Nebenläufige Implementierung der Programmiersprache Curry in Java

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

ausgearbeitet von

Ramin Sadre
Matrikelnummer 191 388

Betreuer: Prof. Dr. Michael Hanus

Aachen, November 1997

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 17. November 1997

(Ramin Sadre)

Ich bedanke mich bei Professor Hanus für die Betreuung, die zahlreichen Vorschläge und Anregungen, bei meinen Eltern für das Korrekturlesen der Arbeit und bei Stefan M., weil er mich auch genannt hat. Mein besonderer Dank gilt Markus Mohnen, von dem die Grundidee der Bindungstabellen stammt und der das Java-System am Hochschulrechner installiert hat.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Welt der deklarativen Programmiersprachen läßt sich grob mit Hilfe von zwei verschiedenen Ideen beschreiben: funktionale Programmiersprachen fassen ein Programm als eine Menge von Funktionen auf, während ein Programm in einer logischen Programmiersprache eine Menge von Aussagen über das zu lösende Problem darstellt. Beide Ansätze haben Vor- und Nachteile, so daß schon seit Jahren mit *funktional-logischen* Programmiersprachen experimentiert wird.

Funktionale Programmiersprachen bieten den Umgang mit Funktionen höherer Ordnung und partiellen Funktionsapplikationen. Der Hauptvorteil ist natürlich, daß der Programmierer mit Funktionen und geschachtelten Ausdrücken arbeiten kann — das Simulieren von Funktionsaufrufen, wie z.B. in Prolog [14] in Form von `p(Arg1,...,ArgN,Result)` mit anschließendem *Cut*, bleibt ihm erspart. Außerdem bieten viele funktionale Programmiersprachen die sogenannte verzögerte Auswertung, die die Verwendung von unendlichen Datenstrukturen ermöglicht [11]. Dies alles führt dazu, daß sogenannte reine funktionale Programmiersprachen seiteneffektfrei sind, selbst bei Dingen wie Ein- und Ausgabe (Monadic IO [17]).

Andererseits bieten logische Programmiersprachen das wichtige Konzept der logischen Variablen. Diese lassen sich nichtdeterministisch binden und bieten die Möglichkeit der Darstellung von partiellen Datenstrukturen. Damit gestaltet sich der Programmablauf zu einer (nichtdeterministischen) Suche nach einer Lösung.

Eine funktional-logische Programmiersprache hat also die Aufgabe Funktionen und logische Variablen zu vereinen. Auf der Suche nach einer geeigneten operationalen Semantik wurden in der Vergangenheit zwei Ansätze verfolgt:

Residuation, wie es zum Beispiel in Oz [18] eingesetzt wurde, behält weiterhin die Idee der deterministischen Auswertung von Funktionen bei: Funktionsaufrufe mit logischen Variablen suspendieren, bis diese Variablen gebunden werden. Auf diese Weise lassen sich nebenläufige Auswertungen durch Variablenbindungen synchronisieren, allerdings muß nichtdeterministische Suche und das Binden von Variablen explizit durch spezielle Operationen durchgeführt werden (z.B. durch einen $\vee$-Operator). Das führt natürlich dazu, daß Ausdrücke, in denen logische Variablen übrigbleiben, nicht ausgewertet werden können: Residuation ist in diesem Sinne unvollständig.

Narrowing lehnt am Ausführungsmechanismus von Prolog an: logische Variablen werden bei einem Funktionsaufruf gebunden, d.h. ein Aufruf der Form `p(X)` mit

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

bindet `X` an `a` und `b`. Andererseits ist diese Vorgehensweise bei manchen Funktionen (wie z.B. Arithmetik) nicht erwünscht.

Mit der Programmiersprache Curry [10] wurde daher der (erfolgreiche) Versuch unternommen beide Ansätze in einem operationalen Modell zu vereinen: der Benutzer kann angeben, welche Funktionsargumente von einer Funktion benötigt werden (Residuation) und welche Argumente Nichtdeterminismus auslösen, wenn die Funktion mit logischen Variablen aufgerufen wird (Narrowing). Damit bietet Curry als funktional-logische Sprache alle Vorteile beider Welten: es besitzt ein polymorphes Typsystem und bietet den monadischen Ansatz zur Ein- und Augabe. Andererseits sind ebenso nichtdeterministische Programmabläufe und die sogenannte eingekapselte Suche [19] möglich.

Die erste Implementierung von Curry wurde in Prolog vorgenommen und nennt sich Taste-Curry [15]. Diese Implementierung hat vor allem das Ziel, die Eigenschaften von Curry zu demonstrieren und weiterhin werden Erweiterungen der Sprache Curry zuerst an diesem System vorgenommen. TasteCurry verwendet intern die operationale Semantik aus [9]: ein Curry-Programm wird als Termersetzungssystem aufgefaßt, mit dessen Hilfe Ausdrücke auf Lösungen reduziert werden. Kapitel 2 gibt eine Einführung in Curry und beschreibt diese operationale Semantik.

Dieser Ansatz hat einen großen Nachteil: dem System fehlen Informationen über die Art und Position des letzten ausgeführten Auswertungsschritts. Bei jedem Auswertungsschritt muß erneut nach einer geeigneten Position im auszuwertenden Ausdruck gesucht werden. Außerdem führt das TasteCurry-System kein term sharing durch, was zur Folge hat, daß unter Umständen die gleiche Funktion mehrmals ausgewertet wird. Gerade Funktionen ohne Nichtdeterminismus (und um solche handelt es sich meistens) würden aber von einer „geradlinigeren" Ausführung profitieren. Ein weiterer wichtiger Punkt ist die Behandlung von Konjunktionen: [9] beschreibt ausführlich unter welchen Umständen welche Teile der Konjunktion bearbeitet werden können. Auch diese Informationen werden vom TasteCurry-System nicht ausgenutzt.

Das Ziel dieser hier vorgestellten Diplomarbeit war es daher, eine effizientere Implementierung eines Curry-Systems zu realisieren. Als Implementierungssprache sollte Java verwendet werden, ich habe mich jedoch für einen Java-Dialekt namens Pizza entschieden: wie wir in Kapitel 3 noch sehen werden, bleiben Pizza-Programme auf gewisse Weise hundertprozentig kompatibel zu Java-Programmen. Als besondere Herausforderung sollte die in Curry vorhandene implizite Nebenläufigkeit (in Form von Nichtdeterminismus und Konjunktion) mit Hilfe von Java-Threads realisiert werden. Wie diese Implementierung nun aussieht und welche Maßnahmen zur Verwaltung der Nebenläufigkeit getroffen wurden, wird in Kapitel 4, dem eigentlichen Kern dieser Ausarbeitung, beschrieben.

Vorher wird allerdings in Kapitel 3 eine kurze (und unvollständige) Einführung in Java und Pizza gegeben. Java ist eine relativ junge Sprache und bietet viele überraschende Eigenschaften, die ausgiebig vom Curry-System ausgenutzt werden. Daher ist dieses Kapitel wohl auch für die, die Java schon kennen, interessant.

Abschließend versuchen Kapitel 5 und 6 zu analysieren, inwieweit die Erwartungen in diese Implementierung, hauptsächlich der Wunsch nach einer deutlich schnelleren Abarbeitung von Curry-Programmen, erfüllt werden konnten.

Auf der beiliegenden Diskette befindet sich sowohl der Quellcode als auch das ausführbare Curry-System. Im Anhang wird seine Benutzung beschrieben und einige Erläuterungen zum Quellcode gegeben.

Kapitel 2

Curry

Curry ist eine deklarative Sprache, die mit dem Ziel entwickelt wurde, die Eigenschaften funktionaler, logischer und nebenläufiger Programmierung zu vereinen. Curry bietet die Vorteile funktionaler Programmiersprachen (Programmieren mit Funktionen, Funktionen höherer Ordnung, verzögerte Auswertung), logischer Programmiersprachen (logische Variablen, Nichtdeterminismus, eingebaute Suche) und nebenläufiger Programmierung (Und-parallele Auswertung mit Synchronisation auf Variablenbindungen).

In diesem Kapitel sollen die Fähigkeiten der Programmiersprache Curry besprochen werden: zuerst anhand des Sprachumfangs und dann anhand einer operationalen Semantik. Dieses Kapitel stellt daher das Pflichtenheft für die nachfolgende Implementierung dar [1]. Da die Implementierung als back-end für das bestehende TasteCurry-System [15] konzipiert wurde (siehe auch Abschnitt 4.1), werden im folgenden einige Aspekte von Curry, insbesondere die, die durch den Parser des TasteCurry-Systems behandelt werden, nicht oder nur teilweise besprochen, so z.B. das Modulsystem oder die `where`-Deklaration. Aus diesem Grund sei hier auf die Originaldokumentation [10] verwiesen.

Für die folgenden Beispielprogramme wurde die Prolog-ähnliche Syntax des TasteCurry-Systems gewählt, d.h. Variablennamen werden durch Großschreibung von den Konstruktor- und Funktionsnamen unterschieden und alle Ausdrücke (bis auf Infixoperatoren) werden geklammert.

2.1 Die Programmiersprache Curry

Ein Curry-Programm besteht aus einer Folge von Datentypdeklarationen, Funktionsdeklarationen und -definitionen, und sogenannten *evaluation annotations* (siehe unten). Curry ist eine streng getypte Sprache mit einem parametrisch polymorphen Hindley/Milner-Typsystem [4]. Das bedeutet (unter anderem), daß durch Typinferenz jedem Objekt in einem Curry-Programm ein eindeutiger Typ zugeordnet werden kann [2].

[1] daher besitzen dieses Kapitel und das Kapitel über die Implementierung einen ähnlichen Aufbau.

[2] das wird in der Implementierung in Abschnitt 4.2.1 ausgenutzt.

2.1.1 Datentypdeklarationen

Neue Datentypen lassen sich mit Hilfe des Schlüsselwortes `data` deklarieren:

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

Der dadurch deklarierte n-stellige Typkonstruktor t enthält k Konstruktoren $c_1,\ldots,c_k$. Jedem Konstruktor c_i wird der Typ

$$\tau_{i1} \to \ldots \to \tau_{in_i} \to \tau(A_1,\ldots,A_n)$$

zugeordnet, wobei n als die *Stelligkeit* des Konstruktors bezeichnet wird.

Als Beispiel die Deklaration eines Listentyps:

```
datatype list(A) = [] ; cons(A,list(A)).
```

Der Typkonstruktor `List` mit der Typvariablen `A` enthält damit zwei Datenkonstruktoren:

- den 0-stelligen Konstruktor `[]` (d.h. eine Konstante) mit dem Typ `list(A)`.
- den zweistelligen Konstruktor `cons` mit dem Typ `A` $\to$ `list(A)` $\to$ `list(A)`.

Für die Typvariable `A` läßt sich jeder beliebige Typ einsetzen (Polymorphismus).

2.1.2 Datenterme und Ausdrücke

Definition 1 (Datenterm) Ein *Datenterm* ist eine Variable X oder eine Konstruktorapplikation $c(t_1,\ldots,t_n)$ des n-stelligen Konstruktors c auf die Datenterme $t_1,\ldots,t_n$. Da jeder Konstruktor einem Datentyp angehört, läßt sich jedem Datenterm (abgesehen von einzelnen Variablen) ein eindeutiger, nicht funktionaler (d.h. ohne '$\to$') Typ zuordnen: hat c den Typ $\tau_1 \to \ldots \to \tau_n \to \tau(A_1,\ldots,A_n)$ und t_i den Typ τ_i für $i = 1,\ldots,n$, so hat die Konstruktorapplikation den Typ $t(A_1,\ldots,A_n)$, wobei $A_1,\ldots,A_n$ von den t_i abhängig sind.

Definition 2 (Ausdruck und Kopfnormalform) Ein *Ausdruck* ist eine Variable oder eine (partielle) Applikation $\varphi(e_1,\ldots,e_m)$ eines Konstruktors oder einer Funktion φ, wobei die Argumente e_i ihrerseits Ausdrücke sind. Da sich Funktionen (siehe unten) genauso wie Konstruktoren ein Typ und eine Stelligkeit n zuordnen läßt ($m \leq n$), ist der Typ eines Ausdrucks wiederum eindeutig.

Ein Ausdruck befindet sich in *Kopfnormalform*, wenn er eine Variable oder eine Konstruktorapplikation auf Ausdrücke ist.

2.1.3 Funktionsdeklarationen

Eine Funktionsdeklaration für eine Funktion f besteht aus einer (optionalen) Typdeklaration

$$f\ ::\ \tau_1\ \texttt{->}\ \ldots\ \texttt{->}\ \tau_n\ \texttt{->}\ \tau.$$

und einer Folge von *Gleichungen* (auch *Regeln* genannt) der Form

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

wobei t_i Datenterme vom Typ τ_i sind und e ein Ausdruck vom Typ τ ist.

Zusätzlich gibt es noch *bedingte Gleichungen* der Form

$$f(t_1, \ldots, t_n) \texttt{ if } \{c\} = e.$$

wobei die Bedingung c ein Constraint ist (siehe Abschnitt 2.1.8). Um eine bedingte Gleichung anwenden zu können, muß die Bedingung c erfüllbar sein.

Definition 3 (Argument- und logische Variablen) Variablen, die in den t_i vorkommen, werden als *Argumentvariablen* bezeichnet, während Variablen, die nur in e und c vorkommen, *logische Variablen* genannt werden. Da erst die Verwendung von logischen Variablen Nichtdeterminismus auslöst, ist diese Unterscheidung für die Implementierung wichtig.

Für die Definition von Gleichungen gibt es zwei grundlegende Einschränkungen:

1. eine Argumentvariable kommt innerhalb der linken Seite einer Gleichung nur *einmal* vor.

2. die rechten Seiten der Gleichungen einer Funktion haben alle den gleichen Typ.

Dank der zweiten Bedingung läßt sich auch einer Funktion f ein Typ und eine Stelligkeit zuordnen.

2.1.4 Funktionale Sichtweise

Curry verhält sich wie eine funktionale Programmiersprache, wenn Ausdrücke keine Variablen enthalten. Das bedeutet insbesondere, daß in Funktionsgleichungen keine logischen Variablen vorkommen. Das Curry-System hat dann die Aufgabe, vom Benutzer eingegebene Ausdrücke auszuwerten, d.h. zu einem Ausdruck einen *Wert* zu berechnen (ein Wert ist ein Audruck ohne Funktionssymbole).

Definition 4 (Substitution und unifizierbar) Sei σ eine Abbildung von Variablen auf Ausdrücke. Diese Abbildung kann zu einem Homomorphismus auf Ausdrücken erweitert werden, indem man definiert, daß $\sigma(\varphi(t_1, \ldots, t_n)) = \varphi(\sigma(t_1), \ldots, \sigma(t_n))$. Solche eine Abbildung heißt *Substitution*.
Zwei Terme s und t sind *unifizierbar*, falls eine Substitution σ existiert mit $\sigma(s) = \sigma(t)$.

Die Berechnung des Wertes kann durch sogenannte *Berechnungsschritte* dargestellt werden. Jeder Berechnungsschritt stellt eine Umformung des Originalausdruckes dar mit dem Ziel, Funktionssymbole zu eliminieren. Sei dazu $f(e_1, \ldots, e_m)$ eine Funktionsapplikation (im folgenden auch *Funktionsaufruf* genannt), $l = r$ eine Gleichung der Funktion f und sei σ eine Substitution mit $\sigma(l) = f(e_1, \ldots, e_m)$. Dann kann der Ausdruck mit Hilfe dieser Gleichung *reduziert* werden zu $\sigma(r)$ (siehe Abschnitt 2.2 für eine formalere Definition). Die Suche nach einer geeigneten Gleichung wird *Pattern Matching* genannt.

Für den Fall, daß kein Berechnungsschritt durchgeführt werden kann, obwohl der Ausdruck weiterhin Funktionsaufrufe enthält, schlägt die Berechnung fehl und terminiert ohne Ergebnis („keine passende Gleichung").

Schauen wir uns dazu das Beispiel 2.1 an. Mit Hilfe dieser Gleichungen kann der Ausdruck `add(s(s(z)),s(z)` reduziert werden zu `s(s(s(z)))`:

$$\texttt{add(s(s(z)),s(z))} \rightarrow \texttt{s(add(s(z),s(z)))} \rightarrow \texttt{s(s(add(z,s(z))))} \rightarrow \texttt{s(s(s(z)))}$$

```
data bool = true ; false.
data Nat  = z ; s(nat).

add(z,    Y) = Y.
add(s(X)),Y) = s(add(X,Y)).

leq(z,X)       = true.
leq(s(X),z)    = false.
leq(s(X),s(Y)) = leq(X,Y).

f(X) = f(X).
```

Beispiel 2.1: Curry-Beispielprogramm

Enthält ein Ausdruck mehrere Funktionsaufrufe, wählt Curry den äußersten für den nächsten Berechnungsschritt. Dies führt zu einer *verzögerten* Auswertung von Funktionen (*lazy evaluation*) — unter Umständen braucht eine Funktion überhaupt nicht ausgewertet zu werden, wie z.B. die Funktion `add` im Ausdruck `leq(z,add(z,z))`.

Für den Fall, daß mehrere Funktionsaufrufe für einen Reduktionsschritt in Betracht kommen, ist es wichtig zu wissen, in welcher Reihenfolge diese abgehandelt werden. So würde der Ausdruck `leq(add(z,z),f(z))` zu einer unendlichen Reduktion führen, falls das zweite Argument von `leq` *vor* dem ersten Argument ausgewertet würde.

Curry wählt standardmäßig den am weitesten links stehenden, äußersten Funktionsaufruf für einen Reduktionsschritt aus, was zu der, aus anderen funktionalen Programmiersprachen, wie Haskell [17], bekannten, Pattern-Matching-Strategie führt. Damit wird der Ausdruck `leq(add(z,z),f(z))` ausgewertet zu `true`. Allerdings kann mit Hilfe der *evaluation annotations* (siehe unten) Curry angewiesen werden, von dieser Auswertungsreihenfolge abzuweichen.

2.1.5 Nichtdeterminismus und logische Variable

Definition 5 (Lösung) Die Verwendung von logischen Variablen erweitert die oben erläuterte funktionale Sichtweise um den Aspekt des Nichtdeterminismus. Treten im auszuwertenden Ausdruck im Laufe der Berechnungsschritte (freie) logische Variablen auf, so können diese durch Funktionsaufrufe *nichtdeterministisch* gebunden werden. Die Auswertung eines Ausdruckes kann also nun zu einer Menge von (disjunktiven) *Lösungen* führen. Dabei besteht eine Lösung '$\sigma \mid e$' — ähnlich wie in logischen Sprachen — aus einer Substitution σ und einem Ausdruck e. Ein Fehlschlag ist durch eine leere Lösungsmenge gekennzeichnet.

Wird eine Funktion mit einer logischen Variablen als Argument aufgerufen, so gibt es zwei Möglichkeiten:

1. der Aufruf suspendiert, d.h. kann erst ausgewertet werden, wenn die logische Variable gebunden wird. Diese Vorgehensweise entspricht dem *residuation*-Prinzip aus Sprachen wie Oz [18].
   Beispiel: arithmetische Funktionen.

2. der Ausdruck wird reduziert, indem mehrere Lösungen erzeugt werden — für jede anwendbare Gleichung eine. Das führt zur einer nichtdeterministischen Bindung der logischen Variablen. Dies entspricht dem *narrowing*-Prinzip aus Sprachen wie ALF [8].

Flex und Rigid

Um für eine Funktion eine der beiden Vorgehensweisen vorzuschreiben, existieren die *evaluation annotations*. Mit den evalutation annotations kann angegeben werden,

- in welcher Reihenfolge die Argumente einer Funktion ausgewertet werden sollen, falls sie sich nicht in Kopfnormalform befinden.
- ob ein Argument verlangt wird, d.h. ob eine (freie) logische Variable als Argument zu einer Suspendierung oder zu Nichtdeterminismus führt.

Die Syntax der evaluation annotations soll anhand der `leq`-Funktion aus Beispiel 2.1 gezeigt werden. Besitzt `leq` die evaluation annotation

```
leq eval 1:flex
```

(die Positionsangabe '1' bezeichnet das erste Argument), so führt eine Reduktion des Ausdrucks `leq(X,z)` zu den disjunktiven Lösungen

`X=z | true` und `X=s(Y) | false`

(`Y` ist eine neue freie Variable).

Hat die Funktion dagegen die annotation

```
leq eval 1:rigid
```

suspendiert die Auswertung des gleichen Ausdrucks.

Das bestehende TasteCurry-System ist in der Lage unvollständige oder fehlende evaluation annotations zu vervollständigen. Die Art und Weise, wie dies geschieht, kann durch Schlüsselwörter beeinflußt werden. Hat man zum Beispiel das System angewiesen, alle Argumentpositionen als `rigid` aufzufassen, wird die eigentlich unvollständige annotation `leq eval 1:rigid` um Anweisungen für das zweite Argument erweitert. Falls das erste Argument die Form `s(...)` hat, muß auch das zweite Argument in Kopfnormalform sein:

```
leq eval 1:rigid(s => 2:rigid)
```

Man beachte die Tatsache, daß erst die Positionsangaben (1 und 2) Curry anweisen, die Argumente von links nach rechts auszuwerten. Normalerweise wird der Benutzer jedoch gar nicht auf diese Darstellung der evaluation annotations zurückgreifen (wollen). Die Voreinstellung (Patternmatching von links nach rechts) und die oben erwähnten Compilerpragmas decken die meisten Fälle ab, z.B. läßt sich die rigid-Funktion `leq` auch so beschreiben:

```
leq eval rigid
```

Überlappende linke Regelseiten

Die `leq`- und die `add`-Funktion aus dem vorherigen Abschnitt haben eines gemeinsam: es läßt sich eine Reihenfolge für die Auswertung der Argumente angeben, die dazu führt, daß genau eine Gleichung ausgewählt werden kann. Dies ist bei Gleichungen mit überlappenden linken Seiten, wie z.B. beim „parallelen Oder“, nicht der Fall:

```
 true por X     = true.
    X por true  = true.
false por false = false.
```

Hat man einen Ausdruck `f por g`, in dem die Argumente `f` und `g` beide nicht in Kopfnormalform vorliegen, ist nicht klar, welches Argument zuerst ausgewertet werden soll. In solchen Fällen wertet Curry *nichtdeterministisch* das erste oder das zweite Argument aus. Diese Art von Nichtdeterminismus wird (genauso wie der Nichtdeterminismus durch Verwendung von logischen Variablen) „don't know“-Nichtdeterminismus genannt (d.h. in Ermangelung von Informationen werden beide Möglichkeiten in Betracht gezogen).

Curry stellt überlappende linke Gleichungsseiten durch das Schlüsselwort `or` in der zugehörigen evaluation annotation dar. Im `por`-Beispiel lassen sich die erste und dritte Gleichung (beide benötigen das erste Argument) mit Hilfe von `or` von der zweiten Gleichung trennen:

```
por eval 1:rigid(false => 2:rigid) or 2:rigid
```

Alternativ wäre es auch möglich, die zweite und die dritte Gleichung zusammenzufassen, was zu

```
por eval 1:rigid or 2:rigid(false => 1:rigid)
```

führen würde.

2.1.6 Funktionen höherer Ordnung

Wie schon erwähnt unterstützt Curry die Verwendung Funktionen höherer Ordnung: Funktionen lassen sich wie Konstruktoren als Argumente an andere Funktionen weitergeben und applizieren. Da es der Parser des TasteCurry-Systems nicht erlaubt, Variablen in der in funktionalen Sprachen üblichen Schreibweise auf Argumente zu applizieren, existiert in TasteCurry der sogenannte `Apply`-Operator `@`. Das Standardbeispiel für Funktionen höherer Ordnung, die `map`-Funktion, zeigt seine Anwendung:

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

Da Curry auch eine logische Sprache ist, wäre es theoretisch möglich, Anfragen der Form `map(F,cons(1,cons(2,[]))) = cons(2,cons(3,[]))` zu stellen und als Lösung eine Bindung für `F` zu erwarten, z.B. `F=inc`. Da die Berechnung solcher Lösungen sehr aufwendig ist, ist der Apply-Operator „rigid“ auf sein erstes Argument definiert, d.h. suspendiert, falls eine (freie) logische Variable appliziert werden soll.

```
data bool = true ; false.

(&&) :: bool->bool->bool.
true  && X = X.
false && X = false.

== :: A->B->bool.
```

und

c==c = true   für alle 0-stelligen Konstruktoren c
c(x_1,...,x_n)==c(y_1,...,y_n) = x_1==y_1 && .. && x_n==y_n
                                                      für alle n-stelligen Konstruktoren c
c(x_1,...,x_n)==d(y_1,...,y_m) = false
                                                  für verschiedene Konstruktoren c und d

Abbildung 2.1: die testende Gleichheit

2.1.7 Die testende Gleichheit

Um Terme auf Gleichheit zu prüfen, bietet Curry die Gleichheit `==`. Bei dieser Funktion handelt es sich nur um einen *Test*: ein Aufruf der Form `X==t`, in der `X` eine freie Variable ist, führt nicht zu einer Bindung der Variablen, vielmehr suspendiert der Aufruf. Die Funktion `==` ist also „rigid" auf beide Argumente. Die Definition der testenden Gleichheit ist in Abbildung 2.1 zu sehen.

2.1.8 Constraints

Bis jetzt wurde die Anwendung einer Gleichung in einem Berechnungsschritt durch ihre linke Seite vorgegeben: war sie unifizierbar mit dem auszuwertenden Ausdruck, konnte die Gleichung angewendet werden. Wie in Abschnitt 2.1.3 schon erwähnt, kann durch sogenannte *bedingte Gleichungen* der Form

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

die Anwendbarkeit einer Gleichung weiter eingeschränkt werden: eine Gleichung ist nur dann verwendbar, wenn die Bedingung c erfüllbar ist. Dabei liegt c in Form eines *Constraints* vor: ist ein Constraint *erfüllbar*, d.h. seine Auswertung erfolgreich, so kann die Gleichung angewendet werden — im anderen Fall schlägt die Berechnung fehl. Im Gegensatz zu booleschen Funktionen wie `==` (diese liefern den Wert `true` oder `false`) schränken Constraints also tatsächlich den Lösungsraum ein. Eine besondere Bedeutung hat der leere Constraint {}: er ist immer erfüllbar und stellt damit eine erfolgreiche Berechnung dar.

Die lösende Gleichheit

Der fundamentalste Constraint ist die Gleichheit `=`. Ihre Definition steht in Abbildung 2.2. Wie man sieht, gibt es keine Gleichung für den Fall, daß die Argumente verschiedene Konstruktoren sind: dieser Fall löst also einen Fehlschlag der Auswertung aus. Außerdem ist die lösende

```
data constraint = {}.

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

c=c = true                                  für alle 0-stelligen Konstruktoren c
c(x1,...,xn)=c(y1,...,yn) = x1=y1 ∧ .. ∧ xn=yn
                                            für alle n-stelligen Konstruktoren c
```

Abbildung 2.2: die lösende Gleichheit

Gleichheit „flex" auf beide Argumente, d.h. logische Variablen werden durch `=` gebunden. Der in der Definition verwendete $\wedge$-Operator wird im nächsten Abschnitt beschrieben; es ist aber klar, daß man für Constraints nicht die boolesche `&&`-Funktion aus Abbildung 2.1 verwenden kann. Weitere Erläuterungen zur Funktionsweise der Gleichheit werden im Kapitel zur Implementierung gegeben.

Hier nochmal die testende und die lösende Gleichheit im Vergleich (seien `c` und `d` verschiedene Konstruktoren):

1. `c=c` ist erfolgreich, `c==c` liefert `true`
2. `c=d` schlägt fehl, `c==d` liefert `false`
3. `X=c` bindet `X` an `c`, `X==c` suspendiert

Während also `==` Gleichheit *testet*, versucht `=` die Gleichheit zu *erfüllen*.

Und-Verknüpfung von Constraints

Mit dem $\wedge$-Operator lassen sich Constraints zu einem neuen Constraint, einer Konjunktion, verknüpfen. Der Constraint $c_1 \wedge c_2$ [3] ist nur erfüllbar, wenn c_1 *und* c_2 erfüllbar sind, d.h. wenn beide Constraints erfolgreich ausgewertet werden können.

Im Gegensatz zu dem booleschen (sequentiellen) Und-Operator `&&` findet die Auswertung der Argumente *parallel* statt. Erst diese parallele Auswertung ermöglicht es, suspendierte Funktionsaufrufe zu reaktivieren: ein Constraint kann durch die Bindung einer Variablen einen parallel ausgewerteten und suspendierten Constraint „wiedererwecken", vorausgesetzt der suspendierte Constraint wartet auf die Bindung genau dieser Variablen.

Damit lassen sich Auswertungsstrukturen der Form `test and generate` aufbauen. Der Generator kann durch eine Funktion dargestellt werden, die ihr Argument an potentielle Lösungen bindet (also „flex" ist):

```
generate eval 1:flex.

generate(1)={}.
generate(2)={}.
generate(3)={}.
```

[3] die Syntax von TasteCurry erlaubt es, $c_1 \wedge c_2$ als $\{c_1, c_2\}$ einzugeben.

