$\mathcal{L}i$SA

Realisierung eines interaktiven Lazy–Evaluators mit Patternmatch

Diplomarbeit
von Sven-Bodo Scholz

am Institut für Informatik und praktische Mathematik
der Christian Albrechts Universität, Kiel

Kiel, den 30.September 1991

Inhaltsverzeichnis

1 Einleitung

Ein Berechnungsparadigma für funktionale Sprachen ist der angewandte λ-*Kalkül* [Back78, Bird88, Fiel88, Hend80, Kogg91]. Er ist ein Kalkül, in dem Programme als Ausdrücke beschrieben werden, die aus Konstanten, Funktionen und Anwendungen (*Applikationen*) von Funktionen auf Argumente bestehen.

Der Kalkül beruht auf Transformationsregeln, die das Anwenden von Funktionen auf aktuelle Parameter beschreiben. Primitive Funktionen wie $+$, $-$, $*$, etc. ordnen konstanten Argumentwerten neue Werte zu (δ-*Reduktion*). Deshalb müssen vor ihrer Anwendung die Argumente ausgerechnet werden. Funktionsanwendungen von parametrisierten Funktionen hingegen erfordern die Substitution der formalen Parameter im Rumpf der Funktion durch Argumente (β-*Reduktion*). Ein Ausdruck, in dem keine Reduktionen mehr möglich sind, ist in *Normalform*. Für den λ-Kalkül [Bare84, Chur41, Curr58, Hind86] gilt die Church-Rosser-Eigenschaft. Gibt es also zu einem Ausdruck mehrere Folgen von Transformationsanwendungen (Reduktionen), die ihn in eine Normalform überführen, so sind diese bis auf Namensumbenennungen gebundener Parameter (α-*Konversion*) gleich. Daher ist es möglich, für alle Ausdrücke, die zu einer Normalform reduziert werden können, diese als Wert derselben zu definieren. Um den Wert eines Ausdruckes zu bestimmen, reicht es also aus, *irgendeine* Folge von Reduktionen zu finden, die den Ausdruck in seine Normalform überführt, falls eine solche existiert. Dafür gibt es unterschiedliche Strategien [Bare84, Hind86]. Die beiden wichtigsten sind die *Applicative Order Reduction* und die *Normal Order Reduction*; andere Strategien sind zumeist Mischformen dieser beiden. Bei Applicative Order Reduction werden die Argumente grundsätzlich zur Normalform reduziert, bevor die Funktionsanwendung erfolgt. Bei Normal Order Reduction werden die Argumente nur so weit reduziert, wie es für die Anwendbarkeit der Funktion erforderlich ist. Bei parametrisierten Funktionen werden deshalb die Argumente unausgewertet in die Funktionsrümpfe eingesetzt.

Ein Rechensystem für funktionale Sprachen, das auf dem angewandten λ-Kalkül beruht, ist π-RED* [Bloe86, Klug86, Leic89, Sche86, Stra89]. Es ermöglicht die Reduktion von Ausdrücken der funktionalen Programmiersprache K*i*R [Bloe89] (*Ki*el *R*eduction Language). Die dabei zugrunde liegende Berechnungsstrategie ist die Applicative Order Reduction. Im Gegensatz zu imperativen Programmiersprachen wie Pascal oder C können *Programmtransformationen* durchgeführt und die Resultate einzelner Reduktionsschritte betrachtet werden. Dazu wird die Hochsprachendarstellung eines Programms mit einem Editor eingegeben und nach der Reduktion das Resultat in der Hochsprachendarstellung zurückgeliefert. Das Ergebnis kann wieder als Eingabeprogramm für eine erneute Reduktion dienen.

Die Applicative Order Strategie ist jedoch nicht *vollständig*; d.h. es gibt Ausdrücke, zu denen es zwar terminierende Reduktionsfolgen gibt, die Applicative Order Reduction liefert jedoch eine Reduktionsfolge, die nicht terminiert. Dies ist immer dann der Fall, wenn ein Argument, das keine Normalform besitzt, an eine Funktion übergeben wird, die dieses gar nicht oder zumindest nur teilweise benötigt. Der Benutzer muß also bei der Entwicklung von Programmen darauf achten, ausschließlich terminierende Teilausdrücke in Argumentposition zu spezifizieren. Es gibt jedoch etliche Anwendungsbeispiele, in denen eine Nutzung solcher Programmstrukturen eine konzisere Spezifikation zulassen, so z.B.:

- Repräsentationen von unendlichen Datenstrukturen (*Generatoren*), die an selektive Funktionen (*Filter*) übergeben werden;
- Anwendungen von Funktionen auf Argumente, deren Werte von dem Resultat der Funktionsanwendung abhängen;

- Verwendung zyklischer Strukturen schlechthin.

Um solche Strukturen nutzen zu können, bedarf es der Verwendung einer Strategie, die vollständig ist; eine solche ist die Normal Order Strategie.

Die Verwendung der Normal Order Strategie bedeutet jedoch, daß die Argumente unausgewertet in den Funktionsrumpf eingesetzt werden. Wird ein Argument an mehreren Stellen des Rumpfes substituiert (dupliziert), so kann es zu mehrfachen Berechnungen des Argumentes kommen. Dieses Problem läßt sich mittels sogenannter *Lazy Evaluation* umgehen. Dabei werden lediglich Referenzen auf das Argument in den Funktionsrumpf substituiert. Nach der Berechnung des Argumentes zeigen dann alle Referenzen auf das Ergebnis (*sharing*). Das führt dazu, daß ein Argument höchstens einmal evaluiert wird.

Aufgabe der vorliegenden Arbeit ist es, ein Rechensystem für funktionale Sprachen zu entwickeln, das genauso wie π-RED*eine schrittweise Reduktion von K*i*R-Ausdrücken ermöglicht, dabei jedoch Lazy Evaluation verwendet. Des weiteren soll diese auf einem UNIX / C System implementiert werden, um Vergleiche des Ausführungsverhaltens der beiden Systeme zu ermöglichen.

In enger Zusammenarbeit mit Carsten Rathsack (siehe auch [Rath91]) entsteht $\mathcal{L}i$SA (*L*azy *I*nteractive *S*imulator For An *A*pplicative Language). Die realisierte Sprache wird $\mathcal{L}$K*i*R genannt.

Der Berechnungsvorgang wird durch eine abstrakte Maschine, die eine interne Programmdarstellung manipuliert, realisiert. Die Operationalität dieser Maschine, der sog. *Ausführungsmechanismus*, wird zunächst in Form von Funktionsgleichungen, die die Zustandsübergänge der Maschine beschreiben, spezifiziert. Deshalb liegt es nahe, zunächst eine funktionale Implementierung vorzunehmen. Dies ermöglicht nicht nur eine Validierung der Realisierung, sondern stellt aufgrund der Syntaxkompatibilität von K*i*R und $\mathcal{L}$K*i*R außerdem die Generierung eines großen Testbeispiels für $\mathcal{L}i$SA dar. Weiterhin kann durch die Anwendung der *Bootstraptechnik* (Selbstanwendung) erreicht werden, daß große Programmteile nicht in der Programmiersprache C recodiert werden müssen und zudem leicht auf andere UNIX / C Systeme portierbar sind. Die Recodierung des Ausführungsmechanismus in der Programmiersprache C ermöglicht es, Vergleiche zwischen den Laufzeiten von Programmen auf π-RED* und $\mathcal{L}i$SA vorzunehmen.

In der funktionalen Implementierung dienen als Ein- und Ausgabeformate Ascii-Files. Eine interaktive Nutzung des Systems bedarf der Verwendung eines Editors. Ein speziell für die Erstellung funktionaler Programme ausgelegter syntaxgesteuerter Editor Anne [Seeg91] übernimmt diese Aufgabe. Seine Oberfläche basiert auf dem X-Windows System X11-Release4.

Um detaillierte Einblicke in die internen Reduktionsvorgänge zu ermöglichen sowie gute Debug-Möglichkeiten für das Reduktionssystem zu schaffen, wird ein Graphik-Paket entwickelt. Nicht zuletzt aus Kompatibilitätsgründen macht es ebenfalls von dem X-Windows-System Gebrauch. Dabei ist es möglich, eine ursprünglich für Anne erstellte Schnittstelle für Ein-und Ausgaben xtermio [Seeg91] mitzubenutzen. Dieses Graphikpaket gestattet eine baumförmige Darstellung von Programmgraphen sowie mausgesteuerte Veränderungen an diesen. Ebenso sind neben einer schrittweisen Ausführung *einer* Reduktion die Darstellung und Manipulation der für den Ausführungsmechanismus benötigten internen Datenstrukturen wie z.B. Environments möglich.

Nach dieser kurzen Einführung wird in Kapitel 2 die Problemstellung der Arbeit vertieft. In Kapitel 3 werden die bei der Realisierung von $\mathcal{L}i$SA verfolgten Grundkonzepte beschrie-

ben. Dabei werden alternative Ansätze diskutiert und bereits eine grobe Vorstellung der Arbeitsweise des Ausführungsmechanismus vermittelt. Bevor jedoch eine detaillierte Realisierungsbeschreibung erfolgt, wird es in Kapitel 4 dem Leser durch eine Darlegung der Implementierungskonzeption ermöglicht, die einzelnen Komponenten des Systems in die Gesamtimplementierung einzuordnen. Die ausführliche Realisierungsbeschreibung gliedert sich in zwei Abschnitte: Kapitel 5 behandelt die eigentliche Reduktionsphase, während in Kapitel 6 die Transformation der externen in die interne Darstellung und zurück dargestellt werden. Nach einem Resümee der Erfahrungen mit $\mathcal{L}i$SA in Kapitel 7 finden sich schließlich im Anhang die für eine vollständige Beschreibung unerläßlichen formalen Darstellungen.

2 Problematisierung

2.1 Die Grundkonzeption des Reduktionssystems π-RED*

Das Reduktionssystem π-RED*, im weiteren nur noch kurz π-RED* genannt, ist ein System, das eine schrittweise Transformation von funktionalen Programmen in der Programmiersprache K*i*R [Bloe89] (*Ki*el *R*eduction Language) erlaubt. Die Transformationen beruhen auf dem angewandten λ-Kalkül.

Grundlegende Begriffe des λ-Kalküls wie Syntax, Reduktionen und Reduktionsstrategien werden in der Arbeit ohne formale Definition benutzt. Bei Bedarf dafür sei auf die Literatur verwiesen [Bare84, Chur41, Curr58, Hind86].

Die Menge der Konstanten in K*i*R umfaßt Zahlen, Strings, Boolsche Werte sowie primitive Funktionen. Außerdem gibt es höhere Datenstrukturen wie z.B. Listen ("$< \langle expr\_list \rangle >$"). Funktionsanwendungen (Applikationen) können durch "ap $\langle Fkt\_expr \rangle$ to [$\langle arg\_list \rangle$]" oder kurz " $\langle Fkt\_expr \rangle$ [$\langle arg\_list \rangle$] " und Funktionsdefinitionen (Abstraktionen) durch "sub [$\langle var\_list \rangle$] in $\langle expr \rangle$ " dargestellt werden. Die Definition von rekursiven Ausdrücken bzw. Funktionen kann in Form von Gleichungen, eingeleitet durch das Schlüsselwort "def", erfolgen. Dabei stehen auf den linken Seiten der Gleichungen die Namen der Funktionen, gefolgt von einer optionalen Liste von Parametern; auf der rechten Seite befinden sich die Rümpfe der Funktionen. Eine solche Definitionsliste wird durch das Schlüsselwort "in" beendet; ihm folgt der zu berechnende Ausdruck.

Obwohl es sich bei der Fakultätsfunktion um ein sehr bekanntes Beispiel handelt, soll es hier aus zwei Gründen Verwendung finden. Zum einen ist es für den Leser gerade aufgrund der Bekanntheit leicht verständlich und zum anderen soll es später auf andere Art neu formuliert werden.

Beispiel 2.1: Fakultät in K*i*R

```
def
  Fac [N] = if (N le 1)
            then 1
            else (N * Fac[(N - 1)])
in Fac [5]
```

Mit Hilfe eines Editors läßt sich ein zu berechnender Teilausdruck selektieren. Er wird nach der Reduktion durch den transformierten Ausdruck ersetzt. Durch einen sog. Reduktionszähler kann außerdem vorgegeben werden, wieviele Transformationsschritte maximal vorgenommen werden sollen. Dies garantiert für sämtliche Programme ein Terminieren von π-RED* und ermöglicht erst die schrittweise Berechnung von Ausdrücken. Um dies zu verdeutlichen, betrachten wir hier eine Berechnung des Fakultätsprogrammes in drei Stufen. Zunächst wird lediglich ein Transformationsschritt vorgenommen. Dadurch transformiert sich Beispiel 2.1 zu:

```
if (5 le 1)
then 1
else (5 * ap def
               Fac[N] = if (N le 1)
                        then 1
                        else (N * Fac[(N - 1)])
```

```
    in Fac
to [(5 - 1)]
```

Anschließend wird mittels des Editors die Applikation in der Alternative als zu berechnender (Unter-) Ausdruck selektiert. Eine vollständige Berechnung dieses Ausdruckes ergibt:

```
if (5 le 1)
then 1
else (5 * 24)
```

Nach drei weiteren Reduktionsschritten ergibt sich schließlich `120`.

Zur Realisierung derartiger Transformationen muß eine vollständige β-Reduktion unterstützt werden. Sie erfordert bei Namenskonflikten die Umbenennung gebundener Variablen. Um dies zu vermeiden, wird ein Äquivalent zur vollständigen β-Reduktion, das *Berkling'sche Komplement zum λ-Kalkül* [Berk76], realisiert. Dabei werden unter gleichnamige Abstraktionen substituierte Variablen mittels sog. *Protect-Schlüssel* - dargestellt durch "\"-Zeichen vor der Variablen - vor parasitären Bindungen geschützt.

Bei K*i*R handelt es sich um eine ungetypte Sprache. Das ermöglicht die Berechnung von Ausdrücken, die zwar nicht typbar (gemäß Hindley-Milner [Card85, Miln78]), wohl aber reduzierbar sind. Ein Beispiel dafür ist die Funktion Twice angewandt auf die primitive Funktion "`Hd`", die das erste Element einer Liste selektiert.

