1=E_1\ /\backslash\ []=R_1))$	D C
$\leadsto (\{E_1,R_1\},id,1=E_1\ /\backslash\ []=R_1)$	$(\{E_2,R_2\},id,2=E_2\ /\backslash\ []=R_2))$	C D
$\leadsto (\{E_2,R_2\},id,2=E_2\ /\backslash\ []=R_2)$	$(\{R_1\},\{E_1=1\},[]=R_1))$	D C
$\leadsto (\{R_1\},\{E_1=1\},[]=R_1)$	$(\{R_2\},\{E_2=2\},[]=R_2))$	C D
$\leadsto (\{R_2\},\{E_2=2\},[]=R_2)$	$(\emptyset,\{E_1=1,R_1=[]\},\{\}))$	D C
$\leadsto (\emptyset,\{E_1=1,R_1=[]\},\{\})$	$(\emptyset,\{E_2=2,R_2=[]\},\{\}))$	C D
$\leadsto$ `[1|merge([],[2])]`		

Im letzten Schritt wird der lokale Raum des ersten Choice-Raums durch *cslocal* in einen gelösten Raum der Form $(\emptyset,\{E_1=1,R_1=[]\})$ überführt. Wir haben also einen erfüllbaren Wächter gefunden und reduzieren daher zu dessen Rumpf. Die erneute Auswertung von `merge` wird wieder vier Choice-Räume erzeugen, von denen diesmal der zweite und dritte fehlschlagen. Übrig bleiben die Räume für den ersten Wächter `[]=[]` und den vierten `[2]=[E|R]`. Der erste Wächter benötigt nur einen Reduktionsschritt und wird daher schneller zu `{}` recduziert, so daß `merge([],[2])` durch `[2]` ersetzt wird und wir das Ergebnis `[1,2]` erhalten. ◇

Im nächsten Beispiel demonstrieren wir, wie trotz eines suspendierenden und eines unendlich reduzierenden Wächters eine Lösung gefunden wird.

Beispiel 5.23
In der folgenden Reduktion suspendiert der Wächter für `f(X)`, da für eine Reduktion von `f` der Parameter keine Variable sein darf. Der zweite Wächter reduziert hingegen unendlich. Der Einfachheit halber betrachten wir diesmal bei der Reduktion nur die Wächter selbst, da die zugehörigen lokale Räume immer leere Variablenmengen und die identische Substitution enthalten. Zur besseren Übersicht markieren wir die Wächter durch Buchstaben.

```
f eval 1:rigid.
f(X) = {}.
list = [1|list].

choice {f(X)}       -> false;   A
       {list=list}  -> false;   B
       {add(z,z)=z} -> true     C
```

```
⇝ A: f(X)          B: list=list             C: add(z,z)=z
⇝ C: add(z,z)=z    A: f(X)                  B: [1|list]=list
⇝ A: f(X)          B: [1|list]=list         C: z=z
⇝ C: z=z           A: f(X)                  B: [1|list]=[1|list]
⇝ A: f(X)          B: [1|list]=[1|list]     C: {}
⇝ C: {}            A: f(X)                  B: 1=1 /\ list=list

⇝ true
```

Man beachte, daß der Raum für `f(X)` immer suspendiert und daher sofort der nächste Raum ausgewertet wird. Die Reduktion erfolgt also abwechselnd zwischen dem `list` und dem `add`-Wächter, wodurch wir die geforderte faire Strategie verwirklichen. Die Lösung wird gefunden, obwohl einer der Wächter suspendiert und die Berechnung des anderen nicht terminiert. Ohne Committed-Choice wäre dies nicht möglich.

```
choiceSim if {f(X)}        = false;
          if {list=list}   = false:
          if {add(z,z)=z}  = true.
```

Der Aufruf von `choiceSim` terminiert nicht, da alle drei Bedingungen vollständig geprüft werden müssen, um alle möglichen Lösungen zu berechnen. Da jedoch die Auswertung der zweiten Bedingung niemals terminiert, können wir nach Definition der Reduktionssemantik aus Abschnitt 2.3.4 kein Ergebnis ausgeben lassen, denn dazu müssen alle Berechnungen suspendiert oder erfolgreich beendet worden sein. ◇

Das nächste Beispiel zeigt, wieso wir im Gegensatz zur Substitution die Menge der Variablen des lokalen Raums zurückliefern müssen (vgl. Beispiel 4.13).

Beispiel 5.24

```
choice local X in {local N in X=add(s(z),N)} -> X
⤳ cSpaces(({X₁},id,local N in X₁=add(s(z),N)) -> X₁)

*
⤳ cSpaces(({N₁},{X₁=s(N₁)},{}) -> X₁)
⤳ ∃{N₁}: []s(N₁)
```

Im letzten Schritt lösen wir den lokalen Raum auf, wenden die Substitution auf den Rumpf X_1 an und ersetzen den `choice`-Ausdruck durch $s(N_1)$. Da N_1 nun im globalen Raum auftritt, muß es also auch zur Menge der dort deklarierten Variablen hinzugefügt werden. Anderfalls würde die folgende Reduktion suspendieren, da im letzten Schritt N_1 nicht gebunden werden dürfte.

```
{(choice local X in X=add(s(z),N) -> X) = s(z)}.

*
⤳ ∃{N₁}: []s(N₁)=s(z)
⤳ ∃{N₁}: []N₁=z
⤳ ∃∅: {N₁=z}
```

Bei der Definition der operationalen Semantik des Suchoperators in Abschnitt 5.2 haben wir aus genau demselben Grund bei der Umwandlung des Ergebnisses in eine Lambda-Abstraktion die verbliebenen lokalen Variablen wieder deklariert (vgl. Beispiel 5.14). ◇

Ein Choice-Raum kann genau wie ein Suchraum aus verschiedenen Gründen suspendieren: weil er einen nichtdeterministischen Schritt auszuführen versucht, aber nicht stabil ist, weil er eine nicht lokale Variable binden will oder weil er wie in Beispiel 5.23 eine Funktion mit einer Variable als Parameter aufruft, die durch Residuation ausgewertet wird. Das Stabilitätskriterium ist aus den in Abschnitt 5.1 erläuterten Gründen sinnvoll, kann aber manchmal auch hinderlich sein.

Beispiel 5.25

Die Funktion `last_c`, die wir am Anfang von Abschnitt 5.3 definiert haben, ist im Gegensatz zur Funktion `last` aus Beispiel 4.7 nicht in der Lage, für die Liste `[1,X]` als Ergebnis das letzte Element `X` zu berechnen.

```
last_c([1,X])
⇝ ∃{X}: id[]choice local E in {local Xs in append(Xs,[E])=[1,X]} -> E
⇝ ∃{X}: id[]cSpaces(({E1},id,local Xs append(Xs,[E1])=[1,X]) -> E1)
⇝ ∃{X}: id[]cSpaces(({E1,Xs1},id,append(Xs1,[E1])=[1,X]) -> E1)
```

An dieser Stelle suspendiert die Berechnung, da wir wie in allen Beispiel `append` durch Narrowing auswerten und daher als nächstes ein nichtdeterministischer Schritt durchgeführt werden müßte. Der lokale Raum ist jedoch nicht stabil, da der Wächter die globale Variable `X` enthält, so daß die Reduktion des gesamten `cSpaces`-Ausdrucks suspendiert. Sie kann nur durch eine Bindung der globalen Variable an anderer Stelle wieder angestoßen werden.

```
last_c([1,X])=L /\ X=2

*⇝ ∃{X,L}: id[]cSpaces(({E1,Xs1},id,append(Xs1,[E1])=[1,X]) -> E1)=L
              /\ X=2
⇝ ∃{L}: {X=2}[]cSpaces(({E1,Xs1},id,append(Xs1,[E1])=[1,2]) -> E1)=L
```

Der lokale Raum ist jetzt stabil und kann den nichtdeterministischen Schritt ausführen, wodurch nach dem Prinzip des „choice splitting" zwei neue Choice-Räume mit demselben Rumpf entstehen.

```
⇝ ∃{L}: {X=2}[]cSpaces(
                ({E1},{Xs1=[]},append([],[E1])=[1,2]) -> E1,
                ({E1,Y,Ys},{Xs1=[Y|Ys]},append([Y|Ys],[E1])=[1,2]) -> E1
                    )=L
```

Die Fortsetzung der Reduktion wird zum Fehlschlag des ersten, einer weiteren Verdoppelung des zweiten Raums und schließlich zum Endergebnis {`L=2,X=2`} führen.

Wir können also durch `last_c` nicht das letzte Element einer Liste berechnen, wenn es sich dabei um eine Variable handelt. In diesem Fall ist das Stabilitätskriterium hinderlich, denn die die Vervielfältigung des Choice-Raums kann wie gesehen auch durch eine Bindung von `X` nicht vermieden werden. Die Funktion `last`, die wir in Beispiel 4.7 definiert haben, kann hingegen im Aufruf `last([1,X])` die Lösung `X` berechnen. Allerdings wird dabei im Gegensatz zur gerade betrachteten Reduktion eine zwischenzeitliche Vervielfältigung des *globalen* Raums vorgenommen, und dies führt zu Konflikten bei Verwendung von Ein-/Ausgabeoperationen (vgl. Abschnitt 6.6). ◇

Genau wie Suchräume können auch Choice-Räume suspendieren, ohne daß die Reduktion jemals wieder angestoßen werden kann.

Beispiel 5.26

```
f eval 1:rigid.
f(X) = {}.

choice local X in {f(X)} -> X
⇝ cSpaces(({X1},id,f(X1)) -> X1)
⇝ ⊥
```

Die Auswertung des Choice-Raums kann niemals fortgesetzt werden, da X_1 durch den Wächter nicht gebunden werden kann und außerhalb des Choice-Raums gar nicht sichtbar ist. Man könnte auf die Idee kommen, derartige Räume zu löschen, da sie nicht zur erfolgreichen Reduktion eines Committed-Choice-Konstrukts beitragen können, sondern nur unnötige Arbeit für *evalChoice* bedeuten. In unserem Beispiel würden wir dadurch statt einer Suspension einen

Fehlschlag bewirken, da alle Choice-Räume gelöscht wurden. Diese Ergebnis ist jedoch falsch, denn ein `choice`-Ausdruck darf nur fehlschlagen, wenn die Unerfüllbarkeit aller Wächter festgestellt wurde. Der Constraint `f(X)` ist aber für jede Bindung von `X` an einen Konstruktorterm erfüllbar, auch wenn wir nie eine dieser Lösungen berechnen können. Wir dürfen daher den Choice-Raum nicht entfernen, obwohl seine Reduktion zu keinem Zeitpunkt fortgesetzt werden kann. ◇

Man könnte natürlich eine Markierung derartiger Choice-Räume vornehmen und sie von *evalChoice* direkt überspringen lassen. Da aber in der Praxis dieser Fall normalerweise nicht auftritt, haben wir darauf bei der Angabe der operationalen Semantik und auch in der TasteCurry-Implementierung verzichtet.

Gegenüber den bisherigen Reduktionsmechanismen bietet die Committed-Choice den Vorteil, daß eine Lösung, die durch unsere Reduktionsfunktionen in endlich vielen Schritten berechnet werden kann, auch gefunden und ausgegeben wird. Mit Hilfe des Suchoperators kann eine solche vollständige Strategie nicht programmiert werden, allerdings bietet er im Gegensatz zur Committed-Choice die Möglichkeit, beispielsweise alle Lösungen oder die beste zu berechnen, einen Fehlschlag kontrolliert abzufangen und Lösungsmengen durch Lambda-Abstraktionen darzustellen. `choice` liefert hingegen Lösungsmengen wie in Beispiel 5.24 als normale Ausdrücke zurück. Wie die Vorteile beider Konstrukte in einfacher Weise kombiniert werden können, zeigen wir im nächsten Kapitel.

Abschließend ziehen wir einen kurzen Vergleich mit der Verwendung von Committed Choice in den Sprachen AKL und Oz.

5.3.3 Committed-Choice in AKL und Oz

Die Sprache AKL unterstützt verschiedene Programmierparadigmen für funktionale, logische, objekt-orientierte, nebenläufige und Constraint-Programmierung. Wir wollen an dieser Stelle nur die Verwendung von „don't-care"-Nichtdeterminismus betrachten, der ähnlich wie in Curry durch einen Committed-Choice-Ausdruck der folgenden Form ermöglicht wird [19]:

$$\begin{array}{l}(\langle\text{statement}\rangle \;\mid\; \langle\text{statement}\rangle \\ ;\ \ldots \\ ;\ \langle\text{statement}\rangle \;\mid\; \langle\text{statement}\rangle\ )\end{array}$$

Jede Zeile ist ein bewachter Ausdruck, wobei das linke <statement> den Wächter und das rechte den Rumpf darstellt. Lokale Variablen können durch

$$X_1, \ldots, X_n : \langle\text{statement}\rangle \;\mid\; \langle\text{statement}\rangle$$

deklariert werden und sind in Wächter und Rumpf sichtbar. Das Reduktionsprinzip ist dasselbe wie in Curry: Ist ein Wächter erfüllbar, wird der gesamte Choice-Ausdruck durch den zugehörigen Rumpf ersetzt, sind mehrere Wächter erfüllbar, wird ein beliebiger ausgewählt, sind alle Wächter unerfüllbar, schlägt der Ausdruck fehl. Andernfalls suspendiert die Berechnung.

Beispiel 5.27
Die `merge`-Funktion wird in AKL als Relation definiert:

```
merge(L1,L2,Erg):=
  ( L1=[] | Erg=L2
  ; L2=[] | Erg=L1
  ; E,R: L1=[E|R] | Erg=[E|Erg1], merge(R,L2,Erg1)
  ; E,R: L2=[E|R] | Erg=[E|Erg1], merge(L1,E,Erg1) )
```

Das Ergebnis der Vereinigung der Listen `L1` und `L2` wird im dritten Parameter `L3` zurückgeliefert. Die `merge`-Relation kann jedoch nur funktional verwendet werden, denn auch in AKL dürfen in einem Wächter nur die lokalen Variablen gebunden werden, so daß ein Reduktionsschritt nur möglich ist, wenn `L1` oder `L2` an eine Liste gebunden sind. Im Rumpf dürfen hingegen auch andere Variablen, hier `Erg` und die neu eingeführte Variable `Erg1`, gebunden werden. ◇

Auch in AKL wird die Berechnung der Wächter eingekapselt[2], da beispielsweise für die Kommunikation mit dem Benutzer oder einem externen Speichermedium Prozesse verwendet werden, die nicht durch Suche in einem nichtdeterministischen Teil des Programms beeinflußt werden dürfen. Die Bedeutung und die Realisierung der Comitted Choice basiert also in AKL und Curry auf denselben Prinzipien.

In Oz wird „don't-care"-Nichtdeterminismus durch die Verwendung eines bedingten Ausdrucks der Form

if $C_1 \,[]\, \ldots \,[]\, C_n$ **else** F **fi**

realisiert, den wir schon in Beispiel 4.9 vorgestellt haben. Wie schon bei dem disjunktiven Ausdruck aus Abschnitt 5.2.3 handelt es sich bei jedem C_i um einen bewachten Ausdruck der Form

$x_1 \ldots x_n$ **in** E **then** D

mit Wächter E, Rumpf D und den (optionalen) lokalen Variablen $x_1, \ldots, x_n$, die in E und D sichtbar sind. Die Reduktion der Wächter wird wie in Curry in lokale Räume eingekapselt, wodurch dasselbe Prinzip der Trennung zwischen lokalen und globalen Variablen verfolgt wird. Ist einer der Wächter erfüllbar, wird der bedingte Ausdruck durch den entsprechenden Rumpf ersetzt, sind mehrere erfüllbar, wird ein beliebiger ausgewählt. Wenn aber alle Wächter unerfüllbar sind, wird statt eines Fehlschlags die Reduktion mit der Auswertung des Ausdrucks F fortgesetzt, der durch den **else**-Teil angegeben wurde. Eine solche Möglichkeit besteht in Curry und AKL nicht.

Beispiel 5.28
Die `merge`-Funktion kann durch einen bedingten Ausdruck definiert werden, wobei der **else**-Fall nie benötigt wird. Ein Aufruf der Prozedur Merge mit den Parametern L1, L2 und Erg1 wird in Oz in der Form {Merge L1 L2 Erg } geschrieben. Listen werden ohne eckige Klammern und eine leere Liste durch nil dargestellt.

```
proc {Merge L1 L2 Erg }
  if L1=nil then Erg=L2
  [] L2=nil then Erg=L1
  [] E R Erg1 in L1=E|R then Erg=E|Erg1 {Merge R L2 Erg1}
  [] E R Erg1 in L2=E|R then Erg=E|Erg1 {Merge L1 R Erg1}
  else false fi
end
```

◇

Das Reduktionsverhalten gleicht dem von Curry und AKL bis auf einen Punkt: Oz erlaubt ebenfalls die Verwendung von „deep guards", d.h. es können beliebige Ausdrücke und Funktionen in den Wächtern auftreten, insbesondere ein disjunktiver Ausdruck der Form

or $C_1 \,[]\, \ldots \,[]\, C_n$ **ro** ,

[2]Das Konzept der Einkapselung ist in AKL nicht so flexibel und mächtig wie in Oz, auf eine mögliche Erweiterung durch sogenannte „engines" wird aber in [19] hingewiesen.

den wir in Abschnitt 5.2.3 vorgestellt haben. Nach der dort beschriebenen Semantik suspendiert ohne Verwendung des Suchoperators die Berechnung, wenn mehr als eine Möglichkeit der Reduktion besteht. Wenn ein bedingter Ausdruck ausgewertet wird, führt daher das Auftreten eines nicht eindeutig reduzierbaren disjunktiven Ausdrucks in einem der Wächter zu dessen Suspension, anstatt wie in Curry und AKL durch „choice splitting“ für jede mögliche Reduktionsrichtung einen neuen lokalen Raum zu erzeugen, der gleichberechtigt mit allen bisherigen an der Auswertung des bedingten Ausdrucks teilnimmt. Dieses unterschiedliche Verhalten ist (ohne die Darstellung lokaler Räume und der Rümpfe) in Abbildung 5.10 skizziert. Der Einfachheit halber nehmen wir an, daß in Curry und AKL der Constraint c_i nur zwei verschiedene Reduktionswege einschlagen kann.

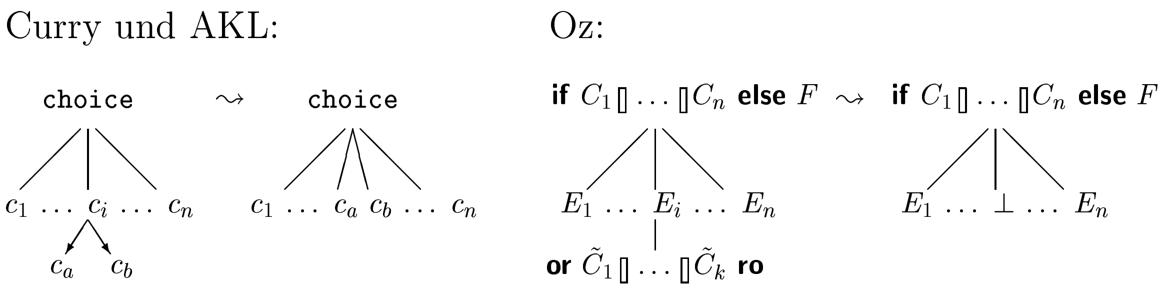


Abbildung 5.10: Behandlung von Nichtdeterminismus in Wächtern

Oz erlaubt jedoch die Verwendung von Suchoperatoren in den Wächtern, wodurch die Suspension bei Auftreten eines disjunktiven Ausdrucks vermieden wird und statt dessen die verschiedenen Alternativen weiter untersucht werden können [32].

Kapitel 6

Anwendungen der eingekapselten Suche

In diesem Kapitel betrachten wir einige Möglichkeiten der Verwendung der eingekapselten Suche. Zunächst demonstrieren wir die Programmierung von Suchalgorithmen und erläutern deren Arbeitsweise anhand des Beispiels der Wegesuche in einem Graph. Wir werden zeigen, daß die Forderungen aus Abschnitt 3.1 nach einer Kontrolle der Suchstrategie und der Behandlung von Lösungen als Datenobjekte ebenso realisierbar sind wie die Kontrolle über einen möglichen Fehlschlag und die Verwendung von nichtdeterministischen Reduktionen im Zusammenhang mit Ein-/Ausgabe. Die Simulation des `findall`-Prädikats aus Prolog durch den Suchoperator und eine Anwendung im Vergleich mit einem Beispiel aus der Sprache Escher werden wir ebenfalls vorstellen Abschließend erläutern wir, wieso wir nicht in der Lage sind, eine direkte Simulation der List-Comprehensions aus funktionalen Sprachen zu erreichen.

6.1 Suchalgorithmen

Der Suchoperator führt Suche nur bis zum Auftreten eines nichtdeterministischen Reduktionsschritts durch und liefert danach die verschiedenen Wege zurück. Die automatische Weiterverarbeitung dieser Wege nach einer bestimmten Strategie kann durch Suchalgorithmen vorgenommen werden, die aufgrund der Definition des Suchoperators genau wie in Oz auf einfache Art und Weise als Programme eingegeben werden können.

Curry stellt standardmäßig vier vordefinierte Algorithmen zur Verfügung, so daß der Benutzer in den meisten Fällen die direkte Verwendung des Suchoperators vermeiden kann. Drei der Algorithmen zum Auffinden aller (`all`), einer (`dfs`) oder der besten (`best`) der Lösungen arbeiten nach dem Prinzip der Tiefensuche und durchlaufen den Suchbaum sequentiell von links nach rechts. Der vierte Algorithmus (`one`) verwendet das Prinzip der Breitensuche, um eine beliebige Lösung zu finden. Die Arbeitsweise der `dfs`-, `all`- und `best`-Funktionen basiert auf den Algorithmen Depth und Best aus Oz [32], deren Definition sich in der neuesten Version jedoch geändert hat [31] .

Ein Algorithmus erhält eine Lambda-Abstraktion als Parameter und liefert eine Liste von Lösungen der Form

```
\X -> {local ... in X=e}
\X -> {}
```

zurück bzw. eine leere Liste, falls die Anfrage unerfüllbar ist. Zwischenergebnisse, die der Suchoperator möglicherweise erzeugt, werden also so lange weiterverfolgt, bis je nach Algorithmus eine oder mehrere Lösungen gefunden wurden.

```
all :: (A -> constraint) -> list(A -> constraint).

all(G) = evalResult(try(G))
       where
         evalResult([])      = [];
         evalResult([S])     = [S];
         evalResult([A,B|C]) = concat(map(all,([A,B|C]))).
```

Abbildung 6.1: Alle Lösungen durch Tiefensuche

Abbildung 6.1 zeigt den `all`-Algorithmus, der den Suchbaum nach dem Prinzip der Tiefensuche von links nach rechts durchläuft. Wir übergeben die Lambda-Abstraktion `G` an den Suchoperator und werten dessen Ergebnis durch die Funktion `evalResult` aus. Wenn keine oder eine Lösung existiert, gibt `all` entsprechend eine leere oder eine einelementige Liste zurück. Wenn der Suchoperator nach einem nichtdeterministischen Schritt verschiedene Lambda-Abstraktionen liefert, untersuchen wir durch rekursive Anwendung von `all` die verschiedenen Reduktionswege von links nach rechts. Die Ergebnislisten vereinigen wir durch die Funktion `concat` zu *einer* (möglicherweise leeren) Liste, die somit alle Lösungen enthält. `concat` verwendet die Funktion `(++)` aus Abschnitt 2.2.1 zur Verküpfung zweier Listen:

```
concat :: list(list(A)) -> list(A).
concat([])    = [].
concat([A|B]) = A ++ concat(B).
```

```
dfs :: (A -> constraint) -> list(A -> constraint).

dfs(G) = takefirst(all(G))
         where
           takefirst([])    = [];
           takefirst([A|B]) = A.
```

Abbildung 6.2: Eine Lösung durch Tiefensuche

Da Curry die Auswertung von Funktionsaufrufen so lange wie möglich verzögert (vgl. Abschnitt 2.3.1), können wir beispielsweise das erste Element der Ergebnisliste von `all` abgreifen, ohne daß die restlichen Alternativen ausgewertet werden. Für Fälle, in denen man nur an einer Lösung interessiert ist, stellen wir daher den `dfs`-Algorithmus aus Abbildung 6.2 zur Verfügung, der verschiedene Reduktionswege nur verfolgt, wenn es wirklich notwendig ist. Liefert beispielsweise der Suchoperator in der Definition von `all` die beiden Abstraktionen `A` und `B` zurück, wird `all(B)` nur aufgerufen, wenn durch `all(A)` keine Lösung gefunden werden konnte. Auf ähnliche Weise können natürlich auch mehrere Lösungen berechnet werden, indem man der Ergebnisliste von `all` entsprechend viele Elemente entnimmt, oder es kann eine interaktive Suche realisiert werden (vgl. Abschnitt 6.7).