Die Testfunktion überprüft, ob ihr Argument eine korrekte Lösung ist — hier ein einfacher Test:

```
test eval 1:rigid.

test(2)={}.
```

Diese beiden Funktionen lassen sich nun mit dem $\wedge$-Operator parallel ausführen. Als Kanal für den Datenaustausch dient eine logische Variable:

`test(X)` $\wedge$ `generate(X)`

oder anders geschrieben:

{ `test(X) , generate(X)` }

Da die Testfunktion „rigid" ist, wird sie, angewendet auf die logische Variable `X`, suspendieren und erst wieder ausgeführt, wenn die Generatorfunktion `generate` die Variable `X` (in diesem Fall nichtdeterministisch an 1,2 und 3) gebunden hat. Die Variable `X` synchronisiert also die beiden Funktionsaufrufe — unabhängig von deren Ausführungsgeschwindigkeit oder -reihenfolge ist das Ergebnis `X=2 |` {}.

2.1.9 Der Choice-Operator

Eine andere Form von Nichtdeterminismus läßt sich durch den Choice-Operator realisieren. Eine Anwendung des Operators hat die Form

```
choice Constraint1 -> Ausdruck1 ;
       ...
       Constraintn -> Ausdruckn
```

Zur Auswertung dieses Konstrukts wird der Ausdruck $Ausdruck_i$ ausgewertet, dessen Bedingung $Constraint_i$ erfüllbar ist. Sind mehrere Constraints erfüllbar, so werden nicht — wie im Abschnitt 2.1.5 — alle Ausdrücke ausgewertet (*don't know-Nichtdeterminismus*), sondern es wird nichdeterministisch *nur einer* für die weitere Auswertung ausgewählt (*don't care-Nichtdeterminismus*).

Die Auswertung der Constraints geschieht als *eingekapselte Suche* [19]. Dies bedeutet:

1. ein Fehlschlag eines Constraints bewirkt nur, daß dieser aus der Wahl ausgeschlossen wird (statt die ganze Berechnung fehlschlagen zu lassen).

2. erzeugt ein Constraint Nichtdeterminismus, so wird nur *seine* Berechnung vervielfältigt (statt die ganze Berechnung zu vervielfältigen).

3. lokale Variablen sind außerhalb des choice-Ausdrucks nicht sichtbar.

4. globale Variablen, d.h. Variablen, die außerhalb des choice-Ausdrucks erzeugt wurden, sind *read-only*.

```
merge :: list(A) -> list(A) -> list(A).

merge(L1,L2)
   = choice
       {L1=[]}  -> L2;
       {L2=[]}  -> L1;
       {L1=cons(H,T)} -> cons(H,merge(T,L2));
       {L2=cons(H,T)} -> cons(H,merge(L1,T)).
```

Beispiel 2.2: Nichtdeterministisches „merge“

Das hat also zur Folge, daß ein Constraint eine Variable nur dann binden darf, wenn sie nur im Constraint selber und im zugehörigen Ausdruck vorkommt. Das Binden von Variablen, die von außen in den Constraint gereicht werden, führt zu einer Suspendierung des Constraints. Im folgenden Beispiel

```
f(X) = choice test1(X,Y) -> g(Y) ;
              test2(X)   -> ...
```

darf also `test1` zwar `Y` binden, aber nicht logische Variablen, die im Argumentausdruck `X` vorkommen.

Das Beispiel 2.2 zeigt eine Anwendung: die Funktion `merge` faßt seine beiden Argumenten (zwei Listen) zu einer Liste zusammen. Da hier der `choice`-Operator verwendet wurde, wird nichtdeterministisch das erste Element der ersten oder der zweiten Liste in die Ergebnisliste geschoben. Mit Hilfe dieser Funktion kann z.B. eine mehrseitige (2:1) Kommunikation realisiert werden:

$$\texttt{produce1(L1)} \wedge \texttt{produce2(L2)} \wedge \texttt{consume(merge(L1,L2))}$$

Die beiden Aufrufe `produce1` und `produce2` erzeugen Kommunikation, indem sie das jeweilige Argument (`L1` oder `L2`) binden. Diese kann dann von der Funktion `consume` empfangen werden.

2.2 Eine operationale Semantik

In diesem Abschnitt wird eine operationale Semantik für Curry beschrieben. Sie wurde in [9] vorgestellt und ist in der Lage, die im vorigen Abschnitt beschriebenen Mechanismen (*Residuation*, *Narrowing*,...) in einem Berechnungsmodell zu vereinen.

Das Berechnungsmodell ordnet jeder Funktion eine Datenstruktur, den sogenannten *Definitional Tree* [2], zu. Diese Struktur ist eine Verschmelzung der evaluation annotation mit den Gleichungen einer Funktion und enthält damit alle Informationen, die für die Bearbeitung eines Funktionsaufrufs nötig sind. Das TasteCurry-System erzeugt die Definitional Trees aus einem Curry-Programm und benutzt sie für die weitere Ausführung.

Die operationale Semantik faßt die Funktionsgleichungen eines Curry-Programms als Termersetzungssystem auf. Deshalb wird im folgenden nicht mehr von Ausdrücken sondern von *Termen* gesprochen.

2.2.1 Definitionen

Definition 6 (Position und Subterm) Eine *Position* p in einem Term t wird durch eine Folge von natürlichen Zahlen dargestellt. $t|_p$ bezeichnet den Subterm von t an der Position p. $t\,[s]_p$ bezeichnet den Term, der aus t hervorgeht, wenn man den Subterm $t|_p$ durch s ersetzt.

Definition 7 (Variante) Eine Regel $l \rightarrow r$ ist Variante einer anderen Regel, wenn sie aus ihr durch Variablenumbenennung hervorgeht.

Definition 8 (Reduktionsschritt, irreduzibel) Jeder Funktionsgleichung $l = r$ wird eine Regel $l \rightarrow r$ in einem Termersetzungssystem $\mathcal{R}$ zugeordnet. Ein *Reduktionsschritt* ist die Anwendung einer Regel $l \rightarrow_{\mathcal{R}} r$ auf einen Ausdruck e: es gilt $t \rightarrow_{\mathcal{R}} s$, wenn eine Position p in t, eine Regel $l \rightarrow r$ und eine Substitution σ existieren mit $t|_p = \sigma(l)$ und $s = t\,[\sigma(r)]_p$. Falls kein Term s mit $t \rightarrow_{\mathcal{R}} s$ existiert, heißt der Term t *irreduzibel bezüglich* $\mathcal{R}$ [6].

Bedingte Gleichungen der Form „l `if` $\{c\} = r$“ lassen sich einfach zu unbedingten Gleichungen umformen, indem man die Bedingung c durch einen Aufruf einer geeigneten Test-Funktion auf der rechten Seite der Gleichung ersetzt (ähnlich wie in der Sprache BABEL [16]), z.B. „$l = cond(c, r)$“. Die `cond`-Funktion hat dann die Aufgabe, c auszuwerten und als Ergebnis r zu liefern.

2.2.2 Definitional Trees

Eine hierarchische Struktur endlicher Höhe ist ein *Definitional Tree mit Pattern* π, wenn sie eine der folgenden Formen hat:

- $rule(l \rightarrow r)$

  wobei $l \rightarrow r$ eine Variante einer Regel ist und $l = \pi$.

- $branch(\pi, o, r, T_1, \ldots, T_k)$

  wobei $\pi|_o$ eine Variable ist, $r \in \{rigid, flex\}$ und für $i = 1, \ldots, k$ gilt:
  T_i ist ein Definitional Tree mit Pattern π_i, so daß $\pi_i = \pi\,[c_i(X_{i1}, \ldots, X_{in_i})]_o$ für einen Konstruktor c_i mit Stelligkeit n_i und (neuen) Variablen $X_{i1}, \ldots, X_{in}$
  ($c_1, \ldots, c_k$ sind paarweise verschieden).

- $or(T_1, T_2)$

  wobei T_1 und T_2 Definitional Trees mit Pattern π sind.

Die in [10] vorgestellten *and*-Trees werden hier nicht besprochen, sondern getrennt durch den $\wedge$-Operator behandelt (siehe unten).

Der *Definitional Tree einer n-stelligen Funktion f* ist ein Definitional Tree T mit Pattern $f(X_1, \ldots, X_n)$ (mit X_i paarweise verschieden), sodaß für jede Gleichung $l = r$ der Funktion f ein Knoten der Form $rule(l' \rightarrow r')$ mit l' Variante von l in T existiert.

Mit Hilfe der Definitional Trees läßt sich nun eine Funktion *cst* definieren, die auf einen Funktionsterm der Form $f(t_1, \ldots, t_n)$ und dem Definitional Tree der Funktion f einen Berechnungsschritt durchführt. Wie schon in Abschnitt 2.1.5 gesagt, ist das Ergebnis eines Berechnungsschrittes eine Menge von Paaren der Form '$\sigma \mid t$', wobei t ein Term und σ eine Substitution für die in t vorkommenden Variablen ist.

Die Funktion *cst* ist rekursiv auf die Definitional Trees definiert:

$\boxed{cst(t, rule(l \to r))}$

Dies ist der grundlegendste Berechnungsschritt: die Anwendung einer Funktionsgleichung $l = r$ auf einen Funktionsaufruf. Dementsprechend ist das Ergebnis (id ist die leere Substitution):

$$cst(t, rule(l \to r)) = \{id \mid \sigma(r)\} \text{ wobei } \sigma \text{ eine Substitution mit } \sigma(l) = t \text{ ist.}$$

Allerdings ist ein $rule$-Knoten nicht für das Pattern Matching verantwortlich; das wird durch die $branch$-Knoten durchgeführt:

$\boxed{cst(t, branch(\pi, o, r, T_1, \ldots, T_k))}$

Ein $branch$-Knoten hat die Aufgabe, den Subterm von t an der Position o zu untersuchen und einen oder mehrere Unterbäume T_1 bis T_k für die weitere Berechnung auszuwählen. Die $branch$-Knoten entsprechen damit den evaluation annotations. Für den Subterm $t|_o$ sind mehrere Fälle zu unterscheiden:

- Fall $t|_o$ ist ein Konstruktor, d.h. $t|_o = c(t_1, \ldots, t_n)$.

  Dieser Fall entspricht dem Patternmatching: es kommen nur noch die Funktionsgleichungen für den nächsten Berechnungsschritt in Frage, die an der Position o auf der linken Seite den Konstruktor c stehen haben. Diese Gleichungen sind durch den Definitional Tree T_i mit $pattern(T_i)|_o = c(X_1, \ldots, X_n)$ repräsentiert. Mit diesem wird dann die Berechnung fortgesetzt:

  $$cst(t, branch(\pi, o, r, T_1, \ldots, T_k)) = cst(t, T_i)$$

  Existiert kein solcher T_i (Patternmatching-Fehler), haben wir einen Fehlschlag, d.h. keine Lösung:

  $$cst(t, branch(\pi, o, r, T_1, \ldots, T_k)) = \emptyset$$

- Fall $t|_o$ ist ein Funktionsaufruf, d.h. $t|_o = f(t_1, \ldots, t_n)$

  Da sich der Subterm nicht in Kopfnormalform befindet, wird die Auswertung mit der Funktion f fortgesetzt. Das Ergebnis dieser Auswertung wird dann mit Hilfe der Funktion $replace$ an die Position o in den Term t eingesetzt:

  $$cst(t, branch(\pi, o, r, T_1, \ldots, T_k)) = replace(t, o, cst(t|_o, FT))$$

  wobei FT der Definitional Tree von f und $replace$ wie folgt definiert ist:

  $$replace(t, o, suspend) = suspend$$
  $$replace(t, o, \sigma_1 \mid t_1, \ldots, \sigma_n \mid t_n) = \{\sigma_1 \mid \sigma_1(t)[t_1]_o, \ldots, \sigma_n \mid \sigma_n(t)[t_n]_o\}$$

- Fall $t|_o$ ist eine logische Variable, d.h. $t_o = X$

  Hier ist wichtig zu wissen, ob $r = rigid$ oder $r = flex$. $r = rigid$ bezeichnet den Fall, daß das Argument an dieser Position benötigt wird, d.h. die Funktion suspendiert für eine logische Variable:

  $$cst(t, branch(\pi, o, r, T_1, \ldots, T_k)) = suspend$$

Falls $r = flex$, soll Nichtdeterminismus ausgelöst werden: die Variable wird an alle in den Subtrees $T_1, \ldots, T_n$ beschriebenen Konstruktoren gebunden. Das Ergebnis ist dann eine Disjunktion von Lösungen:

$$cst(t, branch(\pi, o, r, T_1, \ldots, T_k)) = \bigcup_{i=1}^{k} compose(\sigma_i(t), T_i, \sigma_i)$$

mit $\sigma_i = \{X = pattern(T_i)|_o\}$ und

$$\begin{aligned} compose(t, T, \sigma) &= \{\sigma \mid t\}, \quad \text{falls } cst(t, T) = suspend \\ &= \{\sigma_1 \circ \sigma \mid t_1, \ldots, \sigma_n \circ \sigma \mid t_n\}, \\ &\qquad \text{falls } cst(t, T) = \{\sigma_1 \mid t_1, \ldots, \sigma_n \mid t_n\} \end{aligned}$$

$\boxed{cst(t, or(T_1, T_n))}$

Ein *or*-Knoten entspricht dem `or`-Schlüsselwort aus Abschnitt 2.1.5. Da hier Nichtdeterminismus erzeugt werden soll, ist das Ergebnis wieder eine Disjunktion:

$$cst(t, or(T_1, T_n)) = cst(t, T_1) \cup cst(t, T_2)$$

(Für den Fall, daß ein Aufruf $cst(t, T_i)$ suspendiert, wird sein Ergebnis *suspend* natürlich nicht zurückgegeben, siehe dazu Abschnitt 2.2.4).

Anhand der zwei Beispielfunktionen aus Abschnitt 2.1.5 soll nochmal der Zusammenhang zwischen Funktionsgleichungen, evaluation annotations und Definitional Tree verdeutlicht werden:

1. Die Funktion `leq` (diesmal als „flex"):

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

leq(z,X)       = true.
leq(s(X),z)    = false.
leq(s(X),s(Y)) = leq(X,Y).
```

besitzt den folgenden Definitional Tree:

```
branch(leq(X,Y),1,flex,
        rule(leq(z,X) -> true),
        branch(leq(s(X),Y),2,flex,
              rule(leq(s(X),z) -> false),
              rule(leq(s(X),s(Y)) -> leq(X,Y))))
```

2. Die Funktion `or`

```
por eval 1:rigid(false => 2:rigid) or 2:rigid.

 true por X     = true.
    X por true  = true.
false por false = false.
```

besitzt den Definitional Tree:

```
or(branch(por(X,Y),1,rigid,
        rule(por(true,X) -> true),
        branch(por(false,X),2,rigid,
            rule(por(false,false) -> false))),
   branch(por(X,Y),2,rigid,
        rule(por(X,true) -> true)))
```

2.2.3 Auswertung beliebiger Terme

Die Semantik eines Berechnungsschrittes wurde mit Hilfe von *cst* nur auf Funktionstermen definiert. Natürlich sollen auch beliebige Terme ausgewertet werden können, d.h. Terme, deren oberstes Symbol keine Funktion ist. Dazu wird eine Funktion *cs* definiert:

1. Eine Variable X als auszuwertender Term suspendiert:

$$cs(X) = suspend$$

2. Zur Auswertung von Funktionen wird *cst* benutzt:

$$cs(f(t_1,\ldots,t_n)) = cst(f(t_1,\ldots,t_n), FT)$$

   wobei FT der Definition Tree von f ist.

3. Ist der Term eine Konstruktorapplikation, dann werden die Argumente nach einem Funktionsaufruf durchsucht, der als nächstes ausgewertet werden soll. Suspendieren alle Argumente, supendiert auch der gesamte Term:

$$\begin{aligned} cs(c(t_1,\ldots,t_n)) &= replace(c(t_1,\ldots,t_n), k, cs(t_k)), \\ &\qquad \text{falls } cs(t_i) = suspend, i = 1\ldots, k-1 \text{ und } cs(t_k) \neq suspend \\ &= suspend, \quad \text{falls } cs(t_i) = suspend, i = 1,\ldots,n \end{aligned}$$

Mit der Funktion *cs* läßt sich nun ein *Berechnungsschritt für eine Disjunktion (BS)* definieren. Es gilt

$$(\sigma \mid t) \vee D \xrightarrow{BS} \{\sigma_1 \circ \sigma \mid t_1, \ldots, \sigma_n \circ \sigma \mid t_n\} \vee D$$

falls $\sigma \mid t$ noch (nicht suspendierende) Funktionssymbole enthält und

$$cs(t) = \{\sigma_1 \mid t_1, \ldots, \sigma_n \mid t_n\}$$

2.2.4 Der $\wedge$-Operator

Laut Definition soll die Auswertung des Terms $t_1 \wedge t_2$ genau dann erfolgreich terminieren, wenn beide Terme t_1 und t_2 erfolgreich ausgewertet werden konnten.

Eine operationale Semantik für eine sequentielle Implementierung des $\wedge$-Operators wird in [9] beschrieben:

1. Die Auswertung beginnt mit dem Term t_1,
2. Terminiert die Auswertung von t_1 erfolgreich, wird der Term t_2 ausgewertet,
3. Suspendiert die Auswertung von t_1, wird ein Berechnungsschritt von t_2 ausgeführt,
4. da der Berechnungsschritt von t_2 möglicherweise eine Variable gebunden hat, wird wieder mit der Auswertung von t_1 fortgefahren.

Dies verlangt eine Anpassung der Semantik des *or*-Knotens: um zu verhindern, daß suspendierte Berechnungspfade eines *or*-Knotens durch den $\wedge$-Operator „übersehen" werden, suspendiert der ganze Oder-Knoten im Falle einer Suspendierung eines Teilbaumes:

$$\begin{aligned} cst(t, or(T_1, T_2)) &= cst(t, T_1) \cup cst(t, T_2), \\ &\qquad \text{falls } cst(t, T_1) \neq suspend \textbf{ und } cst(t, T_2) \neq suspend \\ &= suspend, \quad \text{sonst} \end{aligned}$$

Das test-and-generate–Beispiel aus Abschnitt 2.1.8

$$\texttt{test(X)} \wedge \texttt{generate(X)}$$

läuft mit Hilfe dieser Semantik wie folgt ab: der $\wedge$-Operator bewirkt, daß zuerst `test(X)` ausgewertet wird. Da `test` „rigid" ist, suspendiert seine Auswertung. Daraufhin wird der rechte Constraint um einen Schritt ausgewertet. Das Ergebnis ist:

$$\begin{array}{ll} & \{\ \{X=1\} \mid \texttt{test(1)} \wedge \{\}\ \} \\ \vee & \{\ \{X=2\} \mid \texttt{test(2)} \wedge \{\}\ \} \\ \vee & \{\ \{X=3\} \mid \texttt{test(3)} \wedge \{\}\ \} \end{array}$$

Nun kann wieder mit dem linken Constraint fortgefahren werden. Da sowohl `test(1)` und `test(2)` fehlschlagen (Pattern-Matching-Fehler), bleibt übrig:

$$\{\ \{X=3\} \mid \{\} \wedge \{\}\ \}$$

Nach einem letzten Schritt auf dem ersten Argument, bleibt nur noch das zweite Argument übrig:

$$\{\ \{X=3\} \,[]\, \{\}\ \}$$

was dann das endgültige Ergebnis darstellt.

Eine nebenläufige Implementierung des $\wedge$-Operators wird im Kapitel über die Implementierung vorgestellt. Dort werden auch sogenannte *Stabilitätsregeln* vorgestellt, die die (teuere) Erzeugung von Nichtdeterminismus so weit wie möglich verzögern.

2.2.5 Erweiterungen

Der Apply-Operator

Die Semantik des Apply-Operators aus Abschnitt 2.1.6 kann mit Hilfe einer Funktion @ und einem Konstruktor $\$_f$ für jede n-stellige Funktion f definiert werden:

$$f \quad @ \quad t_1 = \$_f(t_1)$$

$$
\begin{aligned}
\$_f(t_1) \quad @ \quad & t_2 = \$_f(t_1, t_2) \\
& \vdots \\
\$_f(t_1, \ldots, t_{n-1}) \quad @ \quad & t_n = f(t_1, \ldots, t_n)
\end{aligned}
$$

Der Choice-Operator

Der Choice-Operator aus Abschnitt 2.1.9 mit der Form

```
choice Constraint1 -> Ausdruck1 ;
       ...
       Constraintn -> Ausdruckn
```

hat die Aufgabe nichtdeterministisch einen der Ausdrücke $Ausdruck_i$ mit einem erfüllbaren Constraint $Constraint_i$ auszuwählen. Dies läßt sich auf verschiedene Weise implementieren:

- eine unfaire sequentielle Implementierung würde einfach die Constraints der Reihe nach auswerten.
- eine sequentielle Implementierung könnte abwechselnd bei jedem der Constraints $Constraint_1, \ldots, Constraint_i$ einen Auswertungsschritt durchführen. Der erste Constraint, der erfolgreich ausgewertet wurde, wäre der entscheidende.
- eine nebenläufige Implementierung könnte eine parallele Ausführung der Constraints als Threads starten. Terminiert ein Thread erfolgreich, werden alle anderen Alternativen gelöscht.

Die beiden letzten Möglichkeiten würden den „schnellsten“ Constraint auswählen.

Etwas mehr Aufwand benötigt die Implementierung der *eingekapselten Suche*: hier sollen ja Schreibzugriffe auf „externe“ Variablen zur Suspendierung führen. Hier seien nur zwei mögliche Erweiterungen der operationellen Semantik vorgestellt:

- man führt mit der *cst*-Funktion eine Liste der lokalen Variablen: ein Aufruf des Choice-Operators wird immer mit einer leeren Liste begonnen (eingekapselte Suche [19]).
- man markiert die Variablen mit einer Zahl, die die Schachtelungstiefe der Choice-Operatoren angibt (*Level*): es dürfen nur Variablen gebunden werden, die in der gleichen oder einer tieferen Schachtelung erzeugt wurden (d.h. nur lokale Variablen).

Kapitel 3

Java und Pizza

Dieses Kapitel gibt eine kurze Übersicht über die Syntax und Semantik von Java-Programmen. Es soll weder eine Sprachdefinition noch eine Einführung in die objektorientierte Programmierung darstellen [Java].

Der größte Teil dieses Kapitels beschäftigt sich daher mit den Besonderheiten von Java — insbesondere die Java Virtual Machine und die Unterstützung von Threads —, die einen wesentlichen Einfluß auf Implementierungsentscheidungen bei der Entwicklung des Curry-Compilers hatten.

3.1 Java: eine objektorientierte Sprache

3.1.1 Klassen und Objekte

Java ist eine objektorientierte Sprache: in Java besteht die Welt aus *Objekten*, die Daten speichern, andere Objekte verändern, neue Objekte erzeugen,... Jedes Objekt ist eigenständig, d.h. enthält sowohl seine Daten als auch die auf ihn erlaubten Operationen.

Um den Aufbau eines Objektes zu beschreiben, verwendet man eine Klassendefinition. In dieser wird nicht ein spezielles Objekt definiert, sondern eine *Klasse* von Objekten. Jedes Objekt aus dieser Klasse (eine sogenannte *Instanz* der Klasse) besitzt somit den gleichen Aufbau. Eine Klassendefinition besteht aus:

1. Definitionen der Daten, die ein Objekt dieser Klasse speichern kann. Solche Daten werden *Felder* oder *member* genannt [1].

2. Definitionen der Operationen, die auf diesen Objekte ausgeführt werden können, die sogenannten *Methoden*.

Das Beispiel 3.1 zeigt eine mögliche Definition für die Klasse der Kreis-Objekte in Java: laut dieser Definition speichern Kreisobjekte also eine reelle Zahl namens `radius` und besitzen eine Methode `berechneUmfang`, die den Umfang des Kreises zurückgibt.

[1] die aus den imperativen Sprachen bekannten Datenfelder im Sinne von `array [0..n] of Type` nenne ich zur Unterscheidung *Arrays*.

```
class Kreis {
    float radius ;

    float berechneUmfang() { return radius*2.0f*3.14f ; }
}
```

Beispiel 3.1: eine Kreis-Klasse

Diese Klasse läßt sich nun weiterverwenden, z.B. in der Klasse `Test`:

```
class Test {
    static void main(String[] args) {
        Kreis k1,k2,k3 ;

        k1=new Kreis() ; k2=new Kreis() ;
        k1.radius=13.5f ;
        float umfang=k1.berechneUmfang() ;

        k3=k1 ;
    }
}
```

Die Klasse `Test` besteht aus einer einzigen Methode `main`. Zum Kopf der Methode `main` sei nur gesagt, daß damit signalisiert wird, daß das Betriebssystem diese Methode als Einstiegspunkt verwenden kann — ähnlich der `main`–Funktion in C-Programmen.

Die Methode `main` deklariert lokal drei Kreisobjekte (die Deklaration `Kreis k1,k2,k3` befindet sich *innerhalb* der Methode). In Java werden alle Objekte durch Referenzen repräsentiert: Referenzen sind die einzige Möglichkeit, auf ein Objekt zuzugreifen. Bei `k1`,`k2` und `k3` handelt es sich damit um Referenzen: um tatsächlich ein Objekt zu erhalten, muß dieses zuerst mit `new` erzeugt und einer Referenzvariablen zugewiesen werden. Dies geschieht hier zweimal mit `k1=new Kreis()` und `k2=new Kreis()`.
Anschließend weist das Programm dem ersten Kreis den Radius `13.5` zu und ruft dessen `berechneUmfang`-Methode auf [2]. Das Ergebnis steht dann in der Variable `umfang`.
Aus der Verwendung von Referenzen folgt, daß die letzte Zeile, `k3=k1`, keine Kopie des Kreises `k1` erstellt: es wird nur die Referenz kopiert und damit repräsentieren `k3` und `k1` den selben Kreis.

Auffallend ist, daß :

1. die Kreise `k1` und `k2` zwar erzeugt (mit Hilfe von `new`), aber nicht wieder aufgelöst werden. In der Tat sieht Java keine explizite Freigabe eines Objektes vor. Dies geschieht automatisch, wenn keine Referenz mehr auf das Objekt existiert (in diesem Beispiel also nach Beenden der Methode `main`).

2. zwar Kreisobjekte explizit erzeugt werden müssen, nicht aber `float`-Objekte. Java unterscheidet zwischen Basistypen und Klassen: Objekte (wie zum Beispiel Kreise) werden

[2] in anderen objektorientierten Sprachen, wie z.B. Smalltalk, würde man sagen, daß der Kreis aufgefordert wird, seinen Umfang zu berechnen.

(wie oben beschrieben) durch Referenzen dargestellt, während Basistypen (dazu zählen `float`, `int`, `boolean`,...) aus Effizienzgründen direkt abgespeichert werden. Deshalb kann alles, was zu Klassen und Objekten gesagt wird, *nicht* auf Basistypen übertragen werden[3].

3.1.2 Konstruktoren und Zugriffsattribute

Abgesehen von den Methoden unterstützen Klassen eine weitere Art von Operationen: die sogenannten *Konstruktoren* (nicht zu verwechseln mit den Konstruktoren aus Curry). Konstruktoren sind Operationen, die ausgeführt werden, wenn mit Hilfe von `new` ein neues Objekt erzeugt werden soll. Konstruktoren erkennt man daran, daß sie den gleichen Namen wie die Klasse tragen. Das nächste Beispiel erweitert die `Kreis`-Klasse um einen Konstruktor, der es erlaubt den Radius des Kreises schon bei der Erzeugung anzugeben:

```
class Kreis {
    float radius ;

    Kreis(float r) { radius=r ; }

    float berechneUmfang() { return radius*2.0f*3.14f ; }
}
```

Damit bleibt dem Benutzer erspart, den Radius explizit setzen zu müssen:

```
class Test {
    static void main(String[] args) {
        Kreis k1 ;

        k1=new Kreis(13.5f) ;
        float umfang=k1.berechneUmfang() ;
    }
}
```

Hier wäre es angebracht, den Zugriff auf das Feld `radius` noch weiter einzuschränken: mit `private` markierte Elemente sind nur noch in Methoden der gleichen Klasse sichtbar. Wir können damit z.B. die Implementierung des Radius verbergen, indem wir `radius` als `private` deklarieren und zwei Methoden für den Zugriff auf `radius` definieren:

```
class Kreis {
    private float radius ;

    Kreis(float r) { setzeRadius(r) ; }

    float gibRadius() { return radius ; }
    void setzeRadius(float r} { radius=r ; }

    float berechneUmfang() { return radius*2.0f*3.14f ; }
}
```

[3] Es gibt allerdings die Möglichkeit Basistypen mit Hilfe von *Wrapper*-Klassen in normale Objekte umzuwandeln — z.B. `int` nach `Integer`.

Weiterhin gibt es die Möglichkeit, Zugriffsrechte zu erweitern. Die obengenannten Beispiele funktionieren nämlich nur dann, wenn die Klassen `Kreis` und `Test` zu einem sogenannten *Package* zusammengefaßt werden. Um Klassen, Felder und Methoden anderen Packages zugänglich zu machen, muß man diese mit `public` markieren. Man kann dann mit Hilfe einer `import`-Anweisung auf die mit `public` markierten Elemente anderer Packages zugreifen.

3.1.3 Klassenfelder und -methoden

Angenommen wir möchten die Konstante π auch anderen Klassen zugänglich machen, z.B. indem wir die Klasse `Kreis` erweitern (`final` signalisiert eine Konstante):

```
class Kreis {
    final float pi=3.1415f ;
    ...
}

class Test {
    static void main(String[] args) {
        Kreis k1=new Kreis(13.5) ;

        System.out.print("Das ist pi: ") ; System.out.print(k1.pi) ;
    }
}
```

Das hat natürlich den Nachteil, daß jedes Kreis-Objekt ein Datum mit dem Inhalt 3.1415 enthält (ganz abgesehen davon, daß wir erst einen Kreis erzeugen müssen, um π zu erhalten). Deshalb ist es sinnvoller, `pi` nicht als Feld jedes Kreis-*Objektes*, sondern als Feld der Kreis-*Klasse* zu deklarieren. Das geschieht mit dem Schlüsselwort `static`:

```
class Kreis {
    static final float pi=3.1415f ;
    ...
}

class Test {
    static void main(String[] args) {
        System.out.print("Das ist pi: ") ; System.out.print(Kreis.pi) ;
    }
}
```

Wie man an der `main`-Methode sieht, ist es auch möglich, Methoden als `static` zu markieren. Ein weiteres Beispiel für `static`-Felder ist das Feld `out` der Klasse `System`. Dieses Feld stellt die Standardausgabe dar und besitzt einige grundlegende Methoden zur Ausgabe, z.B. `print` und `println`. Die Klasse `System` befindet sich im Package `java.lang` und wird automatisch in jedes Java-Programm importiert (auch ohne `import`-Anweisung).