Beispiel 2.2: Twice in K*i*R

```
def
  Twice [F, X] = F[F[X]]
in Twice[Hd]
```

Aus der Definition von Twice ergibt sich für den Parameter `F` ein Typ $\alpha \to \alpha$. Da "`Hd`" vom Typ $list\alpha \to \alpha$ ist, läßt sich Beispiel 2.2 nicht typen. Trotzdem sind bei geeigneter Argumentwahl (Listen von Listen) sinnvolle Anwendungen denkbar.

Die Anwendung von Twice auf Hd demonstriert außerdem die Möglichkeit, auch partielle Anwendungen bis zur Normalform berechnen zu können (Funktionen als Ergebnis). So läßt sich Beispiel 2.2 zu einer Funktion reduzieren, die auf ein Argument zweimal hintereinander die Funktion '`Hd`" anwendet. Da sich aus dem Programm aus Beispiel 2.2 kein Name für diese Funktion ableiten läßt, entsteht eine namenlose Funktion (Abstraktion) der Form:

Beispiel 2.3: Ergebnis von Beispiel 2.2

```
sub [x]
in Hd[Hd[x]]
```

Um derartige Transformationen effizient realisieren zu können, ist es erforderlich, den eigentlichen Berechnungsvorgang auf einer konziseren Darstellung (*der internen Darstellung*) vorzunehmen; sie enthält keinerlei "Füllwörter" wie z.B. "to, "in" etc., sondern nur die für die Berechnung erforderlichen Informationen über die Struktur bzw. die Konstanten des zu berechnenden Ausdruckes. Aufgrund dieser zweiten Darstellungsform unterteilt sich der Reduktionsprozeß in drei Phasen: erstens die *Preprocessing Phase*, in der der zu berechnende Ausdruck in die interne Darstellung transformiert wird; zweitens die *Processing Phase*,

in der die eigentliche Reduktion stattfindet; und schließlich die *Postprocessing Phase*, die für die Rückübersetzung des Resultates von der internen in die externe Darstellung (K*i*R) sorgt.

Da das System die Transformation von K*i*R Ausdrücken realisieren soll, muß für die Konvertierungsfunktionen von K*i*R in die interne Darstellung (Ψ) und zurück (Ψ^{-1}) gelten:

1. Erfolgt keine Reduktion, so wirkt die Komposition von Ψ und Ψ^{-1} wie die Identität: $\forall e \in E_{\mathrm{K}i\mathrm{R}} : \Psi^{-1}(\Psi(e)) = e$.
2. Jeder Ausdruck, der im Verlauf des Processing entsteht, ist in einen K*i*R Ausdruck transformierbar: $\forall e_i \in E_{int.Darst.} : \exists e \in E_{\mathrm{K}i\mathrm{R}} : \Psi^{-1}(e_i) = e$.

Um während der Processing Phase die Identifizierung zu substituierender Variablen durch Namensgleichheit zu vermeiden, erfolgt die Darstellung von Variablen mittels de Bruijn Indizes [DeBr72]. Dabei wird von Variablennamen abstrahiert und die Bindungsstruktur durch Indizes beschrieben; λ-Bindungen (Abstraktionen) werden als Λ dargestellt, Variablen als $\#\langle index\rangle$. Ein solcher Index bezeichnet die Anzahl der λ-Bindungen, die zwischen dem bindenden Λ und der Variablen liegen (siehe Beispiel 2.4 1.). Global freie Variablen werden mit eindeutigen Indizes größer oder gleich der Anzahl der λ-Bindungen, in deren Rumpf sie auftreten, bezeichnet (siehe Beispiel 2.4 2.).

Beispiel 2.4: De Bruijn Darstellung von Ausdrücken des λ-Kalküls

1. $\lambda z.(\lambda x.\lambda y.x \;\; z)$ wird dargestellt durch $\Lambda.(\Lambda.\Lambda.\#1 \;\; \#0)$.
2. $\lambda z.(\lambda x.z \;\; p)$ wird dargestellt durch $\Lambda.(\Lambda.\#1 \;\; \#1)$.

Die ordnungsgemäße Behandlung von Namenskonflikten erfordert im de Bruijn-Kalkül bei der β-Reduktion die Modifikation von Indizes. Diese Notwendigkeit entsteht in zwei Situationen:

(∗) Wenn Ausdrücke, die relativ freie Variablen enthalten, in Funktionsrümpfe hineinsubstituiert werden, müssen die Indizes der freien Variablen erhöht werden. So wird $(\Lambda.\Lambda.\#1 \;\; \underline{\#0})$ (vergl. Beispiel 2.4 1.; der bei der Reduktion zu modifizierende Index ist unterstrichen) reduziert zu $\Lambda.\#1$.

(∗∗) Wenn der Funktionsrumpf eines Redex relativ freie Variablen enthält, so müssen deren Indizes dekrementiert werden. $(\Lambda.\underline{\#1} \;\; \#1)$ (vergl. Beispiel 2.4 2.; der bei der Reduktion zu modifizierende Index ist unterstrichen) wird zu $\#0$.

In π-RED* wird die laufzeitaufwendige Modifikation von Indizes während der Processing Phase vermieden. Das wird dadurch erreicht, daß eine sog. *Superkombinator-Reduktion* erfolgt. Dazu werden in der Preprocessing Phase alle Funktionen durch β-Extensionen in Superkombinatoren verwandelt.

1. Ein Ausdruck E heißt *abgeschlossen* oder *Kombinator* genau dann, wenn gilt:

$$\begin{array}{ll} E = \lambda x_1 ... x_n.M & \text{mit} \quad n > 0 \\ \wedge & M \neq \lambda x.N \\ \wedge & M \text{enthält außer } x_1, ..., x_n \text{ keine freien Variablen.} \end{array}$$

2. Ein Ausdruck E heißt *Superkombinator* genau dann, wenn gilt:

   E ist Kombinator, und alle in E vorkommenden Abstraktionen sind ebenfalls Superkombinatoren.

Dadurch enthält keine Funktion mehr relativ freie Variablen, und die Situation $(**)$ ist vermieden. Betrachten wir noch einmal Beispiel 2.4 2., so liefert die Superkombinator-Bildung $\Lambda.((\Lambda.\Lambda.\#1\ \ \#0)\ \ \#1)$, und die Reduktion von $((\Lambda.\Lambda.\#1\ \ \#0)\ \ \#1)$ ergibt ohne Indexmodifikation #0. Um die Situation $(*)$ vermeiden zu können, reicht es jedoch nicht aus, alle Funktionen zu Superkombinatoren zu machen. Dies zeigt Beispiel 2.4 1., da $\Lambda.(\Lambda.\Lambda.\#1\ \ \#0)$ bereits ein Superkombinator ist. Ursache des Problems ist, daß bei einer partiellen Anwendung Ausdrücke und somit ggf. relativ freie Variablen in einen Funktionsrumpf substituiert werden. Deswegen werden bei der Superkombinator-Reduktion keine partiellen Anwendungen vorgenommen. Beispiel 2.4 1. ist im Sinne der Superkombinator-Reduktion nicht reduzierbar.

Die Verwendung der Superkombinator-Reduktion dient jedoch nicht nur der Vermeidung von Indexmodifikationen, sondern erleichtert auch die Zuführung der aktuellen zu den formalen Parametern. Das liegt daran, daß alle benötigten Parameter der Funktion direkt (via Applikation) zugeführt werden. So kann für die Substitution der formalen Parameter eine Stackstruktur genutzt werden, die die Argumente bereitstellt.

Die Beschränkung der Processing Phase auf eine naive Substitution mittels Superkombinatoren macht in der Postprocessing Phase ein Auflösen der Superkombinatoren sowie eine Vervollständigung der β-Reduktion (partielle Anwendungen!) erforderlich.

2.2 Das Patternmatch

Um die Analyse bzw. Dekomposition von komplexen Datenstrukturen zu vereinfachen, stellt π-RED* das sogenannte *Patternmatch* zur Verfügung [Bark89]. Dabei werden Argumente mit Mustern (*Pattern*) verglichen. Ein Pattern kann aus Konstanten, Variablen sowie daraus zusammengesetzten Datenstrukturen bestehen. Ein Pattern *paßt* genau dann auf ein Argument, wenn es eine Variablenbelegung für die Variablen des Patterns gibt, sodaß das Pattern unter der Variablenbelegung dem Argument bedeutungsgleich ist.

1. Zwei Ausdrücke E_1, E_2 sind *gleich* ($E_1 = E_2$), genau dann, wenn sie bis auf Namensumbenennungen von gebundenen Variablen (*α-Konversionen*) gleich sind.

2. Zwei Ausdrücke E_1, E_2 sind *bedeutungsgleich* ($E_1 =_m E_2$), genau dann, wenn es zwei gleiche Ausdrücke E_1' und E_2' gibt, so daß sich E_i zu E_i' mit $i \in \{1, 2\}$ reduzieren läßt.

Der Vorgang der Überprüfung, ob ein Pattern auf ein Argument paßt, heißt *Patterntest*. Eine *π-Abstraktion* besteht aus einem oder mehreren Pattern, einem *Guard* und einem *Patternrumpf*. Der Guard dient dazu, Bedingungen an die Argumente stellen zu können, die über rein strukturelle Anforderungen hinausgehen. Eine π-Abstraktion *paßt* genau dann auf n Argumente, wenn sie n Pattern enthält, die auf die n Argumente passen und der Guard sich zu true reduzieren läßt. Nur in diesem Fall findet eine Berechnung des Rumpfes statt. Dabei gelten die durch die Pattern gefundenen Variablenbelegungen sowohl im Guard als auch im Rumpf. Die Ermittlung, ob eine π-Abstraktion auf einen Satz von Argumenten paßt, heißt *Matchvorgang*. Ein π-Abstraktion hat in K*i*R folgende Syntax[1]:

[1] Die vorliegende Beschreibung beschränkt sich auf die wesentlichen Konstrukte; eine vollständige Beschreibung findet sich in [Bloe89]

$$
\begin{array}{lll}
\langle pattern\_expr \rangle & \Longrightarrow & \text{when } \langle patterns \rangle \text{ guard } \langle expr \rangle \text{ do } \langle expr \rangle \\
\langle patterns \rangle & \Longrightarrow & \langle pattern \rangle \quad | \quad (\langle pattern \rangle \Big(, \langle pattern \rangle \Big)^{+}) \\
\langle pattern \rangle & \Longrightarrow & \langle konst \rangle \quad | \quad \langle var \rangle \quad | \\
 & & < \langle pattern \rangle \Big(, \langle pattern \rangle \Big)^{*} >
\end{array}
$$

Die eigentliche Ausdruckskraft (*expressive power*) des Patternmatches einer funktionalen Sprache entsteht jedoch erst durch die Möglichkeit, alternative π-Abstraktionen spezifizieren zu können. Eine solche *Case-Abstraktion* hat in K*i*R folgende Syntax:

$$
\begin{array}{lll}
\langle case\_expr \rangle & \Longrightarrow & \text{case } \langle pattern\_expr \rangle \Big(, \langle pattern\_expr \rangle \Big)^{*} \text{ otherwise } \langle expr \rangle \text{ end } | \\
 & & \text{case } \langle pattern\_expr \rangle \Big(, \langle pattern\_expr \rangle \Big)^{*} \text{ end}
\end{array}
$$

Dabei ist die Bedeutung einer Anwendung auf Argumente die erste π-Abstraktion, deren Pattern auf die Argumente passen. Gibt es keine solche, so bildet der hinter dem `otherwise` stehende Ausdruck das Resultat; existiert dieser nicht, so läßt sich die Case-Abstraktion nicht anwenden.

π-RED* arbeitet gemäß Applicative Order Strategie. Daraus ergibt sich, daß zu Beginn des Matchvorganges einer Case-Abstraktion sämtliche Argumente ausgewertet vorliegen. Deshalb müssen während der Patterntests keine Reduktionen erfolgen und jeder Patterntest kann konzeptuell als ein Schritt betrachtet werden. Dies wird in der Realisierung dahingehend genutzt, daß jedes Pattern während der Preprocessing Phase in Code übersetzt wird und eine abstrakte Maschine [Bark89] diesen während der Processing Phase interpretiert. Der Matchvorgang einer Case-Abstraktion wird durch eine sequentielle Abfolge solcher Code-Interpretationen realisiert. Liegt eine strukturelle Ähnlichkeit in aufeinanderfolgenden Pattern einer Case-Abstraktion vor (*überlappende Pattern*), so bleiben diese unberücksichtigt. Diese Vorgehensweise führt bei überlappenden Pattern zwar zu redundanten Tests, der Testvorgang ist jedoch aus dem Quelltext ohne großen Zeitaufwand ableitbar.

2.3 Auswertungsstrategien

π-RED* selektiert den als nächstes zu reduzierenden Redex nach der Applicative Order Strategie. Daraus ergibt sich, daß vor jeder Funktionsanwendung zunächst alle Argumente berechnet werden. Das schränkt zwar nicht die Menge der berechenbaren Probleme ein, da z.B. die Simulation einer Turing-Maschine durch ein K*i*R-Programm möglich ist, wohl aber die Menge der in K*i*R-Syntax spezifizierbaren Programme, deren Berechnung zu einer Normalform führt. So führt z.B. die Codierung der einfachen Rekursion mittels des Y-Kombinators in π-RED* unabhängig von dem Funktionsrumpf nie zu einer Normalform. Um allen Ausdrücken, die eine Normalform besitzen, diese auch zuordnen zu können, ist die Verwendung einer vollständigen Strategie wie der Normal Order Reduction erforderlich. Sie bewirkt, daß die Argumente nur soweit ausgerechnet werden, wie es für die Anwendung der Funktion notwendig ist.

Um eine δ-Reduktion ausführen zu können, dürfen die Argumente zwar nicht in Form von Applikationen vorliegen, aber es bedarf auch nicht der Normalform der Argumente. So reicht es z.B. für die Anwendung von `Hd` auf ein Argument aus, festzustellen, ob das Argument zu einer Liste reduzierbar ist; die Berechnung der Listenkomponenten ist für die δ-Reduktion nicht erforderlich. Zur besseren Beschreibung dieses Sachverhaltes bedarf es eines neuen Normalform-Begriffes, der *Weak Head Normal Form.*

Ein Ausdruck E ist in *Weak Head Normal Form* (WHNF) genau dann, wenn gilt:

$$E = FA_1...A_m \quad \text{mit} \quad m \geq 0$$
$$\wedge \quad F \text{ ist keine Applikation}$$
$$\wedge \quad \forall n \leq m : (FA_1...A_n) \text{ ist kein Redex.}$$

Damit ergibt sich: Vor der Anwendung einer primitiven Funktion müssen die zugehörigen Argumente zur WHNF (falls existent) berechnet werden.