Der `best`-Algorithmus in Abbildung 6.3 ermittelt durch Verwendung einer „branch-and-bound"-Strategie die beste Lösung bezüglich einer Ordnung, indem er die erste Lösung sucht und anschließend alle weiteren Alternativen derart modifiziert, daß sie nur noch eine bessere Lösung berechnen können. Mit jeder neuen gefundenen Lösung werden die Bedingungen für die verbleibenden Alternativen verschärft, so daß der Suchraum immer weiter beschnitten wird.

Die Ordnungsrelation übergeben wir in Form einer binären Funktion `Compare` des Typs `A -> A -> Bool`. Sie muß erfüllt sein, wenn die erste von zwei übergebenen, möglicherweise partiellen Lösungen die bessere ist. Die Vergleichsfunktion darf erst reduzieren, wenn Lösungen berechnet wurden, und daher muß sie durch Residuation ausgewertet werden. In Beispiel 6.3 werden wir sehen, daß es für den Vergleich oftmals nicht notwendig ist, beide Lösungen vollständig zu berechnen.

```
best :: (A -> constraint) -> (A -> A -> bool) -> list(A -> constraint).

best(G,Compare) = bestHelp([],try(G),[])
    where
      bestHelp([],[],Best)     = Best;
      bestHelp([],[B|Bs],Best) = evalB(try(constrain(B,Best)),Bs,Best);
      bestHelp([A|As],Bs,Best) = evalA(try(A),As,Bs,Best);

      evalB([],Bs,Best)        = bestHelp([],Bs,Best);
      evalB([NewBest],Bs,Best) = bestHelp([],Bs,[NewBest]);
      evalB([C,D|As],Bs,Best)  = bestHelp([C,D|As],Bs,Best);

      evalA([],As,Bs,Best)        = bestHelp(As,Bs,Best);
      evalA([NewBest],As,Bs,Best) = bestHelp([],As++Bs,[NewBest]);
      evalA([C,D|E],As,Bs,Best)   = bestHelp([C,D|E]++As,Bs,Best);

      constrain(B,[])  = B;
      constrain(B,[Best]) = \X -> {local Y in Best@Y /\ Compare@X@Y=true
                                                              /\ B@X}.
```

Abbildung 6.3: Beste Lösung durch Tiefensuche

Im letzten Parameter der lokalen Funktion `bestHelp` speichern wir die bisher beste berechnete Lösung, im ersten Parameter alle Alternativen, die bereits bezüglich dieser Lösung beschränkt wurden, und im zweiten alle noch nicht beschränkten. Bevor eine Alternative `B` aus dem zweiten Parameter untersucht wird, erweitern wir den enthaltenen Constraint um eine Vergleichsoperation mit der besten Lösung `Best` und erstellen eine neue Lambda-Abstraktion der Form

```
\X -> {local Y in Best@Y /\ Compare@X@Y=true /\ B@X}
```

Wir entpacken also den Rumpf von `B` und der bisher besten Lösung `Best` durch Applikation und fügen als zusätzlichen Constraint den Vergleich zwischen den beiden in der Form `Compare@X@Y=true` hinzu. Dieser ist nur erfüllbar, wenn `B` eine bessere Lösung als `Best` berechnen kann. Liefert die neue Lambda-Abstraktion ein Ergebnis zurück, können wir es daher als neue beste Lösung speichern. Da wir die Beschränkung einer Alternative erst unmittelbar vor ihrer Reduktion vornehmen, vermeiden wir in den meisten Fällen unnötige Erweiterungen um Vergleiche mit vormals besten Lösungen, die zum Zeitpunkt der Reduktion bereits verworfen wurden.

Falls bei der Reduktion der neuen Lambda-Abstraktion ein nichtdeterministischer Schritt auftritt, sind natürlich alle möglichen Wege bereits beschränkt, und daher sammeln wir sie im ersten Parameter von `bestHelp` und untersuchen sie weiter, bevor wir eine neue Alternative aus dem zweiten Parameter betrachten. Liefert das Verfolgen einer dieser Wege eine Lösung `NewBest`, speichern wir sie als neue beste und müssen alle anderen Wege in den zweiten Parameter schieben, da diese nun vor ihrer weiteren Reduktion bezüglich `NewBest` beschränkt werden müssen. In Abschnitt 6.2 werden wir die Arbeitsweise des `best`-Algorithmus anhand einer Beispielrechnung verdeutlichen.

```
one :: (A->constraint)->list(A->constraint).

one(G) = dfs(\X -> {X = choice local Y in {G@Y} -> Y }).
```

Abbildung 6.4: Eine Lösung durch Breitensuche

Abbildung 6.4 zeigt die nichtdeterministische Suchfunktion `one`, die durch Verwendung von Committed-Choice die Suche nach einer fairen Strategie durchführt. Wir entpacken die Lambda-Abstraktion im Wächter und lassen den Constraint dort lösen. Falls ein Ergebnis gefunden wird, liefern wir es im Rumpf des `choice`-Ausdrucks zurück und verpacken es wieder in eine Lambda-Abstraktion. Durch die zusätzliche Anwendung von `dfs` wird ein möglicher Fehlschlag der Berechnung abgefangen und durch eine leere Liste signalisiert. `one` hat also dasselbe Ein- und Ausgabeformat wie alle anderen Suchalgorithmen und vereint daher die Vorteile des Suchoperators (kontrollierter Abbruch und Lambda-Abstraktionen) mit der fairen Suchstrategie der Committed-Choice.

Ob die Anwendung von `one` oder von `dfs` sinnvoller ist, hängt von der jeweiligen Problemstellung ab. Die Breitensuche durch `one` ist in vielen Fällen ineffizienter, allerdings kann sie auch in vielen Fällen eine Lösung berechnen, in denen `dfs` versagt (vgl. Abschnitt 6.4).

Anhand der vorgestellten Algorithmen sollte deutlich geworden sein, auf welche Art und Weise in Curry Suchstrategien programmiert werden können. Weiterhin denkbar sind beispielsweise Strategien, die den Suchbaum von rechts nach links durchlaufen, oder auch die Realisierung einer bedingten Suche der Form

```
condSearch(G,Cond) = one(\X -> {G@X /\ Cond@X=true}),
```

bei der wir neben der Lambda-Abstraktion `G` eine Bedingung `Cond` vom Typ `A -> Bool` übergeben und ähnlich wie bei der `best`-Suchstrategie die ursprüngliche Abstraktion erweitern. Auf diese Weise kann nur eine Lösung zurückgeliefert werden, die die Bedingung erfüllt. Durch Verwendung von `all` statt `one` kann man natürlich auch alle Lösungen sammeln, die eine bestimmte Bedingung erfüllen.

In den folgenden Abschnitten betrachten wir verschiedene Anwendungsmöglichkeiten der Suchalgorithmen und zeigen dabei, daß der Suchvorgang auch durch Interaktion mit dem Benutzer gesteuert werden kann.

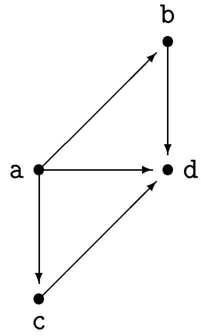
6.2 Wegesuche im Graph

Das folgende Programm zur Wegesuche im Graph haben wir in Abschnitt 2.2.2 erstmals vorgestellt.

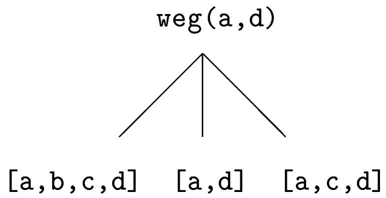
```
data knoten = a;b;c;d.

kante(a)=b.
kante(a)=d.
kante(a)=c.
kante(b)=c.
kante(c)=d.

weg(X,Y)  if {X=Y}         = [X]
          if {kante(X)=Z} = [X] ++ weg(Z,Y).
```



Zur Verdeutlichung der Arbeitsweisen unserer Suchalgorithmen betrachten wir den Aufruf `weg(a,d)`, der alle Wege von Knoten `a` nach Knoten `d` berechnet. Wenn wir davon ausgehen, daß wie in der aktuellen TasteCurry-Version die Regeln der `kante`-Funktion in der Reihenfolge ihrer Deklaration im Programm angewendet werden, erhalten wir die folgende Form eines einfachen Suchbaums, bei der wir die Wurzel mit der Anfrage und die Blätter mit den Lösungen bezeichnen:



Ein Algorithmus, der diesen Baum so wie unsere `all`-, `best`- und `dfs`-Algorithmen sequentiell von links nach rechts durchläuft, wird die Lösungen also in der Reihenfolge `[a,b,c,d]`, `[a,d]`, `[a,c,d]` finden.

Der Einfachheit halber gehen wir im folgenden davon aus, daß Regeln, deren Bedingungen nicht erfüllbar sind, nicht angewendet werden, und daß erfüllbare Bedingungen in einem Schritt reduziert werden. Statt bewachter Ausdrücke lassen wir daher durch den Suchoperator nach einem nichtdeterministischen Schritt immer sofort die rechten Regelseiten zurückliefern. Außerdem werden wir in manchen Fällen für eine übersichtlichere Darstellung unwichtige Berechnungsschritte sofort ausführen, auch wenn dies der eigentlichen Reduktionsreihenfolge widerspricht. Ein solcher Schritt wäre beispielsweise die Konkatenation mit der leeren Liste `[] ++` e , den wir immer sofort zu e vereinfachen. Wenn wir mehrere Reduktionsschritte zusammenfassen, markieren wir wie im letzten Kapitel den Reduktionspfeil mit einem Stern.

Beispiel 6.1 (Wegesuche durch `dfs`)
Um eine Lösung durch Tiefensuche zu finden, verpacken wir den Aufruf `weg(a,d)` in eine Lambda-Abstraktion und übergeben sie an `dfs`. Durch Anwendung der zweiten Regel der `weg`-Funktion wird ein nichtdeterministischer Schritt durchgeführt, da es drei Ergebnisse des Aufrufs `kante(a)=Z` gibt.

```
dfs(\X -> {X=weg(a,d)})

↝* takefirst(concat(map(all,[\X -> {X = [a] ++ weg(b,d)},
                             \X -> {X = [a] ++ weg(d,d)},      L2
                             \X -> {X = [a] ++ weg(c,d)}])))   L3
↝ takefirst(concat([all@(\X -> {X = [a] ++ weg(b,d)})|map(all,L2,L3)]))
```

`all` wird nun auf die erste Alternative angewendet und berechnet die Lösung `X=[a,b,c,d]`. Der vorige Ausdruck reduziert dann wie folgt weiter:

```
⤳* takefirst([\X -> {X=[a,b,c,d]}|concat(map(all,L2,L3))])
⤳  [\X -> {X=[a,b,c,d]}]
```

Aufgrund der verzögerten Auswertung und der Definition von `concat` (vgl. letzter Abschnitt) wird also die erste gefundene Lösung zurückgeliefert, ohne daß ein Reduktionsschritt für die beiden anderen Alternativen durchgeführt werden mußte. ◇

Bei Verwendung des `all`-Algorithmus werden alle drei Wege nacheinander in der oben erwähnten Reihenfolge berechnet.

Beispiel 6.2 (Wegesuche durch `all`)
Bis nach der Berechnung der ersten Lösung stimmt die Reduktion durch `all` mit der durch `dfs` bis auf das Fehlen der `takefirst`-Funktion überein. Daher wird die Anwendung von `all` auf die beiden verbleibenden Alternativen, die wir wieder mit `L2` und `L3` bezeichnen, tatsächlich ausgeführt.

```
all(\X -> {X=weg(a,d)})

⤳* [\X -> {X=[a,b,c,d]}|concat(map(all,L2,L3))]
⤳  [\X -> {X=[a,b,c,d]}|concat([all@L2|map(all,L3)])]

⤳* [\X -> {X=[a,b,c,d]},
    \X -> {X=[a,d]}|concat(map(all,L3))]

⤳* [\X -> {X=[a,b,c,d]},
    \X -> {X=[a,d]},
    \X -> {X=[a,b,d]}]
```

◇

Mit Hilfe der `best`-Suchstrategie sind wir in der Lage, beispielsweise den kürzesten Weg von `a` nach `d` zu finden, da wir während des Suchvorgangs Lösungen miteinander vergleichen können, so wie wir es in Abschnitt 3.1.2 gefordert hatten.

Beispiel 6.3 (Suche nach dem kürzesten Weg)
Die folgende Funktion vergleicht zwei Wege und reduziert zu `true`, wenn der erste kürzer ist, und ansonsten zu `false`.

```
shorter([],X)           = true.
shorter(X,[])           = false.
shorter([X|Xs],[Y|Ys]) = shorter(Xs,Ys).
```

Der Aufruf `best(\X -> {X=weg(a,d)},shorter)` soll also den kürzesten Weg von `a` nach `d` berechnen. In der folgenden Reduktion betrachten wir immer nur die drei Parameter von `besthelp`. Für eine einfachere Darstellung betrachten wir dabei statt der Lambda-Abstraktionen `\X -> {X=weg(..)}` nur die Constraints `X=weg(..)` und vereinfachen zudem die Form einer beschränkten Alternative. Statt

```
\X -> {local Y in Best@Y /\ shorter@X@Y=true /\ B@X)
```

liefern wir für `Best` = `\Z -> {Z=`e`}` und `B` = `\X ->` c direkt den Constraint

```
shorter(X,e)=true /\ c
```

zurück, den wir nach Anwendung des Suchoperators auf die Lambda-Abstraktion sowieso nach einigen Reduktionsschritten erhalten.

```
best(\X -> X=weg(a,d),shorter)
⇝* []                   | [X=weg(a,d)]                        | []
⇝* [X=[a|weg(b,d)],     | []                                  | []
     X=[a|weg(d,d)],
     X=[a|weg(c,d)]]
⇝* []                   | [X=[a|weg(d,d)], X=[a|weg(c,d)]]    | [X=[a,b,c,d]])
```

Da zu Beginn keine beste Lösung existiert, werden die drei Alternativen in den ersten Parameter veschoben, da sie ja derart „beschränkt“ sind, daß sie nur eine bessere Lösung berechnen können als „keine“. Das Ergebnis, das durch die erste Alternative gefunden wird, kann also zur momentan besten Lösung erklärt werden. Die zweite Alternative wird nun vor ihrer Untersuchung derart verändert, daß sie nur einen kürzeren Weg als `[a,b,c,d]` berechnen kann.

```
    [] | [X=[a|weg(d,d)], X=[a|weg(c,d)]]      | [X=[a,b,c,d]])
                  ↓
        shorter(X,[a,b,c,d]) /\ X=[a|weg(d,d)]

⇝* [] | [[X=[a|weg(c,d)]]                      | [X=[a,d]]
```

Da die zweite Alternative trotz der Beschränkung eine Lösung berechnet hat, kann diese nur besser sein als die bisherige, und wir speichern sie daher im dritten Parameter. Die letzte verbliebene Alternative wird nun bezüglich dieser neuen Lösung beschränkt.

```
    [] | [[X=[a|weg(c,d)]]                          | [X=[a,d]]
                 ↓
         shorter(X,[a,d])=true /\ X=[a|weg(c,d)]
      ⇝* shorter(X1,[d])=true  /\ X1=weg(c,d)
      ⇝* shorter(X1,[d])=true  /\ X1=[c|weg(d,d)]
      ⇝* shorter(X2,[])=true   /\ X2=weg(d,d))
      ⇝* false=true            /\ X2=weg(d,d))

⇝* [] | []                                          | [X=[a,d]]
```

Da wir den `shorter`-Constraint nach dem „test and generate“-Prinzip *vor* die Berechnung des neuen Weges gestellt haben und `shorter` durch Residuation ausgewertet wird, erfolgt die Reduktion der Vergleichsfunktion, sobald ein neuer Knoten berechnet wurde. Daher ist in diesem Fall eine Entscheidung bereits möglich, bevor der neue Weg vollständig berechnet wurde. Bei Vertauschung des Vergleichs und der Wegberechnung würden wir nach dem „test and generate“-Prinzip vorgehen und erst den kompletten neuen Weg berechnen, bevor er mit dem bisher kürzesten verglichen wird. Man beachte, daß diese Vorgehensweise im Gegensatz zu der von uns verwendeten nicht nur mehr Aufwand erfordern, sondern sogar versagen würde, wenn wir eine bisher beste Lösung gefunden hätten und danach eine Alternative untersuchen müßten, die einen unendlichen Weg erzeugt. ◇

Anhand des `best`-Algorithmus erkennen wir den in Abschnitt 3.3 erwähnten Vorteil in der Verwendung eines Suchoperators gegenüber der Suche durch List-Comprehensions. Die Erweiterung der Alternativen um zusätzliche Constraints während der Suche entspräche gewissermaßen einer Änderung der Generatoren einer List-Comprehension in Abhängigkeit von den bisher berechneten Lösungen. Da dies aber nicht möglich ist, müßten wir das funktionale Programm zur Wegesuche in einem Graph, das wir in Abschnitt 3.3 vorgestellt haben, selbst ändern, um nicht alle Wege vollständig berechnen lassen zu müssen. In Curry sind wir hingegen in der Lage, verschiedene Suchstrategien anzuwenden, ohne das Programm selbst anpassen zu müssen.

Beispiel 6.4 (Wegesuche durch `one`)
Wenn wir durch Kombination von `dfs` und `choice` einen Weg von `a` nach `d` suchen lassen, wird der Suchbaum Ebene für Ebene durchlaufen. In unserer sequentiellen Implementierung wird automatisch die Lösung gefunden, die durch den kürzesten Pfad im Suchbaum erreichbar ist. Im Beispiel der Wegesuche handelt es sich dabei tatsächlich auch um den kürzesten Weg von `a` nach `d`.

`one(\X -> {X=weg(a,d)})`
$\leadsto$ `dfs(\X -> {X = choice(local Y in {(\X -> {X=weg(a,d)})@Y} -> Y)})`

$\overset{*}{\leadsto}$ searchSpace(X_1,({X_1},id,(X_1=cSpaces(({Y_1},id,Y_1=weg(a,d)) -> Y_1))

Der `one`-Algorithmus arbeitet also mit einer Schachtelung lokaler Räume. Durch „choice splitting" führt die Reduktion im Choice-Raum zur Erzeugung drei neuer Räume, deren Auswertung nach der in Abschnitt 5.3 erläuterten fairen Strategie erfolgt. Wir betrachten die Reduktion nur ausschnittweise und verzögern der Einfachheit halber die Auswertung von `Y=`e so lange, bis e ein Grundterm ist.

..cSpaces(({Y_1},id,Y_1=weg(a,d)) -> Y_1)

$\overset{*}{\leadsto}$ ..cSpaces(({Y_1},id,Y_1=[a|weg(b,d)]) -> Y_1,
({Y_1},id,Y_1=[a|weg(d,d)]) -> Y_1,
({Y_1},id,Y_1=[a|weg(c,d)]) -> Y_1)

$\overset{*}{\leadsto}$ ..cSpaces(({Y_1},id,Y_1=[a,d]) -> Y_1,
({Y_1},id,Y_1=[a,c|weg(d,d)]) -> Y_1,
({Y_1},id,Y_1=[a,b|weg(c,d)]) -> Y_1)

$\overset{*}{\leadsto}$ searchSpace(X_1,({X_1},id,(X_1=[a,d])))

$\overset{*}{\leadsto}$ `[\X -> {X=[a,d]}]`

Die Reduktion des Wächters `Y\downn{1}=[a|weg(d,d)]` wird als erste beendet, und daher wird die Lösung `[a,d]` als Rumpf zurückgeliefert, die durch den umgebenden Suchraum in eine Lambda-Abstraktion verpackt wird. ◇

Nach diesen etwas ausführlicheren Reduktionen, die die Arbeitsweise der Suchalgorithmen verdeutlichen sollten, betrachten wir kurz noch einige weitere Auswertungen. Wir verwenden dazu eine Hilfsfunktion, um die Ergebnisse aus den Lambda-Abstraktionen entpacken und direkt ausgeben zu können:

```
unpack([]) = [].
unpack([A|B]) if A@X = [X|unpack(B)].
```

`unpack` erhält eine Liste von Lambda-Abstraktionen und liefert eine Liste mit den für die Suchvariable berechneten Lösungen zurück. `L1` bezeichnet die bisher betrachtete Lambda-Abstraktion `\X -> {X=weg(a,d)}`.

```
unpack(dfs(L1))             ⇝ ... ⇝ [[a,b,c,d]]
unpack(all(L1))             ⇝ ... ⇝ [[a,b,c,d],[a,d],[a,b,d]]
unpack(best(L1,shorter))    ⇝ ... ⇝ [[a,d]]
unpack(one(L1))             ⇝ ... ⇝ [[a,d]]
```

Mit Hilfe von `condSearch` können wir beispielsweise einen Weg suchen, der eine bestimmte Länge (Anzahl Knoten - 1) unterschreitet. Die vordefinierte Funktion `length` (vgl. Anhang B) berechnet die Länge einer Liste.

```
shorterN(N,List) = length(List) < N+1.

unpack(condSearch(L1,shorterN(1))) ⇝ ... ⇝ []
unpack(condSearch(L1,shorterN(2))) ⇝ ... ⇝ [[a,d]]
```

Durch `L2` und `L3` lassen sich alle von `a` ausgehenden Wege bzw. alle Knoten, von denen aus `d` erreichbar ist, berechnen.

```
L2 = \X -> {local L in X=weg(a,L)}.
L3 = \X -> {local L in L=weg(X,d)}.

unpack(all(L2)) ⇝ ... ⇝ [[a],[a,b],[a,b,c],[a,b,c,d],[a,d],[a,c],[a,c,d]]
unpack(one(L2)) ⇝ ... ⇝ [[a]]
unpack(best(L2,shorter)) ⇝ ... ⇝ [[a,d]]

unpack(all(L3)) ⇝ ... ⇝ [d,a,a,a,b,c]
unpack(dfs(L3)) ⇝ ... ⇝ [d]
```

Da drei Wege von `a` nach `d` existieren, berechnet `all(L3)` dreimal das `a`, und da jeder Knoten einen direkten Weg zu sich selbst besitzt, findet `dfs(L3)` das `d`, da dies die erste Lösung im Suchbaum ist.

Ein sinnvolle Anwendung dieser Wegesuche kann man beispielsweise erreichen, wenn man die `weg`-Funktion derart erweitert, daß in einem Graph mit Zykeln kein Knoten zweimal besucht wird. Durch die Verwendung von Städten als Knoten und die Markierung der Kanten durch Entfernungen oder Zeiträume, könnte man beispielsweise für das Streckennetz der Deutschen Bundesbahn eine beliebige, alle oder die kürzeste bzw. schnellste Verbindung zwischen zwei Städten berechnen lassen. Durch `searchCond` kann man z.B. nur diejenigen Verbindungen ausgeben lassen, die über eine bestimmte Stadt führen. Ein entsprechendes Beispielprogramm ist auf der beigefügten Diskette enthalten (vgl. Anhang A)[1].