```
Kreis k1,k2 ;
FarbigerKreis fk1,fk2 ;
boolean result ;

// hier k1,k2,fk1,fk2 initialisieren

result=k1.kleinerAls(k2) ;
result=k1.kleinerAls(fk1) ;
result=fk1.kleinerAls(fk2) ;
result=fk1.kleinerAls(k1) ;
```

Beispiel 3.2:
Verwendung von Klassen und Subklassen

3.1.4 Vererbung

Einer der wichtigsten Aspekte der objektorientierten Programmierung ist die *Vererbung*: es ist möglich, aus einer bestehenden Klasse eine neue Klasse (eine *Subklasse* der bestehenden Klasse) zu definieren [4], die zwar die alte Klasse erweitert, aber trotzdem deren Eigenschaften behält („erbt"). D.h. jede Operation, die mit Objekten einer Klasse A arbeitet, kann dies auch mit Objekten einer Subklasse von A. Als Beispiel sei die folgende Klasse gegeben:

```
class Kreis {
    float radius ;

    boolean kleinerAls(Kreis andererKreis) {
        return radius<andererKreis.radius ;
    }
}
```

Diese Klasse läßt sich nun erweitern zu einer Klasse `FarbigerKreis`, die zusätzlich eine Farbe für den Kreis abspeichert:

```
class FarbigerKreis extends Kreis {
    private int farbe ;

    void setzeFarbe(int f) { ... }
    int gibFarbe() { ... }
}
```

Da die Klasse `FarbigerKreis` eine Subklasse der Klasse `Kreis` ist, sind alle Anweisungen in Beispiel 3.2 korrekt, d.h. die Methode `kleinerAls`, die ja als Argument ein `Kreis`-Objekt erwartet, arbeitet auch mit `FarbigerKreis`-Objekten.

[4] die sogenannte *Mehrfachvererbung* ist in Java über das *Interface*-Konzept realisierbar.

3.1.5 Überschreiben von Methoden

Subklassen ist es erlaubt, Methoden aus ihrer Oberklasse zu *überschreiben*, d.h. neu zu implementieren, vorausgesetzt die Methode wurde nicht als `private` oder `final` markiert. Da die Überschreibung völlig transparent sein soll, muß die neue Methode exakt den gleichen Typ wie die zu überschreibende Methode haben. Z.B.:

```
class Kreis {
   String alsText() { return "ich bin ein Kreis" ; }
}

class FarbigerKreis {
    int farbe ;

    String alsText() { return "ich bin ein Kreis mit Farbe "+farbe ; }
}
```

Die alte (überschriebene) Implementierung der Methode bleibt weiterhin erhalten. Man erhält sie, indem man auf die Oberklasse von `FarbigerKreis`, d.h. auf die Klasse `Kreis`, mit `super` zugreift. Daher könnte man `FarbigerKreis` auch so definieren:

```
class FarbigerKreis {
    int farbe ;

    String alsText() {
        String alterText=super.alsText() ;
        return alterText+" mit Farbe "+farbe ;
    }
}
```

Im Normalfall (d.h. ohne `super`) ist immer nur die aktuellste Version der Methode sichtbar, unabhängig davon mit welchem Typ das Objekt angesprochen wird. Z.B. gibt die folgende Methode

```
    void gibKreisAus(Kreis k) {
        System.out.println(k.alsText()) ;
    }
```

für Objekte aus der Klasse `FarbigerKreis` korrekt "ich bin ein Kreis mit Farbe ..." aus, auch wenn das *Argument* `k` als `Kreis` deklariert wurde. Die Methode `alsText` wurde also tatsächlich überschrieben.

3.1.6 Abstrakte Klassen und Methoden

Manchmal ist es sinnvoll, eine Methode in einer Klasse zu deklarieren, deren Implementierung jedoch den Subklassen zu überlassen.

Zum Beispiel:

```
class Figur {
    int farbe ;
    float pos_x,pos_y ;

    float berechneFlaeche() ;
}
```

Die Klasse `Figur` soll eine Oberklasse für Klassen wie `Kreis` und `Rechteck` darstellen. Unter der Annahme, daß alle Figuren Objekte aus diesen beiden Klassen sind, ist es nicht sinnvoll, in der Klasse `Figur` eine Implementierung für `berechneFlaeche` vorzugeben. Dazu wird in Java die Klasse und die betreffende Methode als *abstrakt* deklariert:

```
abstract class Figur {
    int farbe ;
    float pos_x,pos_y ;

    abstract float berechneFlaeche() ;
}
```

Abstrakte Klassen können nicht instanziiert werden, d.h. eine abstrakte Klasse wird erst durch ihre Subklassen benutzbar [5].

3.1.7 Weitere Sprachelemente

Interfaces

Wie schon erwähnt, läßt sich Mehrfachvererbung in Java nur mit Hilfe von sogenannten *Interfaces* realisieren. Interfaces beschreiben — im Gegensatz zur Vererbung — keine Verwandtschaften zwischen zwei Klassen, sondern nur deren gemeinsame Eigenschaften. So könnte man sich die folgenden Interfaces vorstellen:

```
interface Vergleichbar {
    boolean istGleich(Object anderesObjekt) ;
}

interface Kopierbar {
    Object erstelleKopie() ;
}
```

Eine Klasse kann nun beide Eigenschaften implementieren, was dann einer Mehrfachvererbung entspricht:

```
class Kreis implements Vergleichbar,Kopierbar {
    // Implementierung der Methoden istGleich und erstelleKopie:
    ...
}
```

[5] falls auch die Subklasse auf eine Implementierung der abstrakten Methoden verzichtet, wird auch sie zu einer abstrakten Klasse.

Damit ist der Kreis überall dort einsetzbar, wo ein „kopierbares“ Objekt verlangt wird:

```
void doit(Kopierbar obj) {
    ...
}
```

Exceptions

Java unterstützt das Auslösen und Abfangen von Exceptions. Mit Hilfe von `throw` kann eine neue Exception ausgelöst werden. Dadurch wird die momentan bearbeitete Methode abgebrochen und zur aufrufenden Methode zurückgekehrt. Dies setzt sich solange fort, bis die Exception durch einen `try...catch`-Block abgefangen und verarbeitet wird. Ein kleines Beispiel verdeutlicht die Syntax:

```
void doIO() throws IOException {
    ...
    if(errorOccured) throw new IOException("Fehler in IO") ;
    ...
}

void tryIO() {
    try{
        doIO() ;
    }catch(Exception e) {
        System.out.println("Exception abgefangen: "+e) ;
    }
}
```

Das besondere an Exceptions ist, daß `try-catch`-Blöcke zur Laufzeit nichts kosten, wenn keine Exception ausgelöst wurde.

Synchronisation

Java enthält Sprachelemente zur Synchronisation von Threads durch sogenannte Monitore. Dies wird im Abschnitt 3.3 beschrieben.

3.2 Das Ausführungsmodell von Java

Um ein Java-Programm, das in Form von Quellcode vorliegt, verwenden zu können, muß es durch einen Compiler verarbeitet werden. Dieser Compiler erzeugt aus dem Quellcode Binärdateien, die dann ausgeführt werden können. Dies ähnelt der bekannten Vorgehensweise bei anderen imperativen Sprachen wie z.B. Pascal, C oder C++.

Jedoch verlieren bei diesen Sprachen viele der auf Quellcodeebene existierenden Eigenschaften zur Laufzeit ihre Gültigkeit. So ist es z.B. in C++ problemlos möglich, die `private`-Felder eines Objektes auszulesen: man braucht nur den Zeiger auf das Objekt in einen Zeiger auf ein Byte-Array umzuwandeln.

Warum dies in Java nicht möglich ist und wie diese Eigenschaft realisiert wurde, wird in den nächsten Abschnitten beschrieben.

3.2.1 Der Java-Compiler

Java-Quellcode liegt als Text in Dateien mit der Endung `.java` vor. Mit Hilfe eines Java-Compilers, z.B. dem `javac`-Programm des Java Development Kits (JDK) von Sun [Java], können diese Dateien compiliert werden. Dabei wird für jede vorkommende Klassendefinition eine `*.class`-Datei erzeugt (d.h. eine Datei mit dem Namen der Klasse und der Endung `.class`). Den aus anderen Sprachen bekannten statischen Linkvorgang gibt es bei Java nicht.

Solche `.class`-Dateien enthalten nicht nur die zu Binärcode compilierten Methoden, sondern auch die Deklarationen der Felder und Methoden der Klasse — einschließlich der Typen und der Zugriffsrechte. Mit Hilfe eines Java-Disassemblers kann praktisch die gesamte Klasse rekonstruiert werden.

Das Importieren bestehender Klassen mit Hilfe von `import` geschieht übrigens auf Basis der `.class`-Dateien, d.h. der Compiler kann alle nötigen Überprüfungen durchführen, auch wenn der Quellcode dieser Klassen nicht vorliegt.

3.2.2 Ausführen von Java-Programmen

`.class`-Dateien haben eine herausragende Eigenschaft: der Binärcode der Methoden besteht nicht aus Maschinenbefehlen für eine existierende Hardware, sondern aus Befehlen (dem sogenannten *Bytecode*) für eine abstrakte Maschine, der JVM (Java Virtual Machine). Die Aufgabe, den Maschinencode der JVM in maschinenspezifischen Code umzuwandeln, wird einem Java-Interpreter überlassen. Da diese Umwandlung immer nur zur Laufzeit erfolgt, sind Java-Programme (d.h. die `.class`-Dateien) unabhängig von einer bestimmten Plattform. Die einzige Voraussetzung ist, daß für die gewünschte Plattform ein Java-Interpreter existiert.

Folgende Schritte werden also zum Ausführen eines Java-Programms durchgeführt:

1. ein Java-Interpreter wird gestartet. Als Argument erhält der Interpreter den Namen der zu auswertenden Java-Klasse. Voraussetzung ist, daß die Klasse eine Methode namens `main` besitzt (siehe dazu das Beispiel aus Abschnitt 3.1.3).

2. der Interpreter lädt die spezifizierte Klasse und die von ihr importierten Klassen (d.h. ihre `.class`-Dateien).

3. der Bytecode der Methode `main` wird ausgeführt.

Für die Auswertung des Bytecodes gibt es zwei Ansätze:

- mit Hilfe eines Interpreters, der Befehl für Befehl der JVM in Maschinencode für die aktuelle Plattform umsetzt. Oder, da diese Vorgehensweise relativ langsam ist,

- mit Hilfe eines Just-in-Time-Compilers: solch ein Compiler übersetzt den Bytecode während des *Ladevorgangs* in plattformspezifischen Code. Auch hier bleibt die Portabilität erhalten.

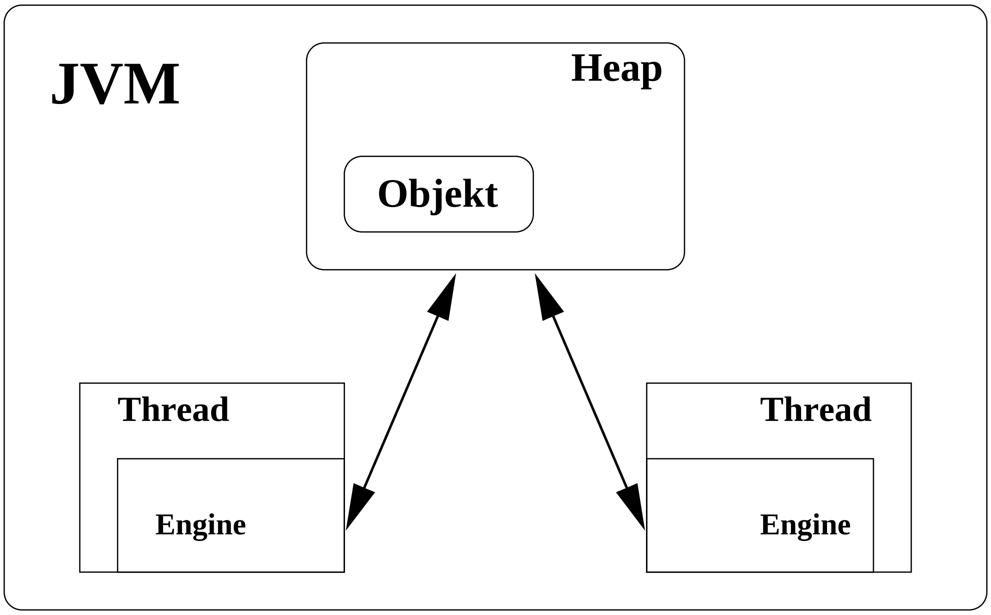

Abbildung 3.1: Aufbau der JVM

Die Plattformunabhängigkeit des Bytecodes und die Organisation von Java-Programmen in Form von `.class`-Dateien prädestiniert Java für Anwendungen in heterogenen Netzwerken: nicht lokal vorhandene Klassen können einfach über das Netzwerk nachgeladen werden, ungeachtet der eingesetzten Plattform. Weiterhin besitzt die JVM ein umfassendes Sicherheitskonzept, das den „Amoklauf" von fremden Klassen (z.B. Viren) weitgehend verhindern kann (siehe Abschnitt 3.2.4). [6]

3.2.3 Die JVM

Die von der JVM simulierte Maschine besteht aus

- einem **Heap**: der Heap stellt den Hauptspeicher der Maschine dar. Hier befinden sich alle erzeugten Objekte. Ein Garbage Collector sorgt dafür, daß Objekte, auf die keine Referenzen mehr existieren, aus dem Heap entfernt werden.
- **Threads**: Threads werden ausführlicher in Abschnitt 3.3 besprochen. Hier soll nur das wichtigste Element der Threads erläutert werden: die Ausführungseinheit (*execution engine*), die den Bytecode interpretiert.

(Abbildung 3.1)

Der Bytecode

Java-Programme liegen in Bytecode vor. Wenn eine Klasse geladen wird, untersucht der *Code-Verifier* den Bytecode auf Verstöße gegen die Spezifikation, z.B. muß die Zieladresse von Sprungbefehlen auf einen korrekten Befehl zeigen.

[6] Java wurde in der Tat in Hinblick auf den Einsatz von Java-Programmen als WWW-Applets entwickelt

Beim Entwurf des Befehlssatzes wurde versucht, soweit wie möglich Optimierungen und Überprüfungen in die Compilezeit zu verlagern. Z.B. existieren viele Befehle in Versionen für Integer, Long-Integer, usw. Natürlich ist die JVM auf die Ausführung von Java-Programmen optimiert: der Befehlssatz enthält spezielle Befehle zum Aufruf von Methoden, Zugriff auf Felder, ...; hardwarenahe Operationen sind nicht vorgesehen. Andere Programmiersprachen lassen sich deshalb nur sehr schwer nach Bytecode übersetzen.

Um plattformspezifische Eigenschaften trotzdem der JVM zugänglich zu machen (z.B. Grafikoperationen), können Methoden in Java als `native` markiert werden: ihre Implementierung wird dann einer plattformspezifischen Bibliothek überlassen.

Die Ausführungseinheit

Die Ausführungseinheit der JVM ist eine Stackmaschine: alle Operationen werden auf einem Datenstack durchgeführt. Dazu müssen die Operanden vorher auf den Stack geladen werden, wo sie dann nach einer Operation durch das Resultat ersetzt werden. Die Optimierung von Speicherzugriffen, z.B. durch Caching-Techniken, wird der jeweiligen Implementation der JVM überlassen.

Beispielsweise wird die Methode

```
void test() {
    int i ;
    int j=1 ;

    i=5+j ;
}
```

zu folgendem Bytecode übersetzt:

```
iconst_1            // 1
istore_2            // j:=

iconst_5            // 5
iload_2             // j
iadd                // +
istore_1            // i:=
return
```

3.2.4 Das Sicherheitskonzept

Das Sicherheitskonzept tritt an drei Stellen in Aktion:

1. **beim Laden einer Klasse**: das Laden von Klassen wird von einer Instanz der Klasse `ClassLoader` durchgeführt. Von dieser Klasse lassen sich eigene ClassLoader ableiten. Denkbar wäre ein ClassLoader, der Klassen mit Hilfe einer Internet-URL lokalisiert. Damit könnte der ClassLoader externe von lokalen Klassen unterscheiden und sie für den SecurityManager (siehe unten) markieren.

   Außerdem gibt es die Möglichkeit, Klassen mit einer digitalen Unterschrift zu versehen und deren Authentizität durch den ClassLoader überprüfen zu lassen.

2. **beim Zugriff auf kritische Elemente**, z.B. Dateioperationen. Diese Aktionen werden durch eine Instanz der `SecurityManager`-Klasse überwacht. Auch hier wäre eine eigene Subklasse für Netzwerkanwendungen denkbar. Z.B. erlaubt der SecurityManager des WWW-Browsers von Netscape keine Festplattenzugriffe durch externe (= fremde) Klassen.

3. **beim Zugriff auf Objekte und Klassen**: die Struktur der JVM bedingt, daß Überprüfungen von Zugriffsrechten oder Typumwandlungen zur Lade- und Laufzeit durchgeführt werden. Es ist in Java *nicht* möglich, sich in ein Objekt „einzuhacken“, indem man ein (aus C oder C++ bekanntes) Typecasting durchführt:

   ```
   void hack(Object *obj) {
       char *bytefeld=(char*)obj ;
       // jetzt das bytefeld auslesen
   }
   ```

   Ein ähnliches Programm könnte auch in Java compiliert werden, doch zur Laufzeit würde die Typumwandlung eine Exception auslösen, außer natürlich wenn das Argument `obj` tatsächlich eine Instanz der `Byte`-Klasse ist.

3.3 Threads und Kommunikation

3.3.1 Threads

Die JVM ist in der Lage, mehrere Threads gleichzeitig auszuführen. Ein Thread stellt einen eigenständigen Programmablauf dar, d.h. jeder Thread hat einen eigenen Programmcounter und einen eigenen Registersatz. Da sich jedoch alle Threads den Objekt-Heap der JVM teilen, sind sie in der Lage miteinander zu kommunizieren (siehe dazu den nächsten Abschnitt). Auf welche Art die JVM die verschiedenen Threads ausführt — es existieren sogar Multiprozessorimplementierungen — ist nicht spezifiziert und jeder Implementierung der JVM überlassen. Allerdings müssen einige Regeln eingehalten werden [Java].

Beim Start des Java-Interpreters wird ein Hauptthread generiert, der dann das Java-Programm (d.h. die `main`-Methode) ausführt. Der Programmierer kann nun in seinem Programm dynamisch neue Thread generieren und wieder löschen. Als neuen Thread kann jedes Objekt gestartet werden, das das `Runnable`-Interface implementiert:

```
interface Runnable {
    public void run() ;
}
```

Der Thread selbst ist eine Instanz der Klasse `Thread`. Entweder implementiert der Thread selbst die `run`-Methode oder erhält bei seiner Erzeugung als Argument solch ein Objekt. Die beiden Klassen `Test1` und `Test2` aus Beispiel 3.3 sind also gleichwertig.

Allerdings wird ein Thread mit seiner Erzeugung nicht automatisch gestartet. Dazu existiert die Methode `start()` und analog dazu die Methode `stop()` zum Beenden eines Threads, vorausgesetzt der Thread hat sich nicht schon selbst durch einen Ausstieg aus der `run`-Methode

```
class MeinThread extends Thread {
    public void run() {
        System.out.println("Ich laufe parallel") ;
    }
}

class Test1 {
    public static void main(String args[]) {
        MeinThread mt=new Thread() ;
    }
}

class RunObject implements Runnable {
    public void run() {
        System.out.println("Ich laufe parallel") ;
    }
}

class Test2 {
    public static void main(String args[]) {
        Thread mt=new Thread(new RunObject()) ;
    }
}
```

Beispiel 3.3: Erzeugung von Threads

beendet. Ein komplettes Beispiel sieht also so aus (mit der Methode `join` kann ein Thread auf das Ende eines anderen Threads warten):

```
class MeinThread extends Thread {
    public void run() { System.out.println("Ich bin ein neuer Thread") ; }
}

class Test {
    public static void main(String args[]) {
        MeinThread mt=new Thread() ;

        System.out.println("Ich bin der Hauptthread") ;
        mt.join() ;
        System.out.println("Der neue Thread hat sich beendet") ;
    }
}
```

Zusätzlich enthält die `Thread`-Klasse noch einige Methoden, um das Laufzeitverhalten eines Threads zu beeinflussen: es existieren Methoden, um einen Thread für eine bestimmte Zeit schlafenzulegen, seine Ausführungspriorität zu ändern, usw.

Die JVM läuft nach dem Start eines Programmes solange, bis alle erzeugten Threads beendet worden sind (was für den Hauptthread einem Austieg aus der `main`-Methode entspricht) oder bis ein Abbruch durch den Benutzer erfolgt. Wenn dieses Verhalten der JVM nicht erwünscht ist, kann man einzelne Threads als *Daemons* markieren. Daemon-Threads werden automatisch nach Beenden des Hauptthreads gelöscht (z.B. der im Hintergrund laufende Garbage-Collector).

3.3.2 Thread-Gruppen

Die Klasse `ThreadGroup` bietet dem Programmierer die Möglichkeit, Threads zu Gruppen zu organisieren. Thread-Gruppen dürfen auch weitere Thread-Gruppen enthalten, so daß sich ganze Ausführungsbäume aufbauen lassen. Mit Hilfe von Thread-Gruppen lassen sich Thread-Operationen (z.B. `stop`) auf alle Mitglieder der Gruppe anwenden.

Aus Sicherheitsgründen werden Operationen auf Threads und auf Thread-Gruppen vom SecurityManager überwacht, d.h. man kann z.B. einzelnen Threads verbieten, eine Thread-Gruppe zu verändern.

3.3.3 Monitore

Wo immer mehrere Threads gleichzeitig laufen, ist es nötig, gewisse Maßnahmen zur Synchronisierung von zeitgleichen Zugriffen auf Datenstrukturen zu ergreifen. Wie in vielen anderen Programmiersprachen sind Operationen wie z.B.

```
zaehler=zahler+1 ;
```

in Java nicht elementar, d.h. gliedern sich auf in Einzeloperationen — in diesem Fall: lesen des alten Zählerstandes, erhöhen und neuen Zählerstand zurückschreiben. Würden die Threads diese Einzeloperationen gleichzeitig bzw. verschränkt ausführen, wäre die Wirkung dieser Programmzeile nichtdeterministisch. Deshalb bietet Java die Möglichkeit, daß Threads Zugriffe auf Objekte exklusiv durchführen, d.h. unter Ausschluß aller anderen Threads. Dazu benutzt die JVM das Konzept der *Monitore* [1].

Jedes Objekt in Java besitzt einen Monitor. Ein Monitor besteht aus einem Eintritts- und einem Austrittsprotokoll. Diese Protokolle gewährleisten, daß immer nur ein Thread in den Besitz des Monitors gelangen kann, d.h. der Thread, der im Besitz des Monitors ist, hat exklusiven Zugriff auf das Objekt. Java unterstützt syntaktisch diese beiden Protokolle durch das Schlüsselwort `synchronized`:

```
... ausserhalb des Monitors ...
synchronized(object)       <- hier wird das Eintrittsprotokoll fuer
{                                   das Objekt object durchgefuehrt
    ... jetzt im Monitor ...