Bei der Anwendung einer vom Benutzer spezifizierten Funktion werden die Argumente unausgewertet in den Funktionsrumpf substituiert. Zur Vermeidung von daraus resultierenden Mehrfachberechnungen findet bei der Realisierung Lazy Evaluation Verwendung.

Sie kombiniert den Vorteil der Vollständigkeit der Normal Order Reduction mit der Eigenschaft der Applicative Order Reduction, jedes Argument höchstens (bei Applicative Order Reduction: genau) einmal zu berechnen. Dies erschließt dem Anwender die Möglichkeit, Algorithmen zu verwenden, die die Verwendung von nicht strikten Funktionen für die Terminierung der Berechnung nutzen.

Eine ganze Klasse von Beispielen nutzt Repräsentationen von unendlichen Datenstrukturen (*Generatoren*), die an selektive Funktionen (*Filter*) übergeben werden. Exemplarisch soll hier eine solche Datenstruktur genutzt werden, um das Fakultätsbeispiel aus Beispiel 2.1 zu recodieren. Da solche Datenstrukturen immer Zyklen enthalten, erweist sich eine Darstellung durch Binärlisten als sinnvoll. Die Verwendung von "[⟨*expr*⟩.⟨*expr*⟩]" als externe Darstellung von Binärlisten und "`Tl`" als Funktion, die die rechte Komponente einer Binärliste selektiert, führt zu folgendem Programm:

Beispiel 2.5: Fakultät mittels der unendlichen Liste aller Fakultäten

```
def
  Map2f [F] = when ([A.B],[C.D]) guard true do [ F[A,C] . Map2f[F,B,D] ]
  Ones [] = [ 1 . Ones ]
  Nats [] = [ 1 . Map2f[+,Ones,Nats] ]
  Facs [] = [ 1 . Map2f[*,Tl[Nats],Facs] ]
in select[5,Facs]
```

Die Funktion `Map2f` erhält drei Argumente: eine dyadische Funktion `F` und zwei Binärlisten konzeptuell beliebiger Länge. Sie verknüpft diese Listen elementweise mittels der Funktion `F`. `Ones` stellt eine Liste von Einsen dar. Durch geeignete Verwendung von `Map2f` repräsentiert `Nats` die Liste der natürlichen Zahlen und `Facs` die Liste der Fakultäten der natürlichen Zahlen. Es liegen also drei Generatoren vor. Als Filter dient die primitive Funktion `select`.

Während ein Applicative Order System wie π-RED* versucht, die Generatoren vollständig zu berechnen, werden in einem Lazy Evaluator die Generatoren nur soweit berechnet, wie es für die Selektion des fünften Elementes von `Facs` erforderlich ist. Auf diese Weise wird die Normalform `120` gefunden.

Eine andere Klasse von Beispielen beruht auf der Attributierung von Grammatiken [Bird84, Burg75]. Die Problemstellung ist dabei, einen Ableitungsbaum mit generischen Attributen zu versehen, die von synthetischen abhängen, ohne den Baum zweimal traversieren zu müssen. Das bekannteste Beispiel dafür ist das "Mintree-Beispiel", bei dem die Blätter eines Baumes durch ihr Minimum ersetzt werden sollen. Diese Art von Problemen ist mittels Lazy Evaluation lösbar, indem das unberechnete synthetische Attribut schon bei dem Abstieg in den Baum für die Berechnung der generischen Attribute mitgegeben

werden kann. In einem Applicative Order System ist dies nicht möglich und es bedarf zwei Traversiervorgängen.

Die hier erwähnten Beispiele machen es neben anderen wünschenswert, ein auf Lazy Evaluation basierendes System zur Reduktion von Ausdrücken zu nutzen. Das bekannteste solche System ist die G-Maschine. Die Konzeption dieses Systems soll hier kurz vorgestellt werden, um sie in Relation zu der vorliegenden Arbeit stellen zu können.

2.4 Die G-Maschine

Die G-Maschine [Augu85, John84, John85, John86, PJ87] ist ein auf Lazy Evaluation basierendes System, das daraufhin konzipiert wurde, funktional spezifizierte Berechnungen möglichst effizient auf eine von Neumann Architektur abzubilden. Deshalb wird das funktionale Programm zunächst in eine Code-Darstellung, den sog. G-Code, transformiert, der so gestaltet ist, daß er leicht auf von Neumann Architekturen abbildbar ist [John86]. In der eigentlichen Berechnungsphase führt die Ausführung des Codes zur Erzeugung bzw. Analyse von Graphstrukturen (*compilierte Graphreduktion*). Danach wird das Ergebnis mittels des sog. *Print*-Befehls ausgegeben.

In der G-Maschine erfolgen zur Laufzeit keine Typüberprüfungen. Deshalb ist es erforderlich, sicherzustellen, daß keine Typfehler auftreten. Eine notwendige Voraussetzung dafür ist es, nur getypte Programme als Eingabe zuzulassen.

Die Reduktion wird mittels Superkombinator-Reduktion realisiert. Das gestattet bei eindeutiger Benennung aller Funktionen das Anheben lokaler Funktionsdefinitionen auf globales Niveau (λ-*Lifting*) [John85], da keine relativ freien Variablen in den Funktionsrümpfen enthalten sind und ermöglicht somit eine einfache Compilierbarkeit. Um Namenskonflikte zu vermeiden, bedingt die Superkombinator-Reduktion jedoch, daß keine partiellen Anwendungen erfolgen (vergl. Kapitel 2.1). Liegt eine partielle Anwendung als Ergebnis vor, so wird diese nicht wie in der π-RED* durch die Postprocessing Phase aufgelöst, sondern führt zu einer Fehlermeldung. Auf eine schrittweise Transformation von Programmen wird verzichtet und nur vollständige Berechnungen von Programmen vorgenommen. Im Gegensatz zu π-RED* liefert also die G-Maschine als Resultat einer Berechnung ausschließlich Konstanten bzw. Datenstrukturen darüber.

2.5 Die Aufgabenstellung

Wie aus Kapitel 2.3 hervorgeht, kann die Beschränkung der zur Normalform transformierbaren Ausdrücke, die durch die Verwendung der Applicative Order Strategie entsteht, mittels Lazy Evaluation behoben werden. Dies wird zwar von der G-Maschine erreicht, jedoch schränkt der Verzicht auf Typüberprüfungen während der Laufzeit sowie die fehlende Auflösung von partiellen Anwendungen als Ergebnis wiederum die Menge der berechenbaren Ausdrücke ein.

Ziel dieser Arbeit ist es also, einen Lazy Evaluator - im weiteren kurz $\mathcal{L}i$SA genannt - zu realisieren und implementieren, der *alle* Ausdrücke, die eine Normalform besitzen, berechnen kann. Die Menge der mit $\mathcal{L}i$SA berechenbaren Programme umfaßt also nicht nur die der π-RED* und die der G-Maschine, sondern auch Programme, die neben Lazy Evaluation eine ungetypte Sprache bzw. partielle Anwendungen erfordern. Die von $\mathcal{L}i$SA realisierte Sprache heißt $\mathcal{L}$KiR.

Die architektonischen Merkmale von π-RED* sollen bis auf die Reduktionsstrategie übernommen werden, d.h. es sollen ungetypte Programme berechnet werden können und schritt-

weise Transformationen sowie partielle Anwendungen möglich sein. Die Reduktion von mittels eines Editors selektierten Unterausdrücken erfordert weiterhin den ordnungsgemäßen Umgang mit freien Variablen (Berklingsches Komplement). Um die Austauschbarkeit von Programmen zu gestatten, wird die Syntax von $\mathcal{L}$K*i*R zu der von K*i*R kompatibel gehalten. Soweit die Änderung der Auswertungsstrategie es noch als sinnvoll erscheinen läßt, wird die Beibehaltung der Grundkonzepte von π-RED* angestrebt. Dies soll die Möglichkeit eröffnen, Vergleiche im Ausführungsverhalten der beiden Systeme π-RED* und $\mathcal{L}i$SA vorzunehmen.

3 Die Grundkonzepte des Ausführungsmechanismus

3.1 Die Darstellungsebenen funktionaler Programme

Die Konzeption der Darstellungsebenen kann von π-RED* übernommen werden. Das bedeutet, die Reduktion findet in drei Phasen statt (Preprocessing-, Processing- und Postprocessing Phase) und für die Konvertierungsfunktionen Ψ und Ψ^{-1} gilt (vergl. Kapitel 2.1):

1. Erfolgt keine Reduktion, so wirkt die Komposition von Ψ und Ψ^{-1} wie die Identität: $\forall e \in E_{\mathcal{L}\mathrm{K}i\mathrm{R}} : \Psi^{-1}(\Psi(e)) = e$.
2. Jeder Ausdruck, der im Verlauf des Processing entsteht, ist in einen $\mathcal{L}$KiR Ausdruck transformierbar: $\forall e_i \in E_{int.Darst.} : \exists e \in E_{\mathcal{L}\mathrm{K}i\mathrm{R}} : \Psi^{-1}(e_i) = e$.

3.1.1 Die externe Darstellung

Die externe Darstellung (Syntax von $\mathcal{L}$KiR) stimmt im wesentlichen mit der Syntax von KiR überein. Deshalb sollen hier nur kurz die Unterschiede gegenüber KiR beschrieben werden. Eine vollständige Darstellung der Syntax von $\mathcal{L}$KiR findet sich in Anhang A.

Die Syntax von KiR bietet als einziges Listenkonstrukt eine Darstellung der Form "$< a_1, ..., a_n >$". Da π-RED* gemäß Applicative Order Reduction reduziert, werden Listen auch in Argumentposition stets vollständig ausgerechnet.

Im Zusammenhang mit Lazy Evaluation sind aber gerade Listen mit unausgewerteten Komponenten, sog. *Lazy Listen* von Interesse (vergl. Kapitel 2.3).

Die Programmierpraxis mit π-RED* zeigt, daß Listen häufig als Zustandsrepräsentation für abstrakte Ausführungsmechanismen verwendet werden. Bei schrittweiser Transformation solcher Anwendungen ist es für den Anwender hilfreich, wenn die vollständige Berechnung der Listen bzw. Zustände forciert wird.

Um diese Hilfestellung trotz Lazy Evaluation bereitzustellen, bedarf es in $\mathcal{L}i$SA also neben dem Lazy Listen Konstrukt eines Listenkonstruktes, das die Berechnung seiner Komponenten forciert. Dieses Konstrukt soll im weiteren *Tupel* genannt und durch"$< a_1, ..., a_n >$" dargestellt werden. Die Darstellung der Lazy Listen erfolgt in Form von binären Listen der Art "$[hd.tl]$".

Da die komplexen Datentypen Vektor und Matrix im bisherigen Programmierbetrieb kaum Verwendung fanden und für eine Umstellung der Auswertungsstrategie von geringer Relevanz sind, wird ihre Implementierung zunächst zurückgestellt. Genauso fehlen vorerst Gleitkommazahlen sowie Digitstrings konzeptuell beliebiger Länge. Bei Bedarf sind diese Konstrukte problemlos einfügbar.

Schließlich gibt es vier in KiR nicht vorhandene primitive Funktionen, die der Zerlegung des neuen Listenkonstruktes bzw. der Zerlegung von Strings dienen. Eine ausführliche Beschreibung der primitiven Funktionen findet sich in Anhang B.

3.1.2 Die interne Darstellung

Lazy Evaluation beruht darauf, nur Verweise auf Argumente anstelle der Argumente selbst in Funktionsrümpfe zu substituieren (vergl. Kapitel 2.3). Es ist also für den Ausführungsmechanismus erforderlich, Repräsentationen von Programmteilen (Knoten) und Verweise darauf (Kanten) zu handhaben. Die Verwendung einer graphenförmigen Darstellung als interne Darstellung liegt deshalb nahe. Das erleichtert außerdem die Realisierung der schrittweisen

Transformation, da nach jedem Reduktionsschritt der Programmausdruck als vollständiger Graph vorliegt.

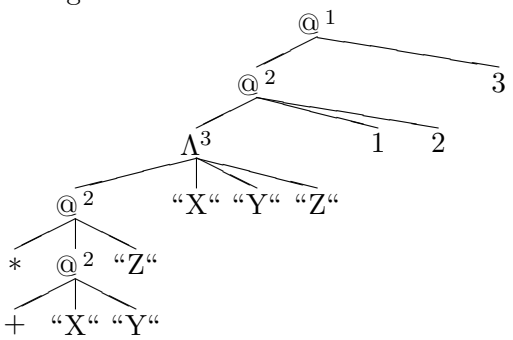
Um eine Menge von Graphen beschreiben zu können, bedarf es einer Menge $\mathcal{K}$ von Graphknoten, einer Menge $\mathcal{K}' \subset \mathcal{K}$ von zulässigen Wurzelknoten, einer Funktion $\nu_{anz} : \mathcal{K} \to \mathbb{N}_0$, die jedem Knoten die Anzahl seiner Nachfolge-Knoten zuordnet, sowie Nachfolgerfunktionen $\nu_k : \{1, \ldots, \nu_{anz}(k)\} \to Pot(\mathcal{K})$, für alle $k \in \mathcal{K}$ mit $\nu_{anz} \geq 1$, die zu einem Graphknoten k für jede Unterkomponente die Menge der möglichen Nachfolgeknoten festlegt. Eine durch die Definition der oben beschriebenen Komponenten charakterisierte Menge von Graphen wird mit $\mathcal{G}_{\mathcal{K}}^{\mathcal{K}'}$ bezeichnet.