6.3 Simulation des `findall`-Prädikats

In Abschnitt 3.2 haben wir das `findall`-Prädikat aus Prolog vorgestellt, mit dem alle Lösungen einer Anfrage in einer Liste gesammelt werden können. Das gleiche tut auch unser `all`-Algorithmus, der allerdings Lambda-Abstraktionen zurückliefert, die wir aber durch `unpack` extrahieren können. Wir definieren daher in Curry eine Funktion `findall` wie folgt:

```
findall(G) = unpack(all(G)).
```

Sie unterscheidet sich von dem `findall`-Prädikat aus Prolog dadurch, daß alle lokalen Variablen explizit deklariert werden müssen, während dies in Prolog automatisch getan wird.

Beispiel 6.5
In Abschnitt 2.2.4 haben wir die Großvater-Relation in Curry durch boolesche Funktionen definiert:

```
vater(hans,klaus)  = true.
vater(hans,karl)   = true.
vater(klaus,peter) = true.
```

[1]Wenn also in einigen Jahren beispielsweise ein Kunde an den Bahnschalter kommt und wissen möchte, wie er am schnellsten von Aachen nach München fahren kann, wird der Bahnbeamte seinen Curry-Interpreter starten und die gewünschte Auskunft geben können...

```
vater(klaus,egon)  = true.
vater(karl,stefan) = true.

grossvater(X,Y) if {vater(X,Z)=true, vater(Z,Y)=true} = true.
```

Wir nehmen an, daß in Prolog eine entsprechende Definition durch Prädikate vorgenommen sei und zudem das Programm zur Wegesuche mit dem im letzten Abschnitt verwendeten Graph gemäß Abschnitt 3.2 ebenfalls in Prolog definiert sei. Dann sind folgende Reduktionen möglich:

```
findall(X,weg(a,d,X),L).           ⇝  L = [[a,b,c,d],[a,d],[a,c,d]] ?

findall(X,grossvater(hans,X),L).   ⇝  L = [peter,egon,stefan] ?

findall(X,vater(X,Y),L).           ⇝  L = [hans,hans,klaus,klaus,karl] ?
```

Durch die letzte Anfrage berechnen wir alle Väter. Da `findall` das `Y` existenzquantifiziert, werden dessen Bindungen nicht zurückgeliefert. In Curry formulieren wir die entsprechenden Anfragen wie folgt:

```
findall(\X -> {X=weg(a,d)})                    ⇝  [[a,b,c,d],[a,d],[a,c,d]]

findall(\X -> {grossvater(hans,X)=true})       ⇝  [peter,egon,stefan]

findall(\X -> {local Y in vater(Y,X)=true})    ⇝  [hans,hans,klaus,klaus,karl]

findall(\X -> {vater(Y,X)=true})               ⇝  ⊥
```

Anhand der letzten Reduktion sehen wir, daß eine Simulaton des `findall`-Prädikats aus Prolog nur möglich ist, wenn wir wirklich alle lokalen Variablen explizit angeben, da sonst der Versuch, `Y` zu binden, suspendiert. ◇

Die Formulierung in Curry ist durch Verwendung von Lambda-Abstraktionen und die Notwendigkeit der expliziten Deklaration etwas komplizierter als in Prolog. Dafür bietet Curry im Gegensatz zu Prolog die Möglichkeit der flexiblen Programmierung verschiedener Suchstrategien und die Koordinierung der Berechnung durch Suspension und Variablenbindung bei Verwendung nicht lokal deklarierter Variablen in den Lambda-Abstraktionen (vgl. Beispiele in Kapitel 5). In Prolog kann hingegen ein solcher Suspensionsmechanismus nur durch zusätzliche Deklarationen erreicht werden, beispielsweise durch `block`-Anweisungen in SICStus Prolog [28].

Eine weitere Anwendung der `findall`-Funktion betrachten wir anhand eines Beispielprogramms aus der funktional-logischen Sprache Escher, das Lloyd in [22] vorstellt und das zusammen mit der Formulierung der nachfolgend betrachteten Beispielreduktionen von Hanus nach Curry übertragen und freundlicherweise für diese Arbeit zur Verfügung gestellt wurde. Aus Platzgründen drucken wir das Curry-Programm nicht ab, es liegt aber auf der Diskette unter dem Namen `england.fl` bei (vgl. Anhang A). Es unterscheidet sich nur geringfügig von der Formulierung in der Sprache Escher, die eine Notation mit Mengen verwendet, die in Curry nicht zur Verfügung steht.

In dem Programm werden eine Menge von Bezirken und Städten sowie ihre Beziehung zueinander durch die drei Funktionen `neighbours`, `distance` und `isin` definiert. `neighbours(X,Y)` und `isin(T,X)` sind wahr, wenn die Bezirke `X` und `Y` aneinander grenzen bzw. wenn die Stadt `T` im Bezirk `X` liegt. Die Funktion `distance(X,Y)` liefert die Distanz zwischen den Städten `X` und `Y` zurück und ist in Escher als dreistelliges Prädikat definiert.

Die Mengennotationen, die Escher in den folgenden Beispielreduktionen verwendet, sollten ohne große Erklärungen verständlich sein. In Curry können wir durch Verwendung des `findall`-Prädikats Listen berechnen, die dieselben Elemente wie die Antwortmengen von

Escher enthalten. Dabei simulieren wir den „in"-Operator durch das `member`-Prädikat und die boolesche Disjunktion `||` aus Escher durch zwei boolesche Funktionen `\/` für die Disjunktion boolescher Terme und `\\//` für die Disjunktion von Constraints. Neben einem Existenzquantor `exists` gibt es jedoch in Escher anders als in Curry auch einen Allquantor. Der Ausdruck

forall [x_1,...,x_n] (*domain* --> *cond*)

ist wahr, wenn für alle $x_1, \ldots, x_n$, deren Bereich durch *domain* festgelegt wird, die Bedingung *cond* erfüllt ist. Beispielsweise ist der Ausdruck

```
forall [y] (y 'in' {Cricket, Tennis} --> likes(Mary,y))
```

wahr, wenn Mary sowohl Cricket als auch Tennis mag. Ein relationaler Gebrauch, um alle Leute zu ermitteln, die Cricket und Tennis mögen, ist ebenfalls möglich. In Curry kann dieser Allquantor mit Hilfe der `findall`-Funktion durch

```
forall :: (A->constraint) -> (A->constraint) -> constraint.
forall(Domain,Cond) = foldr(/\,{},map(Cond,findall(Domain))).
```

simuliert werden. Das Beispiel aus Escher wird wie folgt formuliert:

```
forall(\X -> {member(X,[cricket,tennis])=true}, \Y -> {likes(mary,Y)=true})
```

Das `X` entspricht der allquantifizierten Variable in Escher. Zuerst wird durch `findall(Domain)` die Wertemenge berechnet, dann wird die Bedingung auf jedes enthaltene Element angewendet. Wenn ein Element die Bedingung nicht erfüllt, schlägt die Berechnung fehl. Anderfalls erhält man eine Liste von leeren Constraints, die durch `foldr` zu *einem* leeren Constraint verknüpft werden, der als Ergebnis zurückgeliefert wird.

Nachfolgend führen wir einige der Beispiele aus dem Escher-Programm zusammen mit der von Hanus erstellten äquivalenten Formulierung in Curry auf. Wir bilden aber nur die Antwort ab, die der Escher-Ausdruck berechnet. Wenn man die Mengen- durch Listenklammern ersetzt und die Städte mit Kleinbuchstaben notiert, erhält man den Antwortausdruck in Curry, der durch das auf der Diskette beigefügte Programm berechnet wird.

Beispiel 6.6

1. *Finde alle Städte, die weniger als 40 Meilen von Bristol entfernt sind.*

```
{x | exists [y] (distance(Bristol,x,y) && y<40)}
forall(\X -> {distance(bristol,X)<40 = true})

⇝ {Gloucester,Bath}
```

2. *Finde alle Städte, die in Nachbarbezirken von Oxfordshire liegen.*

```
{x | exists [Y] ((neighbours(Oxfordshire,y) || neighbours(y,Oxfordshire))
                  && isin(x,y))}
findall(\X ->{local Y in
                 ((neighbours(oxfordshire,Y) \/ neighbours(Y,oxfordshire))
                  && isin(X,Y)) = true})

⇝ {Salisbury,Gloucester,Cirencester,Cheltenham}
```

3. *Finde alle Städte in den Bezirken Devon, Cornwall, Somerset und Avon.*

```
{x | exists [y] (y `in` {Devon,Cornwall,Somerset,Avon} && isin(x,y)}
findall(\X -> {local Y in (member(Y,[devon,cornwall,somerset,avon])
                           && isin(X,Y)) = true})

~> {Torquay,Plymouth,Exeter,Penzance,Truro,Taunton,Bristol,Bath}
```

4. *Liegt in allen Nachbarbezirken von Avon mindestens eine Stadt?*

```
forall [x]   ((neighbours(Avon,x) || neighbours(x,Avon)) --> isin(_,x))
forall(\X -> {(neighbours(avon,X) \/ neighbours(X,avon)) = true},
       \X -> {local Y in isin(Y,X) = true})
~> True
```

Curry liefert in diesem Fall den leeren Constraint zurück.

5. *Liegen alle Städte, die weniger als 40 Meilen von Bristol entfernt sind, im selben Bezirk wie Bristol?*

```
exists [x] in (isin(Bristol,x) &&
 forall [z]
  (exists [y] (((distance(Bristol,z,y) || distance(z,Bristol,y)) && y<40))
                --> isin(z,x)))
local X in isin(bristol,X) = true /\
 forall(\Z -> {local Y in
               (({distance(bristol,Z) = Y} \\// {distance(Z,bristol) = Y})
                  && Y<40) = true},
        \Z -> {isin(Z,X) = true})

~> False
```

Curry liefert in diesem Fall „`No solution`“ zurück. ◇

Weitere Beispielreduktionen können in [22] nachgelesen werden, ihre Formulierungen in Curry sind in der Datei `england.fl` enthalten.

6.4 Tiefen- und Breitensuche

Da die Suchfunktionen `all`, `best` und `dfs` nach dem Prinzip der Tiefensuche arbeiten, den Suchbaum also sequentiell von links nach rechts durchlaufen, kann das Berechnen einer Lösung durch einen unendlichen Zweig im Suchbaum verhindert werden. Wenn wir beispielsweise die `add`-Funktion wie bisher durch die beiden Regeln

```
add(z,N)    = N.
add(s(M),N) = s(add(M,N)).
```

definieren und davon ausgehen, daß die Regeln in der Reihenfolge der Deklaration angewendet werden, berechnet der Aufruf `add(Y,z)` bei der Auswertung durch Narrowing unendlich viele Lösungen

```
{Y=z}  {Y=s(z)}  {Y=s(s(z))}  ... ,
```

die jedoch niemals ausgegeben werden, da die Berechnung nicht terminiert. Dasselbe gilt für den Aufruf

```
all(\X -> {local Y in X=add(Y,z)}),
```

der die Berechnung der unendlichen Liste von Lösungen niemals beenden kann. Wir können aber aufgrund der verzögerten Auswertung von Curry bereits berechnete Lösungen aus der Ergebnisliste von `all` entnehmen und ausgeben, beispielsweise durch den `dfs`-Algorithmus:

```
unpack(dfs(\X -> {local Y in X=add(Y,z)})) ⇝ ... ⇝ z
```

Wenn wir aber die beiden Regeln für die Additionsfunktion vertauschen, können wir mit den bisherigen Aufrufen überhaupt keine Lösung mehr berechnen:

```
addRev(s(M),N) = s(add(M,N)).
addRev(z,N)    = N.
```

Wenden wir wieder `all` auf eine entsprechende Lambda-Abstraktion an, wird nach einem nichtdeterministischen Schritt immer zunächst die erste der Alternativen verfolgt, die jeweils durch Anwendung der ersten Regel von `addRev` entsteht und daher eine unendliche Lösung zu berechnen versucht:

```
all(\X -> {local Y in X=addRev(Y,z)})
⇝ ... ⇝ [\X -> {local A in X=addRev(s(A),z)},
          \X -> {X=addRev(z,z)}]
⇝ ... ⇝ [\X -> {local [A,B] in X=s(A) /\ A=addRev(s(B),z)},
          \X -> {local A in X=s(A) /\ A=addRev(z,z)},
          \X -> {X=addRev(z,z)}]
⋮
```

Diese Listen erhalten wir natürlich nicht als Ergebnis, sie werden innerhalb des `all`-Algorithmus generiert. Das erste Element der Ergebnisliste von `all` wird also niemals abschließend berechnet, und daher terminiert der Aufruf

```
dfs(\X -> {local Y in X=addRev(Y,z)})
```

nicht. Natürlich könnte man einen Algorithmus `dfsRev` entwerfen, der den Suchbaum nicht von links nach rechts, sondern in umgekehrter Reihenfolge durchläuft, indem er die Alternativen, die der Suchoperator nach einem nichtdeterministischen Schritt zurückliefert, von rechts nach links untersucht. Ein solcher Algorithmus würde aber wiederum bei Verwendung der `add`- anstelle der `addRev`-Regeln keine Lösung berechnen können.

Wenn die Gefahr besteht, daß durch einen unendlichen Berechnungszweig das Auffinden einer Lösung verhindert wird, sollte man daher auf den `one`-Algorithmus zurückgreifen, der nach dem Prinzip der Breitensuche den Suchbaum Ebene für Ebene durchläuft und daher sowohl für `add` als auch für `addRev` eine Lösung berechnet.

```
unpack(one(\X -> {local Y in X=add(Y,z)})) ⇝ ... ⇝ z

unpack(one(\X -> {local Y in X=addRev(Y,z)})) ⇝ ... ⇝ z
```

Wenn wir an mehreren Lösungen interessiert sind, gibt es jedoch keine Möglichkeit, mit Breitensuche zu arbeiten, da die Committed-Choice sich immer für *einen* der erfüllbaren Wächter entscheidet, wir ohne ihre Verwendung aber keine faire Strategie realisieren können. Zudem erfordert in den meisten Fällen eine Breitensuche einen erheblich höheren Aufwand als eine

Tiefensuche. Wenn wir uns beispielsweise einen Graph ohne Zykel vorstellen, in dem fünf verschiedene Wege der Länge 5 zwischen zwei Knoten existieren, und annehmen, daß wir einen Schritt benötigen, um von einem Knoten aus zum nächsten zu gelangen, berechnet `dfs` die Lösung in 5, `one` hingegen in 21 Schritten. `one` wird außerdem eine Menge überflüssiger lokaler Räume erzeugen, was beispielsweise bei der Realisierung lokaler Räume durch eigene Threads in einer parallelen Implementierung einen ziemlich hohen, unnötigen Aufwand bedeuten kann.

Es gibt also keine optimale Suchstrategie, sondern wir müssen uns je nach Problemstellung für eine bestimmte entscheiden. Wollen wir mehr als eine oder eine beste Lösung berechnen, müssen wir auf Tiefensuche durch Verwendung von `all` oder `best` zurückgreifen. Falls wir nur an einer Lösung interessiert sind und keine Kentnisse über die Struktur des Suchbaums besitzen, können wir durch Verwendung von Breitensuche mittels der `one`-Strategie eine Lösung finden, sofern diese in endlich vielen Schritten berechenbar ist. Wissen wir sicher, daß keine unendlichen Zweige im Suchbaum existieren, können wir durch Verwendung von `dfs` statt `one` den Effizienzvorteil von Tiefensuche gegenüber Breitensuche ausnutzen.

6.5 Kontrolle des Fehlschlags

Eine Motivation für die Implementierung einer eingekapselten Suche war der Wunsch nach einer Kontrolle des Fehlschlags.

Beispiel 6.7
In Abschnitt 3.1.3 hatten wir die Funktion

```
sumOne(X,Y) if {add(X,Y)=s(z)} = true.
```

betrachtet, die nicht in der Lage ist, `false` zurückzuliefern, wenn die Summe ihrer Argumente nicht `s(z)` ergibt.

sumOne(s(z),z) $\rightsquigarrow$ ... $\rightsquigarrow$ true
sumOne(s(z),s(z)) $\rightsquigarrow$ ... $\rightsquigarrow$ $\emptyset$

Durch Verwendung eines der Suchalgorithmen läßt sich dieses Manko beheben:

```
sumOne(X,Y) if {SearchResult = [A]} = true
            if {SearchResult = []}  = false
             where
               SearchResult = dfs(\L -> {add(X,Y)=s(z)}).
```

sumOne(s(z),z) $\rightsquigarrow$ ... $\rightsquigarrow$ true
sumOne(s(z),s(z)) $\rightsquigarrow$ ... $\rightsquigarrow$ false

Wenn wir an `sumOne` Variablen als Parameter übergeben, können wir jedoch keine Lösung berechnen, da `X` und `Y` im Suchraum, in dem `dfs` die Suche durchführt, nicht gebunden werden dürfen. Auf dieses Problem gehen wir im nächsten Abschnitt noch einmal ein. $\diamond$

Da ein Suchalgorithmus in jedem Fall eine Liste zurückliefert, müssen wir in manchen Fällen gar keine Sonderbehandlung eines Fehlschlags vornehmen, sondern können leere und nicht leere Listen durch dieselbe Routine behandeln lassen.

Beispiel 6.8
Wir möchten aus einer Liste von Peano-Zahlen diejenigen löschen, die kleiner oder gleich `s(z)` sind.

[z, s(s(s(z))), s(z), s(s(A))] $\rightarrow$ [s(s(s(z))), s(s(A))]

Wir müssen also jedes Element der Liste mit dem Muster `s(s(X))` zu unifizieren versuchen, und dabei kann es natürlich zu einem Fehlschlag kommen, der nicht zu einem Fehlschlag des gesamten Programms führen darf. Wir kapseln daher die Unifiktion durch einen Suchalgorithmus ein. Anstatt aber dessen Ergebnis auf eine leere oder eine nicht leere Liste zu unterscheiden, konkatenieren wir einfach sämtliche Ergebnislisten.

```
biggerOne([])     = [].
biggerOne([X|Xs]) = unpack(dfs(\Y -> {local B in s(s(B))=X /\ Y=X}))
                    ++ biggerOne(Xs).
```

Für Elemente, die nicht mit `s(s(X))` unifizierbar sind, wird durch `dfs` eine leere Liste zurückgeliefert, so daß sie in der Ergebnisliste nicht mehr auftauchen.

```
biggerOne([z, s(s(s(z))), s(z), s(s(A))]) ⇝ ... ⇝ [s(s(s(z))), s(s(A))]
```

Falls wir übrigens ein Element übergeben, von dem nicht entscheidbar ist, ob es kleiner oder gleich `s(z)` ist, suspendiert die Berechnung an dieser Stelle (und dadurch für den gesamten Ausdruck).

```
biggerOne([z, s(s(s(z))), s(z), s(A)])
⇝ ... ⇝ [s(s(s(z)))|... try(\Y -> {local B in s(s(B))=s(A)...}...]
⇝ ⊥
```

Je nachdem, an welchen Wert `A` noch gebunden wird, kann `s(A)` in der Liste verbleiben oder muß entfernt werden. Die Suspension wird durch den Versuch hervorgerufen, bei der Reduktion von `s(s(B))=s(A)` im Suchraum die globale Variable `A` an den Term `s(B)` zu binden. ◇

In Abschnitt 6.7 betrachten wir zwei weitere Beispiele, die einen Fehlschlag einer Berechnung durch eine entsprechende Routine behandeln.

6.6 Verhinderung gobaler Vervielfältigung

In Abschnitt 3.1.4 haben wir erklärt, wieso im Zusammenhang mit Ein-/Ausgabe eine Vervielfältigung des globalen Berechnungsraums verhindert werden muß. Wir werden in diesen Abschnitt zeigen, wie nichtdeterministische Reduktionen eingekapselt werden können, dabei aber auch die Notwendigkeit zur Einkapselung bestimmter deterministischer Funktionen kennenlernen. Außerdem werden wir sehen, daß wir nicht in der Lage sind, den *Dynamic-Cut* aus Babel zu simulieren, und daß nicht in allen Fällen eine Einkapselung möglich ist, ohne Lösungen zu verlieren.

Beispiel 6.9
Betrachten wir die folgende boolesche Funktion:

```
p(1) = true.
p(2) = true.
```

Bei Übergabe einer Variable reduziert `p` durch einen nichtdeterministischen Narrowingschritt, der für die Verdoppelung des Berechnungsraums sorgt:

```
p(X) ⇝ ... ⇝ {X=1}[]true | {X=2}[]true
```

Um diese Berechnung einzukapseln, können wir beispielsweise die Committed Choice verwenden:

```
choice {local X in p(X)=true} -> true ⇝ ... ⇝ true

choice local X in {p(X)=true} -> X  ⇝ ... ⇝ 1
```

Auf diese Weise liefern wir allerdings eine andere Form des Antwortausdrucks zurück als in der ursprünglichen Reduktion. Wir können dieses Problem lösen, indem wir eine Funktion definieren und die Suche in ihrer Bedingung durchführen lassen. Der vielleicht intuitive Ansatz

```
f(X) if choice {p(X)=true} -> true  = true.
```

ist jedoch nicht erfolgreich, da wir `X` als globale Variable im Wächter des `choice`-Ausdrucks nicht binden können. Wir müssen daher einen Umweg wählen und die berechnete Bindung außerhalb von `choice` noch einmal vornehmen:

```
f(X) if {X = (choice local Y in {p(Y)=true} -> Y)} = true.

f(X) ⇝ ... ⇝ {X=1}[]true
```

Wenn wir davon ausgehen, daß der `choice`-Ausdruck immer die Lösung `Y=1` berechnet, führt der Aufruf von `f(2)` nun zu einem Fehlschlag, obwohl `p(2)` erfüllbar ist. Wir können also nicht durch ein- und dieselbe Funktion einerseits Lösungen überprüfen und andererseits die Berechnung von Lösungen eingekapselt durchführen lassen.

Wenn wir in einem Programm mehr als eine Lösung verwenden wollen, müssen wir die Einkapselung durch `all` oder `findall` vornehmen:

```
findall(\X -> {p(X)=true})) ⇝ ... ⇝ [1,2]
```

Wenn eine Funktion verschiedene Substitutionen *und* Ergebnisse berechnet, kann man beispielsweise Paare verwenden, um beide Anteile zurückzuliefern. ◇

Bei nichtdeterministischen Berechnungen müssen wir uns also entscheiden, welche oder wieviele Lösungen wir benötigen und in welcher Form sie zurückgeliefert werden sollen. Dabei muß beachtet werden, daß durch die Einkapselung die Verwendungsmöglichkeit einer Funktion eingeschränkt werden kann.

Auch bei deterministischen Reduktionen, die nur ein Ergebnis berechnen können, kann es in Curry zur Vervielfältigung des globalen Raums kommen.