}                          <- hier wird das Austrittsprotokoll
                                                  durchgefuehrt
... und wieder ausserhalb ...
```

Um beim Aufruf einer Methode automatisch in den Monitor des zugehörigen Objektes einzutreten, gibt es eine kürzere Schreibweise:

```
synchronized void eineMethode(...) {
    ...
}
```

Befindet sich schon ein Thread im Monitor, muß der Thread, der eintreten will, warten. Verläßt ein Thread den Monitor, darf einer der wartenden Threads in den Monitor. Um keinen Thread zu bevorteilen, kann dieser Mechanismus z.B. mit einer FIFO-Warteschlange implementiert werden.

3.3.4 Kommunikation

Zusätzlich zum Monitor besitzt jedes Objekt noch zwei Methoden, die nur von Threads aufgerufen werden dürfen, die sich im Monitor des Objektes befinden: [7]

- `wait()` bewirkt, daß der aufrufende Thread sich in die *Wartemenge* des Objektes einträgt und den Monitor verläßt. Dadurch kann ein anderer Thread den Monitor betreten.
- `notify()`[8] bewirkt, daß ein Thread aus der Wartemenge sich in die Liste der auf Eintritt wartenden Threads einträgt.

  Dementsprechend gibt es noch die Methode `notifyAll()`, die alle Threads aus der Wartemenge „wiedererweckt“.

Mit Hilfe dieser beiden Methoden können Threads miteinander kommunizieren: ein Thread, der auf das Signal eines anderen Threads warten will, ruft `wait()` eines gemeinsamen Objektes auf und kann dann mit der `notify()`-Methode durch den anderen Thread benachrichtigt werden.

[7] mit dem Ausdruck „jedes Objekt“ ist die Oberklasse aller Java-Klassen, die Klasse `Object` gemeint.
[8] die `notify()`-Methode ist in der Literatur auch als `signal()` bekannt.

In Anhang C wird beschrieben, wie sich mit diesen Methoden Kommunikationskanäle konstruieren lassen.

3.4 Pizza

Pizza erweitert Java um einige neue Sprachelemente, ohne jedoch die Kompatibilität zu reinen Java-Programmen zu verlieren. Der Pizza-Compiler erzeugt, genauso wie ein Java-Compiler, Bytecode für die JVM, d.h. bei der Ausführung von Pizzaprogrammen erkennt man keinen Unterschied zu Java-Programmen — es ist sogar möglich, den Pizza-Compiler Java-*Quell*code erzeugen zu lassen.

Im folgenden möchte ich die zwei Erweiterungen beschreiben, die ich wegen ihrer Nützlichkeit oft benutzt habe. Für eine komplette Beschreibung von Pizza sollte die Originaldokumentation zu Rate gezogen werden [Pizza].

3.4.1 Funktionale Objekte

In Java sind Methoden keine Objekte: man kann nicht Methoden als Parameter an andere Methoden übergeben. Pizza erweitert Java um einen Funktionstyp. Ein Objekt des Typs

$$(a_1, \ldots, a_n) \texttt{ -> } returntype$$

steht für eine Methode mit der Deklaration:

$$returntype \text{ MethodenName}(a_1 \text{ Arg1}, \ldots, a_n \text{ ArgN})$$

Damit ist das folgende Programm möglich:

```
class Test {
    private (int,int)->boolean vergleichsFunktion ;

    void speichereVergleichsFunktion((int,int)->boolean vf) {
        vergleichsFunktion=vf ;
    }

    boolean vergleiche(int i1,int i2) {
        return vergleichsFunktion(i1,i2) ;
    }
}
```

Partielle Applikationen sind weiterhin *nicht* möglich.

3.4.2 Typvariablen

Java fehlt die Möglichkeit, die aus C++ bekannten Patterns zu definieren. Patterns sind Klassendefinitionen mit Typvariablen. Diese Typvariablen werden zur Compilezeit aufgelöst: so als ob man die Klasse mehrmals definiert hätte, jedesmal mit anderen Werten für die Typvariablen.

An einem Beispiel läßt sich die durch die Verwendung von Typvariablen gewonnene Lesbarkeit und Sicherheit verdeutlichen. Die folgende Klasse soll einen Paartyp implementieren, zuerst in reinem Java:

```
class Paar {
    private Object obj1,obj2 ;

    void setzeWert(Object o1,Object o2) {
        obj1=o1 ; obj2=o2 ;
    }
    Object leseWert1() { return obj1 ; }
    Object leseWert2() { return obj2 ; }
}
```

Wie man sieht, muß man das Paar mit Instanzen der Klasse `Object`, die Oberklasse aller Java-Klassen, implementieren, um keine Einschränkungen auf den Typ der gespeicherten Objekte zu machen. Dies hat den Nachteil, daß bei Leseoperationen eine Typumwandlung nötig ist:

```
    Paar meinPaar=new Paar() ;

    meinPaar.setzeWert(new Kreis(),new Quadrat()) ;
    Kreis meinKreis=(Kreis)meinPaar.leseWert1() ;
                    ^^^^^^^
```

Nun die gleiche Klasse in Pizza mit Hilfe der Typvariablen `A` und `B` (angegeben in spitzen Klammern):

```
class Paar<A,B> {
    private A obj1 ;
    private B obj2 ;

    void setzeWert(A o1,B o2) {
        obj1=o1 ; obj2=o2 ;
    }
    A leseWert1() { return obj1 ; }
    B leseWert2() { return obj2 ; }
}
```

Der Gebrauch des Paares gestaltet sich nun komfortabler:

```
    Paar<Kreis,Quadrat> meinPaar=new Paar() ;

    meinPaar.setzeWert(new Kreis(),new Quadrat()) ;
    Kreis meinKreis=meinPaar.leseWert1() ;
```

Es ist auch möglich Typvariablen einzuschränken, z.B. wenn man für `A` nur die Klasse `Kreis` und deren Subklassen zulassen möchte:

```
class Paar<A extends Kreis,B> {
    ...
}
```

Kapitel 4

Implementierung eines Curry-Compilers

4.1 Entwicklung eines Ausführungsmodells für Curry-Programme

Das Ziel der Entwicklung eines Curry-Compilers war, ein System zu erhalten, das

1. die Semantik von Curry, so wie sie im bestehenden TasteCurry-System implementiert wurde, beibehält,
2. Curry-Programme mit höherer Geschwindigkeit als das TasteCurry-System ausführt und
3. in Java (bzw. in Pizza) geschrieben ist und von Threads zur Darstellung des $\wedge$-Operators Gebrauch macht.

Um die Implementierung zu vereinfachen, sollte das Einlesen der Curry-Programme und die Erzeugung der zugehörigen Definitional Trees dem TasteCurry-System überlassen werden. Dem Curry-Compiler standen dann die Definitional Trees in Form einer Textdatei vor, die nur ausgelesen zu werden braucht.

Die niedrige Ausführungsgeschwindigkeit von TasteCurry wird vor allem durch die verwendete operationale Semantik (siehe Abschnitt 2.2) verursacht: sie basiert auf Termersetzungen, d.h. ein Goal wird als ein Term aufgefaßt, der Schritt für Schritt in einen Lösungsterm umgewandelt werden soll. Dazu wird in jedem Schritt der gesamte Term nach einer geeigneten Stelle für die nächste Termersetzung durchsucht.

Um eine direktere Vorgehensweise zu finden, soll zunächst an einem Beispiel gezeigt werden, welche Operationen laut Definitional Tree bei der Auswertung eines Goal durchgeführt werden müssen. Als Beispiel nehmen wir wieder die `add`-Funktion:

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

mit dem Definitional Tree:

```
branch add(X,Y),1,flex,
```

```
  rule add(z,Y) -> Y,
  rule add(s(X),Y) -> s(add(X,Y))
```

Die Semantik verbindet mit dem `branch`-Tree eine Unterscheidung auf dem 1. Argument zwischen Variablen, Konstruktoren und Funktionen:

- **Variablen** sollen Nichtdeterminismus auslösen, da der Baum als `flex` markiert ist.
- **Konstruktoren** sollen Pattern Matching auslösen und somit eine der beiden Regeln für den nächsten Schritt auswählen.
- **Funktionen** führen zu der Auswertung des zugehörigen Definitional Trees. Aus der Sicht einer imperativen Programmiersprache stellt diese Auswertung nichts anders als einen Funktionsaufruf dar.

Daher kann der Definitional Tree der `add`-Funktion auch durch folgenden Algorithmus dargestellt werden:

```
wenn <1. Argument ist eine Funktion>
    werte Funktion aus bis das Resultat in Kopfnormalform ist
    (d.h. entweder Variable oder Konstruktor)
wenn <1. Argument ist eine freie Variable>
    binde die Variable an die moeglichen Konstruktoren
    und gehe dadurch zum naechsten Schritt:
wenn <1. Argument ist ein Konstruktor>
    Fallunterscheidung auf <Konstruktor>
        Fall 'z': gebe <Argument 2> als Ergebnis zurueck
        Fall 's':
             T:=<1. Argument des Konstruktors 's'>
             gebe 's(add(T,Argument 2))' als Ergebnis zurueck
```

Auf den ersten Blick bietet es sich natürlich an, diesen Algorithmus direkt in Quellcode einer imperativen Sprache umzuwandeln. Es darf jedoch nicht vergessen werden, daß Funktionen — und damit auch Funktionsaufrufe — in Curry Nichtdeterminus auslösen können. Z.B. im folgenden Funktionsaufruf:

```
f: Funktion f ruft g auf
   g: Funktion g ruft h auf
      h: Funktion h erzeugt Nichtdeterminismus
         Rueckkehr zur Funktion g mit den Ergebnissen A und B
   g: Rueckkehr zur Funktion f mit den Ergebnissen A'(A) und B'(B)
f: ....
```

Die meisten Hochsprachen merken sich Funktionsaufrufe auf einem sogenannten *call stack*: wird die Funktion verlassen, so kann anhand des Eintrags im call stack festgestellt werden, von welcher Stelle die Funktion ausgeführt wurde, und dort die Ausführung fortgesetzt werden. Allerdings erlauben die üblichen Programmierspachen keine Operationen auf den call stack.

Das führt dann zu einem Problem, wenn, wie beim Nichtdeterminismus, Eingriffe in den call stack nötig sind: so spaltet die Funktion `h` durch den Nichtdeterminus den Ausführungspfad auf. Unabhängig von der Implementierung des Nichtdeterminismus (z.B. durch Backtracking) muß eine Kopie des call stacks erstellt werden, um jeder Lösung von `h` zu ermöglichen, zur Funktion `g` und damit zu `f` zurückzukehren.

Aus diesem Grund können Funktionsaufrufe von Curry nicht direkt als Funktionsaufrufe in einer imperativen Sprache implementiert werden: es muß explizit ein call stack verwaltet werden, der die erforderlichen Möglichkeiten bietet.

Der Aufruf einer Funktion `f` würde also ungefähr so aussehen:

```
push <Befehlszaehler> auf eigenen call stack
springe zu Code fuer Funktion <f>
```

und die Rückkehr aus einer Funktion so:

```
<Addr>:=pop von eigenem call stack
goto <Addr>
```

Die Sache hat jedoch einen Haken: Java kennt keinen `goto`-Befehl. Es bringt auch nichts, den `goto`-Befehl durch einen Methodenaufruf zu simulieren: da dem Methodenaufruf der zugehörige Methodenausstieg fehlt, würde sich der call stack der eingesetzten Programmiersprache nach und nach füllen und nach einiger Zeit sogar überlaufen.

Aus den oben genannten Gründen kommt die direkte Übersetzung in eine Hochsprache (wie sie z.B. im `wamcc`-Projekt [3] realisiert wurde) nicht in Frage.

Daher wurde für den Curry-Compiler ein anderer Ansatz gewählt: die Definitional Trees werden nicht in eine Hochsprache übersetzt, sondern in Code für eine abstrakte Curry-Maschine. Dadurch werden die obengenannten Probleme auf einfache Weise vermieden, da man hier die volle Kontrolle über den call stack hat. Der Curry-Compiler legt dazu den Code für die abstrakte Maschine in Form eines Java-Programms ab, das anschließend von einem Java-Compiler in Bytecode übersetzt werden muß. [1]

Das Curry-System besteht also aus zwei Teilen, beide in Java geschrieben: dem Compiler, der den Code erzeugt, und einem Laufzeitsystem, das die abstrakte Curry-Maschine beinhaltet.

4.2 Darstellung der Terme

Da Java eine objektorientierte Sprache ist, ist es naheliegend, Terme durch Java-Objekte darzustellen. [2] Damit stehen dem Curry-Laufzeitsystem die Fähigkeiten der JVM, insbesondere die Garbage Collection und die Möglichkeit, Klassen zur Laufzeit zu laden, zur Verfügung.

Abbildung 4.1 zeigt die Klassen, die zur Darstellung von Termen verwendet werden.

Terme, die Argumente erhalten können (d.h. Konstruktoren und Funktionen) leiten sich aus der Klasse `NodeTerm` ab. Da die *Maximal*anzahl der Argumente solcher Terme durch ihren Typ statisch vorgegeben ist, kann man die Argumente in einem Array abspeichern:

[1] Wie wir noch sehen werden, übersetzt der Compiler zur Optimierung der Ausführungsgeschwindigkeit *doch* einige Teile der Definitional Trees direkt nach Java.

[2] es handelt sich bei der Curry-Maschine also nicht um eine echte abstrakte Maschine, da sie die Implementierung eines Heaps der JVM überläßt.

- `abstract class Term`: die Oberklasse aller Terme
  Aus ihr leiten sich ab:
  + `abstract class NodeTerm`: Oberklasse der Terme, die Argumente besitzen
    Aus ihr leiten sich ab:
    + `abstract class Constructor`: Oberklasse aller Konstruktorterme
      Aus ihr leiten sich ab:
      + `class CustomConstructor`: vom Benutzer definierte Konstruktoren
      + `class PrimConsInt`: Darstellung von Integer-Zahlen
      + `class PrimConsFileReader` und `class PrimConsFileWriter`:
        diese Klassen dienen zur Darstellung von Filedescriptor, wie sie z.B. von einer IO-Monade benötigt werden.
    + `abstract class Function`: Oberklasse aller Funktionsterme
  + `class Variable`

Abbildung 4.1: Hierarchie der Term-Klassen

```
abstract public class NodeTerm extends Term implements Cloneable{
    public int nargs=0 ;            // aktuelle Anzahl der Argumente
    private Term [] args ;          // Argumente

    public final Term getArg(int n) { return args[n] ; }
    public final void setArg(int n, Term t) { args[n]=t ; }
    public final void addArg(Term t) { setArg(nargs++,t) ; }
    ...
}
```

4.2.1 Konstruktoren

Konstruktorterme sind Instanzen einer Subklasse von `Constructor`. Um ein effizientes Patternmatching zu ermöglichen, besitzen Konstruktorterme eine Methode `int getIndex()`, die den Konstruktoren eines Typs Integer-Zahlen zuordnet. Da Curry eine streng getypte Programmiersprache ist (insbesondere ist der Typ der Argumente einer Funktion eindeutig), braucht dieser Index nur *innerhalb* eines Typs eindeutig an die Konstruktoren vergeben zu werden. Daher brauchen zur Compilezeit nicht alle zur Laufzeit benötigten Konstruktoren bekannt zu sein.

4.2.2 Funktionen

Im Gegensatz zu Konstruktoren übersetzt der Compiler jede in einem Curry-Programm definierte Funktion in eine eigene Klasse, die aus der Klasse `Function` abgeleitet wurde. Das Besondere an einer solchen Funktionsklasse ist, daß sie den (durch den Compiler) erzeugten Code für die abstrakte Curry-Maschine enthält.

Wird eine Funktion zum ersten Mal benutzt, d.h. zum ersten Mal eine Instanz der zugehörigen

Funktionsklasse erzeugt, lädt die JVM automatisch die Klasse und damit den Code für die abstrakte Curry-Maschine. Voraussetzung ist natürlich, daß ein Java-Compiler vorher den Quelltext der Funktionsklasse in eine Java-Klasse übersetzt hat. Ein ähnlicher Ansatz wurde im JProlog-Projekt, einem Übersetzer von Prolog nach Java, verfolgt [5].

Der Definitional Tree einer Funktion `'Name'` wird also in eine Klasse namens `'Function_Name'` übersetzt, wobei das `code`-Feld der Klasse den Code für die abstrakte Maschine enthält:

```
public class Function_Name extends Function {
    ...
    private static final Command code[]={
        ...Code f"ur Curry-Maschine...
    } ;
}
```

Hat man dann eine Funktionsapplikation t der Funktion `Name` (d.h. t ist eine Instanz der Klasse `Function_Name`), kann man einfach auf den zugehörigen Code zugreifen:

$$t\texttt{.code}$$

Da der Code für alle Instanzen der Funktion gleich ist, wurde `code` als `static` deklariert (siehe Abschnitt 3.1.3).

4.2.3 Variablen

Instanzen der Klasse `Variable` stellen logische Variablen dar. Dabei wird während eines Programmablaufs jeder vorkommenden logischen Variablen jeweils nur ein solches Objekt zugeordnet. Die Abbildung 4.2 veranschaulicht das nochmal am Term `reverse(cons(X,cons(X,nil)))`

4.3 Die abstrakte Maschine

4.3.1 Darstellung des Codes

Befehle für die abstrakte Maschine werden durch Instanzen von Subklassen der Klasse `Command` dargestellt:

```
abstract public class Command {
    abstract public void execute(Machine machine) ;
}
```

Wie man sieht, besitzt jeder Befehl eine Methode `execute`. Diese Methode wird von der Maschine aufgerufen, wenn sie den Befehl auswerten möchte.

Eine Folge von Befehlen wird in einem Array des Typs `Command[]` abgespeichert. Die Maschine kann also ihren Programcounter (siehe unten) als Index im Array benutzen, um den nächsten auszuführenden Befehl zu ermitteln.

Beispiel 4.1 zeigt einen einfachen Befehl: `CmdJump`. Dieser Befehl tut nichts anderes als den Programcounter der Maschine auf einen neuen Wert zu setzen. Mit diesem Befehl kann auf einfache Weise eine Endlosschleife produziert werden:

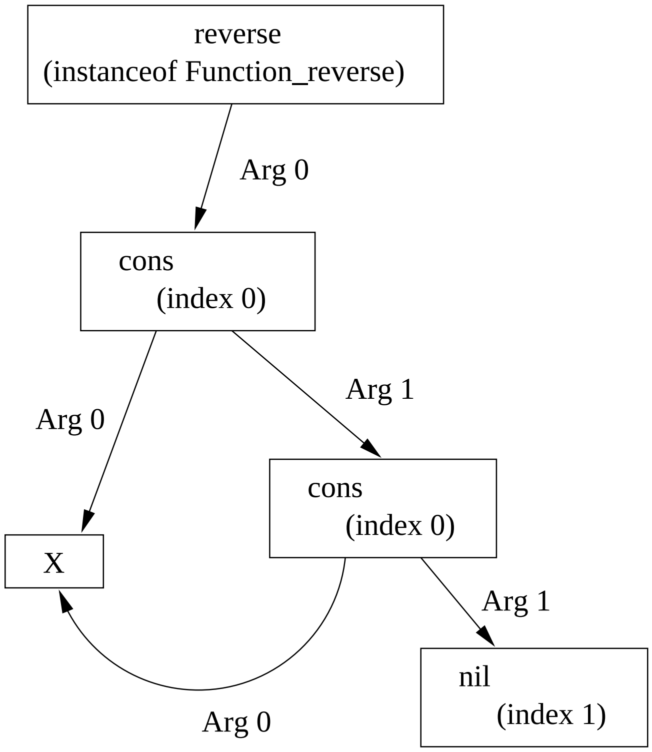

Abbildung 4.2: Term `reverse(cons(X,cons(X,nil)))`

```
public class CmdJump extends Command {
    private int index ;
    public CmdJump(int index) { this.index=index ; }

    final public void execute(Machine machine) {
        machine.setPC(index) ;
    }
}
```

Beispiel 4.1: Jump-Befehl

```
Command[] code={ new CmdJump(0) } ;
```

4.3.2 Aufbau und Arbeitsweise der Maschine

Der Einfachheit halber soll der Umgang mit logischen Variablen erst im nächsten Abschnitt untersucht werden. Hier beschäftigen wir uns nur mit rein funktionalen Programmen. An dem Beispielalgorithmus auf Seite 45 kann man schon gut erkennen, nach welchem Schema der Compiler Code erzeugen sollte:

- Ein `rule LHS -> RHS` wird übersetzt zu:

```
Result:=neuen Term RHS aufbauen (mit den Argumenten aus LHS)
return Result
```

- Ein `branch PATTERN,POS,MODE,SUBTREES` wird übersetzt zu:

```
T:=Term an Position POS in PATTERN
Test von T auf Funktion oder Konstruktor
Wenn Funktion:
    aufrufen bis Kopfnormalform
Wenn Konstruktor:
    Je nach Konstruktor passenden SUBTREE auswaehlen
    Fuer jeden SUBTREE:
        Konstruktor in Argumente zerlegen
        Code fuer Subtree erzeugen
```

Benötigt werden also Befehle zum Aufruf einer Funktion, zur Konstruktion neuer Terme, zum Verlassen einer Funktion (mit Ergebnis),...

Die Maschine enthält nun folgende Elemente:

Argumentregister: die Argumentregister nehmen bei der Abarbeitung einer Funktion die Argumentterme der Funktion auf. Der Compiler erzeugt aus den Argumentvariablen auf der rechten Seite einer Regel Zugriffe auf die Argumentregister.

Programcounter: der Programcounter besteht aus einem Paar: das erste Element des Paares verweist auf die Funktion, die die Maschine gerade bearbeitet. Mit dieser Information kann die Maschine auf das `code`-Feld (siehe Abschnitt 4.2.2) der zugehörigen Funktion zugreifen.

Das zweite Element dient dann als Index innerhalb des Codes. Das Index-Element wird nach jedem Befehl erhöht und kann durch Sprungbefehle verändert werden.

Um eine Funktion aufzurufen, setzt man das erste Element auf die Funktion und das Indexelement auf 0. Damit führt dann die Maschine den ersten Befehl der Funktion aus.

call stack: bei einem Funktionsaufruf wird ein Eintrag auf dem call stack erzeugt. Dieser Eintrag enthält folgende Informationen:

1. einen Verweis auf den vorherigen Eintrag (der call stack ist als einfach verkettete Liste implementiert).
2. eine Sicherheitskopie der Argumentregister.
3. einen Verweis auf den Befehl, der nach Beendigung der Funktion ausgeführt werden soll (üblicherweise *Rückkehradresse* genannt).
4. in welches Register das Ergebnis des Funktionsaufrufes geschrieben werden soll.

Wenn die Auswertung des Funktionsaufrufs beendet ist, stehen diese Informationen wieder auf der Spitze des call stack zur Verfügung.

Temporäre Register: werden benutzt, um während des Patternmatchings Konstruktoren in ihre Argumente zu zerlegen.

Zusätzlich besitzt die Maschine einen Termstack, mit dem auf einfache Weise neue Terme konstruiert werden können (für rechte Regelseiten). Dieser wird jedoch, wie wir im Abschnitt 4.7 sehen werden, selten benutzt.

Anhand des vom Compiler für das `add`-Beispiel erzeugten Codes in Beispiel 4.2, soll der Befehlssatz der Maschine erläutert werden [3] (die Maschine lädt beim Aufruf der Funktion das 1. Argument in Register 0 und das 2. Argument in Register 1):

0 `CmdSwitchOnType(0,11)`:
falls das 1. Argument (Register 0) eine Funktion ist, wird zu Befehl 11 gesprungen. Ansonsten (Fall Konstruktor) macht die Maschine mit dem nächsten Befehl weiter.

1 `CmdSwitchOnCons(0,{2,4})`:
dieser Befehl benutzt die in Abschnitt 4.2.1 erwähnte `getIndex`-Methode der Konstruktoren um eine Sprungtabelle anzusprechen: Befehl 2 wird für den Fall 'z' und Befehl 4 für den Fall 's' angesprungen.

Falls die Sprungtabelle den Eintrag -1 enthält, wird beim entsprechenden Konstruktor ein Fehlschlag ausgelöst. Dies ist z.B. der Fall bei Funktionen, die nicht über alle Konstruktoren eines Typs definiert sind. Bei einem Fehlschlag terminiert die Maschine ohne Rückgabe eines Ergebnisses.

2,3 `CmdPushReg(1)` und `CmdReturn()` : der erste Befehl schiebt das 2. Argument auf den Termstack, wo es dann von `CmdReturn` als Rückgabewert der Funktion benutzt wird.

4,5 `CmdLoadArgs(0)` und `CmdLoadReg(0,0)`: das 1. (und einzige Argument) des Konstruktors 's' wird in Register 0 geladen.

6,7,8,9,10
```
CmdPushReg(0)
CmdPushReg(1)
CmdPushFunc("add",2)
CmdPushCons("s",1)
CmdReturn()
```

Diese Befehle bauen den Term `s(add(X,Y)` auf dem Termstack auf und geben ihn als Ergebnis der Funktion zurück.

[3] in Anhang D wird der komplette Befehlssatz vorgestellt. Anhang E beschreibt die Codeerzeugung.

```
static final Command code[]={
    new CmdSwitchOnType(0,11),
    new CmdSwitchOnCons(0,{2,4}),
    new CmdPushReg(1),
    new CmdReturn(),
    new CmdLoadArgs(0),
    new CmdLoadReg(0,0),
    new CmdPushReg(0),
    new CmdPushReg(1),
    new CmdPushFunc("add",2),
    new CmdPushCons("s",1),
    new CmdReturn(),
    new CmdNewCall(0,2,0),
}
```

Beispiel 4.2: Code für `add`-Funktion

11 `new CmdNewCall(0,2,0)`

Durch diesen Befehl wird die Funktion aus Register 0 aufgerufen. Vorher wird allerdings ein Eintrag auf dem call stack erzeugt, der die beiden ersten Register sichert. Nach Beendigung der Funktion, wird die Ausführung wieder mit Befehl 0 fortgesetzt.

4.3.3 Funktionsaufrufe

Aus der funktionalen Sichtweise sind die beiden wichtigsten Befehle in dem obigen Beispielprogramm wohl `CmdNewCall` und `CmdReturn`. Deshalb soll ihre Funktionsweise hier nochmal genauer erläutert werden.

Zuallererst soll festgehalten werden, daß Curry eine *lazy*-Sprache ist: Funktionen werden nur wenn nötig und nur einmal ausgewertet. Hat man z.B. eine Funktion `double`

```
double(X)=add(X,X)
```

so wird bei einem Funktionsaufruf der Form `double(f(...))` das Argument `f(...)` *nicht* zweimal ausgewertet. Ein Funktionsobjekt weiß also, ob es schonmal ausgewertet wurde, und liefert in diesem Fall einfach das schon vorhandene Ergebnis. Dazu erweitert man Funktionsobjekte um eine Referenz auf das Ergebnis:

```
class Function {
   ...
   Term result ;
   ...
}
```

Falls `result==null`, weiß man, daß die Funktion noch nicht ausgewertet wurde.

Nun aber zurück zu den Befehlen `CmdNewCall` und `CmdReturn`:

- der Befehl `CmdNewCall(freg,nregs,ra)` überprüft, ob die Funktion im Register `freg` schon ausgwertet wurde. Falls ja, wird das Ergebnis in `freg` geschrieben und zu `ra` gesprungen. Im anderen Fall wird ein neuer Eintrag auf dem call stack erzeugt, der die Register 0 bis `nregs`-1 und die Argumente `freg` und `ra` speichert. Anschließend springt die Maschine zum ersten Befehl der auszuwertenden Funktion.

- das Gegenstück dazu ist der Befehl `CmdReturn`. Er nimmt den obersten Eintrag vom call stack runter und benutzt die darin gespeicherten Informationen `freg`, `nregs` und `ra`: als erstes werden die Register 0 bis `nregs`-1 restauriert. Der oberste Term des Termstacks wird als Ergebnis angesehen und in die Funktion aus Register `freg` gespeichert. Anschließend überschreibt die Maschine das Register `freg` mit dem Ergebnis und springt zu `ra`.

4.3.4 Die `Eval`-Funktion

Wenn der Benutzer nun einen Term auswerten möchte (z.B. „`add(s(z),z)`") führt das Laufzeitsystem folgende Schritte durch:

1. Der auszuwertende Term wird erzeugt. Da der Term eine Instanz der Klasse `Function_add` enthält, wird die Klasse von der JVM geladen.

2. Die abstrakte Maschine wird gestartet. Dazu wird der Programcounter auf den ersten Befehl der auszuwertenden Funktion gesetzt und die Argumente der Funktion in die Argumentregister geladen. Dann tritt die Maschine in eine Schleife ein, die mit Hilfe des Programcounters einen Befehl nach dem anderen ausliest und ausführt.

3. Die Auswertung ist zu Ende, wenn die Maschine versucht einen `CmdReturn`-Befehl mit einem leeren call stack auszuführen (der leere call stack entspricht dem Anfangszustand der Maschine). Wenn die Maschine die Auswertung beendet hat, steht das Ergebnis im Resultatsfeld der Maschine (abgesehen von einem Fehlschlag).

Wie man leicht erkennt, führt die obige Vorgehensweise allerdings noch zu keinem Ergebnis: die Auswertung des Terms `add(s(z),z)` ergibt den Term `s(add(z,z))`. Außerdem ist nicht der Fall berücksichtigt worden, daß der Benutzer möglicherweise gar keine Funktion als obersten Term eingibt, z.B. `s(add(z,z))`.

Aus diesem Grund wird um jeden vom Benutzer eingegebenen Term eine weitere Funktion, die `Eval`-Funktion, gelegt. Diese Funktion lokalisiert Funktionsterme im Goal und ruft sie auf. Erst wenn keine Funktionen mehr im Goal vorkommen oder die Berechnung suspendiert (siehe unten), wird das endgültige Ergebnis ausgegeben (vergleichbar mit der *cst*-Funktion aus der operationalen Semantik in Abschnitt 2.2).

4.4 Logische Variablen und Nichtdeterminismus

Das Auftreten von logischen Variablen wirkt sich dann aus, wenn eine (ungebunden) logische Variable das Argument eines Patternmatching-Schrittes ist. Die Semantik von Curry verlangt, daß in diesem Fall

1. als `flex` markierte `branch`-Bäume zu Nichtdeterminismus führen, d.h. statt einer Lösung soll für jeden, an dieser Stelle des Pattern-Matching möglichen Konstruktor eine eigene Lösung generiert werden.

2. als `rigid` markierte `branch`-Bäume suspendieren.

Die Suspendierung einer Berechnung wird erst unter Zunahme des Und-Operators interessant und daher erst im nächsten Abschnitt besprochen. Vorläufig soll für uns eine suspendierte Berechnung zwar erfolgreich sein, aber als Ergebnis einen unvollständig ausgewerteten Term (d.h. einen Term, der noch Funktionsaufrufe enthält) zurückliefern.

Die Aufspaltung des Berechnungspfades (wie sie beim Nichtdeterminismus nötig ist) kann sehr einfach in Java implementiert werden: jede Curry-Maschine wird durch einen eigenständigen Thread realisiert. Löst eine Maschine Nichtdeterminismus aus, genügt es, die aktuelle Maschine (d.h. den aktuellen Thread) zu kopieren: jede der Maschinen geht anschließend einen der möglichen Lösungswege nach.

Da wir auf ein zeitraubendes Kopieren von Termen verzichten wollen, ist die wichtigste Voraussetzung, daß keine Maschine bestehende (mit anderen Maschinen geteilte) Terme modifiziert, sondern nur neue Terme konstruiert. Da Curry eine seiteneffektfreie Sprache ist, stellt diese Bedingung im Falle von Konstruktoren kein Problem. Allerdings bleibt noch die Frage, wie Funktionen ihre Ergebnisse speichern und Variablen ihre Bindungen darstellen. So muß der Aufruf

```
add(X,z)
```

zwei Maschinen erzeugen, die mit *verschiedenen* Variablenbindungen die Funktion `add` auswerten: die erste mit `X=z` und die zweite mit `X=s(T)` (wobei `T` eine neue Variable ist).

Eine mögliche Lösung soll im nächsten Abschnitt besprochen werden.

4.4.1 Die Berechnungsstruktur

Als erstes wird jede Maschine in eine sogenannte *Berechnungsstruktur* eingebettet. Löst eine Maschine Nichtdeterminismus aus, so wird ihre Berechnungsstruktur (einschließlich der Maschine) kopiert: man erhält eine neue Maschine, die einen anderen Berechnungspfad einschlagen kann.

Jede Berechnungsstruktur besitzt eine eindeutige natürliche Zahl (im folgenden *key* genannt). Weiterhin besitzen alle Variablenobjekte eine *Bindungstabelle*:

```
class Variable {
    BindingTable bindingtable ;
    ...
}
```

Diese Bindungstabelle kann als eine Abbildung $Key \rightarrow Term$ angesehen werden. Eine Berechnungsstruktur kann mit ihrem `key`-Eintrag bei der Variablen ihre „Version" der Bindung anfragen (Abbildung 4.3. Eine Übersicht über verschiedene (andere) Implementierungstechniken geben [7] und [21]).

Die Bindungstabelle wird durch eine Hashtabelle realisiert, was natürlich zur Folge hat, daß Variablen nicht in konstanter Zeit dereferenziert werden können. Die Hashtabelle ist jedoch

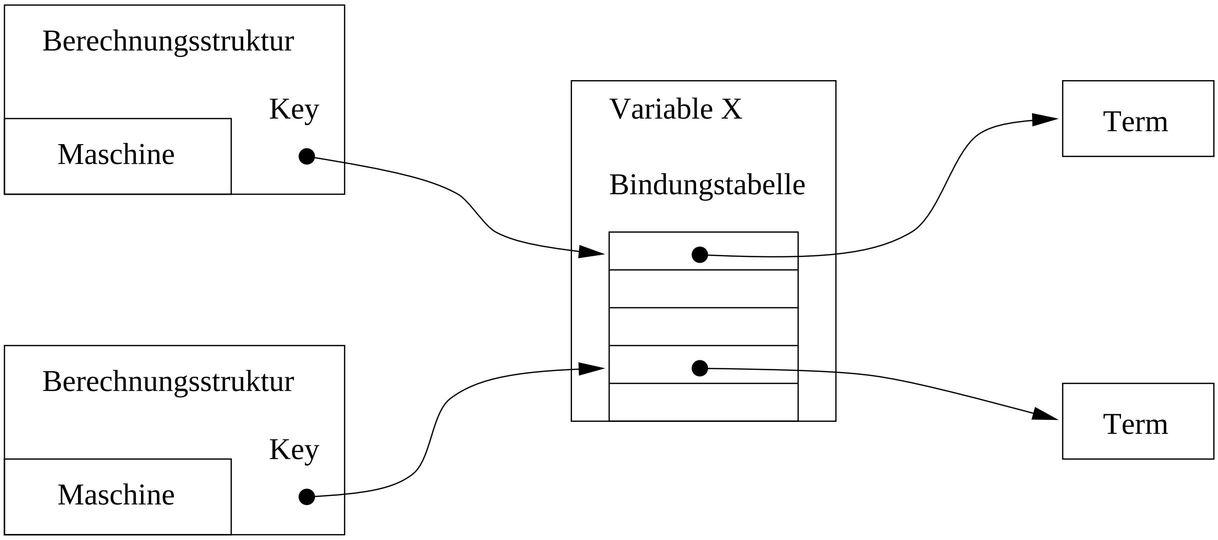

Abbildung 4.3: Die Berechnungsstruktur und die Bindungstabelle

auf eine geringe Anzahl von Einträgen optimiert: in diesem Fall (d.h. bei fast rein funktionalen Programmen) ist die Zugriffszeit fast konstant. [4]

Zwischen Berechnungsstrukturen existiert eine Ordnungsrelation: eine Berechnungsstruktur B, die von einer Struktur A erzeugt wurde, darf alle Variablenbindungen von A lesen, die schon zu dem Zeitpunkt seiner Erzeugung existierten. Diese Eigenschaft wird wie folgt implementiert:

- jede Berechnungsstruktur besitzt einen Zeiger auf die erzeugende Berechnungsstruktur.
- jede Berechnungsstruktur besitzt eine Zeitangabe über ihre Erzeugung (`birthTimeStamp`) und die (persönliche) aktuelle Zeit (`timeStamp`).

  Die Erzeugung einer neuen Berechnungsstruktur `newComp` durch die Berechnungsstruktur `comp` sieht dann so aus:

  ```
  newComp.timeStamp=comp.timeStamp ;
  newComp.birthTimeStamp=comp.timeStamp ;
  comp.timeStamp++ ;
  ```

- jeder Eintrag in die Bindungstabelle einer Variablen wird mit der aktuellen Zeit `timeStamp` der Berechnungsstruktur „signiert“.

Wenn also eine Berechnungsstruktur B eine Variable dereferenzieren möchte, können zwei Fälle eintreten:

- die Bindungstabelle der Variablen besitzt für B einen Eintrag: die Variable ist gebunden.
- die Bindungstabelle besitzt für B keinen Eintrag: in diesem Fall wird die Dereferenzierung nochmal gestartet, aber diesmal mit dem *key* der Vorgängerstruktur (falls vorhanden).

[4] Erreicht die Hashtabelle einen bestimmten Füllgrad, so wird sie automatisch vergrößert.

Wenn sich damit immer noch kein Eintrag finden läßt, ist die Variable tatsächlich aus der Sicht von B ungebunden. Falls doch ein Eintrag gefunden wird, muß mit Hilfe des `timeStamp` des Eintrags entschieden werden, ob dieser Eintrag überhaupt für B gültig ist: es muß nämlich gelten

$$\texttt{entry.timeStamp} \leq B\texttt{.birthTimeStamp}$$

Damit die Dereferenzierung beim nächsten Mal schneller geht, wird die so gefundene Bindung nochmals durchgeführt, aber diesmal für B (aber mit dem gleichen Zeitstempel `entry.timeStamp`).

Die gleichen Techniken werden benutzt, um die Ergebnisse von Funktionsaufrufen in die zugehörigen Funktionsobjekte zu speichern.

4.4.2 Erweiterung der Maschine

Die Maschine wird nun um zwei Befehle erweitert, die

1. eine neue Berechnungsstruktur erzeugen:

   `CmdNewChoice(Programcounter der neuen Maschine,`n`)`

   n gibt an, wieviele Register der bestehenden Maschine in die neue Maschine kopiert werden sollen. Für die Darstellung von or-Knoten im Definitional Tree existiert ein ähnlicher Befehl names `CmdOr`.

2. Variablen binden:

   `CmdSimpleBind(Register)`

Schließlich bekommt der Befehl `SwitchOnType` einen dritten Parameter, der als Sprungziel benutzt wird für den Fall, daß das Argument eine Variable ist:

`CmdSwitchOnType(Register,Sprungziel Funktion,Sprungziel Variable)`

Analog dazu existiert der Befehl `SwitchOnRigid`, der im Falle einer Variable eine Suspendierung der Maschine veranlaßt.

Um den Code der `add`-Funktion (Beispiel auf Seite 52) um die nötigen Befehle für Nichtdeterminismus zu erweitern, schauen wir uns nochmal den Algorithmus auf Seite 45 an: im Falle einer Variablen soll der branch-Baum alle an dieser Stelle möglichen Bindungen erzeugen und mit dem Fall `'ist Konstruktor'` fortfahren (die Variable ist ja nun an einen Konstruktor gebunden). Der zugehörige Code beginnt ab Befehl 14 und wird durch den neuen `SwitchOnType`-Befehl angesprungen (zur Lesbarkeit ohne `'new Cmd...'` vor den Befehlen):

```
    SwitchOnType(0,11,14),
    ...
    NewChoice(18,2),            // Kopie der Maschine an Befehl 18 starten
// 1. Moeglichkeit:
    PushCons("z",0),
```

```
    SimpleBind(0),              // die Variable (Register 0) an 'z' binden
    Jump(2),                    // und zum passenden Konstr.-Fall springen
// 2. Moeglichkeit (fuer die neue Maschine):
    PushVar(),                  // frische Variable T erzeugen
    PushCons("s",1),
    SimpleBind(0),              // Register 0 and 's(T)' binden
    Jump(4)                     // und zum passenden Konstr.-Fall springen
```

4.4.3 Lösungsausgabe

Die Konsequenz aus dem oben besprochenen ist, daß zwar bei Eingabe eines Goals nur eine Berechnungsstruktur erzeugt wird, durchaus aber mehrere Berechnungsstrukturen mit einer Lösung terminieren. Um die laufenden Berechnungsstrukturen bei der Ausgabe einer Lösung untereinander zu synchronisieren, wird die Benutzerschnittstelle mit Hilfe eines eigenen Threads realisiert (im Folgenden einfach *Hauptthread* genannt).

Der Hauptthread wartet auf die Eingabe des Benutzers. Ist ein Term eingegeben worden, wird eine Berechnungsstruktur erzeugt, die mit der Auswertung des Terms beginnt. Daraufhin begibt sich der Hauptthread in eine Empfangsschleife und wartet auf Nachrichten der erzeugten Berechnungsstrukturen:

- die Nachricht **CompCreated** zeigt an, daß eine neue Berechnungsstruktur erzeugt wurde.
- fehlgeschlagene Berechnungen melden sich mit **CompFailed**.
- Lösungsterme werden mit der Nachricht **SolutionFound** an den Hauptthread überreicht und von ihm ausgegeben.

Der Hauptthread verläßt die Empfangsschleife erst dann, wenn genausoviele Berechnungsstukturen terminiert haben, wie erzeugt worden sind. Um eine zügige Abwicklung von terminierten Berechnungsstrukturen zu ermöglichen, hat der Hauptthread eine höhere Ausführungspriorität als die Maschinenthreads.

Eine Folge der Implementierung des Nichtdeterminismus durch Threads ist die Tatsache, daß nichtterminierende Suchpfade *nicht* die Ausgabe von anderen Lösungen behindern können. Im Gegensatz zum Backtracking-Verfahren, wie es zum Beispiel in Prolog [14] implementiert wurde, ist hier jede Alternative gleichberechtigt.

4.5 Der Und-Operator

Im Abschnitt 2.1.8 wurde beschrieben, wie sich mit Hilfe von $\wedge$-Constraints zu Konjunktionen zusammenfassen lassen. Während die operationale Semantik aus Abschnitt 2.2.4 eine sequentielle Ausführung der Konjunktion beschreibt, bietet es sich natürlich auch hier an, zur Implementierung auf Java-Threads zurückzugreifen.

Daher ist die Aussage, daß mit jeder Berechnungsstruktur *eine* Maschine assoziert wird, nicht mehr korrekt: jede Berechnungsstruktur „beschäftigt“ mehrere Maschinen, die parallel die Constraints der erzeugten Konjunktionen auswerten.

4.5.1 Der Und-Baum

Um die und-parallel arbeitenden Maschinen zu verwalten, wird die Berechnungsstruktur um eine neue Datenstruktur erweitert: den Und-Baum. Der Und-Baum ist eine Reprsentation der Konjuktionen und wie folgt definiert:

- die **Blätter** des Baumes stellen die arbeitenden Maschinen dar. Jeder dieser Maschinen ist mit der Auswertung eines Constraints beschäftigt.
- die **inneren Knoten (Und-Knoten)** des Baumes (d.h. alle Knoten des Baumes mit Ausnahme der Blätter) stellen die Anwendung des $\wedge$-Operators dar. Die beiden Söhne solch eines Knotens reprsentieren die beiden Argumente des $\wedge$-Operators.

Erzeugung eines Und-Knotens

Ruft ein Blatt (d.h. eine Maschine) den $\wedge$-Operator auf, z.B. in Form von $t_1 \wedge t_2$, so verwandelt sich das Blatt in einen inneren Knoten, indem folgende Schritte durchgeführt werden:

1. das Blatt wird durch einen inneren Knoten ersetzt. In diesem Knoten wird der aktuelle Zustand der Maschine abgespeichert. Dieser Zustand wird später nach erfolgreicher Bearbeitung der Konjunktion dazu benutzt, die Abarbeitung der Funktion, die die Konjunktion ausgelöst hat, wieder aufzunehmen.
2. der neue Knoten erhält zwei Blätter als Söhne. Das linke Blatt beginnt mit der Abarbeitung des Argumentes t_1, das rechte mit der Abarbeitung von t_2.

   Natürlich brauchen für die Blätter nicht zwei neue Maschinen erzeugt zu werden: die Maschine des ursprünglichen Blattes kann für eines der neuen Blätter „wiederverwertet“ werden.

Als Beispiel betrachten wir die Funktion `f`:

```
f(X,Y) if {X=3,Y=5} = g(X)
```

Der Ausdruck `f(3,5)` veranlaßt das System, zuerst die Konjunktion {`X=3,Y=5`} auszuwerten. Abbildung 4.4 zeigt, wie sich der Und-Baum für `f(3,5)` aufspaltet.

Entfernung eines Und-Knotens

Ein Und-Knoten wird erst dann wieder entfernt, wenn beide Söhne erfolgreich waren. In diesem Fall verwandelt sich der Und-Knoten wieder in ein Blatt und setzt die Bearbeitung mit der Funktion fort, die den $\wedge$-Operator aufgerufen hat. In unserem Beispiel würde die Ausführung mit der rechten Seite der Funktionsdefinition von `f`, also `g(X)`, fortfahren (Abbildung 4.5).

Fehlschlag

Aus der Semantik des $\wedge$-Operators geht hervor, daß die gesamte Berechnungsstruktur fehlschlägt, wenn eines der Blätter des Und-Baumes fehlschlägt.

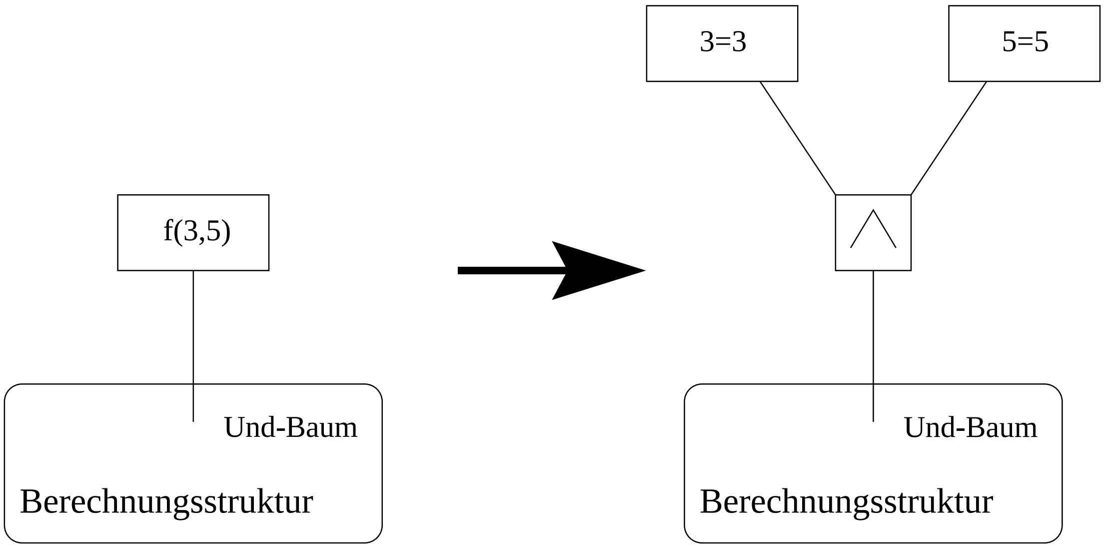

Abbildung 4.4: Auswertung der Funktion `f(3,5)` mit`f(X,Y) if {X=3,Y=5} = g(X)` (Teil 1)

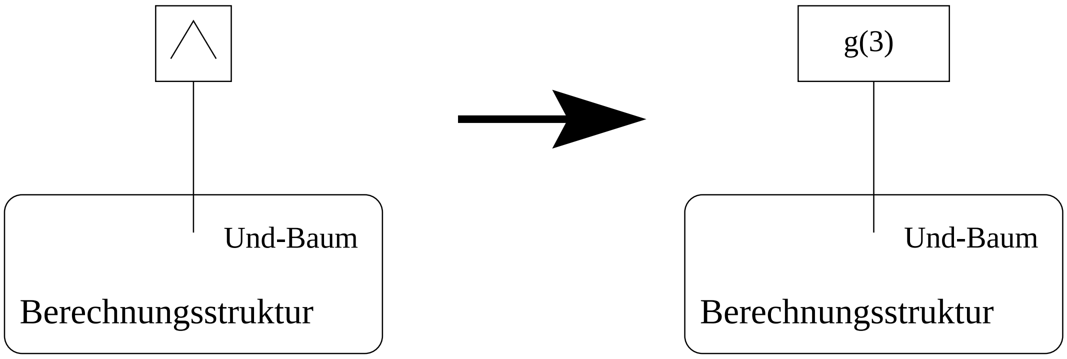

Abbildung 4.5: Auswertung der Funktion `f(3,5)` mit`f(X,Y) if {X=3,Y=5} = g(X)` (Teil 2)

4.5.2 Die Gleichheit

In den obigen Beispielen habe ich die bindende Gleichheitsfunktion `=` verwendet, ohne zu erklären, wie sie implementiert wurde. Das will ich nun nachholen, zumal eine rekursive Definition explizit über alle (!) Konstruktoren (wie in Abschnitt 2.1.8) natürlich nicht in Betracht kommt.

Die wichtigste Voraussetzung ist natürlich, daß sich beide Argumente in Kopfnormalform befinden. Das läßt sich mit dem Befehl `SwitchOnType` der Maschine erreichen. Es bleiben dadurch nur noch vier Fälle zu berücksichtigen (c und d sind Konstruktoren, X ist eine ungebundene Variable):

1. c = d ist genau dann erfolgreich, wenn c und d identisch sind.
2. Die Anwendung der Gleichheit hat die Form

   $$c(a_1, \ldots, a_m) \texttt{ = } d(b_1, \ldots, b_n)$$

   Sind c und d verschieden, so schlägt die Gleichheit sofort fehl. Im anderen Fall, d.h.

   $$c(a_1, \ldots, a_n) \texttt{ = } c(b_1, \ldots, b_n)$$

   wird die Gleichheit umgeformt zu:

   $$a_1\texttt{=}b_1 \ \wedge \ \ldots \ \wedge \ a_n\texttt{=}b_n$$

3. X = c ist erfolgreich und bindet X an c.
4. Die Anwendung der Gleichheit hat die Form

   $$X \texttt{ = } c(a_1, \ldots, a_n)$$

   Da die Argumente von c noch Funktionsaufrufe enthalten können, müssen auch diese ausgewertet werden. Dies geschieht, indem man X an $c(Y_1, \ldots, Y_n)$ bindet ($Y_1, \ldots, Y_n$ sind neue Variablen) und die Gleichheit umformt zu

   $$Y_1\texttt{=}a_1 \ \wedge \ \ldots \ \wedge \ Y_n\texttt{=}a_n$$

   Das ist natürlich im allgemeinen nicht sonderlich effizient und bietet Möglichkeiten zur Optimierung: falls sich a_i schon in Kopfnormalform befindet, ist die Erzeugung der Variablen Y_i eigentlich gar nicht nötig.

   Die Gleichheit führt vorher einen sogenannten *occur check* durch, d.h. schlägt fehl, falls die Variable X in $c(a_1, \ldots, a_n)$ vorkommt.

Die testende Gleichheit `==` wurde ähnlich implementiert, allerdings suspendiert sie, wenn sie auf ungebundene Variablen angewendet wird.

4.5.3 Erweiterung um Oder-Parallelität (Nichtdeterminismus)

Die Vorgehensweise aus dem vorigen Abschnitt zum Nichtdeterminismus kann beibehalten werden: falls eine Variable an mehrere Konstruktoren gebunden werden kann, werden Kopien der Berechnungsstruktur erzeugt. Jede dieser neuen Berechnungsstrukturen verfolgt dann den jeweiligen Berechnungspfad, der sich durch die Variablenbindung ergibt.

Da die Berechnungsstruktur nun auch den Und-Baum enthält, kann die oben genannte Kopieroperation (zeitlich) sehr aufwendig werden. Deshalb wurden für nebenläufige Implementierungen von (funktional–) logischen Programmiersprachen sogenannte Stabilitätsregeln [13] eingeführt:

1. Nichtdeterminismus wird erst dann ausgelöst, wenn es die einzige Möglichkeit ist, die Ausführung fortzusetzen. Konkret bedeutet das, daß ein Constraint nichtdeterministische Variablenbindungen erst dann erzeugen darf, wenn alle anderen (und-parallel) ausgeführten Constraints erfolgreich terminiert haben.
2. Deterministische Variablenbindungen dürfen direkt ausgeführt werden, auch wenn noch andere Constraints und-parallel ausgeführt werden.

Diese beiden Regeln besagen, daß deterministische Ausführungspfade bevorzugt werden. Nichtdeterministische werden so weit wie möglich verzögert und können sich dadurch als überflüssig herausstellen, wie man am folgenden Beispiel erkennt. Der Aufruf

$$\texttt{f(X)} \wedge \texttt{X=3}$$

mit

```
eval f 1:flex.

f(3)={}.
f(5)={}.
```

erzeugt *keinen* Nichtdeterminismus: der Constraint `X=3` wird bevorzugt ausgewertet, wodurch der Constraint `f(X)` zu `f(3)` umgeformt und damit deterministisch wird.

Zur Implementierung dieses Verfahrens unterscheidet der Und-Baum zwischen **aktiven** und **suspendierten** Blättern:

- Blätter (d.h. Maschinen), die einen Constraint auswerten, sind **aktiv**.
- ein aktives Blatt, das eine Variable *nicht*deterministisch binden will,
  - kann dies tun, wenn kein anderes aktives Blatt im Baum existiert
  - **suspendiert** im anderen Fall.

Suspendierte Blättern können aktiviert werden, wenn

- keine aktiven Blätter mehr existieren. In diesem Fall wird eines der suspendierten Blätter reaktiviert.

- eine Variable gebunden wurde. In diesem Fall werden alle suspendierten Blätter reaktiviert, die versucht hatten, diese Variable nichtdeterministisch zu binden.

Dagegen erzeugt der Ausdruck

$$\texttt{f(X)} \wedge \texttt{Y=3}$$

wohl Nichtdeterminismus, aber wiederum erst, wenn der deterministische Teil (`Y=3`) terminiert hat (Abbildung 4.6).

4.5.4 Rigid-Funktionen

Auf den ersten Blick mag die Namensähnlichkeit zwischen suspendierten Blättern aus dem obigen Abschnitt und suspendierten Funktionen aus Abschnitt 2.2 verwirrend wirken: in der operationalen Semantik von Curry wurde der Ausdruck „suspendiert“ nur für rigid-Funktionen gebraucht, die als Argument eine (freie) logische Variablen erhalten haben. Es ist aber so, daß sich das Konzept der suspendierten Blätter leicht um die korrekte Verarbeitung von rigid-Funktionen erweitern läßt.

Damit der Und-Baum auch rigid-Funktionen korrekt verarbeitet, wird die Definition des **suspendierten Blattes** erweitert um:

- versucht eine Maschine mit Hilfe des Befehls `CmdSwitchOnRigid` (Abschnitt 4.4.2) auf eine (freie) logische Variable zuzugreifen, so suspendiert das zugehörige Blatt.

Im folgenden nenne ich Blätter, die durch Nichtdeterminismus suspendiert sind, *suspendierte flex-Blätter*, und Bätter, die aufgrund einer fehlenden Variablenbindung (rigid-Funktionen) suspendiert sind, *suspendierte rigid-Blätter*.

Natürlich gilt auch weiterhin, daß ein suspendiertes Blatt durch die Bindung der entsprechenden Variablen reaktiviert wird. Im Gegensatz zu suspendierten flex-Blättern können allerdings suspendierte rigid-Blätter *nicht* durch die Tatsache, daß kein aktives Blatt mehr existiert, reaktiviert werden: suspendierte rigid-Blätter brauchen *immer* eine Variablenbindung um reaktiviert zu werden. Dadurch wird die gewünschte Synchronisation zwischen Bindungserzeugern (flex-Funktionen) und Bindungskonsumenten (rigid-Funktionen) erreicht.

Falls kein aktives Blatt mehr existiert, aber der Baum nur noch aus suspendierten rigid-Blättern besteht, kann die Auswertung nicht mehr fortgesetzt werden: die gesamte Berechnungsstruktur suspendiert.

Hier noch ein Beispiel (sei `f` eine nichtdeterministische boolesche Funktion). Sei der Ausdruck

$$C_1 \wedge C_2 \wedge C_3$$

mit den Constraints

$$\begin{aligned} C_1 &:= \texttt{Y=3} \\ C_2 &:= \texttt{f(X)=true} \\ C_3 &:= \texttt{(Z==5)=true} \end{aligned}$$

Bei diesem Aufruf suspendieren die Constraints C_2 und C_3: C_2 wegen Nichtdeterminismus, C_3 wegen fehlender Bindung von `Z`. Nachdem C_1 erfolgreich terminiert hat, wird C_2 reaktiviert

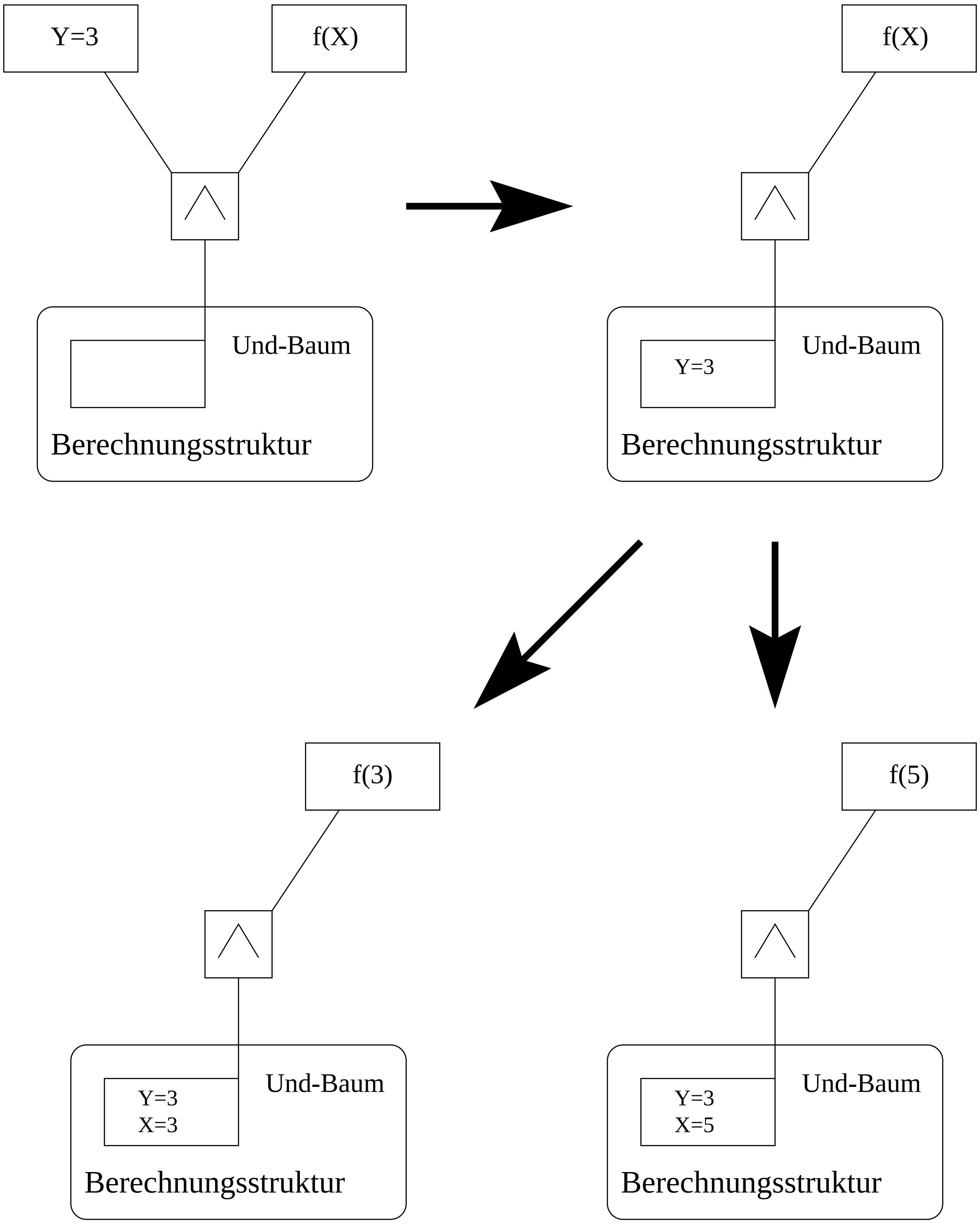

Abbildung 4.6: Auswertung des Ausdruckes `Y=3` $\wedge$ `f(X)` mit `f` nichtdeterministisch

(C_3 kommt nicht in Frage, da immer noch die Variablenbindung fehlt). Sobald C_2 terminiert, suspendiert die gesamte Berechnung: C_3 ist das einzige Blatt im Und-Baum, benötigt aber eine Variablenbindung (Abbildung 4.7).

Mit der oben beschriebenen Verfahrensweise wird das Test and Generate-Beispiel aus Abschnitt 2.1.8

```
test(X) ∧ generate(X)
```

korrekt ausgeführt: das Ergebnis der Auswertung ist weder von der Ausführungsgeschwindigkeit noch von der Ausführungsreihenfolge abhängig. Damit ist die Implementierung mit dem Andorra-Modell [12] vergleichbar [5].

4.6 Erweiterungen

4.6.1 Der Choice-Operator

Im Abschnitt 2.2.5 wurden einige Vorschläge zur Implementierung von Choice-Ausdrücken der Form