So gibt es in der internen Darstellung $\mathcal{L}_{Graph} = \mathcal{G}_{\mathcal{K}}^{\mathcal{K}'}$ Applikations-Knoten $@^n \in \mathcal{K}$ mit $n \in \mathbb{N}$, $\nu_{anz}(@^n) = (n+1)$ und $\nu_{@^n}(m) = \mathcal{K}$ für $m \in \{1, \ldots, (n+1)\}$. Abstraktionen werden dargestellt durch $\Lambda^n \in \mathcal{K}$ mit $n \in \mathbb{N}$, $\nu_{anz}(\Lambda^n) = (n+1)$ und $\nu_{\Lambda^n}(1) = \mathcal{K}$ sowie $\nu_{\Lambda^n}(m) = \mathcal{K}|_{names}$ für $m \in \{2, \ldots, (n+1)\}$. Mit Variablen und Konstanten als Graphknoten ohne Nachfolger wird z.B.:

Beispiel 3.1: Zur internen Darstellung vor Ausdrücken

```
ap ap sub [X,Y,Z]
      in ((X + Y) * Z)
   to [1,2]
to[3]
```

intern dargestellt durch:



Eine vollständige Beschreibung der internen Darstellung $\mathcal{L}_{Graph}$findet sich in Anhang C.

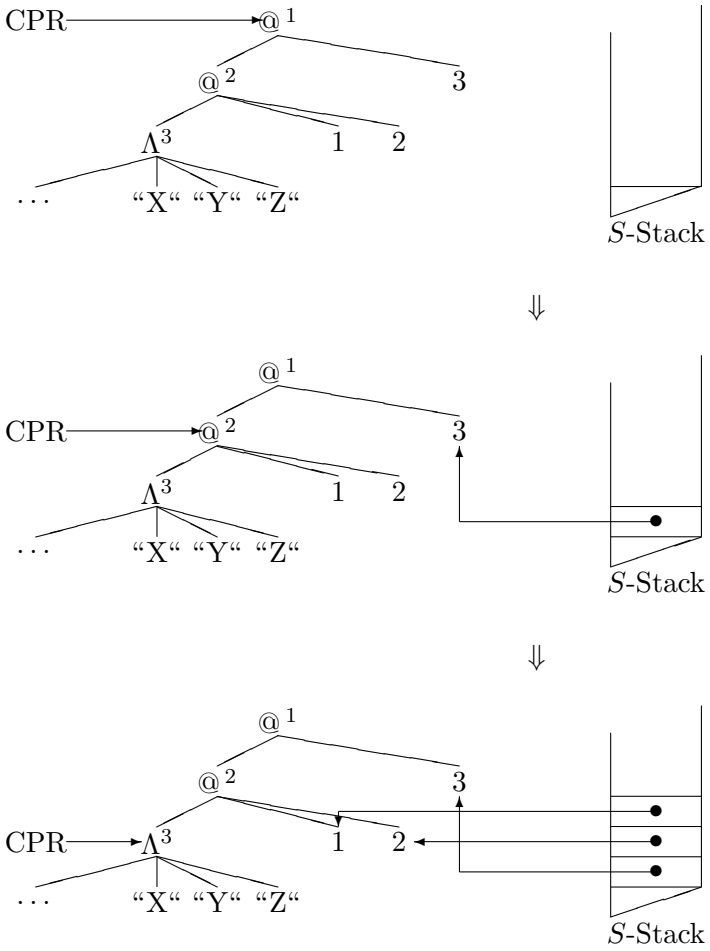
3.2 Das Traversieren von Graphen

Zur Berechnung eines Ausdruckes ist es notwendig, den Programmgraphen geordnet zu durchlaufen, um die Redices identifizieren zu können. Dieser Vorgang heißt *Traversieren*. Das Traversieren des Graphen erfolgt in $\mathcal{L}is$SA mittels eines Zeigers, dem sog. CPR (Current Point Of Reduction), der durch den Graphen bewegt wird.

Ein Redex stellt sich im Graphen so dar, daß der linke Untergraph eines @-Knotens die Graphdarstellung einer Funktion ist. Deswegen wird der CPR solange von einem @-Knoten auf dessen linken Unterknoten umgesetzt, bis er auf die Darstellung einer Funktion (z.B. einen Λ-Knoten) zeigt. In einer solchen Situation wird für die Ausführung der Reduktion ein Zugriff auf die aktuellen Parameter erforderlich; sie befinden sich an den Vaterknoten des Knotens, auf den der CPR verweist. Um diesen Zugriff zu ermöglichen, müssen beim Umsetzen des CPR entweder die Zeiger auf die Argumente (rechte Untergraphen der @-Knoten) in einer gesonderten Datenstruktur, z.B. einem Stack, nachgehalten werden, oder aber der

Zeiger, der vom @-Knoten auf seinen linken Untergraphen zeigt, umgedreht werden (*pointer reversal*) [PJ87]. Da ein häufiges Verändern von Pointern einen hohen Laufzeitaufwand bedeutet, werden in $\mathcal{L}is_{\text{A}}$ die aktuellen Parameter auf einem Stack, dem S-Stack, nachgehalten. Zeigt also der CPR auf eine Funktion, so liegen die aktuellen Parameter auf dem S-Stack. Für Beispiel 3.1 (verkürzte Darstellung) ergibt sich:

Abbildung 3.1: Traversiervorgang für Beispiel 3.1



3.3 Die Instanziierung von Ausdrücken

Ziel der Lazy Evaluation ist es, jedes Argument höchstens einmal zu berechnen. Wird also ein Ausdruck berechnet, der von mehreren Stellen des Graphen referenziert wird, so muß sichergestellt sein, daß nach der Berechnung alle Referenzen auf das Resultat zeigen. Das wird dadurch realisiert, daß der auszurechnende Ausdruck mit dem Ergebnis überschrieben wird (*Update*).

Updates, die in Funktionsrümpfen stattfinden, hängen von der Art der aktuellen Parameter ab. Deshalb ist es zumindest konzeptionell notwendig, Kopien des Funktionsrumpfes zu erzeugen und in diesen dann die formalen Parameter durch die aktuellen zu ersetzen. Dieser Vorgang heißt *Instanziieren*.

Das explizite Kopieren eines Graphen stellt jedoch einen erheblichen Aufwand dar, zumal nach dem Erzeugen der Kopie die berechneten Applikationsknoten mit neuen Ausdrücken

(Resultaten) überschrieben und somit nicht mehr benötigt werden. Deshalb werden in $\mathcal{L}i$SA keine solchen expliziten Graphkopien vorgenommen. Die Substitution der formalen durch die aktuellen Parameter wird zunächst zurückgestellt; stattdessen werden die Substitute in einer gesonderten Datenstruktur, dem sog. *Environment*, nachgehalten. Eine Instanz eines Funktionsrumpfes wird also durch ein Tupel repräsentiert, das aus einem Verweis auf den Graphen und einem Verweis auf das zugehörige Environment besteht. Solche Tupel werden im folgenden *Indirektionsknoten* genannt. Der Update einer solchen Instanz erfolgt durch das Überschreiben des Indirektionsknotens.

Auf diese Weise entstehen bei der Berechnung eines Ausdruckes sehr viele Indirektionsknoten. Sie werden auch nach ihrem Update noch benötigt und können erst freigegeben werden, wenn der Ausdruck, den sie repräsentieren, durch eine primitive Funktion konsumiert wird. Allerdings erscheint dieser Aufwand im Hinblick auf die Kopier- und Überschreibersparungen akzeptabel.

3.4 Die Realisierung der β-Reduktion

Analog zur π-RED* soll die Darstellung der Variablen mittels Indizes und die β-Reduktion mittels naiver Substitution, d.h. ohne Veränderung der Indizes, erfolgen.

Durch die Zurückstellung der Substitution bei der Instanziierung von Ausdrücken muß bei der β-Reduktion für die Argumente nicht nur der Graph, sondern auch das zugehörige Environment als Substitut für die Variable in dem zu generierenden Environment gespeichert werden. Dadurch entstehen zwangsläufig verschachtelte Environment-Strukturen.

Bei einer Superkombinator Reduktion werden somit während einer Rumpfberechnung Kontextwechsel zu alten Environments und zurück notwendig, was bei einer Stackrealisierung einen hohen Zeitaufwand erfordert. Der Verzicht auf eine Realisierung der Environments mittels eines Stacks führt dazu, daß eine Environmentstruktur,die eine offene β-Reduktion direkt unterstützt, kaum Mehraufwand kostet. Um auf die Superkombinator Reduktion verzichten zu können, muß auf andere Weise sichergestellt werden, daß bei der β-Reduktion keine Indexmodifikationen erforderlich sind.

Wie bereits in Kapitel 2.1 erläutert, werden solche Modifikationen in zwei Situationen erforderlich:

1. Wenn Ausdrücke, die relativ freie Variablen enthalten, in Funktionsrümpfe hineinsubstituiert werden, müssen die Indizes der freien Variablen erhöht werden.

2. Wenn der Funktionsrumpf eines Redex relativ freie Variablen enthält, so müssen deren Indizes dekrementiert werden.

Ursache der Modifikation sind also immer relativ freie Variablen.

Um relativ freie Variablen, die im Gesamt-Ausdruck gebunden sind, im Rumpf bzw. Argument eines β-Redex zu vermeiden, reicht es offenbar zu fordern, daß keine Reduktionen im Rumpf einer Abstraktion erfolgen. Dies läßt sich dadurch realisieren, daß zum einen immer der äußerste Redex zuerst reduziert wird und zum anderen wie in der Superkombinator Reduktion keine partiellen Anwendungen gemacht werden. Unter dieser Voraussetzung können relativ freie Variablen im Rumpf bzw. Argument eines β-Redex nur noch dadurch entstehen, daß sie global frei sind. Die für diese Variablen erforderliche Indexmodifikation läßt sich jedoch durch eine Darstellung als Konstanten während der Processing Phase von dieser in die Postprocessing Phase verlagern.

In $\mathcal{L}i$SA erfolgt somit eine naive, offene β-Reduktion, die in der Postprocessing Phase vervollständigt wird. Dadurch, daß im Rumpf von Abstraktionen nicht reduziert wird, entsteht als Ergebnis einer Reduktion wie in π-RED* nicht unbedingt eine Normalform, sondern eine sogenannte *Weak Normal Form*.

Ein Ausdruck E ist in *Weak Normal Form* (WNF) genau dann, wenn E in WHNF ist und es gilt:

$$
\begin{aligned}
& E = FA_1...A_m \quad \text{mit} \quad m \geq 0 \\
\wedge \quad & F \text{ ist eine Datenstruktur} \\
\Rightarrow \; & \text{die Unterkomponenten von } F \text{ sind in WNF.}
\end{aligned}
$$

3.5 Das Patternmatch

Im Gegensatz zu einem auf Applicative Order basierenden Systems wie der π-RED* liegen bei $\mathcal{L}i$SA die Argumente für eine π- bzw. Case-Abstraktion zunächst unausgewertet vor. Um die Mechanismen der Realisierung des Patternmatches von π-RED* ohne großen Aufwand übernehmen zu können, wäre es denkbar, die π-Abstraktionen als *strikt* zu erklären und vor dem Eintritt in den Patternausdruck zu versuchen, die Argumente vollständig zu evaluieren. Eine solche Vorgehensweise würde jedoch die Einsatzmöglichkeiten des Patternmatches erheblich einschränken; eine Verwendung in Filtern für konzeptuell unendliche Listen z.B. wäre nicht sinnvoll, da es zur "vollständigen" Berechnung der Listen führen würde.

Daher ist es notwendig, in der Auswertung der Argumente je nach Patternstruktur zu unterscheiden und dabei nur soviel auszuwerten, wie notwendig ist, um zu entscheiden, ob die Argumente auf die Pattern passen oder nicht. Unter dieser Prämisse stellt es sich aber durchaus als Problem dar, zu entscheiden, welche Argumentteile in welcher Reihenfolge wie weit ausgewertet werden müssen [Augu85, Lavi87, PJ87]. Um die Problematik aufzuzeigen, betrachten wir folgenden Ausdruck:

Beispiel 3.2: Eine *kritische π-Abstraktion*.

```
ap
  when ( 1, 1 ) guard true do 42
to [ arg1, arg2 ]
```

Um aussagen zu können, daß die Argumente passen, ist es offenbar notwendig, beide Argumente zur WHNF zu reduzieren. Für die Aussage, daß die Argumente nicht passen können, reicht es gegebenenfalls jedoch aus, *ein* Argument zu berechnen. Es ist für beide Argumente nicht entscheidbar, ob sie eine WHNF besitzen. Um für den Fall, daß sich eines der beiden Argumente zu einem anderen Wert als eins reduzieren läßt, ein mögliches Terminieren garantieren zu können, ist also eine nebenläufige Berechnung der Argumente erforderlich. Ein solcher Lösungsansatz führt jedoch wiederum zu redundanten Berechnungen. Aus dem Beispiel 3.2 ergibt sich also:

Es ist zwar möglich, zu einer vorgegebenen π-Abstraktion festzustellen, welche Argumentteile mindestens berechnet werden müssen, um nachzuweisen, daß die Pattern passen; es gibt jedoch π-Abstraktionen, bei denen nicht festgelegt werden kann, welche Argumentteile berechnet werden müssen, um mit möglichst wenigen Reduktionen feststellen zu können, daß die Pattern nicht passen.

Solche π-Abstraktionen werden *kritische π-Abstraktion* genannt.

Da für eine π-Abstraktion im Falle nicht passender Pattern die Berechnung ohnehin terminiert, ist die Reihenfolge der Patterntests bei einzelnen π-Abstraktionen von untergeordneter Bedeutung. Bei $\mathcal{L}i\text{SA}$ werden alle benötigten Argumentteile "von links nach rechts" berechnet.

Da es bei einer Case-Abstraktion im Falle einer nicht passenden π-Abstraktion zumeist Alternativen gibt, hat hier die oben aufgezeigte Problematik einen erheblichen Einfluß auf das Reduktions- bzw. Terminierungsverhalten. Auch bei der Case-Abstraktion wäre es denkbar zu fordern, möglichst wenige Teile der Argumente zu berechnen. Ein Algorithmus, der dies zu einer vorgegebenen Case-Abstraktion leistet, heißt *Lazy Patternmatch Algorithmus*. Eine Case-Abstraktion, zu der es einen solchen Algorithmus gibt, heißt *sequentielles Pattern* [Lavi87].