Beispiel 6.10
Wir können die `member`-Funktion beispielsweise durch die drei Regeln

```
member(X,[])                            = false.
member(X,[Y|Ys]) if {X=Y}               = true.
member(X,[Y|Ys]) if {member(X,Ys)=true} = true.
```

definieren. Durch Narrowing können verschiedene Lösungen berechnet werden:

```
member(X,[1,2]) ⇝ ... ⇝ {X=1}[]true | {X=2}[]true
```

Bei Übergabe zweier Grundterme sollte jedoch eigentlich nur *ein* Ergebnis `true` oder `false` berechnet werden:

```
member(1,[1,2,1,3,1]) ⇝ ... ⇝ true | true | true
```

Curry berechnet jedoch dreimal die Antwort `true`, da aufgrund der überlappenden linken Seiten der zweiten und dritten Regel immer beide angewendet werden, solange die Liste nicht leer ist, und daher für jede 1 in der Liste das Ergebnis `true` berechnet wird. Dreimal dasselbe Ergebnis zu berechnen ist natürlich sinnlos und führt zu einer unnötigen Vervielfältigung des globalen Raums. ◇

Abhilfe könnte in solchen Fällen der sogenannte *Dynamic-Cut* schaffen, der beispielsweise in der Sprache Babel verwendet wird und zur Laufzeit entscheiden kann, ob nach Berechnung einer Lösung die Auswertung weiterer Alternativen notwendig ist oder nicht [24]: Wenn ein Ausdruck e durch eine (bedingte) Regel reduziert werden kann, ohne daß dabei Variablen von e gebunden werden, können alle anderen Alternativen verworfen werden, ohne daß dadurch eine Lösung verlorengeht. Allerdings gilt dies nur bei Verwendung einer verzögerten Auswertungsstrategie und wenn alle Regeln Extravariablen nur in der Bedingung verwenden und zudem die schwache Eindeutigkeitsbedingung erfüllen, die wir in Abschnitt 2.2.2 vorgestellt haben.

Beispiel 6.11
Im Falle der `member`-Funktion wäre der Dynamic-Cut anwendbar, da die Regeln die schwache Eindeutigkeitsbedingung erfüllen und keine Extravariablen verwenden. Mit Hilfe der Committed-Choice können wir eine teilweise Simulation erreichen:

```
member_c(X,[])     = false.
member_c(X,[Y|Ys]) = choice {X=Y}                -> true;
                            {member(X,Ys)=true} -> true.

member_c(1,[1,2,1,3,1]) ⇝ ... ⇝ true
member_c(X,[1,2,1,3,1]) ⇝ ... ⇝ ⊥
```

In der ersten Anfrage wird der Dynamic-Cut simuliert, indem das Verfolgen des Wegs `member(1,[2,1,3,1])` verworfen wird, nachdem der Wächter `1=1` erfüllt werden konnte. In der zweiten Anfrage müßten wir jedoch eine globale Variable binden, wodurch mehrere Lösungen berechnet werden können. Da die Bindung globaler Variablen jedoch in lokalen Räumen verboten ist, suspendiert die Berechnung. Bei Verwendung eines echten dynamischen Cut würden hingegen im zweiten Aufruf alle Lösungen berechnet. ◇

Wenn globale Variablen gebunden werden müssen, können wir also keinen Dynamic-Cut simulieren und müssen daher auf andere Weise die Vervielfältigung des globalen Raums verhindern. Die Berechnung von `member(X,[1,2,1,3,1])` kann beispielsweise durch den Aufruf

```
findall(\X -> {member(X,[1,2,1,3,1])=true})
```

eingekapselt werden. Nach dem Prinzip des Dynamic-Cut können wir hingegen immer dann vorgehen, wenn entweder gar keine oder nur lokal deklarierte Variablen in einer Bedingung gebunden werden sollen.

Beispiel 6.12
In Abschnitt 3.1.4 hatten wir die Notwendigkeit zur Einkapselung von Berechnung bei Verwendung von Ein-/Ausgabeoperationen anhand folgender Funktion erläutert:

```
p(1) = true.
p(2) = true.
f if {p(X)=true} = putStr("Anfrage erfüllbar!").

f ⇝ ... ⇝ *** Error: Cannot duplicate the world!
```

Da die Extravariable `X` nur im Wächter auftritt, können wir sie dort lokal deklarieren und nach dem Prinzip des Dynamic-Cut vorgehen:

```
f_enc if choice {local Y in p(Y)=true} -> true = putStr("Anfrage erfüllbar!").

f_enc ⇝ ... ⇝ Anfrage erfüllbar!
```

◇

In diesem Beispiel ist die Anwendung des Cuts naheliegend, denn da der `local`-Operator ein Existenzquantor ist, will man in der Bedingung nur wissen, ob *irgendein* `Y` existiert, so daß `p(Y)` zu `true` reduzierbar ist. Wenn man allerdings Regeln verwendet, die die schwache Eindeutigkeitsbedingung nicht erfüllen oder Extravariablen auch in der rechten Regelseite verwenden, können Lösungen verloren gehen, wenn nach einer erfolgreichen Reduktion ohne Bindung globaler Variablen die restlichen Alternativen verworfen werden. Beispielsweise verwendet die `weg`-Funktion eine Extravariable, um eine in der Bedingung berechnete Bindung in die rechte Seite weiterzureichen:

```
weg(X,Y)  if {X=Y}                    = [X]
          if local Z in {kante(X)=Z}  = [X] ++ weg(Z,Y).
```

Bei Einkapselung der Bedingung würde nur noch eine Bindung von `Z` berechnet, so daß wir Lösungen verlieren würden, wenn in dem untersuchten Graph mehrere Kanten von `X` ausgehen.

Inwieweit eine Einkapselung einer Berechnung ohne Verlust von Lösungen möglich ist, muß also in jedem Einzelfall genau geprüft werden. Oftmals hilft es, die Einkapselung nicht innerhalb sondern außerhalb einer Funktion vorzunehmen, so wie wir es beispielsweise in Abschnitt 6.2 mit der `weg`-Funktion getan haben. Anstatt deren Regeln wie im letzten Beispiel einzukapseln, haben wir die Funktion selber an Suchoperatoren übergeben und dadurch als Ganzes in einen lokalen Raum eingeschlossen.

Auch in Fällen, in denen anders als in den bisher aufgeführten Beispielen nur ein einziger erfolgreicher Weg existiert, um eine Anfrage zu erfüllen, kann eine Einkapselung notwendig sein.

Beispiel 6.13
Die `last`-Funktion, die wir in Abschnitt 4.1.4 definiert haben, berechnet für eine gegebene Liste immer nur eine Lösung:

```
last(L) if {append(Xs,[E])=L} = E.
last([1,2]) ⇝ ... ⇝ 2
```

Trotzdem wird zwischenzeitlich der Berechnungsraum verdoppelt (wir verwenden die Darstellung bedingter Regeln durch bewachte Ausdrücke):

```
last([1,2])
⇝ ∃{Xs,E}: id ▯(append(Xs,[E])=[1,2] ⇒ E)
⇝   ∃{E}: {Xs=[]}           ▯(append([],[E])=[1,2] ⇒ E)
  | ∃{E,Y,Ys}: {Xs=[Y|Ys]}▯(append([Y|Ys],[E])=[1,2] ⇒ E)
```

Die Auswertung der `append`-Funktion bei der Reduktion der Bedingung führt also zu einer Vervielfältigung des globalen Raums. Der zweite Antwortausdruck wird später erneut eine Verdoppelung bewirken, aber da nur eine Lösung existiert, werden alle bis auf einen der Räume fehlschlagen, so daß wir am Ende der Berechnung wieder nur einen globalen Raum vorliegen haben. Die vorübergehende Vervielfältigung reicht jedoch aus, um einen Konflikt mit den Ein-/Ausgabeoperationen zu verursachen. Bei Verwendung der eingekapselten Version `last_c` aus Abschnitt 5.3 vermeiden wir diesen Konflikt

```
last_c(L) = choice local E in {local Xs in append(Xs,[E])=L} -> E.

lastIO(L)   if {last(L)=E}   = putStr("Das letzte Element ist ")>>show(E).
lastIO_c(L) if {last_c(L)=E} = putStr("Das letzte Element ist ")>>show(E).

lastIO([1,2])   ⇝ ... ⇝ *** Error: Cannot duplicate the world!
lastIO_c([1,2]) ⇝ ... ⇝ Das letzte Element ist 2
```

◇

Auch die Verwendung von Regeln mit mehr als einer Bedingung kann je nach Implementierung zu einer Vervielfältigung des Berechnungsraums führen.

Beispiel 6.14
Eine Möglichkeit zur Behandlung einer Regeldefinitionen mit beispielsweise zwei Bedingungen besteht in der Umwandlung in zwei Regeln mit identischer linker Seite und einem bewachten Ausdruck als rechte Seite, der jeweils eine der Bedingungen enthält:

```
max(X,Y) if X>=Y  = X
         if X<Y   = Y.

→ max(X,Y) = (X>=Y = true ⇒ X).
  max(X,Y) = (X<Y  = true ⇒ Y).
```

Diese Vorgehensweise wird beispielsweise in TasteCurry verwendet. Aufgrund der überlappenden linken Regelseiten wird der Reduktionssemantik aus Abschnitt 2.3 zufolge beim Aufruf von `max` ein disjunktiver Ausdruck erzeugt, da beide Regeln angewendet werden:

```
max(1,3)  ⇝  (1>=3  ⇒ 1) | (1<3  ⇒ 3)
```

Diese globale Vervielfältigung ist unnötig, da immer nur einer der Wächter bzw. eine der Bedingungen aus der originalen Definition erfüllbar ist und daher nach Abschluß der Berechnung wieder nur ein globaler Raum existiert. Man könnte daher eine Reduktionssemantik definieren, die bei Regeln mit sich ausschließenden Bedingungen keinen disjunktiven Ausdruck erzeugt, sondern die Bedingungen der Reihe nach prüft, bis die erfüllbare gefunden wird. Bislang haben wir jedoch keine derartige Semantik entwickelt und müssen darum eine Einkapselung vornehmen, um eine vorübergehende Vervielfältigung des Berechnungsraums zu verhindern:

```
max(X,Y) = choice X>=Y -> X;
                  X<Y  -> Y.
```

Man beachte, daß für die Wächter eines `choice`-Ausdrucks auch boolesche Ausdrücke anstelle von Constraints verwendet werden dürfen, solange keine lokalen Variablen deklariert werden müssen. ◇

Um eine Vervielfältigung des globalen Berechnungsraums durch bedingte Regeln zu verhindern, haben wir in Abschnitt 6.1 die Suchalgorithmen durch Verwendung lokaler Funktionen definiert und nicht durch bedingte Regeln. Wenn wir beispielsweise den `all`-Algorithmus in der Form

```
all(G) if {SearchG=[]}      = []
       if {SearchG=[S]}     = S
       if {SearchG=[A,B|C]} = concat(map(all,[A,B|C]))
       where
         SearchG = try(G).
```

programmiert hätten, würde bei jedem Aufruf der Berechnungsraum verdreifacht, wodurch natürlich der Sinn der Einkapselung durch die Verwendung von Suchalgorithmen verloren ginge. Eine Einkapselung durch Committed-Choice wie im vorigen Beispiel wäre auch möglich, würde aber im Gegensatz zu der verwendeten Lösung mit lokalen Funktionen einen zusätzlichen Berechnungsaufwand für das Ezeugen und Verwalten der Choice-Räume erfordern.

Die in diesem Abschnitt behandelten Beispiele zeigen, daß es keine optimale Strategie gibt, um eine Vervielfältigung des globalen Raums zu verhindern. An welcher Stelle und in welcher Form eine Einkapselung notwendig ist, muß daher in jedem Einzelfall entschieden werden und hängt beispielsweise davon ab, ob globale oder lokale Variablen gebunden werden sollen und ob man nur an einer oder an mehreren Lösungen interessiert ist.

6.7 Interaktive Suche

In diesem Abschnitt wollen wir anhand zweier kleiner Beispielanwendungen zeigen, wie die Suche durch Kommunikation mit dem Benutzer gesteuert werden kann. Zunächst betrachten wir eine kleines Programm zur Sitzplatzreservierung in einem Zug.

Beispiel 6.15
Das Programm bietet vier Arten von Sitzplätzen zur Auswahl an: Raucher oder Nichtraucher, jeweils im Abteil oder im Großraumwagen. Nachdem der Benutzer eine Auswahl getroffen hat, wird überprüft, ob sie erfüllbar ist. Gibt es noch einen entsprechenden Sitz, wird die Reservierung vorgenommen, andernfalls werden die noch möglichen Sitzgelegenheiten aufgelistet und zur Auswahl gestellt. Die verschiedenen Sitzarten definieren wir durch einen entsprechenden Datentyp `kindOfSeat`. Der Start der Programms erfolgt dann durch den Aufruf der Funktion `reservation`, der wir die Anzahl der jeweils freien Sitzplätze übergeben. `reservation(2,1,3,2)` startet beispielsweise mit zwei freien Raucherplätzen im Abteil, einem Raucherplatz im Großraumwagen, drei Nichtraucherplätzen im Abteil und zwei Nichtraucherplätzen im Großraumwagen. Nachfolgend listen wir das Programm auszugsweise auf.

```
data kindOfSeat = smokeA;smokeG;nonsmokeA;nonsmokeG.
reservation(A,B,C,D) = nl>>putStrLn(" Wo wollen Sie sitzen?")>>
                       offers(smokeA)>>offers(smokeG)>>
                       offers(nonsmokeA)>>offers(nonsmokeG)>>
                       anyOrAbort>>
                       getAnswer>>=
                       evalAnswer((A,B,C,D)).
%%%%%%%%%%%%%%%%%%%% Ein-/Ausgabe
⋮
offers(smokeA)    = putStrLn(" 1) Raucher im Abteil").
offers(smokeG)    = ...
⋮
outputResult((A,B,C,D),smokeA) =
                        putStrLn("Raucherplatz im Abteil reserviert!")>>
                        reservation(A-1,B,C,D).
outputResult((A,B,C,D),smokeG) = ...
⋮
%%%%%%%%%%%%%%%%%%%% Reservierungsvorgang
ask((A,B,C,D)) if {A>0=true} = smokeA.
ask((A,B,C,D)) if {B>0=true} = smokeG.
ask((A,B,C,D)) if {C>0=true} = nonsmokeA.
ask((A,B,C,D)) if {D>0=true} = nonsmokeG.

evalAnswer(Seats,49) =
        getSeat(Seats,dfs(\X -> {X=smokeA /\ smokeA=ask(Seats)})).
evalAnswer(Seats,50) =          ...smokeG ...
⋮
evalAnswer(Seats,53) = result(unpack(dfs(\X -> {X=ask(Seats)})))
%egal-Antwort          where
                         result([L]) = outputResult(Seats,L);
                         result([])  = putStrLn("Keine Plätze mehr frei!").
evalAnswer(Seats,54) = putStrLn("Auf Wiedersehen!").
```

```
getSeat(Seats,[L]) if {L@X} = outputResult(Seats,X).
getSeat(Seats,[])  = nl>>putStrLn(" Reservierungswunsch nicht erfüllbar!")>>
                       putStrLn(" Folgende Plätze sind verfügbar:")>>nl>>
                       options(findall(\X -> {X=ask(Seats)}))
               where
                 options([])           = putStrLn("Keine Plätze mehr frei!");
                 options([X|Xs])       = browseAndAsk([X|Xs]);
                 browseAndAsk([X|Xs]) = offers(X)>>browseAndAsk(Xs);
                 browseAndAsk([])      = anyOrAbort>>putStrLn("Ihre Wahl?")>>
                                         getAnswer>>=evalAnswer(Seats).
```

Der Wunsch des Kunden wird durch `getAnswer` abgefragt und an `evalAnswer` weitergeleitet. Das Beispielprogramm stammt aus TasteCurry und dort werden Eingaben in ihren Ascii-Code umgewandelt. Der Ascii-Wert 49 entspricht der Eingabe des Zeichens „1“, der Wert 54 der Eingabe von „6“.

Um im folgenden zu ermitteln, ob ein bestimmter oder welche Plätz überhaupt frei sind, verwenden wir die Funktion `ask`, die die Menge der verfügbaren Sitzplatzarten in einem disjunktiven Ausdruck zurückliefert:

```
ask((2,1,3,2)) ⇝ ... ⇝ smokeA | smokeG | nonsmokeA | nonsmokeG
```

Die Formulierung als nichtdeterministische Funktion macht zwar die Einkapselung der Verwendung notwendig, ermöglicht aber gerade die einfache Verwendung verschiedener Suchalgorithmen.

Hat der Benutzer einen bestimmten Platz ausgewählt, starten wir eine Suche durch den `dfs`-Algorithmus und geben dabei den gewünschten Sitz vor. Das Ergebnis wird durch `getSeat` ausgewertet. Wenn ein Platz gefunden wurde, wird er reserviert. Andernfalls werden durch `findall` alle noch verfügbaren Plätz ermittelt und durch `browseAndAsk` ausgegeben. Unter diesen kann der Benutzer erneut wählen.

Wird die Option „5 - egal“ ausgewählt hat, dann wenden wir den `dfs`-Algorithmus ohne Vorgabe auf die `ask`-Funktion an und reservieren die Platzart, die als erste berechnet werden kann. In der nachfolgenden Beispielreduktion führen wir zum besseren Verständnis die Anzahl der Sitzplätze zusätzlich auf.

```
|: reservation(1,1,0,3).

Wo möchten Sie sitzen? (1,1,0,3)
1) Raucher im Abteil             4) Nichtraucher im Großraumwagen
2) Raucher im Großraumwagen      5) egal
3) Nichtraucher im Abteil        6) Ende

5

Raucherplatz im Abteil reserviert!

Wo möchten Sie sitzen? (0,1,0,3)
1) Raucher im Abteil             4) Nichtraucher im Großraumwagen
2) Raucher im Großraumwagen      5) egal
3) Nichtraucher im Abteil        6) Ende

3

Reservierungswunsch leider nicht erfüllbar!
```

```
Folgende Plätze sind verfügbar:
2) Raucher im Großraumwagen            5) egal
4) Nichtraucher im Großraumwagen       6) Ende

Ihre Wahl?

4

Nichtraucherplatz im Großraumwagen reserviert!
```

◇

Das Programm ist an einigen Stellen noch verbesserungsbedürftig, beispielsweise werden Fehleingaben nicht abgefangen und führen zu einem Abbruch. Es ging uns aber nur darum zu demonstrieren, wie in einem etwas größeren Beispiel Suchalgorithmen im Zusammenhang mit Ein-/Ausgabeoperationen verwendet werden können. Wir haben dabei sowohl die Vorteile der Einkapselung als auch des kontrollierten Abbruchs genutzt und gesehen, daß die Verwendung zweier Suchalgorithmen für dieselbe Funktion in einem Programm durchaus sinnvoll sein kann.

Als zweites Beispiel betrachten wir eine ganz kleine Simulation der Prolog-Oberfläche.

Beispiel 6.16
Das folgende Programm nutzt die verzögerte Auswertung von Curry aus. Wir verwenden den `all`-Algorithmus und fordern nacheinander Elemente aus der Ergebnisliste an, die erst im Moment der Anforderung berechnet werden.

```
prolog(G) = loop(all(G))
         where
           loop([])    = putStrLn("no");
           loop([A|B]) = browse(A) >>putStr(" ? ")>>getChar>>=evalAnswer(B);

           evalAnswer(B,59) = getChar>>nl>>loop(B);
           evalAnswer(B,10) = nl>>putStrLn("yes");

           browse(A) if A@X = show(X).
           nl = putChar(10).
```

Wir übergeben der Funktion `prolog` eine Lambda-Abstraktion, auf die der `all`-Algorithmus angewendet wird. Die Hilfsfunktion `loop` sorgt durch ihr Pattern Matching dafür, daß das Ergebnis von `all` so weit ausgewertet werden muß, bis entweder die gesamte Suche fehlgeschlagen oder eine Lösung berechnet worden ist. Im zweiten Fall geben wir die Lösung mit Hilfe der Funktion `browse` aus und warten auf eine Benutzereingabe. Nach Eingabe eines Semikolons (Ascii-Wert 59) wird `loop` erneut aufgerufen, so daß `all` die nächste Lösung zu berechnen versucht (das zusätzliche `getChar` vor dem `loop`-Aufruf ist notwendig, um das Return nach dem Semikolon zu verarbeiten). Die Schleife wird entweder durch Eingabe eines Return statt eines Semikolons beendet, oder wenn keine Lösung mehr existiert. Wir stellen die Benutzereingaben der Deutlichkeit halber in Hochkommata dar und bezeichnen ein Return durch `<-`.

```
prolog(\X -> {X=weg(a,d)})

[a,b,c,d] ? ";"
[a,d] ? ";"
[a,c,d] ? ";"

no
-----
```

```
prolog(\X -> {add(s(z),X)=z})

no
-----
prolog(\X -> {local Y in X=add(Y,z)})

z ? ";"
s(z) ? ";"
s(s(z)) ?  "<-"

yes
```

Aufgrund der verzögerten Auswertung können wir im letzten Fall beliebig viele Lösungen berechnen lassen, obwohl der Suchbaum einen unendlichen Zweig enthält (vgl. Abschnitt 6.4). ◇

Sicherlich sind noch zahlreiche andere interaktive Algorithmen denkbar. Aufgrund der flexiblen Handhabungsmöglichkeiten des Suchoperators könnte man beispielsweise eine erste Lösung suchen und sie dem Benutzer vorschlagen. Ist er damit zufrieden, wird die Suche beendet, andernfalls wird die Suche nur noch nach einer besseren Lösung fortgesetzt, indem man die verbleibenden Alternativen entsprechend beschränkt. Auf diese Weise müßte nicht in jedem Fall der Aufwand für die Suche nach der besten Lösung betrieben werden, sondern man könnte nacheinander immer bessere Lösungen suchen, bis der Benutzer zufrieden ist.

6.8 Vergleich mit List-Comprehensions

In Abschnitt 3.3 hatten wir kurz die Verwendung von List-Comprehensions in funktionalen Sprachen betrachtet, beispielsweise in der Form

```
[x | x <- [1..9]; x mod 2 = 0] .
```

Wie dort schon erwähnt ist es mir im Rahmen dieser Arbeit leider nicht gelungen, eine allgemeingültige Vorschrift zu finden, nach der List-Comprehensions in äquivalente Ausdrücke in Curry übersetzt werden können, die mit Hilfe der eingekapselten Suche dieselbe Liste als Ergebnis berechnen.

In einer sequentiellen Implementierung, in der eine Konjunktion von links nach rechts abgearbeitet wird und die Anwendung der Regeln einer Funktion in der Reihenfolge der Deklaration erfolgt, kann man für eine große Anzahl von List-Comprehensions eine Umwandlungsvorschrift angeben, die wir hier kurz betrachten wollen. Wir verwenden dazu folgende Funktionen:

```
generate(E,[X|Xs]) = {E=X}.
generate(E,[X|Xs]) = generate(E,Xs).

list(Head,GeneratorsAndFilters) if {compute(GeneratorsAndFilters)} = Head
      where
        compute([X]) = X;
        compute([X1,X2|Xs]) = {X1 /\ compute([X2|Xs])}.
```

Die Funktion `generate` erhält einen Ausdruck `E` und eine Liste und erstellt einen disjunktiven Ausdruck, in dem `E` mit jedem Element der Liste unifiziert wird. Auf diese Weise soll ein Generator einer List-Comprehension simuliert werden.

Der Funktion `list` übergeben wir einen Ausdruck, der den Kopf der List-Comprehension bildet, und eine Liste von Generatoren und Filtern. Durch `compute` werden die Generatoren und die Bedingungen, die alle vom Typ Constraint sein müssen, ausgeführt und die dabei erstellten Bindungen der Variablen in den Kopf weitergereicht, dessen Wert schließlich zurückgeliefert wird. Nach Definition von `generate` werden die verschiedenen Bindungen aber in einem disjunktiven Antwortausdruck zurückgeliefert, so daß wir `findall` verwenden müssen, um die Berechnung einzukapseln, Fehlschläge zu kontrollieren und die Ergebnisse zu sammeln. Um Variablenbindungen vornehmen zu können, müssen wir dann aber alle auftretenden Variablen existenzquantifizieren.