```
choice Constraint1 -> Ausdruck1 ;
       ...
       Constraintn -> Ausdruckn
```

gemacht. Hier soll natürlich eine nebenläufige Implementierung betrachtet werden. Dazu muß wieder der Und-Baum erweitert werden. Jedes Blatt des Baumes darf lokal Oder-Parallelismus erzeugen, um die Constraints eines Choice-Ausdrucks auszuwerten (Abbildung 4.8). Dazu erzeugt das Blatt für jedes Constraint eine neue Berechnungsstruktur und verhält sich dann wie folgt (wenn im folgenden von Constraints gesprochen wird, sind damit die erzeugten Berechnungsstrukturen des Choice-Ausdrucks gemeint):

- Der erste Constraint, der erfolgreich terminiert, ist ausschlaggebend für die Wahl des Ausdrucks $Ausdruck_i$. In diesem Fall werden alle anderen Constraints des Choice-Ausdrucks abgebrochen, und das Blatt beginnt mit der Auswertung von $Ausdruck_i$.
- Terminierten alle Constraints mit einem Fehlschlag, schlägt der ganze Choice-Ausdruck (und damit der Und-Baum) fehl.
- Besteht der Choice-Ausdruck nur noch aus suspendierten Constraints, suspendiert das Blatt. Die Berechnung wird erst dann fortgesetzt, wenn ein anderes Blatt des Und-Baumes eine Variable gebunden hat, auf die einer der Constraints wartet.

Wichtig ist, daß die Auswertung der Constraints nur lokal erfolgt: erzeugt ein Constraint Nichtdeterminismus, so werden die erzeugten Kopien des Constraints einfach nur als weitere „Teilnehmer“ am Choice-Operator betrachtet.

[5] Da die meisten Programme wenig Potential zur parallelen Verarbeitung bieten, begünstigt die Implementierung die sequentielle Ausführung von $\wedge$-Ausdrücken: die Auswertung des rechten Teilausdrucks wird erst dann gestartet, wenn der linke suspendiert oder terminiert hat.

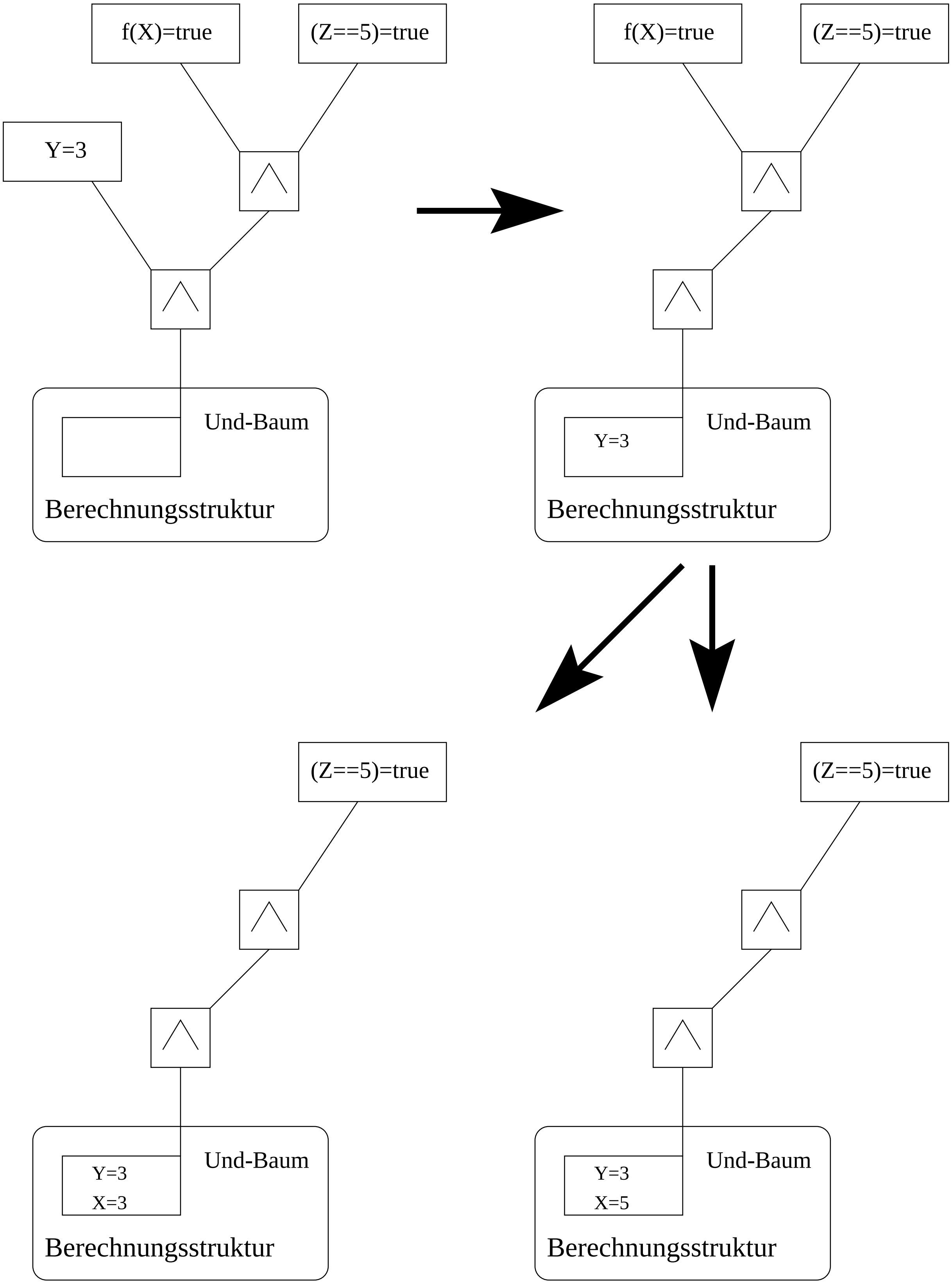

Abbildung 4.7: eine suspendierende Berechnung (`f` ist nichtdeterministisch)

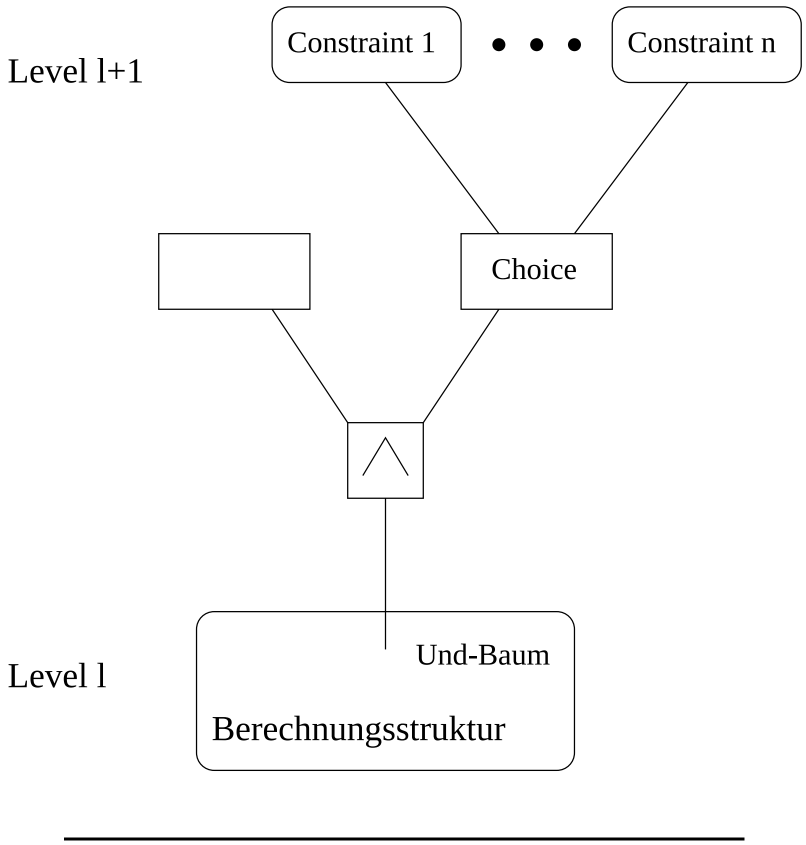

Abbildung 4.8: Auswertung eines Choice-Ausdrucks

Weiterhin dürfen nur Variablen gebunden werden, die lokal für den Constraint (und für den zugehörigen Ausdruck) erzeugt worden sind. Um zu entscheiden, welche Variablen lokal sind und welche nicht, wird jeder Berechnungsstruktur eine natürliche Zahl zugeordnet (im folgenden *Level* genannt):

- bei der Erzeugung von don't care-Nichtdeterminismus (or-Knoten, flex-Knoten) erhalten die ezeugten Berechungsstrukturen alle den gleichen Level (und zwar den gleichen wie die Struktur, die den Nichtdeterminismus ausgelöst hat).
- Berechnungsstrukturen für die Auswertung von Constraints erhalten eine Level-Zahl, die um eins höher liegt als der Level der erzeugenden Struktur (Abbildung 4.8).

Ebenso wird jeder Variablen ein Level zugeordnet: eine Berechnungsstruktur „signiert" eine von ihr erzeugten Variablen mit ihrem Level. Die Variablenzugriffe werden dann wie folgt kontrolliert:

1. Terminiert ein Constraint erfolgreich, so übernimmt die *aufrufende* Berechnungsstruktur alle von dem Constraint erzeugten Variablen (Abbildung 4.9).
2. Das hat natürlich zur Folge, daß die aufrufende Struktur lokal erzeugte Variablen der Constraints erst dann „sieht", wenn ein Constraint erfolgreich terminiert hat. Lokale Variablen sind also während der Wahl-Phase wirklich lokal.
3. Um eine Variable zu dereferenzieren, wird auf den Bindungseintrag der Berechnungsstruktur zugegriffen, die den gleichen Level wie die Variable hat.
4. Das *Binden* einer nicht-lokalen Variablen (d.h. die Variable hat den kleineren Level) führt zu einer Suspendierung des Constraints.

4.6.2 Der Apply-Operator

Die Implementierung des `@`-Operators aus Abschnitt 2.1.6 gestaltet sich relativ einfach durch eine eigenständige `@`-Funktion:

$$f \texttt{ @ } t$$

Ist f ein partiell applizierter Term, so wird f auf t appliziert. Im anderen Fall (d.h. f ist vollständig appliziert) wird f zuerst ausgewertet. Ist f eine Variable, so suspendiert die `@`-Funktion.

Einen partiell applizierten Term erkennt man daran, daß er eine Instanz der Klasse `NodeTerm` ist, deren Maximalanzahl an Argumenten nicht mit der aktuellen Anzahl übereinstimmt. Möchte man f auf t applizieren, so muß folgendes beachtet werden: der Term t darf nicht einfach in das Argumentenarray von f eingetragen werden, denn dadurch würde die Datenstruktur von f verändert. Da f möglicherweise mit anderen Maschinen geteilt wird, muß f `@` t als Resultat eine *Kopie* von f erweitert um t zurückgeben.

4.6.3 Inline-Befehle

Der Compiler, der die Definitional Trees in Code für die abstrakte Curry-Maschine übersetzt, unterstützt das Schlüsselwort `inline` als rechte Seite eines `rule`-Knotens. Trifft der Compiler

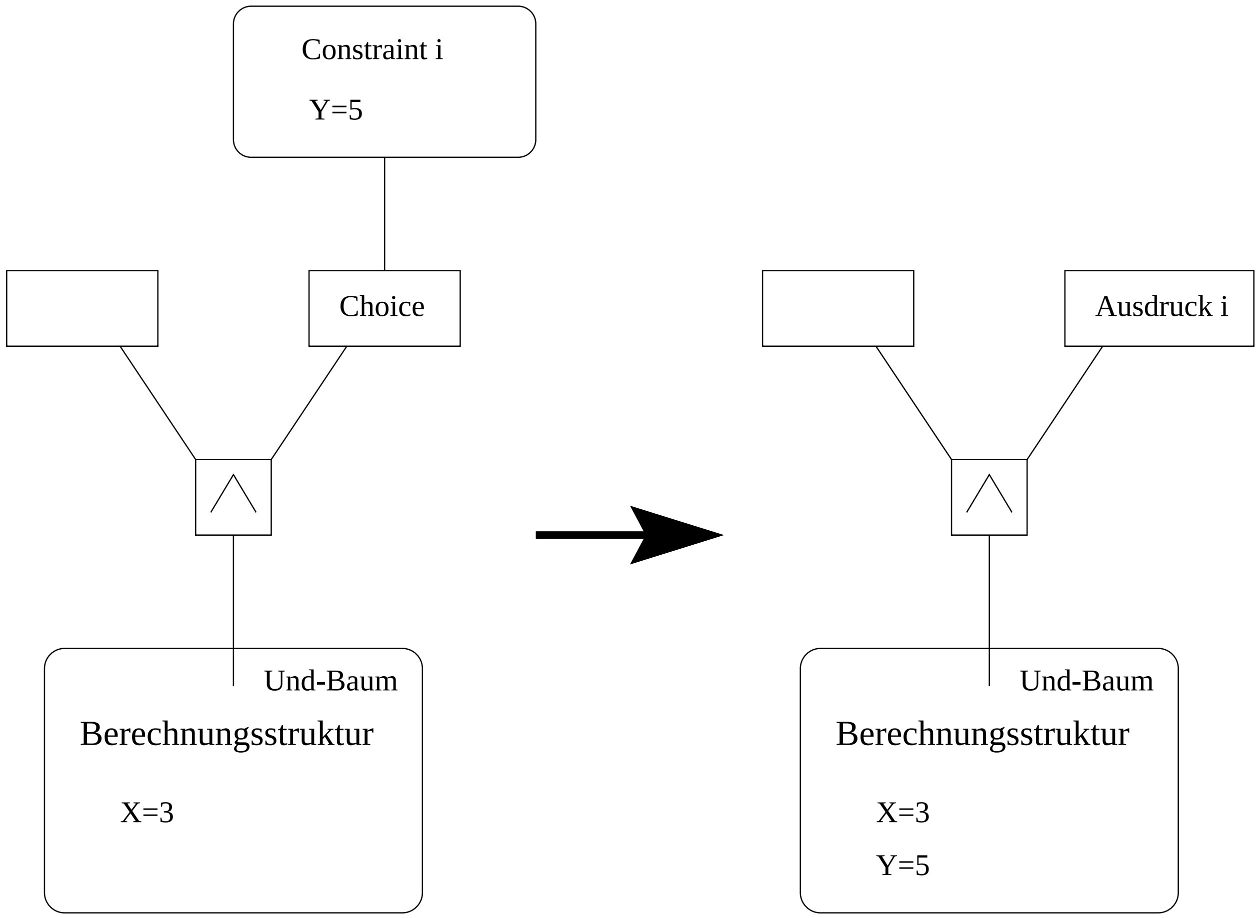

Abbildung 4.9: Übernahme einer Lösung

auf eine rechte Seite der Form `inline("Text")`, so setzt er den angegebenen Text direkt in den Zielcode ein.

Das folgende Beispiel ist ein Ausschnitt aus der `prelude`-Datei des Compilers: hier wurde das `inline`-Schlüsselwort dazu benutzt, mit Hilfe des Additionsbefehls der abstrakten Maschine die `+`-Funktion in Curry zu implementieren.

```
+ branch(+(X,Y),{1},rigid,
        {branch(+(X,Y),{2},rigid,
                {rule(+(X,Y),inline("CmdPrimIntAdd(0,1)"))})})
```

Der erzeugte Code enthält dann den Befehl `CmdPrimIntAdd`:

```
private static final Command code[]={
    new CmdSwitchOnRigid(0,5),
    new CmdSwitchOnRigid(1,4),
    new CmdPrimIntAdd(0,1),
    new CmdReturn(),
    new CmdNewCall(1,1,1),
    new CmdNewCall(0,2,0)
} ;
```

4.7 Optimierungen

Optimierungen am Platz- und Zeitverbrauch können bei einer Programmiersprache an vielen Stufen der Übersetzung durchgeführt werden. In diesem Abschnitt werden zwei Optimierungen vorgestellt, die sich ausschließlich auf die abstrakte Maschine beziehen.

Während die erste Optimierung dazu dient, den Ausführungspfad zu optimieren, versucht man bei der zweiten, soviel Code wie möglich direkt in Java, statt mit der abstrakten Maschine abzuarbeiten. Dadurch spart man sich einige `fetchCommand-executeCommand`–Zyklen der Maschine.

4.7.1 Funktionsaufrufe auf der rechten Regelseite

Eine für viele Sprachen typische Optimierung ist die Vermeidung von unnötigen Einträgen auf dem call stack. Handelt es sich bei der rechten Regelseite um eine *vollständige* Applikation eines Funktionssymbols, z.B.

```
reverse(cons(X,Y)) = append(reverse(Y),cons(X,nil)))
```

so können zwei Optimierungen durchgeführt werden:

1. Die Funktion `reverse` braucht nicht die Funktionsapplikation von `append` als Ergebnis zu liefern, sondern kann direkt mit deren Auswertung beginnen.
2. Beim Aufruf der Funktion `append` braucht *kein* Eintrag auf dem call stack erzeugt zu werden, da das Ergebnis der Funktion `append` gleichzeitig das Ergebnis der Funktion `reverse` darstellt.

Dies wird an einem Beispiel klar. Die Aufruffolge

```
f ruft reverse auf
    reverse ruft append auf
        append terminiert
    reverse terminiert
f terminiert
```

liefert das gleiche Ergebnis wie

```
f ruft reverse auf
    reverse springt zu append
    append terminiert
f terminiert
```

Zur Realisierung dieser Implementierung wird die abstrakte Maschine um den Befehl `CmdExecute` erweitert, der dann an den entsprechenden Stellen einfach statt `CmdReturn` benutzt wird.

4.7.2 Erzeugung von Java-Anweisungen

Am Anfang dieses Kapitels wurde erläutert, warum man Curry-Funktionen nicht direkt in eine Hochsprache übersetzen kann: die Implementierung des Nichtdeterminismus verlangt tiefe Eingriffe in die Kontrolle des Programmpfades. Deshalb wurde auf die direkte Übersetzung zugunsten einer abstrakten Curry-Maschine verzichtet. Die Folge war eine langsamere Ausführungsgeschwindigkeit.

Beim näheren Hinsehen fällt aber auf, daß ein Abschnitt einer Curry-Funktion *nie* Nichtdeterminismus enthält: die rechten Regelseiten, dargestellt durch die *rule*-Knoten des Definitional Trees. Aus diesem Grund übersetzt der Curry-Compiler *rule*-Knoten immer direkt in eine Java-Methode. Mit Hilfe dieser Optimierung wird die Ausführung nochmals um den Faktor 2.5 beschleunigt, während der Unterschied zu einem kompletten Verzicht auf die abstrakte Maschine immer geringer wird: er beträgt nur noch knapp Faktor 2.

Damit die abstrakte Maschine die Methoden, die die rechten Regelseiten darstellen, aufrufen kann, erhält sie einen neuen Befehl:

```
CmdCustom(Methodenname)
```

weist die Maschine an, die Methode `Methodenname` aufzurufen. Der Konstruktor von `CmdCustom` hat daher den Typ

```
CmdCustom((Machine)->void method)
```

Es sei nochmals darauf hingewiesen, daß diese Schreibweise nur von **Pizza** [Pizza] unterstützt wird. Reines Java erlaubt keine Methoden höherer Ordnung.

Beispiel 4.3 und 4.4 zeigen den Code der Funktion `add` (flex) vor und nach der Optimierung. Die optimierte Version enthält zwei Methoden `rule0` und `rule1`, die den Code der rechten Regelseiten enthalten.

```
public class Function_add extends Function {

    static final Command code[]={
        new CmdSwitchOnType(0,11,12),
        new CmdSwitchOnCons(0,{2,4}),
        new CmdPushReg(1),
        new CmdReturn(),
        new CmdLoadArgs(0),
        new CmdLoadReg(0,0),
        new CmdPushReg(0),
        new CmdPushReg(1),
        new CmdPushFunc("add",2),
        new CmdPushCons("s",1),
        new CmdReturn(),
        new CmdNewCall(0,2,0),
        new CmdNewChoice(16,2),
        new CmdPushCons("z",0),
        new CmdSimpleBind(0),
        new CmdJump(2),
        new CmdPushVar(),
        new CmdPushCons("s",1),
        new CmdSimpleBind(0),
        new CmdJump(4)
    } ;

    private static final
        FuncDesc funcDesc=Run.findFuncDesc("add") ;
    public Function_add() { super(funcDesc) ; }
    public final Command getCommand(int index) {
        return code[index] ;
    }
}
```

Beispiel 4.3: Code für add-Funktion (unoptimiert)

```
public class Function_add extends Function {

    private static void rule0(Machine machine) {
        machine.returnWith(machine.getReg(1)) ;
    }

    private static void rule1(Machine machine) {
        Constructor cons=(Constructor)machine.getReg(0) ;
        Constructor cons0=new CustomConstructor(consDesc0) ;
        Function func0=new Function_add() ;
        func0.addArg(cons.getArg(0)) ;
        func0.addArg(machine.getReg(1)) ;
        cons0.addArg(func0) ;
        machine.returnWith(cons0) ;
    }

    private static final Command code[]={
        new CmdSwitchOnType(0,4,5),
        new CmdSwitchOnCons(0,{2,3}),
        new CmdCustom(rule0),
        new CmdCustom(rule1),
        new CmdNewCall(0,2,0),
        new CmdNewChoice(9,2),
        new CmdPushCons("z",0),
        new CmdSimpleBind(0),
        new CmdJump(2),
        new CmdPushVar(),
        new CmdPushCons("s",1),
        new CmdSimpleBind(0),
        new CmdJump(3)
    } ;

    private static final
        ConsDesc consDesc0=Run.findConsDesc("s") ;
    private static final
        FuncDesc funcDesc=Run.findFuncDesc("add") ;

    public Function_add() { super(funcDesc) ; }
    public final Command getCommand(int index) {
        return code[index] ;
    }
}
```

Beispiel 4.4: Code für add-Funktion (optimiert)

Kapitel 5

Ergebnisse und Eigenschaften der Implementierung

In Abschnitt 4.1 wurden einige Vorgaben für die Implementierung des Curry-Compilers vorgestellt. Dieses Kapitel beschäftigt sich mit der Frage,

- inwieweit die Vorgaben erreicht wurden und
- welchen Einfluß die Vorgaben auf die Implementierung hatten.

5.1 Eigenschaften durch Verwendung von Java und Pizza

Die Verwendung von Java (bzw. Pizza) als Implementierungssprache hatte tiefgehende Auswirkungen auf viele Designentscheidungen während der Implementierung. Z.B. wurde in Abschnitt 4.1 erläutert, warum keine direkte Übersetzung von Curry-Funktionen in Java-Programme möglich ist. Natürlich hat Java nicht nur zu Einschränkungen geführt: Anhang C beschreibt einige besondere Fähigkeiten von Java, die in der Implementierung benutzt wurden.

Abgesehen davon besitzt Java noch einige allgemeine Eigenschaften, die praktisch jedem Java-Programm zur Verfügung stehen, ohne daß man sie explizit ausnutzen muß:

Erweiterbarkeit

Wie schon öfters erwähnt, ist die Beziehung zwischen Compilezeit und Laufzeit bei Java viel enger als bei anderen imperativen Sprachen: der Compiler erzeugt für jede definierte Klasse eine .class-Datei, die dann zur Laufzeit geladen, zerlegt, analysiert oder einfach nur ausgeführt werden kann. Da jede Klasse nicht nur den Code der Methoden sondern auch die Schnittstellendefinition enthält, läßt sich die Implementierung einer Klasse auch ohne Neucompilierung und Linken des gesamten Programmes ändern; die einzige Voraussetzung ist die Einhaltung der Schnittstellendefinition.

Wie wirkt sich das nun auf das Curry-System aus? Denkbar wäre die beliebige Erweiterung des Systems um eigene Klassen. Der Benutzer könnte z.B. eigene Suchmechanismen [19] in Java schreiben und sie dem System in Form von Subklassen der Klasse `Function` hinzufügen. Dank des Sicherheitskonzepts von Java wäre es auch möglich, den Zugriff auf die Interna der

Curry-Maschine einzuschränken. Eine andere Möglichkeit wären Curryfunktionen, die Zugriffe auf die Klassen des Abstract Window Toolkit (AWT, plattformunabhängige grafische Benutzeroberfläche) bieten [Java].

Portabilität

Natürlich teilt auch der Curry-Compiler eine besondere Eigenschaft von Java mit allen Java-Programmen. Da ein Java-Compiler immer Bytecode für die JVM erzeugt, kann das gesamte compilierte Curry-System (d.h. Curry-Compiler, Curry-Laufzeitsystem und auch die compilierten Curry-Funktionen) auf jedem Rechner mit installierter JVM *ohne Neucompilierung* ausgeführt werden. Der Benutzer kann sich damit auch neue Curry-Funktionen in Form von .class-Dateien aus jeder beliebigen Quelle beschaffen und direkt einsetzen.

Anbindung an das World Wide Web (WWW)

Momentan werden Ein- und Ausgaben über die Standardein- und -ausgabekanäle des Betriebssystems abgewickelt. Es ist allerdings leicht möglich, das ganze über ein eigenes Fenster durchzuführen; Java bietet dazu die entsprechenden Klassen. Man könnte sogar einen Schritt weitergehen: da die meisten WWW-Browser eine Implementierung der JVM besitzen, wäre es kein Problem Curry-Programme als *Applets*[1] im Fenster eines WWW-Browsers laufen zu lassen.

5.2 Unterschiede zu TasteCurry

Kompatibilität

- die in TasteCurry erlaubten `and`-Knoten in den Definitional Trees [9] werden vom Curry-Compiler nicht unterstützt, hauptsächlich deswegen, weil dadurch die Curry-Maschine um ein vielfaches komplizierter geworden wäre. Andererseits beschränkt sich der Einsatz der `and`-Knoten typischerweise auf den $\wedge$-Operator, und der ist in der Curry-Maschine fest implementiert.

- die aktuelle Version von TasteCurry besitzt ein Modulkonzept: jedes Modul besitzt einen eigenen Namensraum für Funktionen. Auch dies wird nicht vom Curry-Compiler unterstützt, ließe sich aber durch das `package`-Konzept von Java relativ elegant realisieren.

Benutzung

Da die abstrakte Curry-Maschine die Definitional Trees nicht zur Laufzeit interpretiert, fehlt eine interessante Fähigkeit des TasteCurry-Systems: Auswertungen können nicht mehr in einem Einzelschrittmodus mitverfolgt werden.

[1] Applets sind Programme (meist mit grafischer Ausgabe), die in der Darstellung einer WWW-Seite eingelagert sind.

5.3 Laufzeitmessungen

Da die Entwicklung des hier vorgestellten Curry-Compilers vor allem das Ziel hatte, die Ausführung von Curry-Programmen gegenüber dem bestehenden TasteCurry-System deutlich zu beschleunigen, stellt sich natürlich die Frage nach dem Laufzeitverhalten der compilierten Curry-Programme. Folgende Fragen sind deshalb von Interesse:

1. Um welchen Faktor ist die Auswertung schneller geworden (gegenüber dem TasteCurry-System)?
2. Wie schnell ist die Auswertung im Vergleich zu bestehenden funktionalen oder logischen Sprachen?
3. Wie stark wird die Auswertung durch die Verwendung einer abstrakten Curry-Maschine gebremst, d.h. wäre eine direkte Übersetzung der Curry-Funktionen nach Java schneller?
4. Ist die Implementierung von Parallelismus durch Threads zu teuer?
5. Welchen Anteil an der Ausführungssgeschwindigkeit hat die Tatsache, daß die Auswertung von Java-Programmen auf die Interpretation von Bytecode basiert?

Zu Frage 3: diese Frage kann natürlich — wie in Abschnitt 4.1 erläutert — nur für rein deterministische Programme beantwortet werden. Die Beschleunigung beträgt für solche Programme gerade mal Faktor 2 (siehe auch dazu Abschnitt 4.7.2).

Zu Frage 4: in Abschnitt 5.3.3 werden einige Messungen zur Beantwortung dieser Frage vorgestellt. Ungeachtet der Ergebnisse darf aber nicht vergessen werden, daß die Verwendung von Threads einen wesentlichen Vorteil hat: Implementierungen der JVM für Multiprozessorsysteme schenken dem Curry-Programmierer praktisch ohne weiteres Zutun echten Parallelismus.

Zu Frage 1,2 und 5: die Antwort zu Frage 3 macht schon deutlich, daß die Ausführungsgeschwindigkeit von Java-Programmen wesentlich von der JVM bestimmt wird. Im folgenden werden einige Messungen zu diesen Fragen durchgeführt und ihre Ergebnisse besprochen.

5.3.1 Testbedingungen

Messungen an TasteCurry unter Prolog

Die Messungen wurden auf einem Pentium 100 Mhz, 32 MB RAM unter Linux (Kernel 2.0.24) und SICStus-Prolog 3 #3 [20] durchgeführt. Das TasteCurry-System wurde vorher in SICStus mit dem Prädikat `compile` compiliert.

Zur Zeitmessung diente der Schalter `time` der TasteCurry-Shell [15].

Messungen am Curry-System unter Java

Auch hier wurden die Messungen auf einem Pentium 100 MHz, 32 MB RAM durchgeführt. Die Java-Klassen wurden mit der JVM des Sun SDKs 1.1.3 ausgeführt. Als Just-in-Time Compiler kam die Prerelease des mitgelieferten Symantec-Compilers zum Einsatz. Um Einflüsse des Garbage Collectors auszuschließen, wurden der JVM mit den Optionen `ms` und `mx` genügend Speicher zugeordnet.

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

reverse([]) = [].
reverse(cons(X,Xs)) = append(reverse(Xs),cons(X,nil)).
```

Beispiel 5.1: Umkehren von Listen

```
take :: int -> list(T) -> list(T).
take(N,[])      = [].
take(N,[X|Xs]) = if(N==0,[],[X|take(N-1,Xs)]).

from :: int -> list(int).
from(N) = [N|from(N+1)].

primes :: list(int).
primes = sieve(from(2)).

sieve :: list(int) -> list(int).
sieve([X|Xs]) = [X|sieve(filter(indivisible@X,Xs))].

indivisible :: int -> int -> bool.
indivisible(X,Y) = (Y mod X > 0).

filter :: (A -> bool) -> list(A) -> list(A).
filter(P,[])      = [].
filter(P,[X|Xs]) = if(P@X, [X|filter(P,Xs)], filter(P,Xs)).
```

Beispiel 5.2: Ermitteln der Primzahlen

Zur Zeitmessung wurden die Messungen, die das Curry-Laufzeitsystem durchführt, übernommen. Alle Zeitangaben beziehen sich auf wirkliche Auswertungszeit ohne Ausgabe der Lösung (keine CPU-Zeit der einzelnen Berechnungsstrukturen). Vor jeder Messung wurde eine Auswertung des jeweiligen Ausdrucks veranlaßt, um der JVM die Gelegenheit zu geben, alle nötigen Klassen zu laden.