Für Case-Abstraktionen, die nur aus nicht kritischen π-Abstraktionen bestehen, ist ein Lazy Patternmatch Algorithmus direkt ableitbar; er ergibt sich aus der sequentiellen Abfolge der Patterntests der einzelnen π-Abstraktionen. Sind jedoch kritische π-Abstraktionen enthalten, kann diese Vorgehensweise zu einem Patternmatch Algorithmus führen, der nicht lazy ist. Beispiel 3.3 demonstriert einen solchen Fall:

Beispiel 3.3: Ein *sequentielles Pattern*.

```
ap
  case
    when ( 1 , 1 ) guard true do 1
    when ( X , 2 ) guard true do 2
  end
to [ arg1, arg2 ]
```

Ein Lazy Patternmatch Algorithmus muß bei diesem Beispiel offenbar dafür sorgen, daß zuerst die Berechnung des zweiten Argumentes erfolgt, um ggf. eine Evaluation des ersten Argumentes zu verhindern. Es gibt ein Verfahren [Lavi87], das zu jedem sequentiellen Pattern eine Berechnungsfolge der Argumentteile bestimmt, so daß ein Lazy Patternmatch Algorithmus entsteht. Da es jedoch auch viele Case-Abstraktionen gibt, die nicht sequentiell sind, kann dieses Verfahren nur für einen Teil aller Case-Abstraktionen unnötige Berechnungen vermeiden. Beispiele für nicht sequentielle Pattern sind das Berry-Example, Parallel-OR, Beispiel 3.4 und andere.

Beispiel 3.4: Ein Pattern, das nicht sequentiell ist.

```
ap
  case
    when ( 1, 1 ) guard true do 1
    when ( X, Y ) guard true do 2
  end
to [ arg1, arg2 ]
```

Bei diesen Beispielen sind unnötige Berechnungen unvermeidbar, und nur eine nebenläufige Berechnung der Argumente kann ein mögliches Terminieren garantieren.

Damit ergibt sich sowohl aus dem Verfahren zur Auffindung der Lazy Patternmatch Algorithmen als auch aus der abwechselnden Berechnung der Argumente jeweils nur für einen Teil aller Case-Abstraktionen eine befriedigende Lösung. Beide Ansätze erfordern jedoch für *alle* Case-Abstraktionen einen höheren Laufzeitaufwand. Deshalb ergibt sich bei

$\mathcal{L}i$SA genau wie bei π-RED* die Bedeutung der Anwendung einer Case-Abstraktion direkt aus der Bedeutung der Anwendung der einzelnen π-Abstraktionen.

In π-RED* wird eine Case-Abstraktion durch die sequentielle Abfolge der Matchvorgänge der einzelnen π-Abstraktionen realisiert. Diese Vorgehensweise führt jedoch bei überlappenden Pattern zu redundanten Tests (vergl. Kapitel 2.2). In einem auf Lazy Evaluation basierenden System wie $\mathcal{L}i$SA führt dieser Ansatz jedoch nicht nur zu redundanten Patterntests, sondern auch zu redundanten Überprüfungen, ob bestimmte Argumentteile bereits evaluiert sind. Diesen Laufzeitaufwand reduziert ein *baumartiger Ansatz*. Dabei werden bei strukturell ähnlichen Pattern durch Zusammenfassung der Tests Redundanzen vermieden. Deshalb wird vor der eigentlichen Reduktion in $\mathcal{L}i$SA für Case-Abstraktionen statt eines linearen Codes, wie er in der π-RED* erzeugt wird, Code in Form eines Graphen erzeugt.

3.6 Reduktionszähler

Das Konzept des Reduktionszählers wird bei $\mathcal{L}i$SA von der π-RED* übernommen. Es gibt also einen solchen, der für beliebige Programmausdrücke eine Beendigung der Reduktion nach einer vorgebbaren Anzahl von Reduktionsschritten ermöglicht. Dabei werden wie in π-RED* β-, δ -, α (Rekursionen)- sowie π (Patternmatches)-Reduktionen gezählt. Einziger Unterschied zur π-RED* ist der, daß ein rekursiver Funktionsaufruf, der in der π-RED* als ein Schritt gezählt wird, bei der $\mathcal{L}i$SA als zwei zählt; einen Schritt für die Rekursion und einen weiteren für die darauffolgende Parameterübergabe (β -Reduktion).

Um dem Anwender noch differenziertere Möglichkeiten für die Beendigung des Reduktionsvorganges zur Verfügung zu stellen, gibt es vier weitere Reduktionszähler:

1. einen Zähler für β -Reduktionen,
2. einen Zähler für δ -Reduktionen,
3. einen Zähler für α -Reduktionen (Rekusionen) sowie
4. einen Zähler für π -Reduktionen (Patternmatches).

4 Die Implementierungskonzepte

4.1 Der Bootstrap

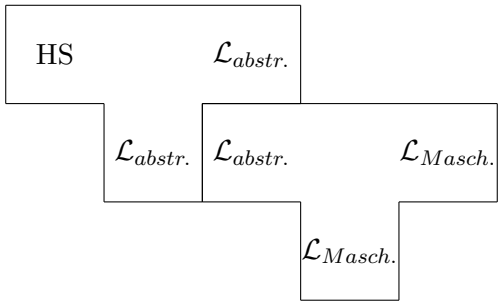
Der Entwurf des Ausführungsmechanismus erfolgt auf einem abstrakten Niveau, d.h. es werden z.B. Tupel oder Graphen gehandhabt, ohne eine Darstellung für sie zu konkretisieren. Je nach Wahl der Spezifikationssprache hat dies Auswirkungen auf die Implementierung.

Eine funktionale Implementierung (z.B. in K*i*R) läßt sich leicht umsetzen. Dies liegt zum einen daran, daß Datenstrukturen mittels benutzerdefinierter Konstruktoren leicht darstellbar bzw. dynamisch erzeugbar sind. Zum anderen stellt das Patternmatch Sprachkonstrukte bereit, mit deren Hilfe Datenstrukturen auf einfache und übersichtliche Weise zerlegt werden können. Die Ausführungszeiten einer solchen Spezifikation sind jedoch sehr hoch. Außerdem erfordert eine Portierung des Systems, daß auf der Zielmaschine zunächst der benötigte Ausführungsmechanismus (z.B. π-RED*) installiert wird.

Wird eine Implementierung in C vorgenommen, so sind zwar bessere Ausführungszeiten erreichbar, aber der Codierungsaufwand steigt erheblich, da die Verwaltung der dynamisch erzeugbaren Datenstrukturen weitaus konkreter spezifiziert werden muß (Speicherreservierungen, Verkettung durch Zeiger, Dereferenzierung von Zeigern, etc.). Zudem kann auch die Portierung von C-Code Probleme bereiten: die Umsetzung von Struktur-Spezifikationen in konkrete Speicherzugriffe sowie die Ausführungsreihenfolge einiger Statements sind nicht standardisiert und somit maschinen- bzw. compiler-abhängig.

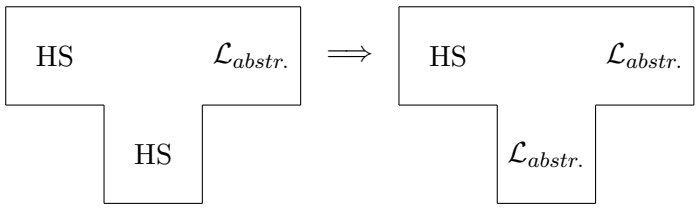
In $\mathcal{L}i$SA findet ein hybrider Ansatz Verwendung; er bedient sich der sogenannten *Bootstrap-Technik* [Aho88]. Die Vorgehensweise ist dabei die folgende: es gibt einen sogenannten *Ausführungskern*, der eine abstrakte Maschinensprache - im weiteren $\mathcal{L}_{abst.}$ genannt - auf einer realen Maschine exekutierbar macht. $\mathcal{L}_{abst.}$ wird dann als Schnittstelle für die Implementierung von Hochsprachen benutzt. Dazu ist es erforderlich, einen Compiler von der Hochsprache (HS) zur abstrakten Maschinensprache *in* der abstrakten Maschinensprache zu erstellen (siehe Abbildung 4.1).

Abbildung 4.1: T-Diagramm für den Bootstrap.



Existiert bereits ein Ausführungsmechanismus für die Hochsprache, so ist es weiterhin möglich, den Compiler in der Hochsprache zu codieren und ihn dann mittels einer Selbstanwendung in die benötigte Form zu transformieren (siehe Abbildung 4.2).

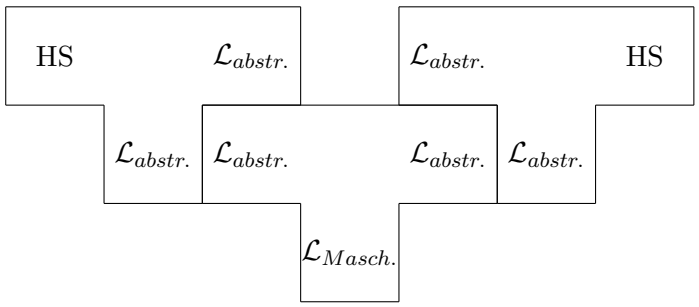
Abbildung 4.2: T-Diagramm für die Transformation des Compilers.



Durch die Schnittstelle $\mathcal{L}_{abst.}$ wird eine leichte Portierbarkeit auf andere Maschinen erreicht, da eine Portierung lediglich eine Installierung des Ausführungskernes auf der neuen Umgebung erfordert. Eine anschließende Selbstanwendung des Compilers auf dem "neuen" Ausführungskern macht das System nicht nur vollständig unabhängig von der alten Umgebung, sondern bietet zugleich einen umfangreichen Test des Ausführungskernes.

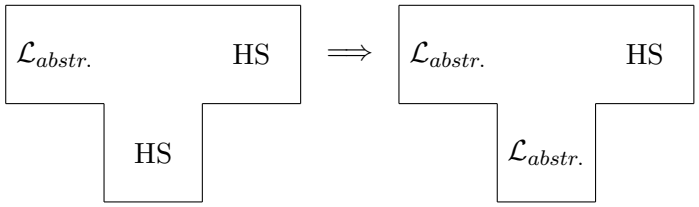
Diese Technik läßt sich auf transformierende Systeme wie $\mathcal{L}i$ISA übertragen. Dazu wird der Ausführungskern so gestaltet, daß er *Transformationen* von Programmen in $\mathcal{L}_{abstr.}$ vornimmt. Neben dem Kompiler von HS nach $\mathcal{L}_{abstr.}$ bedarf es eines zweiten Compilers, der $\mathcal{L}_{abstr.}$-Programme in HS-Programme überführt. Damit ergibt sich als T-Diagramm:

Abbildung 4.3: T-Diagramm für den Bootstrap eines transformierenden Systems.



Der "zweite Compiler" kann ebenfalls in der Hochsprache codiert werden, da er mittels des Compilers von HS nach $\mathcal{L}_{abstr.}$ in die abstrakte Maschinensprache übersetzt werden kann (siehe Abbildung 4.4).

Abbildung 4.4: T-Diagramm für die Transformation des zweiten Compilers.



Konkret auf $\mathcal{L}i$ISA bezogen bedeutet dies: es wird lediglich ein Ausführungskern in C spezifiziert. Die Codierung der Transformationsprogramme von der Hochsprache ($\mathcal{L}$KiR) in die abstrakte Maschinensprache und zurück erfolgt in KiR $\cap$ $\mathcal{L}$KiR. Durch π-RED* können diese Programme dann in die abstrakte Maschinensprache übersetzt werden (vergl. Abbildung 4.2/Abbildung 4.4). Damit steht ein System zur Verfügung, das, obwohl große Teile in KiR spezifiziert sind, unabhängig von π-RED* arbeitet. Eine Portierung erfordert lediglich die Reinstallation des Ausführungskernes auf der "neuen" Maschine. Auf diese Weise kann der zu portierende C-Code gegenüber einer vollständig in C codierten Implementierung

kompakt gehalten werden.

Da die Konzeption der Darstellungsebenen von $\mathcal{L}i$SA neben der hochsprachlichen Darstellung nur eine weitere (interne) Darstellung vorsieht, wird diese als abstrakte Maschinensprache gewählt; jede andere hätte die Hinzunahme einer weiteren Darstellungsebene zur Folge. Die daraus resultierende Spezifikation der kompletten Processing Phase in C ermöglicht zudem Vergleiche in den Ausführungszeiten von π-RED* und $\mathcal{L}i$SA. Da die Spezifikationen der Pre- und der Postprocessng Phase in KiR erfolgen und KiR keine Ausgabe-Konstrukte bereitstellt, kann die Schnittstelle nur aus der KiR-Repräsentation der internen (Graph-) Darstellung bestehen.

4.2 Die C-Implementierung

Den Hauptteil der C-Implementierung nimmt der Ausführungsmechanismus ein. Dazu muß eine Abbildung der internen Darstellung auf C-Datenstrukturen gefunden werden. Dies geschieht in Anlehnung an die Datenstrukturen von π-RED*. Graphknoten werden durch sogenannte *Deskriptoren* repräsentiert. Es handelt sich dabei um Datenstrukturen fester Länge und vergleichbarer Struktur. Verweise auf Unterknoten, die sich nicht im Deskriptor selbst codieren lassen, werden in sogenannten *Heapsegmenten* variabler Länge abgelegt und vom Deskriptor via Pointer referenziert. Für die Verwaltung der Deskriptoren und Heapsegmente finden nicht die zur Verfügung stehenden Systemroutinen Verwendung; stattdessen übernimmt ein eigener Programmteil, der sogenannte *Heapmanager*, in einem initial reservierten Speicherbereich, dem *Heap*, die Verwaltung. Eine ausführliche Beschreibung des Heapmanagers findet sich in [Rath91].

Neben dem Ausführungsmechanismus sind außerdem noch C-Routinen notwendig, die die Eingabe bzw. Ausgabe solcher Graphen gestatten. Die Eingabe muß zum einen aus Ascii-Files erfolgen können, in denen eine K*i*R-Repräsentation der Graphen liegt (Bootstrap!) sowie zum anderen direkt von einem Preprocessor bzw. Editor gesteuert werden. Analog müssen zwei verschiedene Ausgabemechanismen ermöglicht werden.