Wir setzen einige Grundkentnisse über List-Comprehensions voraus und betrachten anhand einiger Beispiele die Umwandlung von List-Comprehensions in Curry-Ausdrücke. Wenn wir zunächst die Verwendung iterativer Generatoren ausschließen, können wir eine einfache List-Comprehension der Form

$$\texttt{[}e\ \texttt{|}\ pat_1\ \texttt{<-}\ expr_1\texttt{;}\ldots\texttt{;}pat_n\ \texttt{<-}\ expr_n\texttt{;}\ filter_1\texttt{;}\ldots\texttt{;}filter_k\texttt{]}$$

in etwa auf folgende Weise in einen Curry-Ausdruck übersetzen, wobei $\{\texttt{X}_1,\ldots,\texttt{X}_j\}$ die Menge aller Variablen bezeichne, die in dem `list`-Ausdruck auftreten:

$$\begin{array}{l}\texttt{findall(\textbackslash X -> \{local [}\texttt{X}_1\texttt{,}\ldots\texttt{,}\texttt{X}_j\texttt{] in X =}\\ \quad\texttt{list(}e\texttt{, [generate(}pat_1\texttt{,}expr_1\texttt{)},\ldots\texttt{,generate(}pat_n\texttt{,}expr_n\texttt{), }filter_1\texttt{,}\ldots\texttt{,}filter_k\texttt{])}\end{array}$$

Wir verdeutlichen dies anhand einiger Beispiele aus [17]. Wir führen zunächst die Formulierung in einer funktionalen Sprache und dann einen entsprechenden Ausdruck in Curry auf, der in der TasteCurry-Implementierung dieselbe Liste wie der funktionale Ausdruck berechnet. Wir verwenden in Curry zusätzlich eine Hilfsfunktion `genList(N,M)`, die eine Zahlenliste der Form `[N,N+1,..,M]` berechnet.

Beispiel 6.17

```
genList(A,B) = if A==B then [A] else [A|genList(A+1,B)].

-----
[x*x | x<-[1..9]; x mod 2=0]  ⇝  [4,16,36,64]

findall(X -> {local Y in X=list(Y*Y,[generate(Y,genlist(1,9)),Y mod 2=0])})

-----
[(i,j) | i <- [1..2]; j <- [2..4])] ⇝ [(1,2),(1,3),(1,4),(2,2),(2,3),(2,4)]

findall(\X -> {local [I,J] in X=list((I,J),[generate(I,[1,2]),
                                             generate(J,[2,3,4])])})
```

Abhängige Generatoren und geschachtelte List-Comprehensions sind ebenso möglich wie Muster in der linken Seite eines Generators.

```
[(i,j) | i <- [1..3]; j <- [1..i])] ⇝  [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]

findall(\X -> {local [I,J] in X=list((I,J),[generate(I,[1,2,3]),
                                             generate(J,genList(1,I))])})

-----
[ [i*j | i <- [1..2]]  | j <- [1..3]]  ⇝  [[1,2],[2,4],[3,6]]

findall(\X -> {local J in X=list(
         findall(\X -> {local I in X=list(I*J,[generate(I,[1,2])])}),
         [generate(J,[1,2,3])])})

-----
```

```
getones x = [a | (a,1) <- x]
getones [(3,1),(0,2),(2,1),(2,3)]  ⇝  [3,2]

getones(X) = findall(\Y -> {local A in Y=list(A,[generate((A,1),X)])}).
```

◇

Iterative Generatoren verwenden keine Listen, sondern Berechnungsvorschriften zur Erzeugung der Elemente. Die Liste der Fibonacci-Zahlen kann beispielsweise durch

```
[a|(a,b) <- (0,1),(b,a+b)..]
```

berechnet werden. In Curry müssen wir solche Generatoren durch Verwendung einer Hilfsfunktion realisieren. Ein Generator

pat `<-` *value*`,` *iterPat*`..`

wird in die Form

`generate(`*pat*`,f(`*value*`))`

übersetzt, wobei `f` durch die Regel

`f(V) = [V|f(`*iterPat*`)].`

definiert wird.

Beispiel 6.18
Da die folgenden List-Comprehensions unendliche Listen berechnen, verwenden wir die Funktion `take` aus Haskell, die beim Aufruf `take n xs` die ersten `n` Elemente einer Liste `xs` zurückliefert. In Curry kann sie beispielsweise durch

```
take(N,Xs) = if N==0 then [] else
                (if Xs==[] then [] else [head(Xs)|take(N-1,tail(Xs))]).
```

formuliert werden.

```
take 10 [a | a <- 0, a+1..]  ⇝  [0,1,2,3,4,5,6,7,8,9]

f(A) = [A|f(A+1)].
take(10, findall(\X -> {local A in X=list(A,[generate(A,f(0))])}))

-----
take 10 [a | (a,b) <- (0,1), (b,a+b)..]  ⇝  [0,1,1,2,3,5,8,13,21,34]

f((A,B)) = [(A,B)|f((B,A+B))].
take(10,
     findall(\X -> {local [A,B] in X=list(A,[generate((A,B),f((0,1)))])}))
```

◇

Es ist mir jedoch nicht gelungen, eine Übersetzungsvorschrift für diagonalisierende List-Comprehensions zu finden. Zudem funktioniert die vorgestellte Art der Simulation nur in einer sequentiellen Implementierung und bei Beachtung der Reihenfolge der Regeldeklaration. Wenn wir beispielsweise die beiden Regeln der Funktion `generate` in umgekehrter Reihenfolge angeben, dreht sich im ersten Beispiel auch die Reihenfolge aller Listenelemente um. Bei der Verwendung der iterativen Generatoren aus dem zweiten Beispiel erhalten wir hingegen gar kein Ergebnis, da zuerst alle Elemente der unendlichen Listen erzeugt werden müßten, bevor eines in die linke Seite des Generators weitergereicht werden könnte.

Man müßte daher versuchen, eine Realisierung ohne Verwendung von nichtdeterministischen Reduktionen zu erreichen, beispielsweise, indem man mit Hilfe der `map`-Funktion die Bedingung auf alle Elemente anwendet, die ein Generator berechnet. Dieses Prinzip hat Hanus beispielsweise bei der Definition der `forall`-Funktion verwendet, die wir in Beispiel 6.3 vorgestellt haben. Bei Verwendung mehrerer, möglicherweise abhängiger Generatoren und Mustern in den linken Generatorseiten, deren Unifikation mit den Elementen der Liste wie bei `getones` auch fehlschlagen kann und daher eingekapselt werden müßte, ist es mir jedoch nicht annähernd gelungen, eine Übersetzungsvorschrift zu finden. Man muß deshalb in jedem konkreten Fall überlegen, wie eine Simulation in Curry realisiert werden kann. Wir haben in Beispiel 6.8 eine Funktion `biggerOne` vorgestellt, die im Grunde genommen der List-Comprehension

```
biggerOne x = [s(s(y)) | s(s(y)) <- x]
```

entspricht, abgesehen davon, daß partielle Strukturen in einer List-Comprehension nicht erlaubt sind. Aber mit Hilfe des dort verwendeten Schemas zur Kontrolle von Fehlschlägen können wir einige List-Comprehensions verwirklichen. Die beiden Ausdrücke

```
[x*x | x<-[1..9]; x mod 2=0]
getones x = [a | (a,1) <- x]
```

lassen sich beispielsweise durch folgende Funktionen realisieren:

```
sq([])      = [].
sq([X|Xs]) = unpack(dfs(\Y -> {X mod 2=0 /\ Y=X*X})) ++ sq(Xs).

sq([1,2,3,4,5,6,7,8,9]) ⇝ ... ⇝ [4,16,36,64]

-----
getones([])      = [].
getones([X|Xs]) = unpack(dfs(\ -> {local A in (A,1)=X /\ Y=A}))
                  ++ getones(Xs).

getones([(3,1),(0,2),(2,1),(2,3)]) ⇝ ... ⇝  [3,2]
```

Die Möglichkeiten zur Simulation von List-Comprehension können sicherlich noch eingehender untersucht werden, als es in dieser Arbeit geschehen ist. Vielleicht können die vorgestellten Ansätze als Anregung dienen, über weitere Lösungsmöglichkeiten nachzudenken. In jedem Fall dürften aber zu List-Comprehensions äquivalente Ausdrücke in Curry wesentlich komplizierter sein als die sehr einfache und knappe Formulierung der List-Comprehensions selber. Zukünftige Version von Curry werden daher möglicherweise die direkte Eingabe von List-Comprehensions in der aus funktionalen Sprachen bekannten Form erlauben.

Kapitel 7

Die TasteCurry-Implementierung

Wir haben in den vorhergegangenen Kapiteln mehrfach auf das TasteCurry-System hingewiesen, die erste Prototyp-Version von Curry, die von Hanus, Sadre und Steiner programmiert wurde. Ihre Realisierung in SICStus-Prolog [28] verwirklicht die grundlegenden Ideen aus [11], die wir in Kapitel 2 erläutert haben, und basiert auf den Mechanismen zur Übersetzung funktional-logischer Programme in Prolog, die in [9] beschrieben werden. Die Implementierung der strikten Gleichheit erfolgt nach dem Schema aus [23].

TasteCurry wurde nicht entwickelt, um ein möglichst effizientes System zu schaffen, sondern um zunächst einmal die grundsätzliche Arbeitsweise von Curry zu demonstrieren und die Realisierbarkeit verschiedener Sprachelemente zu überprüfen. Während der Entwicklung wurden deshalb immer wieder kleinere Änderungen an der operationalen Semantik vorgenommen, beispielsweise durch die Einführung von Constraints, die ursprünglich nicht vorgesehen waren, oder zuletzt durch den Verzicht auf definierende Bäume mit *and*-Knoten, mit deren Hilfe ursprünglich die nebenläufige Konjunktion realisiert wurde.

Die Entwicklung von TasteCurry ist noch nicht abgeschlossen und wird immer wieder durch Hinzufügen einiger der Sprachelemente, die in [16] beschrieben sind, erweitert. Die Ein-/Ausgabe in der in Abschnitt 3.1.4 beschriebenen Form ist mittlerweile ebenso realisiert wie die Verwendung von Lambda-Abstraktionen und lokalen Definitionen; die Entwicklung eines Modulsystems ist beinahe abgeschlossen. Im Rahmen dieser Arbeit erfolgte die Ergänzung um die Mechanismen der eingekapselten Suche.

Die auf der Diskette beigefügte TasteCurry-Version unterliegt einigen Einschränkungen gegenüber der allgemeinen Sprachdefinition von Curry [16]. Die wichtigsten sind:

- Die Syntax wird durch das zugrunde liegende Prolog-System bestimmt (vgl. Abschnitt 2.2 und Anhang C).
- Das Modulsystem ist noch nicht integriert.
- Die Datentypen `Float`, `Char` und `String` für Fließkommazahlen, Zeichen und Zeichenketten sind nicht implementiert.
- Der Anschluß von externen Constraint-Solvern ist nicht möglich.
- Es sind nur einstellige Lambda-Abstraktionen erlaubt.
- Auf der Kommandozeile ist nur eine stark eingeschränkte Verwendung von Lambda-Abstraktionen der Form `\`X `->` c möglich:
  - c muß ein Constraint sein.
  - c darf keine weiteren Lambda-Abstraktionen enthalten.

Beispielsweise ist die Eingabe

```
findall(\X -> {local Y in X=weg(a,Y)}).
```

auf der Kommandzeile möglich, die Eingaben

```
(\X -> X*X)@2.
(\X -> {X=findall(\Y -> {Y=weg(a,d)})}).
```

hingegen nicht. Diese Einschränkung hat programmiertechnische Gründe und wird in zukünftigen Versionen aufgehoben werden. Für das Arbeiten mit dem Suchoperator und das Ausprobieren der Beispiele, die wir in dieser Arbeit vorgestellt haben, sollte die eingeschränkte Form der Lambda-Abstraktionen ausreichen.

Eine Beschreibung sowie ein WWW-Interface zur Benutzung des jeweils aktuellen TasteCurry-Systems ist unter folgender Adresse verfügbar:

`http://www-i2.informatik.rwth-aachen.de/~hanus/tastecurry/`

Allgemeine Informationen zu Curry und der aktuelle Curry-Report können unter folgender Adresse abgerufen werden:

`http://www-i2.informatik.rwth-aachen.de/~hanus/curry/`

Bei der Implementierung der eingekapselten Suche im TasteCurry-System sind wir in einigen Punkten von der vorgestellten Theorie abgewichen, und wir wollen einige der Abweichungen in diesem Kapitel kurz vorstellen, da sie auch für andere Implementierungen von Interesse sein könnten. In Anhang A erläutern wir dann die Installation und Benutzung der auf der beiliegenden Diskette enthaltenen TasteCurry-Version, in Anhang B führen wir die Menge der vordefinierten Funktionen und Datentypen auf.

7.1 Die Behandlung von Variablen

Die Behandlung der Variablen in TasteCurry unterscheidet sich grundsätzlich in 3 Punkten von der vorgestellten Theorie.

- Die in den Abschnitten 4.1.4 und 5.3 eingeführte zusätzliche Verwendung des Existenzquantors außerhalb der geschweiften Klammern bei bedingten Regeln und der Comitted Choice muß in TasteCurry durch einen anderen Operator vorgenommen werden. Dies ist notwendig, um Regeln ohne Konflikte einlesen und in eine interne Darstellung umwandeln zu können. Sobald ein vernünftiger Parser zur Verfügung steht, wird diese Einschränkung aufgehoben werden. Bis dahin müssen Existenzquantoren außerhalb der geschweiften Klammern durch einen zweistelligen Infix-Operator `localIn` anstatt durch `local...in...` deklariert werden:

  ```
  last(L) if E localIn {local Xs in append(Xs,[E])=L} = E.

  last_c(L) = choice E localIn {local Xs in append(Xs,[E])=L} -> E.
  ```

  Wie wir noch sehen werden, ist jedoch eine Unterscheidung zur Laufzeit nicht notwendig, und daher verwenden wir intern für beide Existenzquantoren ein- und dieselbe Darstellung durch einen ebenfalls zweistelligen Infix-Operator `localVars`. Um unnötige Verwirrung zu vermeiden, verwenden wir in den nachfolgenden Beispielen aber in allen Fällen die bisherige Form des Existenzquantors.

- Die Menge der in einem Raum deklarierten Variablen wird nur für lokale Räume gesammelt. Bei der Berechnung in einem globalen Raum darf jede Variable gebunden werden, und daher ersparen wir uns die Arbeit für die Protokollierung. Ob wir uns in einem lokalen Raum befinden oder nicht, wird durch einen booleschen Parameter angezeigt, den wir an *cse*, *equal* und *cs* übergeben. Standardmäßig ist der Parameter auf `false` gesetzt, aber wenn wir bei der Reduktion auf ein `try`, `searchSpace` oder `cSpaces` stoßen, übergeben wir an nachfolgende Aufrufe der Reduktionsfunktionen `true` als Parameter.

- Die Umbenennung lokal deklarierter Variablen zur Laufzeit wird aus Effizienzgründen so weit wie möglich vermieden.

Um den letzten Punkt realisieren zu können, müssen zwei Bedingung erfüllt sein:

1. Jede während der Berechnung neu auftretende Variable muß eine frische sein, die sich von allen bereits aufgetretenen unterscheidet.

2. Innerhalb einer Regel und einer Anfrage dürfen keine zwei verschiedene Variablen mit demselben Namen bezeichnet sein.

Der Existenzquantor benennt die Variablen in der Theorie nur zur Vermeidung von Namenskonflikten um. Wenn wir die beiden Bedingungen einhalten, kann es jedoch während einer Berechnung überhaupt nicht zum Auftreten zweier verschiedener Variablen gleichen Namens kommen, und daher ist auch eine Umbenennung nicht erforderlich.

Die erste Bedingung wird bereits durch die Definition der Reduktionsfunktionen *cse* und *equal* erfüllt. Neue Variablen können in einer Berechnung nur auftreten,

1. wenn eine Regel angewendet wird, die Extravariablen enhält,

2. wenn eine Variable in einem Narrowing-Schritt an ein Muster eines definierenden Baums gebunden wird, das Variablen enthält, oder

3. wenn bei der Bindung einer Variable an einen Konstruktorterm eine Konjunktion von Gleichheitsconstraints erstellt wird, die neue Variablen enthält:

```
X=s(add(z,z)) ⤳ {X=s(Y)}▯Y=add(z,z)
```

Im letzten Fall werden nach Definition von *equal* immer systemweit neue Variablen verwendet. In den ersten beiden Fällen stammen die Variablen aus einem definierenden Baum einer Funktion, und da *cse* an die Reduktionsfunktion *cs* immer einen Definierenden Baum „...mit neuen Variablen“ übergibt (vgl.Abschnitt 2.3.4), sind auch diese Variablen neu und damit systemweit eindeutig. Bei einer Implementierung in Prolog benötigt man keinen besonderen Mechanismus, um bei jeder Regelanwendung den zugehörigen definierenden Baum mit neuen Variablen zu versehen. Wenn man jeden Baum als Faktum speichert, wird er von Prolog bei jeder erneuten Verwendung des Faktums mit frischen Variablen versehen.

Wenn wir jedoch die lokal deklarierten Variablen bei der Reduktion eines Existenzquantors nicht umbenennen, kann es bislang zu einem Konflikt kommen:

Beispiel 7.1
Betrachten wir die folgende Funktion und ihre Reduktion:

```
f = {local X in X=1 /\ {local X in X=2}}.

f ⤳ {local X in X=1 /\ {local X in X=2}}
  ⤳ ∃{X}: id▯X=1 /\ {local X in X=2}
  ⤳ ∃∅: {X=1}▯{local X in X=2}
  ⤳ ∃{X}: {X=1}▯X=2
```

Bereits an dieser Stelle gibt es einen Konflikt, da eine Variable unter demselben Namen in der Variablenmenge und in der Substitution auftritt. Im nächsten Schritt gäbe es zudem einen Bindungskonflikt, da wir nicht zwei verschiedene Bindungen einer Variable mit demselben Namen verarbeiten können. ◇

Wir lösen dieses Problem durch die Verwendung einer erweiterten Normalform, die dafür sorgt, daß jede Variable innerhalb einer Regel einen eindeutigen Namen erhält. Die Funktion `f` aus dem Beispiel würde daher in die Normalform

$$\texttt{f = \{local X}_1\texttt{ in X}_1\texttt{=1 /\textbackslash\ \{local X}_2\texttt{ in X}_2\texttt{=2\}\}.}$$

umgewandelt. Die Existenzquantoren werden nach einer solchen Umwandlung nur noch benötigt, damit wir auf einfache Weise die Extravariablen einer Regel zur Menge der im aktuellen Berechnungsraum deklarierten Variablen hinzufügen können.

Wir nehmen in TasteCurry zusätzlich noch eine weitere kleine Vereinfachung vor, indem wir geschachtelte Constraints aufbrechen und die Existenzquantoren so weit wie möglich nach vorne ziehen und dort zu einem Quantor zusammenfassen. Dies ist unter Verwendung der folgenden Umformungen möglich, die die Reduktionssemantik des Constraints nicht verändern und entsprechend für mehrere Variablen und Konjunktionen mit mehr als zwei Constraints gelten. Wir verwenden die Abkürzung $\overline{X}_n$ für eine Folge von Variablen der Form $X_1, \ldots, X_n$.

$$\begin{array}{lcl}
\texttt{local [}\overline{X}_n\texttt{] in } c_1 \texttt{ /\textbackslash\ \{local } Y \texttt{ in } c_2\texttt{\}} & \rightarrow & \texttt{local [}\overline{X}_n, Y_1\texttt{] in } c_1 \texttt{ /\textbackslash\ } c_2[Y/Y_1] \\
\texttt{local [}\overline{X}_n\texttt{] in \{local } Y \texttt{ in } c_1\texttt{\} /\textbackslash\ } c_2 & \rightarrow & \texttt{local [}\overline{X}_n, Y_1\texttt{] in } c_1[Y/Y_1] \texttt{ /\textbackslash\ } c_2 \\
 & & \text{mit } Y_1 \text{ eine neue Variable}
\end{array}$$

Die geschweiften Klammern des inneren Constraints können wir auflösen, nachdem der Existenzquantor entfernt wurde, ohne dadurch die korrekte Form von Constraints zu zerstören.

Variablen, die vor der Bedingung einer Regel bzw. vor dem Wächter eines `choice`-Ausdrucks deklariert sind, benennen wir in Bedingung und rechter Seite bzw. im Wächter und Rumpf gleichermaßen um, und deklarieren sie anschließend innerhalb der Bedingung bzw. des Wächters. Bei der Reduktion einer Regel mit Extravariablen müssen wir dann im besten Fall nur einen einzigen statt mehrerer Existenzquantoren reduzieren, wodurch Berechnungsschritte eingespart werden können.

Beispiel 7.2
Eine vollständige formale Definition des neuen Normalform-Algorithmus wäre sehr umfangreich, und wir verzichten daher auf eine Angabe und verdeutlichen stattdessen die Arbeitsweise anhand einiger Beispiele[1].

$\widetilde{nf}$`(f = {local X in X=1 /\ {local X in X=2}})` $=$
  `f = {local [`X_1`,`X_2`] in `X_1`=1 /\ `X_2`=2}`

$\widetilde{nf}$`(last(L) if local E in {local Xs in append(Xs,[E])=L} = E)` $=$
  `last(L) if {local [`E_1`,`Xs_1`] in append (`Xs_1`,[`E_1`])=L} = `E_1

$\widetilde{nf}$`(last_c(L) = choice local E in {local Xs in append(Xs,[E])=L} -> E)` $=$
  `last_c(L) = choice {local [`E_1`,`Xs_1`] in append(`Xs_1`,[`E_1`])=L} -> `E_1

$\widetilde{nf}$`(test if local L in {L = \X -> {local Y in Y=1 /\ {local Y in Y=2}}} = L)`
$=$ `test if {local `L_1` in `L_1` = \X -> {local [`Y_1`,`Y_2`] in `Y_1`=1 /\ `Y_2`=2}} = `L_1

[1] Auf der beiliegenden Diskette, deren Benutzung wir in Anhang A erläutern, befindet sich eine Datei `local.pl`, die den Normalform-Algorithmus in einer ausführlich dokumentierten Version enthält.

Man beachte, daß im letzten Beispiel die lokalen Variablen der Lambda-Abstraktion natürlich nur in deren Rumpf gesammelt und nicht nach außen gezogen werden dürfen, da der Lambda-Operator einem Allquantor entspricht und man einen Existenzquantor nicht einfach über einen Allquantor hinaus ziehen kann.

In Abschnitt 4.1.3 haben wir erläutert, daß die Umbenennung der nicht explizit deklarierten Extravariablen einer Regel zur Laufzeit auf jeden Fall unnötig ist. Daher können wir sie einfach ohne vorherige Umbenennung in der Bedingung deklarieren.

$\widetilde{nf}$(`last(L) if {append(Xs,[E])=L} = E`) =
    `last(L) if {local [E,Xs] in append (Xs,[E])=L} = E`

◇

Die Verwendung der zusätzlichen *cse*-Regel zur Reduktion bedingter Regeln, die wir in Abschnitt 4.1.5 angegeben haben, ist nun in TasteCurry nicht mehr notwendig, da die vor der Bedingung deklarierten Variablen auch in der rechten Seite bereits umbenannt wurden. Alle Variablenbindungen aus der Bedingung werden nun automatisch durch die Funktion *replace* (vgl. Abschnitt 2.3.4) in die rechte Seite weitergereicht.

$$\boxed{cse(V, \texttt{local [}X_1,\ldots,X_n\texttt{] in } c) \quad = \quad \{\exists\, V \cup \bigcup_{i=1}^{n}\{X_i\} :\ id[]c\}}$$

Abbildung 7.1: TasteCurry-Reduktion eines Existenzquantors

Die Reduktion des Existenzquantors in TasteCurry erfolgt nun durch die Regel in Abbildung 7.1 ohne Umbenennung der lokalen Variablen. Man beachte, daß es dadurch nicht zu Konflikten bei der Applikation von Lambda-Abstraktionen kommen kann, die der Suchoperator zurückliefert. Wenn dieser beispielsweise ein Ergebnis

```
[\X -> {local N in X=s(N)}, \X -> {local N in X=s(s(N))}]
```

berechnet, könnte ja die Applikation beider Abstraktionen zweimal dieselbe Variable `N` zum globalen Raum hinzufügen, wenn der Existenzquantor sie nicht umbenennt. Da wir aber in TasteCurry das Prinzip des Lambda-Lifting verwenden, werden die beiden Abstraktionen in Funktionen mit jeweils einem eigenen definierenden Baum umgewandelt, der wie erwähnt vor der Anwendung immer mit frischen Variablen versehen wird. Daher erhält das `N` in beiden Bäumen einen eigenen Namen. In Abschnitt 7.3 erläutern wir die Umwandlung der Lambda-Abstraktionen ausführlicher.