5.3.2 Rein funktionale Programme

Dieser Abschnitt beschäftigt sich mit der Laufzeit von Curry-Funktionen ohne logische Variablen und ohne Nichtdeterminismus (alle Funktionen sind `rigid`, keine `or`-Knoten im Definitional-Tree).

Als Testfunktionen wurde die Funktion `reverse` zum Umkehren von Listen und der Ausdruck `take(`n`,primes)` (aus der Datei `inflists.fl` des TasteCurry-Systems) zum Ermitteln der n ersten Primzahlen verwendet (Beispiel 5.1 und 5.2).

reverse (Curry-Compiler)

Länge der Liste	10	20	40	80	120	160
Anzahl Reduktionsschritte	66	231	861	3321	7381	13041
Benötigte Zeit (ms)	20	40	130	510	1040	1800
LIPS	3300	5775	6623	6512	7097	7245

Initial Stack Size=Max Stack Size=16 MByte, no garbage collection occured
Siehe Text für Testbedingungen

Tabelle 5.1: Laufzeitmessungen zu Beispiel 5.1

primes

n	5	10	15	20	25
Benötigte Zeit (ms)	30	90	180	290	431

Initial Stack Size=Max Stack Size=16 MByte, no garbage collection occured
Siehe Text für Testbedingungen

Tabelle 5.2: Laufzeitmessungen zu Beispiel 5.2

Die Meßergebnisse des Curry-Compilers sind als Zahlenmaterial in den Tabellen 5.1 und 5.2 zu sehen.

Es zeigt sich sofort, daß der Curry-Compiler deutlich schneller als das TasteCurry-System ist: die Funktion `reverse`, angewendet auf eine 40-elementige Liste, benötigt unter TasteCurry 6 Sekunden (statt 0,13 Sekunden). Da der SICStus-Prologcompiler für seine gute Codeerzeugung bekannt ist, ist wohl der ausschlaggebende Faktor die Tatsache, daß das TasteCurry-Systems intern auf Termersetzung basiert. Das `primes`-Beispiel profitiert besonders vom term sharing: ab $n = 20$ liefert das TasteCurry-System in annehmbarer Zeit kein Resultat. Hier dürfte die Beschleunigung wohl über Faktor 100 liegen.

Um diese Ergebnisse mit einer logischen Programmiersprache wie Prolog vergleichen zu können, ist in der Tabelle 5.1 die Anzahl der nötigen Reduktionsschritte (ein Reduktionsschritt ist die Umwandlung eines Funktionsaufrufs in die rechte Seite einer passenden Funktionsgleichung) und die zugehörige LIPS-Zahl (Anzahl Reduktionsschritte pro Sekunde) angegeben. Auch wenn Curry wegen seiner auf lazy-evaluation basierenden Auswertungsstrategie gegenüber logischen Sprachen im Nachteil ist, erkennt man die niedrige Ausführungsgeschwindigkeit; der SICStus-Compiler erreicht über 500000 LIPS. Unter Berücksichtigung des Faktors 2 für die Verwendung einer abstrakten Curry-Maschine (siehe oben), schafft das entsprechend als Java-Klasse compilierte Curry-Programm gerade mal 14000 LIPS — allerdings mit einem deutlich niedrigeren Speicherverbrauch. Die Ursache ist wohl der gezwungenermaßen langsamere Ausführungsmechanismus der JVM, da vergleichbare Systeme wie z.B. JProlog [5] zu ähnlich schlechten Resultaten führen.

Das gleiche Bild liefert ein Vergleich mit rein funktionalen Lazy-Sprachen. Die Funktion `reverse` erreicht unter Hugs 1.1.4 ungefähr 80000 LIPS und mit dem ghc-Haskellcompiler sogar über 500000 LIPS.

Abbildung 5.1 zeigt nochmals die LIPS-Ergebnisse der Funktion `reverse` als Grafik. Der plötz-

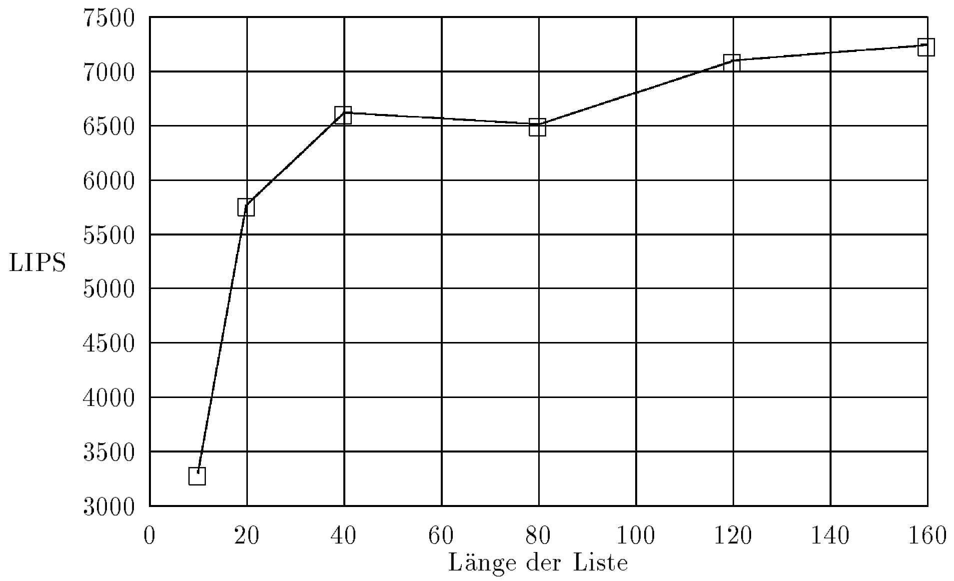

Abbildung 5.1: LIPS-Messung der Funktion `reverse`

liche Anstieg der Kurve für die Messung wird durch den Overhead der Berechnungsstruktur verursacht. Der Rest der Grafik ist erfreulich gerade; offensichtlich wird die Leistung der JVM nicht durch die großen Datenstrukturen negativ beeinflußt.

5.3.3 Die Erzeugung von Threads: Kostenbetrachtung

Dieser Abschnitt beschäftigt sich mit der Frage, wie teuer (bezüglich Zeit) die Erzeugung von Threads ist. Da Maschinen des Und-Baumes durch Threads implementiert wurden, werden Threads erzeugt, wenn

1. Nichtdeterminismus erzeugt wird, d.h. durch einen flex-Baum, durch einen or-Knoten oder durch den Choice-Operator (siehe das Kapitel über die Implementierung).

2. Constraints zu einer Konjunktion verknüpft werden, d.h. durch den $\wedge$-Operator.

Zweifellos überwiegen im Fall 2 die Vorteile der Verwendung von Threads: der Einsatz von Threads vereinfachte nicht nur die Implementierung, sondern macht auch Curry zu einer echt nebenläufigen Sprache. Bleibt noch der Fall 1: wäre eine Implementierung des Nichtdeterminismus durch andere Mechanismen (z.B. Backtracking) günstiger gewesen?

Um zu ermitteln, wie aufwendig in Java die Erzeugung eines neuen Threads ist, wurden die Laufzeiten der Funktionen `linear1` und `linear2` ermittelt (Beispiel 5.3). Diese beiden Funktionen tun nichts anderes als das übergebene Argument auf 0 „herunterzuzählen“, jedoch auf unterschiedliche Weise:

1. Die Funktion `linear1` besteht aus zwei Regeln mit überlappenden linken Regelseiten. Folglich besteht ihr Definitonal Tree aus einem or-Knoten, der den Programmlauf aufspaltet und auf beide Regeln verteilt (Abschnitt 2.1.5). Dies hat zur Folge, daß ein Aufruf der Form

   `linear1`(n)

```
% linear1
% Die ueberlappenden linken Regelseiten erzeugen
% einen or-Knoten

linear1(0)=0.
linear1(N) | {(N>0)=true}=linear1(N-1).

% linear2

linear2(N)=if (N>0)==true then linear2(N-1) else 0.
```

Beispiel 5.3: Testfunktionen für Threaderzeugung

linear n

n	20	40	80	120	160	200	240	280	320
Mit or-Knoten									
Benötigte Zeit (ms)	80	140	290	440	590	730	880	1030	1180
Anzahl Threads (1 erfolreicher)	22	42	82	122	162	202	242	282	322
Ohne or-Knoten									
Benötigte Zeit (ms)	10	30	60	80	110	130	170	190	220
Anzahl Threads=1									
Faktor (ohne/mit or-Knoten)	8,0	4,7	4,8	5,5	5,4	5,6	5,2	5,4	5,4

Initial Stack Size=Max Stack Size=16 MByte, no garbage collection occured
Siehe Text für Testbedingungen

Tabelle 5.3: Laufzeitmessungen zu Beispiel 5.3

$n+2$ Threads (oder-parallele Maschinen) erzeugt. Natürlich wird immer nur eine Maschine erfolgreich sein — die anderen $n+1$ Threads schlagen praktisch sofort fehl (entweder wegen dem Patternmatching-Fehler in Regel 1 oder wegen dem Constraint in Regel 2). Die Funktion läßt sich also gut als Threaderzeuger mißbrauchen.

2. Im Gegensatz dazu arbeitet die Funktion `linear2` auf „herkömmliche“ Weise. Der Programmablauf bleibt völlig linear: es wird nur ein Thread erzeugt, der dann auch erfolgreich terminiert (vorausgesetzt $n \geq 0$).

Die Laufzeiten für verschiedene Werte von n sind in Tabelle 5.3 und nochmal als Grafik in Abbildung 5.2 zu sehen. Wie zu erwarten war, ist der Zeitverbrauch linear zu n. Das überraschendste Ergebnis dürfte wohl die Tatsache sein, daß sich die Laufzeiten der beiden Funktionen „nur“ um einen Faktor 5,5 unterscheiden, obwohl die erste Funktion bis zu 322 Threads (!) erzeugt. Offensichtlich ist die JVM in der Lage, relativ günstig neue Threads zu erzeugen.

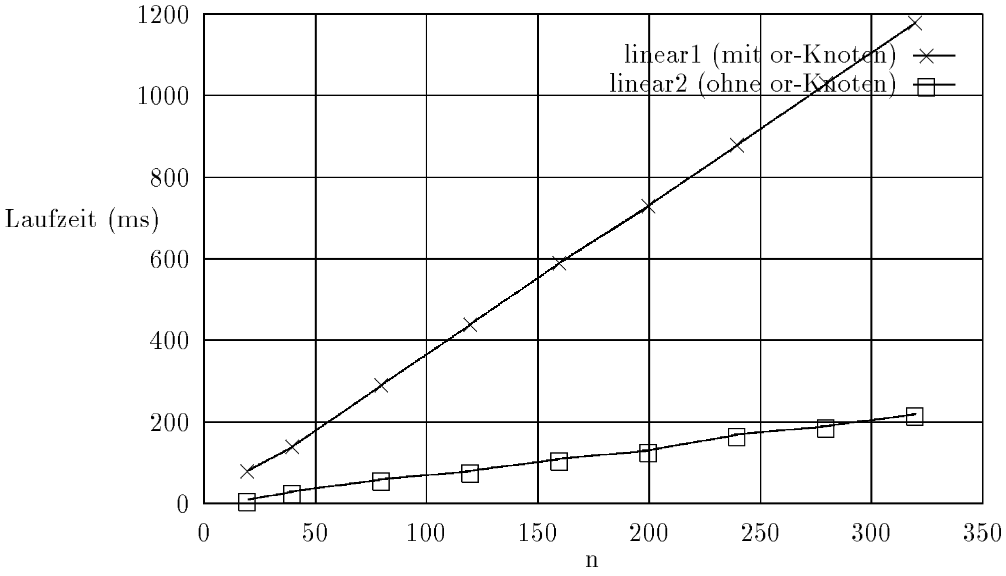

Abbildung 5.2: Laufzeitmessung der Funktionen `linear1` und `linear2`

5.3.4 Und-parallele Programme

Schließlich möchte ich hier noch die Laufzeiten für ein Programm angeben, das massiv Gebrauch vom Und-Baum macht. Es handelt sich um das `colormap`-Programm des TasteCurry-Systems: dieses Programm soll das bekannte Einfärbeproblem lösen und arbeitet nach dem test-and-generate– bzw. generate-and-test–Prinzip (Beispiel 5.4).

Hier die Ergebnisse:

	generate+test	test+generate
TasteCurry		
Laufzeit	6,84 s	2,28 s
Curry-Compiler		
Laufzeit	1,3 s	1,7 s
Erzeugte Berechnungen	256	124
Erfolgreiche Berechnungen	48	48

Interessanterweise ist generate+test bei TasteCurry langsamer als test+generate, während es beim Curry-Compiler umgekehrt ist. Das liegt daran, daß Java relativ günstig unabhängige Threads, wie sie bei generate+test auftauchen, verwalten kann: hier basiert das Programm hauptsächlich auf Oder-Parallelismus. Beim test+generate–Ansatz liegt aber hauptsächlich Und-Parallelismus vor; die Test-Funktion suspendiert laufend und wartet auf die generate-Funktion, was aufwendige Synchronisierungen erfordert. Hier würden Optimierungen am Und-Baum sicherlich viel bringen.

```
% constraint solving (simple generate and test) in Curry:
% graph coloring


% auxiliary functions:

% negation of ==:
diff(X,Y) = {(X==Y)=false}
.

% This is our actual map:
%
% -------------------------
% |       |        |       |
% |       |   L2   |       |
% |       |        |       |
% |  L1   |--------|  L4   |
% |       |        |       |
% |       |   L3   |       |
% |       |        |       |
% -------------------------
%

data color = red ; green; yellow; blue.


color(red)    = true.
color(yellow) = true.
color(green)  = true.
color(blue)   = true.


coloring(L1,L2,L3,L4) = color(L1) && color(L2) && color(L3)
                        && color(L4).


% correct coloring:
correct(L1,L2,L3,L4) = {diff(L1,L2), diff(L1,L3), diff(L2,L3),
                        diff(L2,L4), diff(L3,L4)}.


% generate+test solution:
% goal: {coloring(L1,L2,L3,L4)=true, correct(L1,L2,L3,L4)}.


% test+generate solution:
% goal: {correct(L1,L2,L3,L4), coloring(L1,L2,L3,L4)=true}.
```

Beispiel 5.4: colormap

Kapitel 6

Fazit und Ausblick

Mit dieser Arbeit haben wir unser Ziel erreicht: die Entwicklung eines Curry-Compilers, mit dessen Hilfe sich Curry-Programme in einer einigermaßen realistischen Geschwindigkeit ausführen lassen. Wir haben gesehen, daß durch den Nichtdeterminismus von Curry und einigen Einschränkungen von Java eine direkte Übersetzung von Curry-Funktionen in Java-Programme nicht möglich ist; statt dessen wird Code für eine abstrakte Curry-Maschine generiert. Diese abstrakte Maschine verwendet Java-Threads zur Darstellung der Nebenläufigkeit in Curry-Programmen, was nicht nur zu einer eleganten operationalen Semantik, sondern auch zu einigen Vorteilen in der Ausführung (siehe z.B. Abschnitt 4.4.3) führte.

Nichtsdestotrotz zeigen die Laufzeitmessungen, daß die hier getroffenen Designentscheidungen durchaus ihren Preis haben. Außerdem konnte aus Zeitgründen kein vollständiges Curry-System implementiert werden. Deshalb möchte ich hier noch einige mögliche Verbesserungen und Erweiterungen vorstellen:

Die Verbindung nach Außen: damit TasteCurry durch das hier vorgestellte System ersetzt werden kann, muß der Compiler in der Lage sein, dessen Aufgaben zu übernehmen, als da wären das Einlesen von Curry-Programmen, Typechecking und die Erzeugung der Definitional Trees. Der wichtigste Schritt, nämlich die Entwicklung eines Parsers für Curry-Programme (praktischerweise sogar in Java geschrieben), ist schon getan. Weiterhin benötigt das Curry-System eine vernünftige Shell, in der man Programme laden, compilieren und ausprobieren kann.

Interne Erweiterungen: natürlich sollen auch irgendwann die Erweiterungen, die am TasteCurry-System vorgenommen wurden, in dieses System aufgenommen werden. Für die eingekapselte Suche aus [19] ist das System schon teilweise durch das Levelkonzept der Variablenbindungen vorbereitet.

Wie läßt sich nun die *Leistung des Systems* verbessern? Die abstrakte Maschine selbst läßt sich kaum noch beschleunigen. Ich glaube, daß für rein funktionale Programme, bezüglich der Geschwindigkeit das in Java Machbare erreicht worden ist. Wie schon gesagt, hängt die Ausführungsgeschwindigkeit von Java-Programmen von der Effizienz der JVM ab. In den nächsten Jahren werden hier wohl spürbare Verbesserungen zu erwarten sein, denn theoretisch können Java-Programme mit einem Just In Time–Compiler genauso schnell abgearbeitet werden, wie compilierte C- oder Pascal-Programme.

Vielversprechender sind Analysen des Programmpfades zur Compile- und Laufzeit: besonders interessant wären Analysen, mit deren Hilfe man entscheiden könnte, welche Programmabschnitte rein deterministisch ablaufen. In solchen Situationen könnte die abstrakte Maschine auf den Verwaltungsaufwand wie Ergebnistabellen usw., verzichten und in einen „geradlinigen“ Ausführungsmodus wechseln (der wahrscheinlich Faktor 3 bis 4 schneller ist).

Bleibt natürlich die Frage, ob zukünftige Versionen von Curry überhaupt auf Java basieren sollen. Es muß noch überprüft werden, inwieweit andere Programmiersprachen es wohl erlauben würden, Curry-Funktionen direkt in eine imperative Sprache zu übersetzen.

Anhang A

Bedienung des Curry-Systems

A.1 Installation des Compilers und des Laufzeitsystems

A.1.1 Benötigte Dateien

Der compilierte Code des Compilers und des Laufzeitsystems müssen sich in einem Verzeichnis befinden, das von der JVM erreichbar ist, d.h. in einem Verzeichnis, das in der Environment-Variablen `CLASSPATH` eingetragen wurde [Java].

Da der Compiler das TasteCurry-System als front end benutzt, sollte ein TasteCurry-System installiert sein [15].

Außerdem werden einige Klassen des Pizza-Systems [Pizza] benötigt. Deshalb müssen auch diese Klassen vorhanden und von der JVM erreichbar sein.

Und natürlich braucht man ein installiertes Java-System.

A.1.2 Den Quellcode neu compilieren

Im folgenden wird beschrieben, wie der Quellcode neu compiliert wird. Jedoch sollte beachtet werden, daß das im allgemeinen nicht notwendig ist; compilierte Java- und Pizza-Programme sind maschinenunabhängig.

Da das System in Pizza geschrieben worden ist, verlangt die Compilation einen installierten Pizza-Compiler [Pizza]. Die nächsten Schritte gehen davon aus, daß das Zielverzeichnis `java/classes` heißt und sich der Quellcode des Curry-Systems im Verzeichnis `curry4` befindet. Folgende Befehle sollten in einer Kommandozeilen-Shell des Betriebssystems eingegeben werden.

```
pc -d java/classes curry4/*.pizza
pc -d java/classes curry4/compiler/*.pizza
pc -d java/classes curry4/runtime/*.pizza
```

A.2 Compilierung und Benutzen von Curry-Programmen

An einem Beispiel soll erläutert werden, wie man ein Curry-Programm (in Form einer .fl-Datei) compiliert und anschließend Ausdrücke mit Hilfe dieses Curry-Programms auswertet.

1. Als erstes muß aus dem Curry-Programm eine .def-Datei erzeugt werden. Die .def-Datei enthält die nötigen Informationen (einschließlich der Definitional Trees) für den nachfolgenden Compilationsvorgang.

   Um die .def-Datei zu erzeugen, wird das Curry-Programm in das TasteCurry-System [15] geladen. Dazu gibt man in der TasteCurry-Shell ein (falls das Programm `test.fl` heißt):

   ```
   read test.
   ```

   Nun kann die .def-Datei erzeugt werden:

   ```
   trees('test.def').
   ```

   Die nächsten Schritte sind unabhängig vom TasteCurry-System und sollten in einer Kommandozeilen-Shell des verwendeten Betriebssystems durchgeführt werden.

2. Als nächstes wird die .def-Datei in eine Pizza-Datei übersetzt und vom Pizza-Compiler in Bytecode für die JVM umgewandelt. Dazu wird der Curry-Compiler gestartet:

   ```
   java curry4.compiler.Main test java/classes
   ```

   Das erste Argument gibt den Namen der zu übersetzenden .def-Datei an (in unserem Fall `test.def`). Der Curry-Compiler übersetzt dann die Datei `test.def` in eine Pizza-Datei namens `test.pizza`. Voraussetzung ist allerdings, daß sich die Datei `prelude.def` (zu finden im Quellcode-Verzeichnis `curry4`) im aktuellen Verzeichnis befindet.

   Anschließend wird der Pizza-Compiler automatisch gestartet. Er übersetzt dann die Pizza-Datei in einzelne Java-Klassen. Das Zielverzeichnis für die erzeugten Klassen wird mit dem zweiten Argument (hier `java/classes`) angegeben. Auch hier sollte das Zielverzeichnis von der JVM erreichbar sein.

   Dieser Schritt braucht im allgemeinen nicht durchgeführt zu werden. Der Start des Curry-Laufzeitsystem (siehe nächsten Schritt) beinhaltet eine automatische Compilierung aller noch nicht übersetzten Funktionen. Dazu wird das Erzeugungsdatum der .def- und der zugehörigen .pizza-Datei verglichen.

3. Das Laufzeitsystem wird bei unserer Testdatei mit

   ```
   java curry4.runtime.Run test
   ```

   geladen. Es ist jedoch besser, den Compilationsvorgang dem Laufzeitsystem zu überlassen. Dazu benötigt das System wiederum ein Zielverzeichnis für die erzeugten Klassen:

   ```
   java curry4.runtime.Run test java/classes
   ```

Mit der Option -f kann der Compilationsvorgang erzwungen werden:

```
java curry4.runtime.Run -f test java/classes
```

Nach einer kurzen Initialisierungsphase meldet sich das System mit einem Prompt >. Hier können nun beliebige Ausdrücke zur Auswertung eingegeben werden. Nach dem Beenden der Eingabe mit <Return> wird die Auswertung gestartet. Haben alle erzeugten Berechnungsstrukturen terminiert, erscheint wieder das Prompt-Zeichen > und ein neuer Ausdruck kann eingegeben werden.

Zum Verlassen des Systems kann die Tastenkombination <Control>-C gedrückt werden (was gleichzeitig die laufenden Berechnungen abbricht).

A.3 Syntax von Termen

Der Parser erwartet *alle* Ausdrücke in Prefixschreibweise, dies gilt auch für die Konjunktion von Constraints. D.h. der Ausdruck

```
{ X=3, X+3=Y}
```

sollte eingegeben werden als

```
/\(=(X,3),=(+(X,3),Y))
```

Für den Aufbau von Listen stehen die Konstruktoren [] und : zur Verfügung. Alternativ können Listen in der bekannten Schreibweise eingegeben werden:

```
[1,2,3,4]
```

Da das Zeichen] auch in Funktionsnamen erlaubt ist, muß zwischen dem letzten Element einer Liste und dem Zeichen] ein Freizeichen eingegeben werden (dies gilt nicht für Zahlen):

```
[false,true ]
```

Zeichenketten werden in der bekannten "..."-Notation akzeptiert.

Bei der Ausgabe werden partiell applizierte Konstruktoren und Funktionen mit einem vorangestellten $ markiert.

Anhang B

Übersicht über die Quellcodedateien

B.1 Pizza- und Prologdateien

- TasteCurry:

  Der Quellcode der Shell (in `curry.pl`) wurde um die Anweisung `trees` zur Erzeugung der `.def`-Dateien erweitert.

- Datei `classes.zip`: die compilierten Java-Klassen des Curry-Systems

- `curry4/` (Datei `sources.zip` auf der Diskette)

`prelude.def`	Hier befinden sich die Definitionen der eingebauten Typen und Funktionen (z.B. Addition,...). Diese Datei wird vom Compiler und Laufzeitsystem benötigt. Nicht verwechseln mit `prelude.fl` des TasteCurry-Systems !
`scanner.pizza`	der Scanner für .def-Dateien und Benutzereingaben
`desclist.pizza`	Datenstrukturen zum Speichern von Informationen über die Konstruktoren und Funktionen (z.B. Stelligkeit)

  - `compiler/`

    In diesem Verzeichnis befindet sich der Quellcode des Compilers

`command.pizza`	Repräsentation des erzeugten Codes
`deftree.pizza`	Umwandlung der Definitional Trees in Code
`main.pizza`	Einstiegspunkt in den Compiler für das Betriebssystem und das Laufzeitsystem
`readin.pizza`	Parser für .def-Dateien
`term.pizza`	Darstellung der Terme und Umwandlung in Code (rechte Regelseiten)
`termstack.pizza`	Hilfsdatenstrukturen zum Erzeugen von Termen

– `runtime/`
  In diesem Verzeichnis befindet sich der Quellcode des Laufzeitsystems

`and.pizza`	die Und-Bäume
`apply.pizza`	Implementierung des @-Operators
`bindingtable.pizza`	Implementierung der Bindungstabellen für Variablen
`command.pizza`	die Befehle der abstrakten Maschine
`curry_hashtable.pizza`	Hashtabelle für Bindungs- und Ergebnistabellen
`equal.pizza`	Implementierung von `=` und `==`
`function.pizza`	Funktionen und ihre Ergebnistabellen
`machine.pizza`	Berechnungsstrukturen, abstrakte Maschine und `choice`-Operator
`p_readin.pizza`	Parser für Benutzereingaben
`primitive_int.pizza`	Operationen für Typ `int`
`primitive_io.pizza`	Operationen für IO (Monade)
`run.pizza`	Einstiegspunkt in das Laufzeitsystem für das Betriebssystem
`term.pizza`	Darstellung der Terme
`termstack.pizza`	Hilfsdatenstrukturen zum Erzeugen von Termen

B.2 Die `prelude.def`-Datei

Die `prelude.def`-Datei enthält die Typ- und Funktionsdefinitionen der eingebauten Typen/Funktionen. Daher wird diese Datei bei jedem Start des Curry-Systems benötigt.

Die Datei hat den gleichen Aufbau wie die `.def`-Dateien, die durch den Compiler erzeugt werden:

1. Das Schlüsselwort `constructors` leitet eine Folge von Datentypdefinitionen ein. Solch eine Definition hat die Form

   $$Typname\ \{\ Konstruktor_1\ \ Stelligkeit_1\ \ldots\ Konstruktor_n\ \ Stelligkeit_n\ \}$$

   Zum Beispiel:

   ```
   list{[] 0 : 2 }
   ```

2. Anschließend folgen nach dem Schlüsselwort `functions` die Funktionsdeklarationen in der Form

   $$Funktionsname\ \ Stelligkeit$$

3. Schließlich werden die Definitional Trees angegeben (Schlüsselwort `deftrees`). Für den Syntax der Trees ist ein Blick in die Datei wohl hilfreicher als tausend Worte.

Momentan enthält die Datei folgende Definitionen und Deklarationen:

- Listentyp `list`
- $\wedge$, `=`, `==`, `@` und `$` (partielle Abbildung)
- arithmetische Operationen auf ganze Zahlen
- Vergleichsoperationen auf ganze Zahlen
- IO-Monade für Ein-/Ausgabe in Datei und Konsole
- `ite` (if-then-else)

Anhang C

Ausschnitte aus dem Quellcode

Die Implementierung des Curry-Systems sollte so weit wie möglich von den Eigenschaften der Sprache Java Gebrauch machen. Im folgenden werden einige Teile des Systems vorgestellt, die kaum in einer anderen Sprache mit dieser Einfachheit realisiert werden könnten (mit Ausnahme des Exception-Konzepts, das Java größtenteils von C++ geerbt hat).

C.1 Implementierung von Kommunikationskanälen in Java

Kommunikationskanäle werden in der Implementierung von den Berechnungsstrukuren verwendet, um Lösungen bzw. Fehlschläge an den übergeordneten Kontrollthread zu melden.