Um das mehrstufige Vorgehen (vergl. Abbildung 4.5) für den Anwender des Systems transparent zu gestalten, ist schließlich C-Code erforderlich, der diesen Vorgang automatisiert. Damit ergibt sich für die Hauptkomponenten der C-Implementierung folgendes Diagramm:

Abbildung 4.5: Die Hauptkomponenten der C-Implementierung

4.3 Die funktionale Spezifikation

Nach dem Entwurf des Ausführungsmechanismus ist es notwendig, seine Funktionalität anhand einiger Testbeispiele zu validieren. Um dies mit möglichst geringem Codieraufwand zu erreichen, ist es sinnvoll, zunächst eine funktionale Spezifikation (in K*i*R) vorzunehmen. Diese Vorgehensweise läßt eine Validierung des Pre- und Postprocessors zu, ohne daß eine C-Implementierung des Ausführungskernes benötigt wird, und durch die Möglichkeit der schrittweisen Transformation von K*i*R-Programmen mittels π-RED* sind Programmierfehler leicht auffindbar.

Die Schnittstelle zwischen dem Preprocessor und dem Ausführungskern bildet die K*i*R-Repräsentation der Graphdarstellung (vergl. Kap 4.1). Sie wird von $\mathcal{L}i$SA aufgrund des Bootstraps häufig genutzt. Deshalb wird ihrer Konzeption eine möglichst einfache Bedienbarkeit von seiten der C-Implementierung zugrunde gelegt.

Eine Ausweitung der Darstellungsähnlichkeit von Graphen auf Zustände des Ausführungsmechanismus ermöglicht eine weitere Schnittstelle zwischen der K*i*R- und der C-Implementierung. So kann ein Ausdruck “während“ der Transformation des Ausdruckes von der einen Implementiernug zur anderen überführt werden (siehe Abbildung 4.6), wodurch ein schrittweiser Vergleich beider Implementierungen möglich wird. Zusätzlich ergibt sich aus den in der Fileschnittstelle der C-Implementierung benötigten Routinen ein primitiver, kopierender Garbage-Collektor, was zumindest während der Testphase die Codierung eines Garbage-Collection Mechanismus überflüssig macht.

Abbildung 4.6: Die Hauptkomponenten des Gesamt-Systems.

5 Der Ausführungsmechanismus

Die Bedeutung eines $\mathcal{L}$K*i*R-Ausdruckes ist rekursiv über die Bedeutung seiner Unterausdrücke definiert. Eine direkte Übertragung auf einen Ausführungsmechanismus führt dazu, daß während einer Berechnung rekursive Verschachtelungen mit einer Verschachtelungstiefe der Unterausdrücke entstehen. Dies bedeutet für eine Implementierung, daß der Zustand des Systems in einer vorgegebenen Situation schwer nachvollziehbar wird und daß die ausführende Maschine in der Lage sein muß, beliebige Rekursionstiefen zu handhaben (auf den Apollo-Workstations ist diese z.B. begrenzt). Deshalb wird der Ausführungsmechanismus durch eine abstrakte Maschine realisiert, die auf Zustandstransformationen beruht. Dies ermöglicht sowohl eine einfache Umsetzung in "nicht rekursiven" C-Code, als auch gute Debug-Möglichkeiten durch schrittweise Exekution.

Da der Ausführungsmechanismus auf Graphtransformation beruht, wird der Zustand der abstrakten Maschine hauptsächlich durch einen Speicherbereich für Graphstrukturen und einen Verweis auf den aktuell zu berechnenden Ausdruck charakterisiert. Die Darstellung der Graphstrukturen besteht aus drei Komponenten. Die Hauptkomponente bildet der Datenbereich G. Er enthält den initialen Programmgraphen. Im Verlauf der β -Reduktion bzw. des Patternmatches entstehen Bindungen von formalen an aktuelle Parameter, die in einem weiteren Datenbereich, dem Environment E, nachgehalten werden. Die für das Instanziierungs-Konzept notwendigen Indirektionsknoten, die ebenfalls im Verlaufe der Reduktion entstehen, erfordern den dritten großen Datenbereich I.

Zum Traversieren des Graphen wird ein Stacks S benötigt (siehe Kapitel 3.2). Für die Realisierung von δ-Reduktionen und Listenstrukturen wird ein weiterer Stack, der F-Stack genutzt. Zwei Stacks Pm und Bi finden beim Patternmatch Verwendung.

Bei der Berechnung von Ausdrücken entstehen verschiedene Kontrollsituationen. Das liegt daran, daß im Verlauf einer Berechnung unterschiedliche Normalformen (WHNF und WNF) benötigt werden und daß nach der Beendung von Teilberechnungen die Fortsetzung von zurückgestellten Reduktionen erfolgen muß. Um für diese unterschiedlichen Kontrollsituationen nicht jeweils neue abstrakte Maschinen spezifizieren zu müssen, gibt es ein Flag f, das den aktuellen Kontrollmodus anzeigt. Auf diese Weise lassen sich Kontrollübergänge durch ein Verändern des Flags bewirken. Es kann die Zustände WNF, WHNF, TERMINATED und DONE annehmen. Dabei bedeuten WNF sowie WHNF, daß der aktuelle Graph zu der entsprechenden Normalform reduziert werden soll; TERMINATED, daß der aktuelle (Teil-) Ausdruck berechnet ist und DONE, daß der Reduktionsvorgang vollständig beendet ist. Schließlich gibt es eine Komponente Rc, die die Reduktionszähler enthält.

Vollständig wird ein Zustand der abstrakten Maschine beschrieben durch ein 11-Tupel mit folgenden Komponenten:

Die aktuellen Graphzeiger:

- e Zeiger auf die aktuelle Umgebung in E.
- n Zeiger auf den aktuellen uninstanziierten Graphknoten in G. Es handelt sich dabei um den bisher mit "CPR" beschriebenen Zeiger.

Der Graphbereich:

- G Der eigentliche Graphbereich; er enthält den Programmcode (Graphknoten) sowie sämtliche bei der Evaluation entstehenden Resultate.

- E Der Environmentbereich; hier befinden sich alle noch benötigten Environments.
- I Der Indirektionsknotenbereich; hier werden ausschließlich Paare von Verweisen auf uninstanziierte Graphen in G und dazegehörige Umgebungen in E gehalten.

Die vier Stacks:

- S Der Argumentstack; auf ihm werden instanziierte Argumente zwischengespeichert, auf die der aktuelle Ausdruck angewendet werden soll.
- F Der Funktionsstack; er enthält Verweise auf noch zu berechnende (Teil-) Ausdrücke.
- Pm Der Patternmatchstack; dieser Stack wird ausschließlich während des Matchvorganges von Case-Abstraktionen benötigt. Er hält die beim Fehlschlagen eines Patterns zu restaurierenden Argumentwerte bereit.
- Bi Der Bindungsstack; auch dieser Stack dient ausschließlich dem Patternmatch. Hier werden während des Matchevorganges entstehende Bindungen abgelegt.

Sonstige Komponenten

- f Das Kontrollmodusflag.
- Rc Die Reduktionszählerkomponente; sie besteht aus einem Fünftupel $< B, D, Y, M, S >$. Dabei steht B für β-Reduktionen, D für δ-Reduktionen, Y für Rekursionen, M für Patternmatches und S für akkumulierte Reduktionsschritte.

Die Beschreibung von Zustandstransformationen erfolgt in der Form:

$(Zustand\ vor\ der\ Transformation) \Longrightarrow (Zustand\ nach\ der\ Transformation)$.

Dabei werden bei den Zustandsbeschreibungen nur die für den Zustandsübergang relevanten Ausschnitte der einzelnen Tupelkomponenten explizit angegeben. Bei Stacks bedeutet z.B. $[k.S] : k$ ist das oberste Element und S ist der Reststack. Die Darstellung $G[n = z]$ bedeutet, daß in einem Datenbereich G hinter einem Zeiger n eine Struktur z liegt; von referenzierten Graphstrukturen wird immer nur ein Knoten, gefolgt von den Zeigern auf seine Nachfolger-Knoten, dargestellt. Indirektionsknoten in I werden als Tupel der Form

$$(Verweis\ in\ G,\ Verweis\ in\ E) \qquad \text{und Environments als}$$

$$\langle Substitut_1, \ldots, Substitut_n \rangle \rightarrow Verweis\ auf\ das\ nachfolgende\ Environment$$

dargestellt. Für die Zustände des Kontrollmodusflags werden als Abkürzungen verwendet: t für TERMINATED, f für WHNF oder WNF, w für WNF, h für WHNF und d für DONE.

Als einführendes Beispiel soll hier die Zustandstransformation für Konstanten exemplarisch dargestellt werden.

$$\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \texttt{int}^i/\texttt{bool}^b/\texttt{string}^{`s`}/\texttt{name}^{`x`}/\text{tup}^0/[]\Big], E, f, Rc\Big)$$
$$\Longrightarrow \Big(e, n, S, F, Pm, Bi, I, G, E, t, Rc\Big)$$

Vor der Transformation zeigt der aktuelle Graphzeiger n auf einen Graphknoten des Types $\texttt{int}^i$, $\texttt{bool}^b$, $\texttt{string}^{`s`}$, $\texttt{name}^{`x`}$, tup^0 oder $[]$ und das Kontrolmodusflag hat entweder den Wert WHNF oder den Wert WNF. Für beide Fälle ist der aktuelle Graphknoten vollständig berechnet. Deshalb ändert sich der Kontrollmodus auf TERMINATED (Kontrollmodusflag $:= t$), und alle übrigen Zustandskomponenten bleiben unverändert.

Um die in diesem Kapitel dargestellten Transformationen konziser zu gestalten, wird auf die Darstellung einiger Komponenten verzichtet. In den Abschnitten, die sich nicht auf das Patternmatch beziehen, sind dies die Stacks Pm und Bi; die Rc-Komponente wird im gesamten Kapitel nicht mit dargestellt.

Eine vollständige Darstellung aller Transformationsregeln findet sich in Anhang D.

5.1 Environments

Das Zurückstellen der Substitutionen bei der β-Reduktion führt dazu, daß die Substitute für die formalen Parameter in Environments bereitgestellt werden müssen. Zur Verdeutlichung der dafür erforderlichen Environment-Strukturen soll hier mittels des λ-Kalküls der Begriff der *weiter außen gebundenen Variablen* eingeführt werden.

Seien M, N λ-Ausdrücke mit: M ist Unterausdruck von N. Dann wird die Menge der *für M bezüglich N weiter außen gebundenen Variablen*, kurz $AV(M,N)$, folgendermaßen definiert:

$$AV(M,N) := \begin{cases} \emptyset & \text{falls } M = N \\ AV(M,P) & \text{falls } N = (PQ) \wedge M \text{ ist Unterausdruck von } P \\ AV(M,Q) & \text{falls } N = (PQ) \wedge M \text{ ist Unterausdruck von } Q \\ AV(M,P) \cup \{x\} & \text{falls } N = \lambda x.P \end{cases} .$$

Betrachten wir nun eine Funktionsdeklaration F in einem Programm P in de Bruijn Darstellung mit $F = \Lambda^n body \quad \wedge \quad |AV(F,P)| = m$. Ohne Beschränkung der Allgemeinheit kann davon ausgegangen werden, daß P keine global freien Variablen enthält, da solche Variablen in $\mathcal{L}i$SA während der Processing Phase als Konstanten behandelt werden (vergl. Kapitel 3.4). Weiterhin können wir uns bei der Betrachtung der Variablen im Rumpf der Funktion (*body*) auf diejenigen beschränken, die nicht unter einer Abstraktion in *body* stehen, da jede Abstraktion wiederum eine Funktionsdeklaration darstellt. Für solche Variablen gilt:

$$\begin{array}{ll} i \in \{0,\ldots,(n-1)\} & \text{falls } \#i \text{ formaler Parameter} \quad \text{und} \\ i \in \{n,\ldots,(m+n-1)\} & \text{falls } \#i \text{ relativ freie Variable.} \end{array}$$

Soll der de Bruijn Index einer Variablen während der Processing Phase unverändert bleiben und als Zugriffsoffset für eine lineare Environment-Struktur dienen, so ist also für jede Funktionsanwendung eine Environment-Struktur der Form $< s_{m-1},\ldots,s_0,p_{n-1},\ldots,p_0 >$ mit s_i = Substitut für die für F bezüglich P weiter außen gebundene Variable $\#(i+n)$ und p_i = Substitut für den formalen Parameter $\#i$ erforderlich.

Durch das Zurückstellen der Substitutionen kann es vorkommen, daß mehrere Ausdrücke einen Verweis auf ein Environment haben, das die Substitute s_{m-1} bis s_0 enthält. Deshalb darf dieses Environment bei einer Funktionsanwendung von F nicht um die aktuellen Parameter ergänzt werden, sondern es muß zunächst eine Kopie des Environments erzeugt werden, die dann um die aktuellen Parameter ergänzt wird. Zur Verdeutlichung betrachten wir folgendes Beispiel:

Beispiel 5.1: Eine modifizierte Fakultätsfunktion

```
let A=1
in def
     Fac [ N ] = let B = A
                 in if (N le B)
```

```
                then A
                else (N * Fac[(N - 1)])
in Fac[5]
```

Die Ersetzung von angewandten Vorkommen von Variablen durch Indizes ergibt:

Abbildung 5.1: Darstellung von Beispiel 5.1 mit De Bruijn Indizes

```
let A=1
in def
     Fac [ N ] = let B = #1
                 in if (#1 le #0)
                    then #2
                    else (#1 * Fac[(#1 - 1)])
   in Fac[5]
```

Sollen während der Reduktion die Indices des Funktionsrumpfes unverändert bleiben, so werden als Environment benötigt beim Eintritt in

das Def-Konstrukt	: A=1
die erste Inkarnation von fac	: A=1 | N=5
bzw.	: A=1 | N=5 | B=A
die zweite Inkarnation von fac	: A=1 | N=4
bzw.	: A=1 | N=4 | B=A

⋮

.

Ein solches Vorgehen bedeutet zwar einen schnellen Zugriff auf die Substitute von Variablen, macht jedoch bei jeder Anwendung einer Funktion F die Kopie der Substitute für die für F bezüglich des Gesamtausdruckes weiter außen gebundenen Variablen notwendig. Dieses Kopieren von Environments führt insbesondere bei rekursiven Funktionen zu einem hohen Platz- und Zeitaufwand; eine Rekursion mit Rekursionstiefe r, die nicht tail-end ist, erfordert r Kopien der Environmenteinträge aller weiter außen gebundene Variablen.