Leider gibt es jedoch eine Ausnahme, bei der wir die Notwendigkeit zur Umbenennung lokaler Variablen nicht durch eine entsprechende Normalform aufheben können. Es handelt sich bei TasteCurry um eine „Copying"- und nicht um eine „Sharing"-Implementierung, d.h. ein Parameter, den wir an eine Funktion übergeben, kann mehrfach ausgewertet werden.

Beispiel 7.3

```
coin = z.
coin = s(z).
double(X) = add(X,X).

double(coin) ⇝ add(coin,coin) ⇝ ... ⇝ = z | s(z) | s(z) | s(s(z))
```

Da in einer Copying-Implentierung die beiden `coin`-Parameter unabhängig voneinander zu jeweils zwei Werten reduziert werden können, erhalten wir die vier aufgeführten Ergebnisse.

In einer Sharing-Implementierung würden bei der Reduktion von `double` keine unabhängigen Kopien des Parameters erstellt, sondern beispielsweise nur zwei Zeiger auf ein- und denselben Term `coin`. Daher gäbe es auch nur zwei Ergebnisse:

double(coin) $\rightsquigarrow$ ... $\rightsquigarrow$ z | s(s(z))

◇

Wenn wir nun einen Constraint mit einer lokalen Deklaration als Parameter einer Funktion übergeben, können durch das Copying zwei verschiedene lokale Variablen desselben Namens innerhalb einer Berechnung entstehen.

Beispiel 7.4

```
nd(X) = {X=1}.
nd(X) = {X=2}.

double_c(X) = {X /\ X}.
```

double({local X in nd(X)}) $\rightsquigarrow$ {{local X in nd(X)} /\ {local X in nd(X)}}

In einer Copying-Implementierung muß nun für jeden `nd`-Aufruf eine eigene lokale Variable erstellt werden. Ansonsten wird die Bindung aus der Reduktion des ersten `nd` in den zweiten Teil der Konjunktion weitergereicht und führt zu dem Ausdruck

$\{$X=1$\}\square$ {local 1 in nd(1)} | $\{$X=2$\}\square$ {local 2 in nd(2)}

◇

Um solche Konflikte zu vermeiden, verwendet TasteCurry intern einen zusätzlichen Existenzquantor in Form des zweistelligen Infixoperators `localInPar`, durch den alle lokalen Variablendeklarationen gekennzeichnet werden, die innerhalb einer Funktionsparameters auftreten. Bei der Reduktion dieses Operators nach Abbildung 7.2 werden die Variablen umbenannt.

$$\boxed{cse(V, [X_1, \ldots, X_n] \texttt{ localInPar } c) \quad = \quad \{\exists\, V \cup \bigcup_{i=1}^{n} \{X_{i_{fresh}}\} : \; id \square c[X_i/X_{i_{fresh}}]\}}$$

Abbildung 7.2: Reduktion des Existenzquantors für lokale Variablen in Funktionsparametern

Beispiel 7.5

double({X localInPar nd(X)})
$\rightsquigarrow$ {{X localInPar nd(X)} /\ {X localInPar nd(X)}}
$\rightsquigarrow$ $\exists\{X_1\}: id\square$nd($X_1$) /\ {X localInPar nd(X)}

$\overset{*}{\rightsquigarrow}$ $\{X_1$=1$\}\square$ {X localInPar nd(X)} | $\{X_1$=2$\}\square$ {X localInPar nd(X)}

Die Reduktion des zweiten `nd` kann nun in korrekter Weise erfolgen. ◇

Dieses Beispiel zeigt, daß wir neben den Definitionen eines Programms nun auch die Anfragen, die der Benutzer stellt, in eine Normalform umwandeln müssen, statt wie bisher aus einem Ausdruck e einfach den initialen Ausdruck

$\{\exists free(e) : id \square e\}$

zu erzeugen (vgl. Definition 4.11 in Abschnitt 4.2.1). Wir transformieren dazu jeden Constraint, der in e auftritt, in der beschriebenen Weise durch Umbenennung aller lokalen Deklarationen und Sammeln der Existenzquantoren am Anfang des Constraints. Die freien Variablen von e bilden wie bisher die Ausgangsmenge der deklarierten Variablen.

7.2 Lokale Räume

In der aktuellen TasteCurry-Version ist keine eigene Reduktionssemantik für lokale Räume nach Abschnitt 5.1 realisiert. Ihre Auswertung wird in den jeweiligen Reduktionsfunktionen für Suchräume und Choice-Räume durch direkte Übergabe an die *cse*-Funktion und Interpretation von deren Resultat vorgenommen. Auf der beiliegenden Diskette sind die Reduktionsfunktionen in einer ausführlich kommentierten Version in der Datei `search.pl` aufgeführt.

Einen Unterschied zur Semantik aus Kapitel 5 wollen wir kurz erläutern: die explizite Markierung lokaler Räume, deren Reduktion aufgrund ungebundener globaler Variablen suspendiert werden muß. Solche Räume sind entweder instabil oder versuchen, eine globale Variable während der Reduktion zu binden. Die Reduktionsfunktionen *cseVSearch* und *cseVChoice*, die in der Implementierung die Arbeitsweise von *cslocal* und *returnSearch* bzw. *cslocal* und *choiceEval* in sich vereinigen, markieren einen instabilen Raum ebenso wie einen, der bei der Reduktion durch *cse* suspendiert wurde und globale Variablen enthält.

Beispiel 7.6
Die erste Anfrage suspendiert, weil der Suchraum versucht, eine globale Variable zu binden, die zweite, weil der Suchraum verdoppelt werden müßte, aber instabil ist.

$\texttt{try(\textbackslash X -> \{add(Y,z)=s(z)\})}$

$\overset{*}{\leadsto}\ \exists\{\texttt{Y}\}:id[]\texttt{searchSpace(X}_1\texttt{,(}\{\texttt{X}_1\}\texttt{,}id\texttt{,add(Y,z)=s(z)))}$
$\leadsto\ \exists\{\texttt{Y}\}:id[]\texttt{unboundVars([Y],searchSpace(X}_1\texttt{,(}\{\texttt{X}_1\}\texttt{,}id\texttt{,add(Y,z}_1\texttt{)=s(z))))}$

$\texttt{try(\textbackslash X -> \{add(X,Y)=s(z)\}}$

$\overset{*}{\leadsto}\ \exists\{\texttt{Y}\}:id[]\texttt{searchSpace(X}_1\texttt{,(}\{\texttt{X}_1\}\texttt{,}id\texttt{,add(X,Y)=s(z)))}$
$\leadsto\ \exists\{\texttt{Y}\}:id[]\texttt{unboundVars([Y],searchSpace(X}_1\texttt{,(}\{\texttt{X}_1\}\texttt{,}id\texttt{,add(X,Y)=s(z))))}$

Die `merge`-Funktion haben wir durch Committed-Choice definiert (vgl. Beispiel 5.21). Beim Aufruf `merge(L1,[2])`, können die beiden Wächter, die die Form des ersten Parameters überprüfen, nicht reduzieren, solange `L1` ungebunden ist. Daher werden die entsprechenden Choice-Räume nach ihrem ersten Reduktionsschritt markiert. Wir zeigen die Situation nach Reduktion des ersten und dritten Wächters. Der zweite wurde bereits als unerfüllbar erkannt und gelöscht.

`merge(L1,[2])`

$\overset{*}{\leadsto}\ \texttt{cSpaces((}\{\texttt{E}_2\texttt{,R}_2\texttt{,}id\texttt{,[2]=[E}_2\texttt{|R}_2\texttt{]) -> [E}_2\texttt{|merge([1],R}_2\texttt{)],}$
$\qquad\texttt{unboundVars([L1],(}\emptyset\texttt{,}id\texttt{,L1=[]) -> [2]),}$
$\qquad\texttt{unboundVars([L1],(}\{\texttt{E}_1\texttt{,R}_1\}\texttt{,}id\texttt{,[1]=[E}_1\texttt{|R}_1\texttt{]) -> [E}_1\texttt{|merge(R}_1\texttt{,[2])]))}$

Die derart markierten Choice-Räume werden bei der Reduktion von `cSpaces` so lange übersprungen, bis `L1` gebunden wird bzw. ein anderer Wächter erfolgreich reduziert wurde. Falls alle Choice-Räume aufgrund ungebundener globaler Variablen suspendiert sind, wird das gesamte Konstrukt unter Angabe aller verantwortlichen Variablen markiert:

```
merge(L1,L2) ⇝ ... ⇝ unboundVars([L1,L2],cSpaces(unboundVars([L1],...),
                                                 unboundVars([L2],...),
                                                 ...)
```
◇

Es ist nicht notwendig, derart markierte Such- und Choice-Räume weiter zu untersuchen, bevor nicht mindestens eine der globalen Variablen gebunden wurde, denn auf andere Weise kann sich der Zustand dieser Räume nicht ändern. Für Suchräume und das `cSpaces`-Konstrukt verwenden wir deshalb eine zusätzliche *cse*-Regel, die in in Abbildung 7.3 dargestellt ist. Sie

$$cse(V, \texttt{unboundVars}(VarList, e)) \quad = \quad \begin{cases} \perp & \text{wenn } \texttt{nobound(}VarList\texttt{)=true} \\ cse(V, e) & \text{sonst} \end{cases}$$

Abbildung 7.3: Reduktion von Suchräumen oder `cSpaces`

überprüft, ob wenigstens ein Element aus der Menge der globalen Variablen gebunden wurde und entpackt in diesem Fall den Suchraum oder das `cSpaces`-Konstrukt und wertet es durch einen rekursiven Aufruf aus. Innerhalb der *cseVChoice*-Reduktionsfunktion werden die einzelnen durch `unboundVars` gekennzeichneten Choice-Räume auf ähnliche Weise überprüft. Das Prädikat `noBound` muß erfüllt sein, wenn keine der globalen Variablen gebunden worden ist. Seine interne Realisierung ist von der jeweiligen Implementierung und Behandlung der Variablen abhängig. In TasteCurry können wir dazu das `var`-Prädikat von Prolog verwenden:

```
noBound([]).
noBound(X|Xs]):-var(X),noBound(Xs).
```

Im Unterschied zur vorgestellten Theorie wird also in bestimmten Fällen eine Suspension eines Konstrukts durch Angabe des Suspensionsgrundes markiert, so daß zu einem späteren Zeitpunkt eine Entscheidung möglich ist, ob die Berechnung fortgesetzt werden kann oder nicht. Ohne eine solche Markierung kann ein- und derselbe Such- oder Choice-Raum immer wieder unnötigerweise ausgewertet werden, auch wenn sich seit der ersten Suspension seine Form nicht verändert hat und eine erneute Reduktion daher nicht sinnvoll ist.

Die Markierung der lokalen Räume entspricht in sehr kleinem Rahmen einem Suspensions- und Wiedererweckungsmechanismus, der in einer wesentlich umfangreicheren und effizienteren Form durch Bindung der suspendierten Programmteile an die verantwortliche Variable beispielsweise in AKL [19] und Oz [25] (vgl. Kapitel 8) verwendet wird.

In TasteCurry haben wir außer bei entsprechend darauf ausgerichteten Demonstrationsbeispielen keinen nennenswerten Effizienzvorteil durch die Markierung der lokalen Räume erreichen können. Allerdings ist es in der vorgestellten sequentiellen Reduktionssemantik generell problematisch, Aussagen über die Effizienz von Berechnung zu treffen, da wir in jedem Schritt den zu reduzierenden Ausdruck wieder von der Wurzel an bis zu dem ersten reduzierbaren Teilterm durchlaufen müssen. Ob der einzig reduzierbare Parameter eines Konstruktorterms als erster oder letzter Parameter auftritt, kann bereits einen enormen Laufzeitunterschied bewirken. Ähnlich große Auswirkung kann beispielsweise die Vertauschung zweier Constraints in einer Konjunktion haben.

In einer auf Effizienz ausgerichteten und inbesondere parallelen Implementierung könnte aber ein genereller Überwachungsmechanismus zur Wiedererweckung suspendierter Programmteile vermutlich eine Effizienzsteigerung durch Verhinderung redundanter Berechnung bewirken.

7.3 Lambda-Abstraktionen

In TasteCurry werden Lambda-Abstraktionen durch das Prinzip des Lambda-Lifting implementiert, das wir in Abschnitt 2.3.4 kurz erläutert haben. Eine Abstraktion $\backslash X$ `->` e wird in einem Programm durch die partielle Funktionsapplikation `lambda`$(X_1, \ldots, X_n)$ ersetzt, wobei `lambda` ein neuer Name ist und $\{X_1, \ldots, X_n\} = free(e) \backslash \{X\}$ gilt. Für `lambda` wird ein definierender Baum der Form

$rule($`lambda(`$X_1, \ldots, X_n, X$`) =` $e)$

erstellt. Die freien Variablen von e können außerhalb der Lambda-Abstraktion auftauchen und müssen deshalb als Parameter an `lambda` übergeben werden.

Für jede in einem Programm verwendete Lambda-Abstraktion wird also ein zusätzliches Symbol zur Menge $\mathcal{F}$ der definierten Funktionssymbole hinzugefügt. Bei Verwendung der eingekapselten Suche bleibt $\mathcal{F}$ während der Programmausführung möglicherweise nicht konstant, da wir durch den Suchoperator neue Lambda-Abstraktionen berechnen können, für die nach dem Prinzip des Lambda-Lifting zur Laufzeit neue Funktionssymbole geschaffen und zu $\mathcal{F}$ hinzugefügt werden.

Beispiel 7.7
Die in Beispiel 5.11 vorgestellte Reduktion des Suchoperators

```
try(\X -> {add(X,z)=s(z)})

*
~> [\X -> {X=z /\ add(z,z)=s(z)},
     \X -> {local N in X=s(N) /\ add(s(N),z)=s(z)}]
```

wird in TasteCurry folgendes Ergebnis berechnen:

```
try(\X -> {add(X,z)=s(z)}) ~> ... ~> [$searchLamba1, $searchLambda2]
```

Durch `$` markieren wir in TasteCurry einen partiellen Funktionsaufruf. Für die beiden neuen Funktionen wurden Typinformationen und definierende Bäume der Form

$rule($`searchLambda1(X) = {add(z,z)=s(z)}`$)$

$rule($`searchLambda2(X) = {local N in X=s(N)/\add(s(N),z)=s(z)}`$)$

zum Speicher hinzugefügt. Es sind also tatsächlich während der Ausführung eines Curry-Programms neue Funktionen mit den Namen `searchLambda1` und `searchLambda2` definiert worden. Formal betrachtet wurde also die Menge $\mathcal{F}$ der definierten Funktionssymbole um diese beiden Namen erweitert. $\diamond$

Dieses Verhalten ist ungewöhnlich und tritt beispielsweise in funktionalen Sprachen nicht auf. Auch in Curry kann nur durch Verwendung des Suchoperators während der Programmausführung ein Funktionssymbol geschaffen und weiterverarbeitet werden, für das im Programmtext selbst keine definierende Gleichung aufgeführt ist.

Der Name eines solchen Funktionssymbols ist natürlich bei der Definition des Programms nicht bekannt, da er erst während der Laufzeit erstellt wird, und daher ist die Anwendung der vom Suchoperator berechneten Funktionen nur durch eine indirekte Benennung über Variablennamen möglich.

Beispiel 7.8

```
{dfs(\X -> {add(X,z)=s(z)})=[L] /\ L@Y}
```

Die Variable `L` übernimmt die Rolle eines Platzhalters für die noch unbekannte und unbenannte partielle Funktionsapplikation, die durch `dfs` berechnet wird. `L` kann daher wie eine solche Applikation behandelt und beispielsweise appliziert oder an eine Funktion wie `map` übergeben werden. Wir vernachlässigen die Variablenmenge und die Substitution in der folgenden Reduktion.

```
{dfs(\X -> {add(X,z)=s(z)})=[L] /\ L@Y}

    ~*> searchSpace(X₁,(∅,{X₁=s(z)},{}))
    ~>  $searchLambda1
~>  [$searchLambda1]=[L] /\ L@Y
~>  $searchLambda1@Y
~>  searchLambda1(Y)
~>  Y=s(z)
```

◇

Das Hinzufügen eines neuen Funktionssymbols zum Programm während der Laufzeit kann durchaus einigen Berechnungsaufwand erfordern, da nebem dem definierende Baum auch Informationen über den Typ und die Stelligkeit berechnet werden müssen. Daher sollte diese Arbeit nur dann durchgeführt werden, wenn sie wirklich notwendig ist, d.h. die berechnete Funktion auch im weiteren Verlauf des Programms verwendet wird.

Beispiel 7.9
Bei der Reduktion von

```
dfs(\X -> {add(X,z)=s(z)}
```

berechnet der Suchoperator als erstes Zwischenergebnis die beiden Funktionen `$searchLambda1` und `$searchLambda2` aus Beispiel 7.7. Nach Definition von `dfs` (bzw. `all`) werden diese Ergebnisse durch einen rekursiven Aufruf schließlich wieder an den Suchoperator übergeben, wodurch dann letztendlich eine Lösung

```
[$searchLambda3]
```

mit dem definierenden Baum

rule(`searchLambda3(X) = {X=s(z)}`)

berechnet wird. Der Aufwand für die Erstellung der beiden ersten `searchLambda`-Funktionen war also unnötig, da keine von ihnen für den weiteren Programmablauf verwendet wird. ◇

Wir verzögern daher in TasteCurry die Umwandlung von Suchergebnissen in Funktionen so lange wie möglich, indem wir durch den Suchoperator anstelle von Lambda-Abstraktionen den Suchraum selbst entsprechend gekennzeichnet zurückliefern lassen. Erfolgt ein Zugriff auf diesen Raum, beispielsweise in Form einer Applikation, erstellen wir die entsprechende Funktion und führen sie aus. Wird aber der Suchraum erneut an den Suchoperator übergeben, kann die Reduktion ohne Verzögerung fortgesetzt werden, und wir haben den Aufwand für die Umwandlung in eine Funktion und die erneute Erstellung eines Suchraums vermieden.

Beispiel 7.10
Als erstes Zwischenergebnis des Aufrufs `dfs(\X -> {add(X,z)=s(z)})` liefert der Suchoperator zwei Suchräume zurück, die die beiden Reduktionswege enthalten und zur besonderen Kennzeichnung durch `searchLambda` markiert werden[2]:

[2] Man beachte, daß innerhalb von `searchLambda` Variablenmengen und Substitutionen als Listen und Substitutionen zudem in der Form X/e statt X=e dargestellt werden.

```
[searchLambda(X,([],[X/z],add1(z,z)=s(z))),
 searchLambda(X,([N],[X/s(N)],add(s(N),z)=s(z)))])
```

Die Namensgleichheit der Variablen in beiden Räumen ist unproblematisch, da die Ergebnisse erst nach Umwandlung in eine Lambda-Abstraktion bzw. eine entsprechende Funktion global sichtbar gemacht werden können.

Die erneute Anwendung des Suchoperators auf ein solches `searchLambda`-Konstrukt kann sofort einen Reduktionsschritt ausführen und liefert wieder einen Suchraum zurück:

```
try(searchLambda(X,([],[X/z],add1(z,z)=s(z))))
⇝ searchSpace(A,[],[A/z],z=s(z))
```

Auf diese Weise wird die unnötige Erstellung von `searchLambda1` und `searchLambda2` aus Beispiel 7.7 vermieden. ◇

Durch die verzögerte Umwandlung von Suchergebnissen konnte in TasteCurry die Anzahl der Berechnungsschritte bei Verwendung der `all`- und `best`-Algorithmen beträchtlich gesenkt werden, da für jedes Zwischenergebnis der Suchraum erhalten bleibt und im nächsten Schritt direkt weiter reduziert werden kann. Die Umwandlung in Lambda-Abstraktionen und die erneute Erstellung von Suchräumen nach dem in Abschnitt 5.2.2 beschriebenen Prinzip ist also nicht mehr notwendig.

Wir haben in diesem Kapitel nur die wichtigsten Unterschiede zwischen der vorgestellten Theorie und der TasteCurry-Implementierung aufgeführt und erläutert. Einige kleinere Abweichungen wurden aus programmiertechnischen oder Effizienzgründen vorgenommen, sind jedoch nicht von entscheidender Bedeutung. In Anhang A werden wir die Installation und Benutzung von TasteCurry-Systems erklären.

Kapitel 8

Fazit und Ausblick

Wir haben in dieser Arbeit die Erweiterung von Curry um das Konzept der eingekapselten Suche durch Verwendung eines Suchoperators und der Committed-Choice vorgestellt. Dabei haben wir gesehen, daß zur Unterscheidung zwischen lokalen und globalen Variablen die Einführung eines Existenzquantors und die Erweiterung der operationalen Semantik zur Protokollierung der Variablen eines Berechnungsraums notwendig sind. Wir haben einige Möglichkeiten zur Verwendung der eingekapselten Suche erläutert, dabei aber auch festgestellt, daß mit ihrer Hilfe List-Comprehensions aus funktionalen Sprachen offenbar nicht durch äquivalente Ausdrücke in Curry formuliert werden können. Inwieweit damit die Fähigkeiten des Suchoperators und der Committed-Choice den Anforderungen genügen, die Anwendungsprogramme in Curry an die eingekapselte Suche stellen werden, und ob Curry um die Möglichkeit zur direkten Verwendung von List-Comprehensions erweitert werden sollte, muß sich in der Zukunft noch zeigen.

Die vorgestellten Reduktionssemantiken für den Suchoperator und die Committed-Choice sind so allgemein wie möglich gehalten und sollten daher ohne Schwierigkeiten in anderen Implementierungen als TasteCurry verwendet werden können. Allerdings ist das Konzept auf der Basis der vorgestellten und in TasteCurry verwendeten sequentiellen Reduktionssemantik entwickelt worden. In einer effizienteren Implementierung, in der beispielsweise die nebenläufige Konjunktion zweier Constraints oder die einzelnen Antwortausdrücke eines disjunktiven Ausdrucks durch eigene Threads reduziert werden [15], könnte daher eine andere Verwaltung lokaler Räume sinnvoll sein.

Beispielsweise werden bei der Implementierung von Oz in der abtrakten Maschine AMOZ [25] Berechnungsräume in einer Baumstruktur verwaltet, in der jeder Raum einen Zeiger auf den übergeordneten besitzt und der globale Berechnungsraum die Wurzel bildet. Jeder Raum enthält zusätzlich ein Statusflag, das anzeigt, ob er noch weiter reduzierbar, fehlgeschlagen oder gelöst ist, und Zeiger auf die Instruktionen, die in den letzten beiden Fällen ausgeführt werden sollen. Für die Verwaltung der lokalen Variablenmengen wurde in AMOZ ein anderer Ansatz gewählt, der die einfache Realisierung eines Suspensions- und Wiedererweckungsmechanismus ermöglicht.