Dabei sollen die Kommunikationskanäle die Berechnungsstrukturen untereinander und mit dem Kontrollthread synchronisieren. Dies geschieht, indem die Kanäle als blockierende, einelementige Warteschlangen realisiert wurden: versucht der Kontrollthread eine Nachricht aus einem leeren Kanal zu lesen, wird er blockiert — ebenso Berechnungsstrukturen, die versuchen, eine Nachricht in einen vollen Kanal zu schreiben.

Kanäle werden durch Instanzen der Klasse `Channel` (Beispiel C.1) repräsentiert. Die Klasse besitzt zwei Methoden zum Senden (`void send(Message m)` und Empfangen (`Message receive()`) von Nachrichten. Um zu verhindern, daß gleichzeitig gesendet und empfangen werden kann, sind die beiden Methoden mit dem Monitor des Kanalobjekts synchronisiert (`synchronized`, siehe Abschnitt 3.3). Als Nachricht kann jede Instanz der Klasse `Message` verschickt werden.

Die Implementierung ist sehr einfach gehalten, vor allem deswegen, weil Kanäle selten benutzt werden. Das Kanalobjekt selbst dient dazu, sowohl Sender als auch Empfänger zu benachrichten: stößt ein Sender auf einen vollen Kanal, deaktiviert er sich (`wait()`) und kann erst wieder durch einen Empfänger reaktiviert werden (`notifyAll()`). Analog dazu wird ein Empfänger, der keine Nachricht im Kanal vorfindet, durch einen Sender reaktiviert.

C.2 Nachladen von Klassen

Einer der beeindruckendsten Fähigkeiten von Java ist die Möglichkeit, Klassen dynamisch zur Laufzeit zu laden. Dies geschieht beispielsweise bei der Konstruktion von Termen auf dem Termstack der abstrakten Curry-Maschine. So soll der Aufruf

```
class Channel {
    private boolean filled=false ;
    private Message msg ;

    synchronized public final void send(Message m) {
        try {
            while (filled) wait() ;
        } catch(InterruptedException e) {}
        filled=true ; msg=m ;
        notifyAll() ;
    }

    synchronized public final Message receive() {
        try {
            while (!filled) wait() ;
        } catch(InterruptedException e) {}
        filled=false ; Message m=msg ;
        notifyAll() ;
        return m ;
    }
}
```

Beispiel C.1: Die Channel-Klasse

```
termStack.pushFunc("add",2)
```

auf dem Termstack eine Instanz der Klasse `Function_add` erzeugen, wobei die beiden obersten Elemente des Stacks als Argumente für das so erzeugte Funktionsobjekt dienen.

Java ordnet jeder Klasse ein Objekt der Instanz `Class` zu. Mit Hilfe der Methode

```
Class forName(String name)
```

der Klasse `Class` erhält man das `Class`-Objekt der Klasse mit dem Namen `name`. In unserem Fall also

```
Class funcclass=Class.forName("Function_add") ;
```

Falls die Klasse sich noch nicht im Speicher befindet, wird sie von der JVM geladen.

Durch das Klassenobjekt einer Klasse kann man an alle relevanten Informationen der Klasse gelangen, so zum Beispiel an eine Liste der Methoden und Konstruktoren der Klasse. Da wir eine Instanz der Klassen erzeugen wollen, verlangen wir das Objekt, das den parameterlosen Konstruktor der Klasse darstellt:

```
java.lang.reflect.Constructor funccons=funcclass.getConstructor(null) ;
```

(Die Quellenangabe `java.lang.reflect` ist nötig, damit Java nicht die Klasse `Constructor` mit der `Constructor`-Termklasse des Curry-Systems verwechselt).

Nun da wir den Konstruktor haben, können wir ihn auch anwenden, um ein neues Funktionsobjekt zu erzeugen:

```
final Term pushFunc(String name,int nargs) {
    try{
        Class funcclass=null ;
        try{
            funcclass=Class.forName("Function_"+name) ;
        }catch(Exception e) {}
        if(funcclass==null)
            funcclass=Class.forName("curry4.runtime.Function_"+name) ;

        Function func=(Function)funcclass.newInstance() ;

        for(int i=1;i<=nargs;i++)
            func.setArg(nargs-i,popTerm()) ;
        func.nargs=nargs ;
        return pushTerm(func) ;
    }
    catch(Exception e){
        System.out.println(
            "Can't load function: "+name+"/"+nargs+", because "+e) ;
        return null ;
    }
}
```

Beispiel C.2: Laden von Klassen zur Laufzeit

```
Function func=(Function)funccons.newInstance(null) ;
```

Die oben beschriebenen Schritte können auch abgekürzt werden, da unser Konstruktur keine Argumente verlangt: für diesen Fall bietet die Klasse `Class` eine eigene Methode:

```
Class funcclass=Class.forName("Function_add") ;
Function func=(Function)funcclass.newInstance() ;
```

Beispiel C.2 zeigt nochmals den kompletten Quellcode der Methode `pushFunc`. Falls der Versuch, eine Klasse zu laden, fehlschlägt (z.B. weil die Klasse nicht gefunden werden konnte), erzeugt der `forName`-Aufruf einen Fehlschlag. Der Quellcode geht dann davon aus, daß es sich bei der zu ladenden Funktion um eine interne Funktion des Curry-Systems handelte und startet den Suchvorgang nochmals, diesmal aber mit dem Prefix `curry4.runtime`.

C.3 Abfangen und Behandeln von Fehlern zur Laufzeit

Oft müssen Fehlermeldungen, die in einem Programmteil erzeugt worden sind, zu einem kontrollierten Abbruch der Ausführung führen. Da es zu umständlich wäre, die Fehlermeldungen durch das gesamte Programm als Rückgabewert mitzuschleppen, benötigt man einen Mechanismus, um eine ganze Reihe von Methodenaufrufen bei Bedarf schlagartig abzubrechen und zu einem bestimmen Programmabschnitt zu springen.

Dazu bieten sich Instanzen der Klasse `Throwable` an: diese Objekte haben die Eigenschaft, daß man sie mit `throw ThrowableObject` „abwerfen kann". Die JVM beginnt dann, alle Methodenaufrufe rückgängig zu machen, und zwar solange, bis eine Methode das Objekt mit einem `try-catch`-Kontrollblock „einfängt“ (siehe Abschnitt 3.1.7).

Davon macht zum Beispiel der Eingabe-Parser des Curry-Systems gebrauch. Die Methode zum Parsen von Listen `parseList()` erzeugt im Fehlerfall eine Instanz der Klasse `ParserException`:

```
private void parseList() throws ScannerException,ParserException {
    readTerm() ;
    int t=sc.nextToken() ;
    if(t==',') {
         parseList() ;
         stack.pushCons(":",2) ;
    }else if(t==sc.TT_WORD && sc.sval.equals("]")) {
         stack.pushCons("[]",0) ;
         stack.pushCons(":",2) ;
    }else
         throw new
             ParserException("',' or ']' expected after list-element") ;
    }
```

Dieser Fehler kann von der Haupteingaberoutine (einer Endlosschleife) abgefangen und ausgegeben werden:

```
     while(true) try{
         ...
         Term term=pparser.readNewTerm() ;
         ...
         (start the execution)
         ...
     }
     catch(Throwable e) {
         nl("\n*** Error in input: "+e.toString()) ;
     }
```

Anhang D

Befehlssatz und Methoden der abstrakten Maschine

D.1 Befehlssatz der abstrakten Maschine

Dieser Abschnitt beschreibt die einzelnen Befehle der abstrakten Maschine. Bevor ein Befehl ausgeführt wird, erhöht die Maschine den Programmzähler. Einige Befehle dürfen nur in einem bestimmten Kontext benutzt werden; z.B. erwartet der Befehl `CmdNewChoice` (Erzeugung von Nichtdeterminismus), daß keine anderen aktiven Blätter im Und-Baum existieren (siehe Kapitel zur Implementierung). Natürlich erzeugt der Compiler immer (in diesem Sinne) korrekten Code.

Weiterhin haben viele Befehle Effekte, die nicht von der Maschine, sondern von der umschließenden Berechnungsstruktur bestimmt werden (z.B. `CmdSimpleBind`). Solche Effekte habe ich hier nicht beschrieben: man kann sie im Kapitel zur Implementierung nachlesen. Auch habe ich hier nur die Befehle vorgestellt, die vom Compiler erzeugt werden — interne Befehle, wie zum Auslösen von Und-Parallelismus, sollten im Quellcode nachgeschlagen werden.

Für eine Erklärung der Begriffe $PC, callstack, termstack, \ldots$ siehe den Abschnitt 4.3 zur abstrakte Maschine. Mit $actFunction$ ist die momentan ausgeführte Funktion gemeint.

Kontrolle des Ausführungspfads

`CmdJump(`$index$`)`

$$PC := (actFunction, index)$$

`CmdNewCall(`Reg`,`$NumSave$`,`$Dest$`)`

$$f(arg_1, \ldots, arg_n) := (Function)Reg$$

falls Funktion f schon ausgewertet worden ist:

$$Reg := f.result$$
$$PC := (actFunction, Dest)$$

sonst:

$PC := (actFunction, dest)$
$create\ entry\ on\ callstack\ and\ save$
  f
  PC
  $regs\ 0, \ldots, NumSave - 1$
  $Reg\ as\ ResultReg$
$Reg_0, \ldots, Reg_{n-1} := arg_1, \ldots, arg_n$
$PC := (f, 0)$

`CmdExecute`

$term := pop\ from\ termstack$
$f(arg_1, \ldots, arg_n) := (Function)Term$
$Reg_0, \ldots, Reg_{n-1} := arg_1, \ldots, arg_n$
$PC := (f, 0)$

`CmdReturn`

$restore\ from\ callstack:$
  f
  PC
  $registers$
  $ResultReg$
$Result := pop\ from\ termstack$
$store\ Result\ in\ f.result$
$ResultReg := Result$

Termstackoperationen

`CmdPushCons(`$Name$`,`$NumArgs$`)`
`CmdPushFunc(`$Name$`,`$NumArgs$`)`

$arg_1, \ldots, arg_{NumArgs} := pop\ from\ termstack$
$create\ Name(arg_1, \ldots, arg_{NumArgs})\ on\ termstack$

`CmdPushVar`

$create\ new\ var\ on\ termstack$

`CmdPushPrimInt(`N`)`

$push\ integer\ N\ to\ termstack$

`CmdPushReg(`Reg`)`

$push\ register\ Reg\ to\ termstack$

Registeroperationen

`CmdCopyReg(`$SrcReg$`,`$DestReg$`)`

$$DestReg := SrcReg$$

`CmdNewVar(`Reg`)`

$$Reg := new\ variable$$

`CmdLoadReg(`Reg`,`$TempReg$`)`

$$Reg := TempReg$$

`CmdLoadArgs(`Reg`)`

$$c(arg_1, \ldots, arg_n) := (Constructor)Reg$$
$$TempReg_1, \ldots, TempReg_n := arg_1, \ldots, arg_n$$

Linke Regelseite

`CmdNewChoice(`$Index$`,`$NumArgs$`)`
`CmdOr(`$Index$`,`$NumArgs$`)`

$$create\ or - parallelism\ at\ line\ Index$$
$$copy\ NumArgs\ regs\ to\ the\ new\ machine$$

`CmdSwitchOnType(`Reg`,`$IsFunc$`,`$IsVar$`)`

$Term := (Term)Reg$
if $Term = variable$ then
  $Reg := Term.deref$
  $Term := (Term)Reg$
if $Term = function$ then
  $PC := (actFunction, IsFunc)$
else if $Term = variable$ then
  $suspend\ if\ other\ active\ machines$
  $PC := (actFunction, IsVar)$

`CmdSwitchOnDet(`Reg`,`$IsFunc$`,`$IsVar$`)`

$Term := (Term)Reg$
if $Term = variable$ then
  $Reg := Term.deref$
  $Term := (Term)Reg$
if $Term = function$ then
  $PC := (actFunction, IsFunc)$
else if $Term = variable$ then
  $PC := (actFunction, IsVar)$

`CmdSwitchOnRigid(`Reg`,`$IsFunc$`)`

> $Term := (Term)Reg$
> if $Term = variable$ then
> $\quad Reg := Term.deref$
> $\quad Term := (Term)Reg$
> if $Term = function$ then $PC := (actFunction, IsFunc)$
> if $Term = variable$ then $suspend$

`CmdSwitchOnCons(`Reg`,`$JumpTable$`)`

> $Cons := (Constructor)Reg$
> $Dest := JumpTable[ConsIndex]$
> if $Dest = -1$ then fail else $PC := (actFunction, Dest)$

Bindungen und Nichtdeterminismus

`CmdSimpleBind(`Reg`)`

> $Var := (Variable)Reg$
> $Term := pop\ from\ termstack$
> $bind\ Var\ to\ Term$

`CmdDetBind(`Reg`)`

> In Verbindung mit `CmdSwitchOnDet`:
> $Var := (Variable)Reg$
> $Term := pop\ from\ termstack$
> $bind\ Var\ to\ Term$

Spezielle Befehle

`CmdCustom(`$JavaMethod$`)`

die abstrakte Maschine ruft die Java-Methode *JavaMethod* auf und fährt nach Ausführung der Methode mit dem nächsten Befehl fort. Dieser Befehl dient nur zur Optimierung (siehe Abschnitt 4.7): prinzipiell reicht der Befehlssatz der Maschine aus um Curry-Funktionen zu implementieren.

Für die aufgerufene Methode stellt die Maschine ihrerseits Methoden zur Verfügung, die den vorgestellten Befehlen ähneln (Zugriff auf Register,...). Einige Operationen, wie z.B. der Choice-Operator, sind *nur* über diese Methoden ausführbar. Eine vollständige Beschreibung befindet sich in der Datei `runtime/machine.pizza`, jedoch möchte ich hier nochmal die *häufigsten* Operationen aufzählen:

D.2 Methoden der abstrakten Maschine

`Term getReg(int` reg`)`	gibt den Inhalt des Registers reg zurück
`void returnWith(Term result)`	vergleiche `CmdReturn`
`void continueWith(Function func)`	vergleiche `CmdExecute`
`Variable allocNewVariable()`	vergleiche `CmdPushVar`

Anhang E

Codeerzeugung für die abstrakte Maschine

E.1 Patternmatching

E.1.1 Berechnung der Registerbelegung

Die Übersetzung einer Curry-Funktion in eine Java-Klasse basiert auf dem Definitional Tree der Funktion. Wie im Abschnitt 2.2 erläutert, besitzt jeder Definitional Tree ein Pattern. Diese Pattern sind wichtig für die Übersetzung: sie geben nämlich an, welche Argumente zu welchem Zeitpunkt mit Konstruktoren gematcht werden. Die Vorgehensweise des Compilers besteht darin, die in den Pattern vorkommenden Variablen in Registern der abstrakten Maschine zu halten. Das oberste Pattern (d.h. das Pattern des Definitional Tree der Funktion) enthält natürlich keine Konstruktoren, d.h. hat die Form ($X_0, \ldots, X_{n-1}$ sind Variablen)

$$f(X_0, \ldots, X_{n-1})$$

für eine n-stellige Funktion f. Hier ist die Registerbelegung einfach zu ermitteln: da die Maschine die Argumente einer Funktion in ihre Register lädt, kann man der Variablen X_i das Register i zuordnen. Das bedeutet also, daß die rechte Seite der Funktionsgleichung

```
f(X)=X
```

in einen Zugriff auf das Register 0 übersetzt wird.

Natürlich bleibt die Registerbelegung nicht konstant: ein Patternmatchingschritt (durch einen branch-Baum) führt neue Variablen ein (z.B. `f(X)` nach `f(cons(Y,Z))`) oder eliminiert eine Variable (z.B. `add(X,Y)` nach `add(z,Z)`). Das hat zur Folge, daß, wann immer ein Unterbaum eines Definitional Trees übersetzt werden soll, zuerst eine neue Registerbelegung berechnet werden muß. Die Registerbelegung wird möglichst kompakt gehalten (d.h. keine unbenutzten Register zwischen zwei benutzten Register), damit Befehle wie `CmdNewCall` (siehe Anhang D) sowenig Register wie möglich sichern müssen. Da die Arbeitsweise der Registerbelegung *von der Idee her* relativ einfach ist, verzichte ich hier auf weitere Details.

Während der Übersetzung erhält jede Variable im Definitional Tree das zugehörige Register als Attribut. Im folgenden steht der Ausdruck $register(X)$ für das Register, das mit der Variablen X belegt ist.

Gleichzeitig mit der Berechnung der Registerbelegung wird der Code für die abstrakte Maschine erzeugt, der die Aufgabe hat, einen Konstruktor in seine Argumente zu zerlegen und diese auf die Register der Maschine zu verteilen. Dieser Code wird nur an zwei Stellen benötigt:

1. bei einem Übergang von einem Definitional Tree auf einen Subtree der Form $rule(\ldots)$.
2. bei einem Übergang von einem Definitional Tree auf einen Subtree der Form $branch(\ldots)$

Während eines solchen Übergangs kann sich nur *eine* Variable des Pattern in einen Konstruktor (zu sehen im Pattern des Subtrees) umwandeln.

Im Prinzip können die beiden oben genannten Fälle gleich behandelt werden. Da der Compiler aber für rule-Knoten direkt Java-Code erzeugt, unterscheiden sich die Implementierungen (siehe unten).

E.2 Die Übersetzung der Definitional Trees

Die Übersetzungsfunktion hat allgemein die Form

$$genCode(tree, usedRegs, parentPattern)$$

$parentPattern$ ist das Pattern des übergeordneten Baumes. $usedRegs$ gibt an, wieviele Register momentan in Gebrauch sind.

Die Übersetzung einer n-stelligen Funktion mit Baum DT geschieht dann in zwei Schritten:

1. die Variablen des Patterns von DT auf die Register verteilen (siehe oben)
2. Aufruf der Übersetzung:
$$genCode(DT, n, null)$$

E.2.1 or-Knoten

$$genCode(or(T_1, T_2)) \quad = \quad \left[\begin{array}{ll} & \texttt{CmdOr}(\text{label1}, usedRegs) \\ & genCode(T_1, usedRegs, parentPattern) \\ \text{label1:} & \\ & genCode(T_2, usedRegs, parentPattern) \end{array} \right]$$

E.2.2 branch-Knoten

$$genCode(branch(pattern, pos, rigid, T_1, \ldots, T_n), usedRegs, parentPattern)$$
$$= \left[\begin{array}{ll} & \text{if } parentPattern \neq null \text{ then } prepareArgs \\ \text{start:} & \\ & \texttt{CmdSwitchOnRigid}(switchReg, \text{isFunc}) \\ & genConsCode \\ \text{isFunc:} & \\ & genFuncCode \end{array} \right]$$

$$genCode(branch(pattern, pos, flex, T_1, \ldots, T_n), usedRegs, parentPattern)$$

$$
= \left[\begin{array}{ll}
 & \text{if } parentPattern \neq null \text{ then } prepareArgs \\
\text{start:} & \\
 & \texttt{CmdSwitchOnType}(switchReg,\text{isFunc},\text{isVar}) \\
 & genConsCode \\
\text{isFunc:} & \\
 & genFuncCode \\
\text{isVar:} & \\
 & genVarCode
\end{array} \right]
$$

mit

$$
\begin{array}{rcl}
switchReg & = & register(pattern|_{pos}) \\
genConsCode & = & \left[\begin{array}{ll}
 & \texttt{CmdSwitchOnCons}(switchReg, JumpTable) \\
\text{cons}_1\text{:} & \\
 & genCode(T_1, usedRegs, pattern) \\
cons_2\text{:} & \\
 & \ldots \\
\text{cons}_n\text{:} & \\
 & genCode(T_n, usedRegs, pattern)
\end{array} \right] \\
 & & \text{siehe Abschnitt 4.2.1 für den Aufbau der } JumpTable \\
genFuncCode & = & \texttt{CmdNewCall}(switchReg, usedregs - 1, start), \\
 & & \quad \text{falls } usedregs - 1 = switchReg \\
 & = & \texttt{CmdNewCall}(switchReg, usedregs, start), \quad \text{sonst} \\
genVarCode & = & \left[\begin{array}{ll}
\text{choice}_1\text{:} & \\
 & \texttt{CmdNewChoice}(\text{choice}_2, usedRegs) \\
 & \text{Aufbau eines Konstruktorterms } pattern(T_1)|_{pos} \\
 & \text{auf dem Termstack} \\
 & \texttt{CmdSimpleBind}(switchReg) \\
 & \texttt{CmdJump}(\text{cons}_1) \\
\text{choice}_2\text{:} & \\
 & \ldots \\
\text{choice}_n\text{:} & \\
 & \text{Aufbau eines Konstruktorterms } pattern(T_n)|_{pos} \\
 & \text{auf dem Termstack} \\
 & \texttt{CmdSimpleBind}(switchReg) \\
 & \texttt{CmdJump}(\text{cons}_n)
\end{array} \right]
\end{array}
$$

Die Funktion *prepareArgs* vergleicht das Pattern *parentPattern* des übergeordneten Baumes mit dem Pattern des branch-Baumes. Der Unterschied kann nur die Form

$$\text{Variable } X \text{ gegenüber Konstruktor } cons(A_1, \ldots, A_n)$$

haben. Die Argumente des Konstruktors werden mit Hilfe des `LoadArgs`-Befehls und einer Folge von `LoadReg`-Befehlen in die Register der Maschine geladen. Da das Register von X nicht mehr benötigt wird, wird sein Inhalt durch A_1 überschrieben; die restlichen Argumente werden hinter den schon benutzten Registern geladen.

Für den Fall, daß der Konstruktor 0-stellig ist ($n = 0$), wird mit Hilfe des `CmdCopyReg`-Befehls das letzte belegte Register in das Register von X kopiert. So entsteht keine Lücke und Befehle wie `CmdNewCall` können optimal arbeiten.

E.2.3 rule-Knoten

$$genCode(rule(left, right), usedRegs, parentPattern) = \texttt{CmdCustom}(MethodName)$$

Die Java-Methode $MethodName$ enthält den eigentlichen Code. Der erste Schritt besteht auch hier darin, den Unterschied zwischen dem $parentPattern$ und dem aktuellen Pattern (d.h. $left$) zu finden, wieder in der Form

$$\text{Variable } X \text{ gegenüber Konstruktor } cons(A_1, \ldots, A_n)$$

Der Konstruktor wird dann als erstes in eine lokale Variable der Methode geladen und anschließend der Code für die rechte Regelseite ausgeführt. Mit $register(X) = consReg$ hat man also:

```
private static void MethodName(Machine machine) {
    Constructor cons=(Constructor)machine.getReg(consReg) ;
    code ; exitCode
}
```

wobei

$$(name, code) = genTerm(right)$$

$exitCode$ = `machine.continueWith(`$name$`) ;`,
falls $right$ eine vollständige Funktionsapplikation ist
= `machine.returnWith(`$name$`) ;`, sonst

Die Funktion $genTerm$ gibt das Paar $(name, code)$ zurück. In $code$ steht der erzeugte Code und $name$ ist der eindeutige Name einer lokalen (Java-)Variablen, der der erzeugte Term zugewiesen wurde.
Für Konstruktorapplikationen:

$genTerm(c(a_1, \ldots, a_n))$
= ($neuerName$,

```
Constructor neuerName=new Constructor(...) ;
code_1
neuerName.addArg(name_1)
...
code_n
neuerName.addArg(name_n)    )
```

mit

$$(name_i, code_i) = genTerm(a_i)$$

Für 0-stellige Konstruktoren gibt es eine Optimierung: hier kann der Code direkt als Ausdruck in $name$ zurückgegeben werden.

$$genTerm(c) = (\texttt{new Constructor(...)}, null)$$

Die Behandlung von Funktionsapplikationen möchte ich hier nicht beschreiben, da sie vergleichbar mit der obigen Übersetzungsfunktion ist. Interessanter ist die Übersetzung von Variablen, denn hier gibt es vier Fälle zu berücksichtigen:

1. Die Variable kommt nicht nur auf der linken Seite *left* vor, sondern auch im *parentPattern*, d.h. die Variable ist eine Argumentvariable und befindet sich schon in einem Register:

$$genTerm(X) = (\texttt{machine.getReg(}reg\texttt{)}, null), \qquad \text{mit } register(X) = reg$$

2. Die Variable kommt zwar auf der linken Seite vor, aber nicht im *parentPattern*. Sie ist nämlich erst durch den Patternmatchingschritt entstanden und ein Argument des Konstruktors *cons* (siehe oben):

$$\begin{aligned} genTerm(X) &= (\texttt{cons.getArg(}n\texttt{)}, null), \\ &\qquad \text{falls } X \text{ das } n\text{-te Argument des Konstruktors ist.} \end{aligned}$$

3. Die Variable kommt nur auf der *rechten* Seite vor (d.h. ist eine logische Variable) und muß erst erzeugt werden:

$$genTerm(X) = (Name, \texttt{Variable } Name\texttt{=machine.allocNewVariable()})$$

   wobei *Name* ein neuer Java-Bezeichner ist.

4. Die Variable kommt nur auf der *rechten* Seite vor, wurde aber schon erzeugt (siehe vorigen Schritt) und steht in der Java-Variablen *name* zur Verfügung:

$$genTerm(X) = (name, null)$$

E.3 Ergänzungen

Die hier vorgestellte Übersetzungsfunktion enthält einige Vereinfachungen. Die tatsächliche Implementierung enthält Optimierungen und muß noch einige Spezialfälle behandeln, insbesondere die drei folgenden:

1. Enthält ein flex branch-Knoten nur *einen* Subtree, werden die Befehle `SwitchOnDet` und `DetBind` statt `SwitchOnType` und `SimpleBind` benutzt.

2. Einige Funktionen besitzen einen branch-Knoten in ihrem Definitional Tree, *ohne* einen Patternmatchingschritt ausführen zu wollen, z.B. die rigid-Funktion `+`:

```
+ branch +(X,Y),1,rigid,
      branch +(X,Y),2,rigid,
            rule +(X,Y) -> inline("CmdPrimIntAdd(0,1)")
```

3. Eine Sonderbehandlung verlangen auch Variablen, die in den Constraints des Choice-Operators vorkommen (Abschnitt 2.1.9). Da diese Variablen ja eigentlich für eine Berechnungsstufe mit einem höheren Level gedacht sind (d.h. nur lokal für die Constraints), wird ihre Erzeugung anders gehandhabt als bei den anderen Variablen. Siehe dazu den Quellcode.

Literaturverzeichnis

[1] P. B. Galvin A. Silberschatz. *Operation System Concepts.* Addison-Wesley Publishing Company, fourth edition, 1994.

[2] S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

[3] P. Codognet and D. Diaz. `wamcc`: Compiling Prolog to C. In *Proc. of the 12th International Conference on Logic Programming*, pages 317–331. The MIT Press, 1995.

[4] 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.

[5] B. Demoen and P. Tarau. JProlog.
Available at `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`, 1996.

[6] F. Baader and C. A. Albayrak. *Termersetzungssysteme, Skript zur Vorlesung an der RWTH Aachen.* Verlag der Augustinus Buchhandlung, 1995.

[7] G. Gupta. *Multiprocessor Execution of Logic Programs.* Kluwer Academic Publishers, 1994.

[8] M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[9] 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.

[10] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1997.

[11] R. Hinze. *Einführung in die funktionale Programmierung mit Miranda.* Teubner, Stuttgart, 1992.

[12] S. Janson and S. Haridi. Kernel Andorra Prolog and its computation model. In *Proc. of the 7th International Conference*, pages 31–46. MIT Press, 1990.

[13] S. Janson and S. Haridi. Programming paradigms of the Andorra Kernel Language. In *Proc. 1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[14] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[15] M. Hanus, R. Sadre, and F. Steiner. *TasteCurry, Version of November 29,1996*, 1996.

[16] J. J. Moreno-Navarro and M. Rodríguez Artalejo. Logic programming with functions and predicates: The language BABEL. In *Journal of Logic Programming*, pages 12:191–223. 1992.

[17] P. Hudak, S. Preyton Jones, and P. Wadler. Report on the Programming Language Haskell (Version 1.2). *SIGPLAN Notices*, 27 (5), 1992.

[18] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Development*, pages 324–343. Springer LNCS 1000, 1995.

[19] F. Steiner. Eingekapselte Suchverfahren für logisch-funktionale Programmiersprachen. Master's thesis, Lehr- und Forschungsgebiet Informatik II an der RWTH-Aachen, 1997.

[20] Swedish Institute of Computer Science. *SICStus Prolog User's Manual, Release 3 #3*, 1996.

[21] V. Penner. Parallelisierung Logischer Sprachen, Ausarbeitung zur Vorlesung Sommersemester 1995 an der RWTH Aaachen.

[Java] **Java, Sun Microsystems**

Die Java-Homepage befindet sich auf: `www.javasoft.com`
Dort ist auch die gesamte Dokumentation, einschließlich der Spezifikation der JVM, erhältlich:
`www.javasoft.com/docs/index.html`.

[Pizza] **Pizza**

Die Pizza-Homepage befindet sich auf: `www.cis.unisa.odu.au/~pizza/`
Hier lassen sich die aktuellste Version des Compilers und der Dokumentation herunterladen.