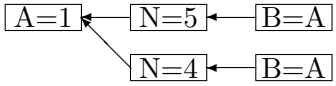
Der beim Kopieren solcher Environmenteinträge entstehende Aufwand läßt sich durch eine Verkettung von einzelnen Environmentteilen - im weiteren *Environmentframes* genannt - vermeiden. Die zugrundeliegende Idee ist dabei die, daß die aktuellen Parameter einer Funktionsanwendung von F in einem eigenen Environmentframe abgelegt werden und dieses dann via Pointer mit dem Environmentframe verkettet wird, das die Substitute der von F bezüglich des Gesamtausdruckes weiter außen gebundenen Variablen enthält.

Um dennoch einem angewandten Vorkommen einer Variablen das zugehörige Substitut mittels eines indizierten Zugriffes auf die Environment-Struktur zuordnen zu können, werden Variablen durch Indextupel $(i, j)^{\lambda}$ dargestellt; dabei spezifiziert i die Anzahl der zu verfolgenden Indirektionen und j die Position im Environmentframe. Diese Indextupel sind genauso wie die de Bruijn Indizes statisch berechenbar. Für obiges Beispiel bedeutet dies in der Indextupeldarstellung für Variablen:

Abbildung 5.2: Darstellung von Beispiel 5.1 mit Indextupeln (1.Variante)

let A=1
in def
Fac [N] = let B = $(1,0)^\lambda$
in if ($(1,0)^\lambda$ le $(0,0)^\lambda$)
then $(2,0)^\lambda$
else ($(1,0)^\lambda$ $*$ Fac[($(1,0)^\lambda$ $-$ 1)])
in Fac [5]

Während der Berechnung entstehen dann als Environments:



Diese Art des Vorgehens vermeidet zwar den Kopieraufwand bei Funktionsanwendungen, der Zugriff auf das Substitut einer Variablen $(i,j)^\lambda$ bedeutet jedoch das Verfolgen von i Indirektionen.

Für die Implementierung der Environment-Strukturen bedarf es dynamisch zu allozierender Speicherbereiche. Der Verwaltungsoverhead für solche Datenstrukturen wird um so gößer, je kürzer die einzelnen zu allozierenden Bereiche (in diesem Fall Environmentframes) sind. Bei der Spezifikation von Funktionen mit wenigen formalen Parametern ergeben sich unter Verwendung des oben geschilderten Ansatzes gerade solche "kurzen" Environmentframes. Weiterhin führt die Verwendung von Let-Konstrukten im Rumpf einer (rekursiven) Funktion dazu, daß in deren Rümpfen der Zugriff auf die aktuellen Parameter der Funktion mindestens eines Indirektionsschrittes bedarf.

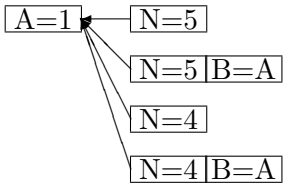
Die auf bisherige Anwendungen von π-RED* gestützte Annahme, daß solche Situationen verhältnismäßig häufig auftreten, führt zur Verwendung eines hybriden Ansatzes in $\mathcal{L}i$SA. Dabei wird bei der Anwendung von Funktionen nur dann ein neuer Environmentframe gegründet, wenn die Funktion mittels eines Def-Konstruktes definiert ist. Ein solcher neugegründeter Environmentframe muß dann mit einem Environmentframe verkettet werden, der ausschließlich die Substitute für die Variablen enthält, die für den Def-Block bezüglich des Gesamtausdruckes weiter außen gebunden sind. Um sicherstellen zu können, daß ein solcher existiert, wird bei jedem Eintritt in einen Def-Block ebenfalls ein neuer Environmentframe erstellt und mit dem bis dahin aktuellen verkettet. Bei Funktionsanwendungen von nicht mittels des Def-Konstruktes definierten Funktionen wird eine Kopie des aktuellen Frames erzeugt.

Auf diese Weise werden die aufwendigen Kopien bei Rekursionen vermieden, es sei denn, der Anwender codiert einen Rekursionsoperator ohne Ausnutzung des Def-Konstruktes. Außerdem wird durch das Kopieren bei nicht rekursiven Funktionsanwendungen eine zu starke Parzellierung des Environments vermieden, und es wird sichergestellt, daß in Funktionsrümpfen, die kein Def-Konstrukt enthalten, der Zugriff auf die Substitute der formalen Parameter ohne Indirektionsschritt erfolgen kann. Da auch bei diesem Verfahren verkettete Environmentstrukturen entstehen, werden hier ebenfalls Indextupel für die Darstellung von Variablen-Bindungen verwendet. Die Indextupel-Darstellung des Beispiel 5.1 sieht bei $\mathcal{L}i$SA folgendermaßen aus:

Abbildung 5.3: Darstellung von Beispiel 5.1 mit Indextupeln (2.Variante)

let A=1
in def
Fac [N] = let B = $(1,0)^{\lambda}$
in if ($(0,1)^{\lambda}$ le $(0,0)^{\lambda}$)
then $(1,0)^{\lambda}$
else ($(0,1)^{\lambda}$ * Fac[($(0,1)^{\lambda}$ − 1)])
in Fac [5]

sowie die Environmentstrukturen zur Laufzeit:



Um eine solche Environmentstruktur bzw. Variablendarstellung verwenden zu können, muß jedoch zunächst überprüft werden, ob sie sich mit dem in Kapitel 3.4 auf Basis des de Bruijn Kalküls vorgestellten Konzept der naiven Substitution vereinbaren läßt.

5.1.1 Die β-Reduktion

Durch die verkettete Environmentstruktur werden zur Darstellung der Bindungsstruktur der Variablen Indextupel $(i,j)^{\lambda}$ benötigt. Aufgrund dessen, daß neue Environmentframes ausschließlich bei der Anwendung mittels des Def-Konstruktes definierter Funktionen bzw. beim Eintritt in Def-Blöcke entstehen sollen, haben die Indextupel folgende Bedeutung:

- $i \geq 0$ gibt die Anzahl der geschachtelten Def-Blöcke an, und
- $j \geq 0$ gibt die Anzahl der λ-Bindungen innerhalb des Def-Blockes an,

die zwischen der Variablenbindung und ihrem Vorkommen liegen. Weiterhin läßt sich definieren:

Eine Variable $(i,j)^{\lambda}$heißt bezüglich eines (k,l) entfernten, bindenden Λ-Konstruktes:

- *blockgeschützt*, falls gilt: $i > k$,
- *geschützt*, falls gilt: $i = k \quad \wedge \quad j > l$,
- *frei*, falls gilt: $i = k \quad \wedge \quad j = l$,
- *gebunden*, falls gilt: $i = k \quad \wedge \quad j < l$,
- *blockgebunden*, falls gilt: $i < k$.

Damit ergibt sich für die vollständige β-Reduktion eines Redex (pre order linearisierte Darstellung): $\dashv \ldots @\,^{1}\Lambda^{1} body\, arg \ldots \vdash$ für die Variablen $(i,j)^{\lambda}$im Rumpf *body*:

- eine Dekrementierung von j, falls $(i,j)^{\lambda}$geschützt;
- das Ersetzen durch arg, falls $(i,j)^{\lambda}$frei;
- keine Veränderung sonst;

für die Variablen $(i,j)^{\lambda}$im Argument arg:

- keine Veränderung, falls $(i,j)^{\lambda}$gebunden oder blockgebunden;
- eine Erhöhung von i bzw. j um die Anzahl der bindenden Λ-Konstrukte respektive Def-Blöcke, in deren Bindungsbereich das Argument neu hineinsubstituiert wird, sonst.

Es zeigt sich also, daß die Ursachen für eine Indexmodifikation bei einer solchen Indextupel-Darstellung genau die gleichen wie bei der Variablendarstellung durch de Bruijn-Indizes sind (vergl. Kapitel 3.4): sie werden durch relativ freie Variablen im Argument bzw. im Rumpf der Abstraktion verursacht. Deshalb reichen auch für die hier vorgestellte Indextupel-Darstellung von Variablen die Realisierung der folgenden Forderungen aus, um eine naive Substitution während der Processing Phase zu ermöglichen:

1. Global freie Variablen werden nicht durch Indextupel dargestellt, sondern wie Konstanten behandelt.
2. Es wird nicht unter Abstraktionen reduziert.

Die Vermeidung der Reduktion unter Abstraktionen wird dadurch realisiert, daß immer der äußerste Redex zuerst reduziert wird und während der Processing Phase keine partiellen Anwendungen vorgenommen werden.

Wie bereits in Kapitel 3.2 erwähnt, erfolgt das Traversieren des Programmgraphen dergestalt, daß Verweise auf Argumente solange auf dem S-Stack gesammelt werden, bis der Verweis auf den aktuellen Knoten (CPR) auf eine Funktion zeigt. Ein β-Redex kennzeichnet sich also dadurch, daß der CPR auf einen Abstraktionsknoten zeigt und der S-Stack bereits Operanden enthält. Für die Realisierung der β-Reduktion sind also Zustandsübergänge für Applikationen ($@^n$ in der internen Darstellung) und Abstraktionen (Λ^n in der internen Darstellung) nötig.

Betrachten wir zunächst den Fall, daß der CPR auf einen Applikationsknoten verweist. Da noch Referenzen auf den Applikationsknoten existieren können, die unter einer anderen Umgebung ausgewertet werden müssen, darf nicht das Argument selbst auf den Stack gelegt werden, sondern lediglich eine Instanz (siehe auch Kapitel 3.3) desselben. Dies geschieht in $\mathcal{L}i$SA mittels eines Indirektionsknotens, der aus einem Pointer auf das Argument und einem auf das aktuelle Environment besteht. Da in dieser Berechnungssituation noch nicht absehbar ist, ob sich in Funktionsposition ein Ausdruck befindet, der tatsächlich zum Konsumieren der Argumente führt, ist es notwendig, die Applikationsstruktur rekonstruierbar zu halten. Um dies zu erreichen, werden zusätzlich Verweise auf den Applikationsknoten auf den S-Stack gelegt. Daraus ergibt sich folgende Transformationsregel:

$$\Big(e, n, S, F, I, G\Big[n = @^k n_0 \ldots n_k\Big], E, f\Big)$$
$$\Longrightarrow \Big(e, n_0, \Big[(p_1, n)\mathbf{.} \ldots \mathbf{.}(p_k, n)\mathbf{.}S\Big], F, I\Big[p_1 = (n_1, e), \ldots, p_k = (n_k, e)\Big], G, E, f\Big).$$

Zeigt der CPR auf eine Abstraktion, so liegt mindestens ein β-Redex vor. Handelt es sich um eine mehrstellige Abstraktion, so darf der äußerste Redex nur dann reduziert werden,

wenn für alle durch diese Abstraktion abstrahierten Parameter ein Argument vorliegt, da sonst eine partielle Anwendung vorgenommen würde, was Indexmodifikationen erforderlich machen könnte. Sind genügend Argumente vorhanden, so liegen Verweise auf diese auf dem S-Stack und die β-Reduktion kann stattfinden. Dazu wird eine Kopie des aktuellen Environmentframes hergestellt und gemäß der Stelligkeit der Abstraktion viele Argumente in Form der Indirektionsknoten hinzugefügt. Die Verweise auf die Applikationsknoten können dabei vom S-Stack genommen werden, da die Argumente ja konsumiert werden. Außerdem muß im Reduktionszähler die β- sowie die akkumulierte Komponente dekrementiert werden. Damit ergibt sich als Zustandstransformation:

$$\begin{aligned}&\Big(e, n, \Big[(p'_1, n_1)\mathbf{.}\ \ldots\ \mathbf{.}(p'_k, n_k)\mathbf{.}S\Big], F, I, G\Big[n = \Lambda^k n_0\Big], E\Big[e = \langle p_1, \ldots, p_m\rangle{\to}e'\Big], f\Big)\\ &\Longrightarrow \Big(e'', n_0, S, F, I, G, E\Big[e'' = \langle p'_k, \ldots, p'_1, p_1, \ldots, p_m\rangle{\to}e'\Big], f\Big).\end{aligned}$$

Überschüssige Verweise auf Argumente verbleiben auf dem S-Stack und können ggf. nach Beendigung der Berechnung wieder zu Applikationen rekonstruiert werden:

$$\begin{aligned}&\Big(e, n, \Big[(p_1, \hat{n})\mathbf{.}\ldots\mathbf{.}(p_k, \hat{n})\mathbf{.}S\Big], F, I, G, E, t\Big) \quad \wedge \quad top(S) \neq (\mathbf{.}, \hat{n})\\ &\Longrightarrow \Big(e, n', S, F, G\Big[n' = \overline{@}^{\,k} n, p_1 \ldots p_k\Big], E, t\Big).\end{aligned}$$

Befinden sich zu wenige Argumentverweise auf dem S-Stack, so liegt eine partielle Anwendung vor. Sie bleibt während der Processing Phase unreduziert. Es werden jedoch die vorhandenen Argumentverweise bereits dem aktuellen Environment hinzugefügt:

$$\begin{aligned}&\Big(e, n, \Big[(p'_1, n_1)\mathbf{.}\ \ldots\ \mathbf{.}(p'_l, n_l)\mathbf{.}S\Big], F, I, G\Big[n = \Lambda^k n_0\Big], E\Big[e = \langle p_1, \ldots, p_m\rangle{\to}e'\Big], f\Big) \quad \wedge \quad l < k\\ &\Longrightarrow \Big(e'', n', S, F, I, G\Big[n' = \overline{\Lambda}^{(k-l)} n_0\, n\, p_1 \ldots p_l\Big], E\Big[e'' = \langle p'_l, \ldots, p'_1, p_1, \ldots, p_m\rangle{\to}e'\Big], t\Big).\end{aligned}$$