Variablen werden als Objekte dargestellt, die einen eindeutigen Index besitzen, der auf eine Position in einem sogenannten *Bindungsarray* verweist. Dort werden die aktuellen Bindungen der Variablen gespeichert. Zusätzlich besitzt eine Variable aber noch einen Zeiger auf den Berechnungsraum, in dem sie deklariert wurde, und eine Struktur zur Speicherung aller Berechnungen, die suspendiert wurden, weil sie auf eine Bindung dieser Variable warten. Im Gegensatz zu unserer Verfahrensweise wird also nicht jedem Raum die Menge seiner lokalen Variablen angehängt, sondern umgekehrt jeder Variable ihr „Heimatraum“. Soll eine Variable gebunden werden, wird geprüft, ob der aktuelle Raum mit ihrem Heimatraum übereinstimmt. Ist dies der Fall, wird die Bindung vorgenommen, ansonsten wird die Berechnung suspendiert, wenn kein anderer Schritt mehr möglich ist. Wenn ein lokaler Raum aufgelöst und mit dem übergeordneten vereinigt wird, müssen die Zeiger der verbliebenen lokalen Variablen verändert werden, da sie nun einem neuen Heimatraum zugeordnet werden. Wir haben dies beim Suchoperator durch erneute Deklaration der lokalen Variablen (vgl. Beispiel 5.14) und bei der Committed-Choice durch Hinzufügen der lokalen Variablen zum übergeordneten Raum erreicht (vgl. Beispiel 5.24).

Auf jeden lokalen Raum, der suspendiert wird, weil er auf die Bindung einer Variable warten muß, wird in der Struktur der Variable durch einen Zeiger verwiesen, so daß nach einer Bindung die suspendierten Räume sofort wieder aufgeweckt werden können. Dieses Vorgehen ist wesentlich effizienter als die Lösung, die wir in TasteCurry versuchsweise realisiert und in Abschnitt 7.2 vorgestellt haben, und bietet sich in einer Implementierung an, in der lokale Räume durch unabhängige und nebenläufig ausgeführte Threads verwirklicht werden. Die Information über eine Variablenbindung kann durch Verwendung von Zeigern sofort in alle betroffenen Räume weitergereicht werden, während es in TasteCurry längere Zeit dauern kann, bis ein lokaler Raum von der Bindung erfährt.

Die skizzierten Ideen aus AMOZ könnten beispielsweise ein sinnvoller Ansatz zur Realisierung der eingekapselten Suche in einer Implementierung von Curry sein, die zur Zeit von Hanus und Sadre in der objektorientierten Sprache Java [26] entwickelt wird [15]. Im Gegensatz zu der in dieser Arbeit vorgestellten sequentiellen Reduktionssemantik werden für die Auswertung der beiden Constraints einer nebenläufigen Konjunktion sowie für die unabhängige Auswertung mehrerer globaler Berechnungsräume eigene Threads verwendet, und dieses Prinzip ließe sich wahrscheinlich ohne große Probleme auf lokale Räume übertragen.

Variablen werden ähnlich wie in AMOZ durch Objekte mit einem Index verwirklicht, der auf ein Bindungsarray verweist, in dem die aktuellen Bindungen der Variablen gespeichert sind. Durch eine Erweiterung der Variablenobjekte um eine Struktur zur Protokollierung der lokalen Räume, die zur Fortsetzung ihrer Reduktion auf eine Bindung der Variable warten müssen, wäre wohl der in AMOZ verwendete Suspensions- und Wiedererweckungsmechanismus in ähnlicher Weise realisierbar.

Die Unterscheidung zwischen lokalen und globalen Variablen könnte auf zwei verschiedene Arten vorgenommen werden. Entsprechend unserem Ansatz könnte man jedem lokalen Raum ein eigenes Bindungsarray zuordnen und nur Bindungen der dort aufgeführten Variablen zulassen. In Anlehnung an Oz würde man hingegen ein globales Array verwalten und die Variablenobjekte um Zeiger auf ihren Heimatraum erweitern. Dieses Vorgehen entspricht dem Prinzip der Nummerierung lokaler Objekte entsprechend ihrer Schachtelungstiefe, das ausführlich in [27] beschrieben wird. Welche der Möglichkeiten leichter zu realisieren und effizienter in bezug auf die Laufzeit ist, muß noch untersucht werden.

Unabhängig von der Implementierung ist zu überlegen, ob die jüngste Erweiterung des Konzepts der eingekapselten Suche in Oz [31] auch in Curry umgesetzt werden sollte. Wie bereits erwähnt, basiert die von uns definierte operationale Semantik auf den ursprünglichen Ideen der Verwendung von Suchoperatoren, die in [32] vorgestellt wurden. Die neueste Version von Oz verwendet jedoch einen stark veränderten, wesentlich flexibleren Ansatz: Berechnungsräume werden als Datenobjekte verfügbar gemacht und können vom Benutzer erzeugt, vervielfältigt, aufgelöst oder beispielsweise auch als Parameter übergeben oder als Ergebnis einer Funktion zurückgeliefert werden.

Anstatt wie in früheren Versionen durch Suchoperatoren einen lokalen Raum zu erzeugen und eine oder mehrere Lambda-Abstraktionen zur Repräsentation seines Reduktionsergebnisses zurückzuliefern, kann nun der Zustand eines Berechnungsraums durch eine entsprechende Prozedur abgefragt werden. Erhält man als Antwort beispielsweise `alternatives`, dann kann die Berechnung verschiedene Wege einschlagen und man kann diese durch Vervielfältigung des Raums unabhängig voneinander untersuchen. Man beachte, daß auf diese Weise der Aufwand für die Umwandlung von Suchräumen in Lambda-Abstraktionen und das Erzeugen neuer Räume bei erneuter Anwendung des Suchoperators vermieden werden. Einen kleinen Ansatz in dieser Richtung haben wir auch in TasteCurry verwirklicht (vgl. Absatz 7.3).

Die sehr umfangreichen neuen Zugriffs- und Verwaltungsmöglichkeiten für Berechnungsräume ermöglichen in Oz die Realisierung weiterer Suchverfahren wie Saturation, Limited Discrepancy Search und visuelle Suche, bei der der Suchbaum graphisch dargestellt wird und vom Benutzer interaktiv erforscht werden kann [31, 30].

Wie wir in Kapitel 6 gesehen haben, lassen sich mit der bisherigen Form der eingekapselten Suche in Curry die Forderungen aus Abschnitt 3.1 verwirklichen und schon einige nützliche Anwendungen programmieren. Ob die Umsetzung des neuen Konzepts aus Oz in zukünftigen Versionen von Curry sinnvoll und erforderlich ist, muß noch überlegt werden.

Anhang A

Verwendung des TasteCurry-Systems

Der Quellcode des in Kapitel 7 beschriebenen TasteCurry-Systems befindet sich auf der beigefügten Diskette. TasteCurry wurde in SICStus Prolog Version 3 #3 [28] unter Solaris und Linux entwickelt. Es ist auf jedem System installierbar, für das SICStus Prolog Version 3 #3 oder höher verfügbar ist und das Dateinamen mit einer Länge von mehr als 8+3 Zeichen verarbeiten kann. Möglicherweise ist das System auch unter einer älteren Version von SICStus lauffähig, dies wurde aber nicht geprüft.

Wir führen zunächst den Inhalt der Diskette mit einer kurzen Beschreibung der Funktion der einzelnen Dateien auf und erläutern anschließend die Installation von TasteCurry.

A.1 Inhalt der Diskette

Auf der Diskette befinden sich zwei Unterverzeichnisse. Das Unterverzeichnis `program` enthält in gepackter Form als `program.tar`- bzw. `program.zip` alle Programmdateien, die zur Installation und Benutzung des Systems notwendig sind, im Unterverzeichnis `examples` befinden sich in `examples.tar` bzw. `examples.zip` die wichtigsten der in dieser Arbeit vorgestellten Beispielanwendungen. Alle Prolog-Dateien sind durch die Erweiterung `.pl` gekennzeichnet, bei Dateien mit der Erweiterung `.fl` handelt es sich um TasteCurry-Programme, deren Inhalt vom Benutzer eingelesen und verarbeitet werden kann.

Inhalt des `program`-Archivs

`curry.pl`	Benutzer-Interface und Implementierung der Hauptreduktionsfunktionen *cse*, *equal* und *cs*.
`search.pl`	Zusätzliche Reduktionsfunktionen für die eingekapselte Suche
`readin.pl`	Einlesen und Konvertieren von `.fl`-Dateien in eine interne Darstellung
`local.pl`	Erstellung der Normalform für Regeln eingelesener `.fl`-Dateien
`typecheck.pl`	Typchecker zur Überprüfung der Typkorrektheit der Regeln nach ihrer Umwandlung in die Normalform
`maketree.pl`	Erstellung der definierenden Bäume für Regeln nach der Überprüfung durch den Typchecker
`operators.pl`	Definition der verwendeten Operatoren und Schlüsselwörter und Festlegung ihrer Präzedenzen
`external.pl`	Einbindung externer Prolog-Funktionen in das TasteCurry-System für Ein-/Ausgabe und arithmetischen Operationen auf Integerzahlen
`prelude.fl`	In TasteCurry vordefinierte Funktionen und Sprachelemente

Der Inhalt der Datei `prelude.fl` ist in Anhang B aufgeführt. Die in ihr definierten Funktionen und Datentypen werden dem Benutzer zusätzlich zu den intern vordefinierten Funktionen `=`, `==`, `@`, `/\`, `try` und `choice` standardmäßig zur Verfügung gestellt.

Die in dieser Arbeit eingeführten Erweiterungen sind überwiegend in den ausführlich in englischer Sprache kommentierten Dateien `local.pl` und `search.pl` enthalten. Sie verwirklichen die vorgestellte Theorie der Kapitel 4 und 5 mit den in Kapitel 7 beschriebenen Abweichungen. Einige darüber hinausgehende kleinere Unterschiede sind kommentiert.

Inhalt des `examples`-Archivs

`bahn.fl`	Erweiterte Wegesuche im Graph mit Zykelkontrolle und gewichteten Kanten (vgl. Abschnitt 6.2)
`england.fl`	Übersetzung des Escher-Programms mit Beispielaufrufen (Abschnitt 6.3)
`lc.fl`	Beispiele zur Simulation von List-Comprehensions (Abschnitt 6.8)
`prolog.fl`	Simulation der Prolog-Benutzerschnittstelle (Abschnitt 6.7)
`reservation.fl`	Platzreservierungsprogramm (Abschnitt 6.7)
`salesman.fl`	Travelling Salesman, eine weitere Anwendung der Wegesuche im Graph.
`various.fl`	Verschiedene Funktionen aus Beispielreduktionen: `add`, `append`, `last`, `member`, `merge`, `sumOne`, `grossvater` u.a.
`wegesuche.fl`	Wegesuche im Graph (Abschnitt 6.2)

A.2 Installation von TasteCurry

Voraussetzung für die Installation ist, daß SICStus Prolog Version 3 #3 oder höher auf dem jeweiligen System installiert ist und die ausführbare Datei `sicstus` in einem Verzeichnis liegt, das in der `PATH`-Umgebungsvariable aufgeführt ist. Folgende Schritte sind dann zur Installation durchzuführen:

1. Eines der Archive aus dem Verzeichnis `program` muß von der Diskette in ein Verzeichnis auf der Festplatte kopiert und dort entpackt werden. Die Datei `program.zip` kann mit Hilfe des `unzip`-Tools durch folgenden Befehl entpackt werden (dazu muß `unzip` auf dem System zur Verfügung stehen und im Pfad liegen):

   ```
   unzip program.zip
   ```

   Die Datei `program.tar` kann mit Hilfe des `tar`-Tools durch folgenden Befehl entpackt werden (dazu muß `tar` auf dem System zur Verfügung stehen und im Pfad liegen):

   ```
   tar -xvf PROGRAM.TAR
   ```

   Anschließend enthält das Verzeichnis auf der Festplatte die in Abschnitt A.1 beschriebenen Programmdateien.

   Damit ist eine mögliche Form der Installation bereits beendet, für die aber zu jeder Verwendung von TasteCurry der Aufruf von SICStus Prolog notwendig ist (siehe Anhang A.3). Die folgenden Schritte erklären eine Installationsmöglichkeit, um eine ausführbare Datei zu erstellen, die unabhängig von SICStus Prolog gestartet und verwendet werden kann.

2. Aus dem Verzeichnis auf der Festplatte heraus muß der SICStus-Prolog-Interpreter durch die Eingabe von `sicstus` gestartet werden. Daraufhin erscheinen eine Meldung und ein Eingabeprompt in der folgenden oder einer ähnlichen Form (Wir haben zuletzt Version 3 #5 verwendet):

   ```
   SICStus 3  #5: Wed May 21 00:38:30 MEST 1997
   | ?-
   ```

3. Durch die Eingabe von

   ```
   [curry].
   ```

   werden die TasteCurry-Programmdateien geladen und der Prompt `| ?-` erscheint wieder.

4. Um nicht für jede Verwendung von TasteCurry den SICStus-Interpreter starten zu müssen, kann nun ein sogenannter „saved state“ erstellt werden. Dazu muß der Befehl

   ```
   | ?- save(tastecurry),cs,halt.
   ```

   eingegeben werden, woraufhin SICStus die Meldung

   ```
   {SICStus state saved in /home/fufu/curry/tastecurry}
   ```

   mit einem entsprechend anderen Verzeichnispfad ausgibt und den TasteCurry-Interpreter startet, der folgende Meldung und einen Eingabeprompt ausgibt:

   ```
   TasteCurry Interpreter (Version of 5/10/97 by Informatik II, RWTH Aachen)

   Type "help." for list of available commands
   |:
   ```

   Durch die Eingabe von

   ```
   |: exit.
   ```

   gelangt man zurück auf die Kommandozeile. Im aktuellen Verzeichnis wurde nun eine ausführbare Datei mit dem Namen `tastecurry` erstellt, durch deren Aufruf der TasteCurry-Interpreter gestartet wird. Diese Datei kann in jedes beliebige Verzeichnis und auch auf andere Computer kopiert werden, und es ist für ihren Aufruf nicht notwendig, daß SICStus Prolog installiert ist oder im Pfad liegt. Allerdings ist die ausführbare Datei nur auf Computern verwendbar, die dieselbe Rechnerarchitektur haben und dasselbe Betriebssystem benutzen wie der Computer, auf dem die Datei erstellt wurde („Binärkompatibilität“).

A.3 Aufruf und Benutzung von TasteCurry

Zur Verwendung der Beispielprogramme müssen die Dateien aus dem Archiv `examples.zip` oder `examples.tar` aus dem `examples`-Verzeichnis auf der Diskette in ein Verzeichnis auf der Festplatte kopiert und dort äquivalent zu der Beschreibung im vorherigen Abschnitt mit Hilfe von `unzip` bzw. `tar` entpackt werden. Die Datei `prelude.fl` muß ebenfalls in das Verzeichnis auf der Festplatte kopiert werden. Nun kann der TasteCurry-Interpreter aus diesem Verzeichnis heraus durch Eingabe von

```
tastecurry
```

gestartet werden, wenn entsprechend dem letzten Abschnitt eine ausführbare Datei `tastecurry` erzeugt wurde und diese sich im gleichen Verzeichnis wie die Beispielprogramme oder in einem durch die `PATH`-Umgebungsvariable aufgeführten Verzeichnis befindet. Daraufhin startet der TasteCurry-Interpreter mit folgender Meldung:

```
TasteCurry Interpreter (Version of 5/10/97 by Informatik II, RWTH Aachen)

Type "help." for list of available commands
|:
```

Wurde keine ausführbare Datei erzeugt, müssen die Beispielprogramme in dasselbe Verzeichnis wie die Programmdateien aus dem `program`-Archiv kopiert und entpackt werden. Aus diesem Verzeichnis heraus muß der SICStus-Interpreter durch Eingabe von `sicstus` gestartet werden. Nach Erscheinen des Prompts ist der Befehl

```
| ?- [curry].
```

einzugeben. Sobald der Prompt wieder erscheint, kann TasteCurry durch die Eingabe von

```
| ?- cs.
```

gestartet werden.

Folgende Kommandos können nun in TasteCurry eingegeben werden:

`read prog.`	Die Programmdatei `prog.fl` wird eingelesen, wenn sie sich im aktuellen Verzeichnis befindet. Wenn andere Zeichen als Buchstaben in `prog` auftreten, muß der Name in Hochkommata eingeschlossen werden, z.B. `"123.test"`. Alle in der Programmdatei enthaltenen Funktionen und Datentypdeklarationen werden eingelesen, in ihre Normalform umgewandelt und durch den Typchecker überprüft. Anschließend werden die definierenden Bäume für die Funktionen erstellt.
<*expression*>.	Der eingegebene Ausdruck wird unter Verwendung aller vordefinierten und der durch das zuletzt eingelesene Programm definierten Funktionen reduziert, d.h. in einen gelösten Ausdruck zu überführen versucht. Vor Beginn der Reduktion wird der Ausdruck in seine Normalform umgewandelt (vgl. Abschnitt 7.1) und durch den Typchecker überprüft. Nach Beendigung der Reduktion wird das Ergebnis auf dem Bildschirm ausgegeben.
`trace./notrace.`	Schaltet den Trace-Modus ein oder aus. Ist er eingeschaltet, wird nach jedem Reduktionsschritt das Zwischenergebnis ausgegeben.
`single./nosingle.`	Schaltet den Einzelschrittmodus ein oder aus. Ist er eingeschaltet, wird nach jedem Reduktionsschritt das Zwischenergebnis ausgegeben und der Benutzer nach dem weiteren Vorgehen gefragt.
`time./notime.`	Schaltet den Zeitmodus ein oder aus. Ist er eingeschaltet, wird nach jeder Reduktion die dafür benötigte Zeit ausgegeben.
`type` <*expression*>.	Gibt den Typ des Ausdrucks < *expression* > aus.
`eval` f.	Gibt den definierenden Baum der Funktion f aus.

`exit.` Beendet den TasteCurry-Interpreter.

Nach Eingabe von `exit.` landet man auf der Kommandozeile, wenn eine ausführbare Datei `tastecurry` verwendet wurde, oder andernfalls wieder im SICStus-Interpreter, der durch Eingabe von `halt.` beendet werden kann.

Nach dem Start des TasteCurry-Interpreters muß man also zunächst ein TasteCurry-Programm einlesen, das in einer Datei mit der Erweiterung `.fl` gespeichert sein muß, bevor man Anfragen stellen und auswerten lassen kann. Das Programm zur Wegesuche im Graph beispielsweise kann durch

```
read wegesuche.
```

eingelesen werden. Danach kann z.B. der Ausdruck

```
|: weg(b,X).
```

eingegeben werden, nach dessen Reduktion das Ergebnis in der Form

```
{X=b} [b] | {X=c} [b,c] | {X=d} [b,c,d]
```

ausgegeben wird. Alle beigefügten `.fl`-Dateien enthalten neben den Funktionsdefinitionen auch einige Beispielanfragen, die wir teilweise in dieser Arbeit verwendet haben. Man beachte, daß für den Existenzquantor vor einer Bedingung bzw. vor einem Wächter in TasteCurry der Infixoperator `localIn` verwendet werden muß und daß bei bestimmten Ausdrücken möglicherweise eine zusätzliche Klammerung erforderlich ist. Beispielsweise muß der Ausdruck

```
choice {local X in X=1} -> choice local X in {X=2} -> X.
```

in der Form

```
choice {local X in X=1} -> (choice X localIn {X=2} -> X).
```

in der Kommandozeile eingegeben werden. Abgesehen davon können aber alle in dieser Arbeit vorgestellten Anfragen oder Programmbeispiele in der dabei verwendeten Form in TasteCurry verwendet werden.

Anhang B

Die Prelude-Datei

Die Datei `prelude.fl` wird vor dem Einlesen eines TasteCurry-Programms automatisch geladen, so daß die in ihr definierten Funktionen in allen vom Benutzer geschriebenen Programmen und in Ausdrücken, die auf der Kommandozeile eingegeben werden, verwendet werden können. Darüber hinaus sind intern die Funktionen `=` und `==` für die strikte Gleichheit, `@` für die Applikation, `/\` für die Konjunktion sowie der Suchoperator `try` und das Committed-Choice-Konstrukt `choice` vordefiniert.

Wir geben nachfolgend den Inhalt der Prelude-Datei an. Dabei fällt auf, daß die Funktionen `/\`, `try`, `choice` und die Existenzquantoren `localVars` und `localInPar` als externe Funktionen definiert sind. Dies ist ein vorläufiger Behelf, um die Überprüfung der korrekten Verwendung dieser Konstrukte durch den Typchecker vornehmen zu können. In einer späteren Version werden diese Deklarationen entfernt werden.

Die aufgeführten Suchalgorithmen unterscheiden sich aus Effizienzgründen von der Theorie. Für das Sammeln aller Lösungen gibt es zum Vergleich die Funktion `all2`, die die Definition aus Abschnitt 6.1 realisiert, und eine schnellere Variante `all`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Standard prelude for TasteCurry (Version of 10/04/97)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Infix operator declarations:

infixr(950,/\).
infixr(800,&&).
infix(700,==).
infix(700,<).
infix(700,>).
infix(700,<=).
infix(700,>=).
infixr(500,++).
infixl(500,+).
infixl(500,-).
infixl(400,*).
infixl(400,div).
infixl(400,mod).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Boolean values:

data bool = true ; false.

% Sequential conjunction (used in "=="):
(&&) :: bool -> bool -> bool.
(&&) eval 1:rigid.

true  && X = X.
false && X = false.

% negation:
not :: bool -> bool.

not(true)  = false.
not(false) = true.

% if-then-else:
ite :: bool -> A -> A -> A.

ite(true ,X,Y) = X.
ite(false,X,Y) = Y.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Pairs:

data pair(A,B) = (A,B).

fst :: pair(A,B) -> A.
fst((X,_)) = X.

snd :: pair(A,B) -> B.
snd((_,Y)) = Y.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Polymorphic lists:

data list(A) = [] ; [A|list(A)].

head([X|_])  = X.

tail([_|Xs]) = Xs.
```

```
% list concatenation:
(++) :: list(A) -> list(A) -> list(A).

[]     ++ Ys = Ys.
[X|Xs] ++ Ys = [X|Xs++Ys].

% list length:
length :: list(A) -> int.

length([])     = 0.
length([_|Xs]) = 1+length(Xs).

% apply a function to all list elements:
map :: (A->B) -> list(A) -> list(B).

map(F,[])     = [].
map(F,[X|Xs]) = [F@X|map(F,Xs)].

% accumulate all list elements:
foldr :: (A->B->B) -> B -> list(A) -> B.

foldr(F,Z,[])    = Z.
foldr(F,Z,[H|T]) = F@H@foldr(F,Z,T).

% filter elements in a list:
filter :: (A -> bool) -> list(A) -> list(A).

filter(P,[])     = [].
filter(P,[X|Xs]) = if P@X then [X|filter(P,Xs)] else filter(P,Xs).

% join two lists to one list of pairs:
zip :: list(A)->list(B)->list(pair(A,B)).

zip([],    [])     = [].
zip([X|Xs],[Y|Ys]) = [(X,Y)|zip(Xs,Ys)].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Types of primitive arithmetic functions and predicates:

external (+)   :: int -> int -> int.
external (-)   :: int -> int -> int.
external (*)   :: int -> int -> int.
external (div) :: int -> int -> int.
external (mod) :: int -> int -> int.
external (<)   :: int -> int -> bool.
external (>)   :: int -> int -> bool.
external (<=)  :: int -> int -> bool.
external (>=)  :: int -> int -> bool.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Types of primitive IO functions:

data io(A) = '$io'(A).   % constructor $io should not be used in user programs

data unit = unit.

(>>)  :: io(A) -> io(B)      -> io(B).
'$io'(_) >> A = A.

(>>=) :: io(A) -> (A->io(B)) -> io(B).
'$io'(V) >>= F = F@V.

external putChar :: int -> io(unit).
external getChar :: io(int).
external done    :: io(unit).
external return  :: Type -> io(Type).

external readFile :: list(int) -> io(list(int)).
         external '#readFileContents' :: filename -> io(list(int)).
         '#readFileContents' eval rule.

external writeFile :: list(int) -> list(int) -> io(unit).
writeFile eval 1:rigid.
          external '#writeFileContents' :: filename -> list(int) -> io(unit).
          '#writeFileContents' eval 2:rigid.

putStr :: list(int) -> io(unit).
putStr([]) = done.
putStr([C|Rest]) = putChar(C) >> putStr(Rest).

putStrLn :: list(int) -> io(unit).
putStrLn(Cs) = putStr(Cs) >> putChar(10).

getLine = getChar >>= \C ->
          if C==10 then return([]) else (getLine>>= \Cs -> return([C|Cs])).

external show :: A -> io(unit).
show eval rule.

external showNl :: A -> io(unit).
showNl eval rule.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Constraints (internal values):

data constraint = {}.

% Conditional expression:
cond :: constraint -> A -> A.
cond eval 1:rigid.

cond({},X) = X.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Declarations of internal functions:

external (/\) :: constraint -> constraint -> constraint.

external choice:: list(pair(constraint,A)) -> A.

external localVars:: A -> constraint ->constraint.

external localInPar:: A -> constraint ->constraint.

external try:: (A->constraint)->list(A->constraint).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Search-Stuff

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tools

% unpack a solution from a lambda-abstraction and write
% it to  to the screen. This is not possible with the putStr function
browse(A) if {A@X} = show(X).

% unpack solutions from a list of lambda abstractions and write
% them to the screen
browseList([])=done.
browseList([A|B]) if {A@X} = showNl(X)>>browseList(B).

% unpack solutions from a list of lambda abstractions, return them
% in a list
unpack([]) = [].
unpack([A|B]) if {A@X} = [X|unpack(B)].

% merge a list of lists into one list
concat :: list(list(A)) -> list(A).
concat([])    = [].
concat([A|B]) = A++concat(B).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Search Algorithms

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Depth First Search
%
% - get the first solution via left-to-right strategy
% - Works as:
% dfs(G) = takefirst(all(G))
%           where
%             takefirst([])    = [].
%             takefirst([A|B]) = A.
% But the following algorithm is about 25% faster.

dfs :: (A->constraint) -> list(A->constraint).

dfs(G) = evaldfs(try(G))
         where
           evaldfs([])      = [];
           evaldfs([X|Xs])  = eval2(try(X),Xs);

           eval2([],Xs)      = evaldfs(Xs);
           eval2([L],Xs)     = [L];
           eval2([A,B|C],Xs) = eval2(try(A),[B|C]++Xs).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% All Search
%
% - get all solutions via left-to-right strategy
% - Works as the following algorithm:
% all(G) = evalResult(try(G))
%          where
%            evalResult([])      = [];
%            evalResult([S])     = [S];
%            evalResult([A,B|C]) = concat(map(all,([A,B|C]))).
%
% The following all algorithm is faster.
% For comparison we have all2, which realizes the above algorithm.

all :: (A->constraint) -> list(A->constraint).
all(G) =  evalall(try(G))
          where
            evalall([])    = [];
            evalall([A|B]) = evalall2(try(A),B);

            evalall2([],B)        = evalall(B);
            evalall2([L],B)       = [L|evalall(B)];
            evalall2([C,D|E],B)   = evalall2(try(C),[D|E]++B).
```

```
all2 :: (A->constraint) -> list(A->constraint).
all2(G) = eval(try(G))
         where
           eval([])      = [];
           eval([S])     = [S];
           eval([A,B|C]) = concat(map(all2,([A,B|C]))).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% findall
%
% - get all solutions via left-to-right strategy and unpack
%   the values from the lambda-abstractions. Similar to
%   Prolog's findall.

findall :: (A->constraint) -> list(A).

findall(G) = unpack(all(G)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Best Search
%
% - get best solution via left-to-right strategy, according to
%   a specified operator that can always take a decision which

%   of two solutions is better.
%   In general, the operation should be rigid in its arguments!

best :: (A->constraint) -> (A->A->bool) -> list(A->constraint).

best(G,Compare) = bestHelp([],try(G),[])
      where
        bestHelp([],[],Best)     = Best;
        bestHelp([],[B|Bs],Best) = evalB(try(constrain(B,Best)),Bs,Best);
        bestHelp([A|As],Bs,Best) = evalA(try(A),As,Bs,Best);

        evalB([],Bs,Best)        = bestHelp([],Bs,Best);
        evalB([NewBest],Bs,Best) = bestHelp([],Bs,[NewBest]);
        evalB([C,D|As],Bs,Best)  = bestHelp([C,D|As],Bs,Best);

        evalA([],As,Bs,Best)        = bestHelp(As,Bs,Best);
        evalA([NewBest],As,Bs,Best) = bestHelp([],As++Bs,[NewBest]);
        evalA([C,D|E],As,Bs,Best)   = bestHelp([C,D|E]++As,Bs,Best);

        constrain(B,[])  = B;
        constrain(B,[Best]) = \X -> {local Y in Best@Y/\Compare@X@Y=true/\B@X}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Search with condition
%
% - searches the first solution that meets a specified condition.
%   Cond must be a unary boolean function.

condSearch(G,Cond) = one(\X -> {G@X /\ Cond@X=true}).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% One Search
%
% - searches one solution via a fair strategy. This is possible because
%   we use choice to compute a solution.
% - to controll failure and to return the solution in a lambda-abstraktion
%   we use dfs around the choice-expression.

one :: (A->constraint)->list(A->constraint).

one(G) = dfs(\X -> {X = choice Y localIn {G@Y} -> Y}).
```

Anhang C

Syntax von Curry

Die Syntax von Curry ist noch nicht abschließend definiert und kann sich in zukünftigen Versionen noch verändern. Die Erläuterungen in Anhang C.1 wurden mit freundlicher Genehmigung von Hanus aus dem Curry-Report [16], Version 6/97, entnommen und übersetzt. Die EBNF zur Definition der kontextfreien Syntax in Anhang C.2 stammt ebenfalls aus diesem Report und wurde uns freundlicherweise von Hanus als LaTeX-Datei zur Verfügung gestellt. Die von uns vorgenommenen Änderungen betreffen die Erweiterung um den Existenzquantor bei der Definition des `choice`-Konstrukts, von Constraints und von bedingten Regeln sowie das Entfernen der "`and`"- und Hinzufügen der "*f* `eval flex`"- bzw. "*f* `eval flex`"-Auswertungsvorschriften.

Es wird die kontextfreie Syntax der Sprache Curry beschrieben, die an Haskell angelehnt ist und sich von der TasteCurry-Syntax, die wir in dieser Arbeit verwendet haben, in einigen Punkten unterscheidet. In Kapitel 2.2 haben wir bereits die wichtigsten Unterschiede erklärt oder an einem Beispiel aufgezeigt. Darüber hinaus ist noch zu erwähnen, daß in Curry das Symbol "`|`" anstelle von "`if`" bei der Deklaration einer bedingten Regel verwendet wird und daß die bewachten Ausdrücke eines `choice`-Konstrukts sowie lokale Regeldeklarationen nicht mit einem Semikolon abgeschlossen werden müssen.

Eine jeweils aktuelle Definition des Sprachumfangs und der Syntax von Curry kann im Curry-Report [16] nachgelesen werden.

C.1 Lexikalische Vorschriften

Für die Definition von Bezeichnern ist die Groß- oder Kleinschreibung des Anfangsbuchstabens von Bedeutung. Es gibt in Curry vier grundsätzliche Modi für die Angabe von Bezeichnern, die zur Compile-Zeit ausgewählt werden können:

- Prolog-Modus: der in TasteCurry verwendete Modus. Variablen beginnen mit einem Großbuchstaben, alle anderen Bezeichner mit einem Kleinbuchstaben.
- Gödel-Modus: die Umkehrung des Prolog-Modus, d.h. Variablen beginnen mit Kleinbuchstaben, die restlichen Bezeichner mit Großbuchstaben.
- Haskell-Modus: siehe Abschnitt 1.3 des Haskell-Report.
- Freier Modus: keinerlei Vorschriften über die Groß- oder Kleinschreibung.

Standardmäßig ist der freie Modus eingestellt.

Die Gestalt der Nichtterminalsymbole *TypeConstrID*, *DataConstrID*, *TypeVarID*, *InfixOpID*, *FunctionID* und *VarID*, die Klassen von Bezeichnern angeben, wird in der nachfolgenden kontextfreien Syntax nicht definiert. Außer *InfixOpID* beginnen alle Bezeichner einer dieser Klassen mit einem Anfangsbuchstaben, der je nach Modus groß- oder kleingeschrieben wird, gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern und/oder Unterstrichen. Ein Bezeichner der Klasse *InfixOpID* ist eine beliebige Folge von Zeichen aus der Zeichenkette “`~!@#$%^&*+-=<>?./|`”

Ebenfalls undefiniert bleiben in der kontextfreien Syntax die primitiven Basisausdrücke aus der Klasse *BasicExpr*. Dabei handelt es sich um Konstanten wie “`1`”, “`True`” oder den Buchstaben “`'a'`”. Sie sind wie in der Sprache Java definiert, die im Gegensatz zu Haskell den Unicode-Standard verwendet, um Zeichen und Zeichenketten darzustellen.

C.2 Kontextfreie Syntax

<table>
<tr><td>Block</td><td>::=</td><td>{ [FixityDeclaration$_1$; … FixityDeclaration$_n$ [;]]</td><td>$n \geqslant 0$</td></tr>
<tr><td></td><td></td><td>BlockDeclaration$_1$; … BlockDeclaration$_m$ [;] }</td><td>$m \geqslant 0$</td></tr>
<tr><td>BlockDeclaration</td><td>::=</td><td>DataDeclaration</td><td></td></tr>
<tr><td></td><td>|</td><td>FunctionDeclaration</td><td></td></tr>
<tr><td>DataDeclaration</td><td>::=</td><td><code>data</code> {TypeDeclaration$_1$; … TypeDeclaration$_n$ [;] }</td><td>$n > 0$</td></tr>
<tr><td>TypeDeclaration</td><td>::=</td><td>TypeConstrID TypeVarID$_1$ … TypeVarID$_n$</td><td>$n \geqslant 0$</td></tr>
<tr><td></td><td></td><td>= ConstrDeclaration$_1$ | … ConstrDeclaration$_m$</td><td>$m > 0$</td></tr>
<tr><td>TypeConstrID</td><td>::=</td><td>see lexicon</td><td></td></tr>
<tr><td>ConstrDeclaration</td><td>::=</td><td>DataConstrID TypeExpr$_1$ … TypeExpr$_n$</td><td>$n \geqslant 0$</td></tr>
<tr><td>DataConstrID</td><td>::=</td><td>see lexicon</td><td></td></tr>
<tr><td>TypeExpr</td><td>::=</td><td>TypeConstrID TypeExpr$_1$ … TypeExpr$_n$</td><td>$n \geqslant 0$</td></tr>
<tr><td></td><td>|</td><td>TypeVarID</td><td></td></tr>
<tr><td></td><td>|</td><td><code>()</code></td><td></td></tr>
<tr><td></td><td>|</td><td>(TypeExpr)</td><td></td></tr>
<tr><td></td><td>|</td><td>(TypeExpr$_1$, … TypeExpr$_n$)</td><td>$n > 1$</td></tr>
<tr><td></td><td>|</td><td><code>[</code> TypeExpr <code>]</code></td><td></td></tr>
<tr><td></td><td>|</td><td>TypeExpr <code>-></code> TypeExpr</td><td></td></tr>
<tr><td></td><td>|</td><td><code>Bool</code> | <code>Int</code> | <code>Char</code> | <code>String</code> | <code>Float</code></td><td></td></tr>
<tr><td>TypeVarID</td><td>::=</td><td>see lexicon</td><td></td></tr>
<tr><td>FixityDeclaration</td><td>::=</td><td>FixityKeyword {FixityArgument$_1$; … FixityArgument$_n$ [;] }</td><td>$n > 0$</td></tr>
<tr><td>FixityKeyword</td><td>::=</td><td><code>infixl</code> | <code>infixr</code> | <code>infix</code></td><td></td></tr>
<tr><td>FixityArgument</td><td>::=</td><td>Natural InfixOpID$_1$, … InfixOpID$_n$</td><td>$n > 0$</td></tr>
<tr><td>Natural</td><td>::=</td><td>Digit | Digit Natural</td><td></td></tr>
<tr><td>Digit</td><td>::=</td><td><code>0</code> | <code>1</code> | <code>2</code> | <code>3</code> | <code>4</code> | <code>5</code> | <code>6</code> | <code>7</code> | <code>8</code> | <code>9</code></td><td></td></tr>
<tr><td>InfixOpID</td><td>::=</td><td>see lexicon</td><td></td></tr>
<tr><td>FunctionDeclaration</td><td>::=</td><td>[Signature] [EvalAnnot] Equat$_1$ … Equat$_n$</td><td>$n > 0$</td></tr>
<tr><td>Signature</td><td>::=</td><td>FunctionName <code>::</code> FunctionType</td><td></td></tr>
<tr><td>FunctionName</td><td>::=</td><td>(InfixOpID)</td><td></td></tr>
<tr><td></td><td>|</td><td>FunctionID</td><td></td></tr>
<tr><td>FunctionID</td><td>::=</td><td>see lexicon</td><td></td></tr>
<tr><td>EvalAnnot</td><td>::=</td><td>FunctionName <code>eval</code> Annotation</td><td></td></tr>
<tr><td></td><td>|</td><td>FunctionName <code>eval</code> FlexRigidKeyword</td><td></td></tr>
<tr><td>Annotation</td><td>::=</td><td>Position : FlexRigid</td><td></td></tr>
<tr><td></td><td>|</td><td><code>rule</code></td><td></td></tr>
<tr><td></td><td>|</td><td>Annotation <code>or</code> Annotation</td><td></td></tr>
<tr><td>Position</td><td>::=</td><td>Natural</td><td></td></tr>
<tr><td></td><td>|</td><td>Natural . Position</td><td></td></tr>
<tr><td>FlexRigid</td><td>::=</td><td>FlexRigidKeyword [(ConsAnnot$_1$ … ConsAnnot$_n$)]</td><td>$n > 0$</td></tr>
<tr><td>FlexRigidKeyword</td><td>::=</td><td><code>flex</code> | <code>rigid</code></td><td></td></tr>
<tr><td>ConsAnnot</td><td>::=</td><td>ConstrID <code>=></code> Annotation</td><td></td></tr>
</table>

$$
\begin{array}{rcll}
\mathit{Equat} & ::= & \mathit{FunctionName}\ \mathit{Pattern}\ \texttt{=}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDefs}] & \\
 & \mid & \mathit{FunctionName}\ \mathit{Pattern}\ \mathit{CondExpr}\ [\texttt{where}\ \mathit{LocalDefs}] & \\
\mathit{Pattern} & ::= & \mathit{ConstrTerm}_1\ \ldots\ \mathit{ConstrTerm}_n & n \geqslant 0 \\
\mathit{ConstrTerm} & ::= & \mathit{VarID} \mid \_ & \\
 & \mid & \mathit{DataConstrID} & \\
 & \mid & \texttt{()} & \\
 & \mid & \texttt{(}\ \mathit{ConstrTerm}_1\ \texttt{,}\ \ldots\ \mathit{ConstrTerm}_n\ \texttt{)} & n > 1 \\
 & \mid & \texttt{(}\ \mathit{DataConstID}\ \mathit{ConstrTerm}_1\ \ldots\ \mathit{ConstrTerm}_n\ \texttt{)} & n > 0 \\
 & \mid & \texttt{(}\ \mathit{ConstrTerm}\ \texttt{:}\ \mathit{ConstrTerm}\ \texttt{)} & \\
 & \mid & \texttt{[}\ \mathit{ConstrTerm}_1\ \texttt{,}\ \ldots\ \mathit{ConstrTerm}_n\ \texttt{]} & n \geqslant 0 \\
\mathit{VarID} & ::= & \text{see lexicon} & \\
\mathit{LocalDefs} & ::= & \{\mathit{ValueDeclaration}_1\ \texttt{;}\ \ldots\ \mathit{ValueDeclaration}_n\ [\texttt{;}]\ \} & n > 0 \\
\mathit{ValueDeclaration} & ::= & \mathit{FunctionDeclaration} & \\
 & \mid & \mathit{PatternDeclaration} & \\
\mathit{PatternDeclaration} & ::= & \mathit{ConstrTerm}\ \texttt{=}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDefs}] & \\
\mathit{CondExpr} & ::= & \texttt{if}\ [\texttt{local}\ \texttt{[}\mathit{VarId}_1\ \texttt{,}\ \ldots\ \mathit{VarId}_k\ \texttt{]}\ \texttt{in}]\ \mathit{Constraint}\ \texttt{=}\ \mathit{Expr}\ [\mathit{CondExpr}] & \\
 & & & k > 0 \\
\mathit{Expr} & ::= & \texttt{\\}\ \mathit{Pattern}\ \texttt{->}\ \mathit{Expr} & \\
 & \mid & \texttt{let}\ \mathit{LocalDefs}\ \texttt{in}\ \mathit{Expr} & \\
 & \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} & \\
 & \mid & \mathit{Expr}\ \mathit{InfixOpID}\ \mathit{Expr} & \\
 & \mid & \mathit{Constraint} & \\
 & \mid & \mathit{ChoiceExpr} & \\
 & \mid & \mathit{FunctExpr} & \\
\mathit{Constraint} & ::= & \{\ [\texttt{local}\ \texttt{[}\mathit{VarId}_1\ \texttt{,}\ \ldots\ \mathit{VarId}_k\ \texttt{]}\ \texttt{in}]\ \mathit{ConstrExpr}_1\ \texttt{,}\ \ldots\ \mathit{ConstrExpr}_n\ \} & \\
 & & & k > 0,\ n \geqslant 0 \\
\mathit{ConstrExpr} & ::= & \mathit{Expr}\ \texttt{=}\ \mathit{Expr} & \\
 & \mid & \mathit{Expr} & \\
\mathit{ChoiceExpr} & ::= & \texttt{choice}\ \{\mathit{ChoiceBranch}_1\ \texttt{;}\ \ldots\ \mathit{ChoiceBranch}_n\ [\texttt{;}]\ \} & n \geqslant 0 \\
\mathit{ChoiceBranch} & ::= & [\texttt{local}\ \texttt{[}\mathit{VarId}_1\ \texttt{,}\ \ldots\ \mathit{VarId}_k\ \texttt{]}\ \texttt{in}]\ \mathit{Constraint}\ \texttt{->}\ \mathit{Expr} & k > 0 \\
\mathit{FunctExpr} & ::= & [\mathit{FunctExpr}]\ \mathit{BasicExpr} & \\
\mathit{BasicExpr} & ::= & \mathit{VarID} & \\
 & \mid & \texttt{()} & \\
 & \mid & \texttt{(}\ \mathit{Expr}\ \texttt{)} & \\
 & \mid & \texttt{(}\ \mathit{Expr}_1\ \texttt{,}\ \ldots\ \mathit{Expr}_n\ \texttt{)} & n > 1 \\
 & \mid & \texttt{[}\ \mathit{Expr}_1\ \texttt{,}\ \ldots\ \mathit{Expr}_n\ \texttt{]} & n \geqslant 0 \\
 & \mid & \mathit{Bool} \mid \mathit{Int} \mid \mathit{Char} \mid \mathit{String} \mid \mathit{Float} & \\
\mathit{Bool} & ::= & \text{see lexicon} & \\
\mathit{Int} & ::= & \text{see lexicon} & \\
\mathit{Char} & ::= & \text{see lexicon} & \\
\mathit{String} & ::= & \text{see lexicon} & \\
\mathit{Float} & ::= & \text{see lexicon} &
\end{array}
$$

In *CondExpr* darf statt eines Constraints auch ein boolescher Ausdruck b verwendet werden, der als Abkürzung für den Constraint $\{b\texttt{=True}\}$ vestanden wird.

Nicht alle Auswertungsvorschriften, die durch *EvalAnnot* generiert werden können, sind gültig, da die Struktur der Vorschriften mit der Struktur der Regeln kompatibel sein muß. Beispielsweise muß jede angegebene Position einer Position eines Konstruktors in einer linken Regelseite entsprechen, und Positionen müssen korrekt geschachtelt werden, d.h. in einer Vorschrift der Form $\mathcal{A}(\cdots\texttt{=>}p.i)$, muß die „Eltern"-Position p in $\mathcal{A}$ auftreten.

Literaturverzeichnis

[1] S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[3] P. Arenas-Sanchez, A. Gil-Luezas, and P. Lopez-Fraguas. Combining Lazy Narrowing with Disequality Constraints. Technical report dia 94/2, Universidad Complutense Madrid, 1994.

[4] F. Baader. *Skript zur Vorlesung „Termersetzungssysteme“*. Verlag der Augustinus Buchandlung, Aachen, 1995.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[6] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.

[7] J.C. Gonzales-Moreno, T. Hortala-Gonzalez, F. Lopez-Fraguas, and M. Rodriguez-Artalejo. A Rewriting Logic for Declarative Programming. In *Proc. ESOP'96*, pages 156–172. Springer LNCS 1058, 1996.

[8] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[9] M. Hanus. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.

[10] M. Hanus. On Extra Variables in (Equational) Logic Programming. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 665–679. MIT Press, 1995.

[11] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[12] M. Hanus. Teaching Functional and Logic Programming with a Single Computation Model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.

[13] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.

[14] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. In *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*, pages 138–152. Springer LNCS 1103, 1996.

[15] M. Hanus and R. Sadre. A Concurrent Implementation of Curry in Java. In *Proc. ILPS'97 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, Port Jefferson (New York), 1997.

[16] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1997. In dieser Arbeit wurde auf Version 6/97 zurückgegriffen.

[17] R. Hinze. *Einführung in die funktionale Programmierung mit Miranda.* Teubner, Stuttgart, 1992.

[18] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell (Version 1.2). *SIGPLAN Notices*, 27(5), 1992.

[19] S. Janson. *AKL – A Multiparadigm Programming Language.* PhD thesis, Swedish Institute of Computer Science, 1994.

[20] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. The Functional Logic Language BABEL and its Implementation on a Graph Machine. *New Generation Computing 14*, pages 391–427, 1996.

[21] J.W. Lloyd. Declarative Programming in Escher. Technical report cstr-95-013, University of Bristol, 1995.

[22] J.W. Lloyd. Programming in an Integrated Funktional and Logic Language. In *The Journal of Functional and Logic Programming*, 1997. To appear.

[23] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.

[24] R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pages 59–87, 1995.

[25] M. Mehl, R. Scheidhauer, and C. Schulte. An Abstract Machine for Oz. In *Proc. 7th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pages 151–168. Springer LNCS 982, 1995.

[26] Sun Microsystems. Java documentation. Available at `http://java.sun.com/docs`, 1997.

[27] G. Nadathur, B. Jayaraman, and K. Kwon. Scoping Constructs in Logic Programming: Implementation Problems and their Solution. *Journal of Logic Programming*, 25(2):119–161, 1995.

[28] Swedish Institute of Computer Science. *SISCtus Prolog User's Manual, Release 3 #3*, 1996.

[29] S.L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Proc. 20th Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[30] C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In *Proc. of the Fourteenth International Conference on Logic Programming*, pages 286–300. The MIT Press, 1997.

[31] C. Schulte. Programming Constraint Inference Engines. In *Proc. of the Third International Conference on Principles and Practice of Constraint Programming*, pages 519–533. Springer-Verlag, 1997. To appear.

[32] C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[33] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.

[34] P. Wadler. How to Replace Failure by a List of Successes. In *Functional Programming and Computer Architecture*. Springer LNCS 201, 1985.