5.1.2 Variablen

Zeigt der CPR auf eine Variable $(i, j)^\lambda$, so reicht es aufgrund der naiven Substitution aus, die Komponenten des entsprechenden Indirektionsknotens vom Environment in den aktuellen Graph- bzw. Environmentzeiger zu übernehmen. Genau betrachtet, beginnt jedesmal, wenn diese Situation eintritt, die Berechnung der Instanz eines Argumentes. Da diese eventuell noch von anderer Stelle referenziert wird, ist es unerläßlich, die Instanz - d.h. den sie repräsentierenden Indirektionsknoten - mit dem Ergebnis zu überschreiben (*Update*). Deshalb wird eine sogenannte Updatemarke auf den S-Stack gelegt. Aufgrund der Tatsache, daß ein Update nur für Ausdrücke, die auch wirklich verändert werden können (also potentiell Redizes enthalten), notwendig ist, ergibt sich als Zustandsübergang:

$$\begin{aligned}&\Big(e, n, S, F, I\Big[p_j = (n_j, e_j)\Big], G\Big[n = (i, j)^\lambda\Big], E\Big[e{\to}e_1{\to}\ldots{\to}e_i = \langle p_0 \ldots p_j \ldots p_m\rangle\Big], f\Big)\\ &\Longrightarrow \begin{cases} \Big(e_j, n_j, \Big[(\$, p_j)\mathbf{.}S\Big], F, P, I, G, E, f\Big) & \text{falls } n_j \in \{@, \alpha, \texttt{cons}\}\\ \Big(e_j, n_j, S, F, I, G, E, f\Big) & \text{sonst}\end{cases}.\end{aligned}$$

Hat die Berechnung terminiert, ist also das Kontrollmodusflag TERMINATED, so führt ein Updateflag auf dem S-Stack zum Überschreiben des Indirektionsknotens mit dem aktuellen Ausdruck:

$$\begin{aligned}&\Big(e, n, \Big[(\$, p)\mathbf{.}S\Big], F, I\Big[p = (n', e')\Big], G, E, t\Big)\\ &\Longrightarrow \Big(e, n, S, F, I\Big[p = (n, e)\Big], G, E, t\Big).\end{aligned}$$

5.1.3 Rekursive Ausdrücke

$\mathcal{L}i$SA stellt genau so wie π-RED* einen primitiven Rekursionsmechanismus zur Verfügung. Er wird durch das Def-Konstrukt realisiert, mit dessen Hilfe sich Ausdrücke der Art

def
$f_1[x_{1,1}, \ldots, x_{1,m_1}] = body_1$
$\vdots$
$f_n[x_{n,1}, \ldots, x_{n,m_n}] = body_n$
in *goal*

spezifizieren lassen. Dabei können beliebige f_i in beliebigen $body_j$ auftreten. Mittels dieses Konstruktes werden also potentiell rekursive Ausdrücke - zumeist Funktionen ($m_i > 0$) - an Variablen ($f_1, \ldots, f_n$) gebunden. Um die Bedeutung eines solchen Ausdruckes sowie insbesondere die Behandlungsweise nicht parametrisierter Ausdrücke (*rekursive Variablen*) zu klären, betrachten wir eine äquivalente Darstellung im reinen λ-Kalkül:

Für $i \in \{1, \ldots, n\}$ sei $b_i := \lambda f_1 \ldots f_n . \lambda x_{i,1} \ldots x_{i,m_i} . body_i$. Dann gilt:
Es existieren $F_1, \ldots, F_n$ mit $F_i = (\ldots (b_i F_1) \ldots F_n) \rightarrow_\beta F_i' := sub_{f_1 \ldots f_n}^{F_1 \ldots F_n} \left[\lambda x_{i,1} \ldots x_{i,m_i} . body_i\right]$
für alle $i \in \{1, \ldots, n\}$.
Damit läßt sich das Def-Konstrukt schreiben als $(\ldots (\lambda f_1 \ldots f_n . goal F_1') \ldots F_n')$.

Das bedeutet, daß die Ausdrücke konzeptuell wie die Argumente eines Let-Konstruktes behandelt werden müssen und somit ihre Berechnung höchstens einmal erfolgen darf. Da jedoch Funktionen ohnehin in WNF sind, bedürfen sie als konstante Objekte keines Updates. Deshalb wird zwischen rekursiven Funktionen und rekursiven Variablen in der Realisierung unterschieden.

Rekursive Funktionen werden nicht in Environments eingetragen; angewandte Vorkommen verweisen direkt auf den zugehörigen Funktionsgraphen. Das Zurückstellen der Substitutionen erfordert aber nicht nur einen Verweis auf den Graphen, sondern auch einen Verweis auf den Environmentframe, der die Substitute für die Variablen enthält, die für die Funktionsdefinition bezüglich des Gesamtprogrammes weiter außen gebunden sind. Da dies bei einem statisch generierten Verweis auf den Graphen nicht möglich ist, muß auf andere Weise der benötigte Environmentframe für die Berechnung der Anwendung bereitgestellt werden. Das Design der Environmentstruktur ist auf die Lösung dieses Problemes ausgerichtet; dadurch, daß bei jedem Def-Block ein neues Environmentframe eingerichtet wird, läßt sich anhand der Blöcke, die zwischen der Deklaration und der Anwendung der Funktion liegen, statisch die Anzahl der Indirektionen bestimmen, die zur Laufzeit vom aktuellen Environmentframe aus verfolgt werden müssen, um den gesuchten Frame aufzufinden. Damit ergibt sich als Transformationsregel für rekursive Funktionsanwendungen:

$$\Big(e, n, S, F, I, G\Big[n = (i)_{off}^{\alpha} n_0 n_\alpha\Big], E\Big[e \rightarrow e_1 \rightarrow \ldots \rightarrow e_j\Big], f\Big)$$
$$\Longrightarrow \Big(e_j, n_0, S, F, I, G, E, f\Big).$$

Da rekursive Variablen (Λ_r) ggf. eines Updates bedürfen, können diese Ausdrücke nicht direkt referenziert werden, sondern benötigen einen Indirektionsknoten, der nach ihrer Berechnung mit dem Ergebnis überschrieben werden kann. Es scheint daher opportun, den Mechanismus der β-Reduktion mitzubenutzen und die Rekursion durch Zyklen in den

den Variablen adjungierten Environments nachzubilden. Daraus ergibt sich folgender Zustandsübergang für das Def-Konstrukt:

$$\Big(e, n, S, F, I, G\Big[n = \alpha^{k,l} n' \Lambda_{r1} \dots \Lambda_{rk} f_1 \dots f_l\Big], E\Big[e = \langle p_1 \dots p_m \rangle \to \hat{e}\Big], f\Big)$$
$$\Longrightarrow \Big(e', n', S, F, I\Big[p'_1 = (\Lambda_{r1}, e) \dots p'_k = (\Lambda_{rm}, e)\Big], G, E\Big[e' = \langle\rangle \to \langle p'_1 \dots p'_k, p_1 \dots p_m \rangle \to \hat{e}\Big], f\Big).$$

Eine ausführliche Beschreibung des Umganges mit rekursiven Variablen findet sich in [Rath91].

5.2 Virtuelle Maschinen

Bei Ausdrücken des reinen λ-Kalküls ist die Berechnungsreihenfolge durch die Normal Order Strategie eindeutig festgelegt; sie erfolgt *leftmost outermost*, d.h. der Kontrollfluß bewegt sich immer vom Wurzelknoten zum linken Unterknoten auf dem sog. *Left Spine* entlang. Dadurch ergibt sich streng sequentieller Berechnungsablauf.

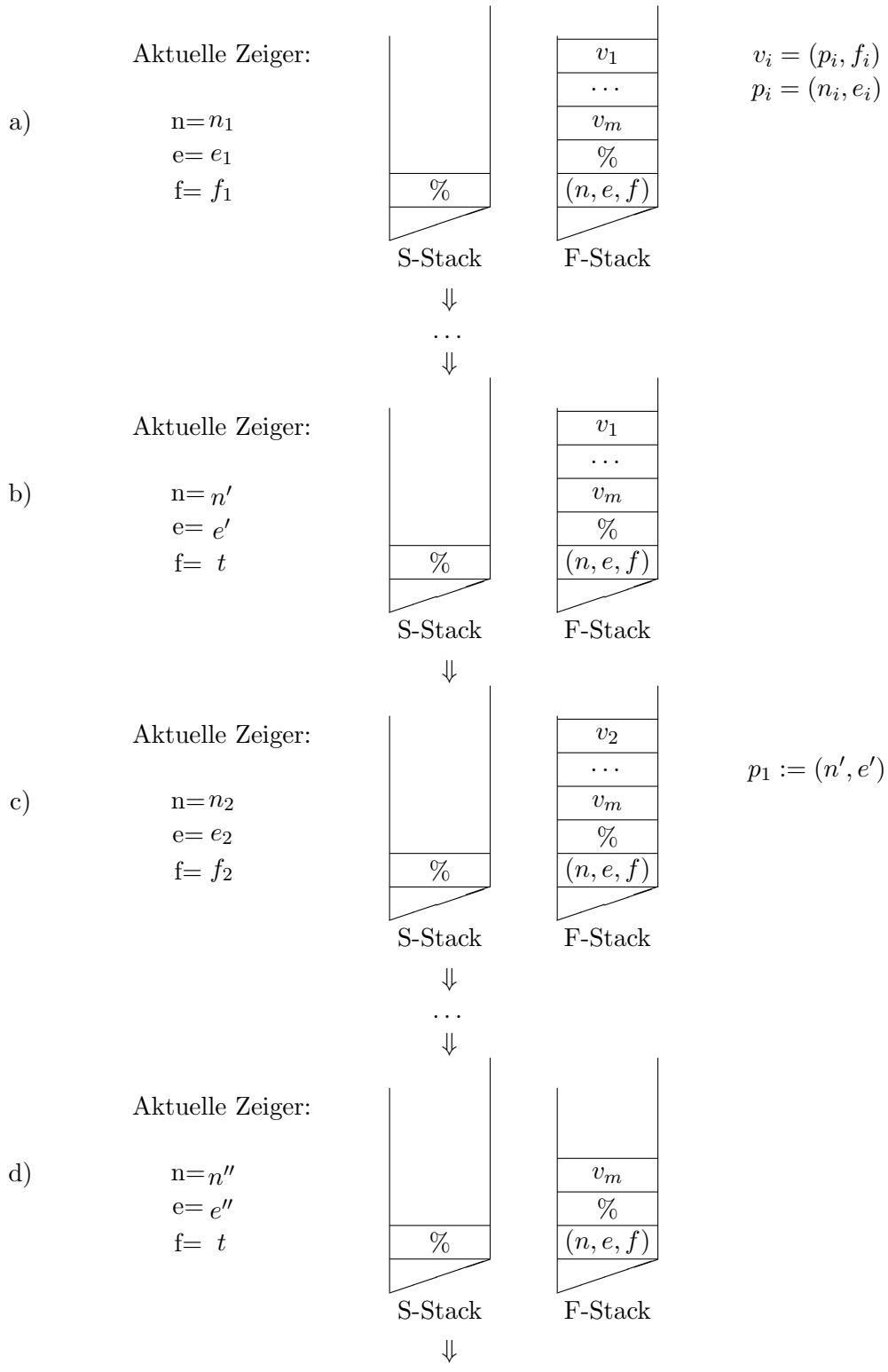
Für die Konstrukte des angewandten λ-Kalküls ist das nicht so. Die primitiven Funktionen erfordern vor ihrer Anwendung die Auswertung der bis dahin zurückgestellten Argumente bis zur WHNF. Auch für die Berechnung der WNF einer Datenstruktur müssen mehrere Ausdrücke, nämlich die Komponenten, ausgerechnet werden. Für beide Fälle ist nach der Berechnung der benötigten Ausdrücke die Wiederherstellung der alten Berechnungssituation erforderlich. Somit bedarf es eines Mechanismus, der mehrere eigenständige Berechnungen nacheinander initiiert, anschließend mit den Ergebnissen ein Update im Graphen vornimmt und schließlich einen bis auf den Graphen unveränderten Maschinenzustand hinterläßt. Konzeptuell bedeutet dies das Gründen vollständig neuer abstrakter Maschinen. Da aber das Ergebnis der Berechnungen ohnehin in den ursprünglichen Graphen übertragen werden muß und durch die Zustandsübergänge gerade rekursive Spezifikationen vermieden werden sollen, entstehen keine *realen Maschinen*, sondern *virtuelle Maschinen*. Sie arbeiten zwar alle in dem gleichen System, Interferenzen werden jedoch durch zwei Mechanismen vermieden:

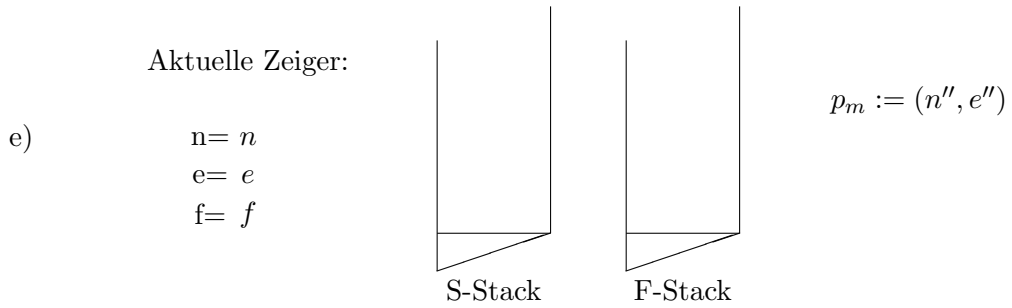
1. Es werden konzeptuell leere Stacks durch Ablegen eines Trennsymbols (%) geschaffen.
2. Zustandskennzeichnende Komponenten wie der Graphzeiger, der Environmentzeiger und das Kontrollmodusflag werden auf eine Art Laufzeitstack, den F-Stack, gelegt, um eine Restaurierung des Zustandes zu ermöglichen.

Mittels des F-Stacks ist es auf einfache Weise möglich, die aufeinander folgende Gründung mehrerer virtueller Maschinen sowie deren Update mit den Ergebnissen zu realisieren:

Zur Charakterisierung einer virtuellen Maschine bedarf es nur der zustandskennzeichnenden Komponenten (n, e, f). Da nach der Berechnung ein Update erfolgen soll, muß ohnehin ein Indirektionsknoten (falls noch nicht vorhanden) gebildet werden. Deshalb werden auf dem F-Stack die zu gründenden virtuellen Maschinen als Tupel, bestehend aus einem Zeiger auf den Indirektionsknoten und einem Wert für das Kontrollmodusflag, abgelegt. Wird also in der Situation (n, e, f) die Gründung der virtuellen Maschinen $v_1 = (p_1, f_1), \dots, v_m = (p_m, f_m)$ notwendig, so entsteht die Situation aus Abbildung 5.4 a).

Abbildung 5.4: Der Mechanismus zur Handhabung virtueller Maschinen


Aktuelle Zeiger:
a)
n= n_1
e= e_1
f= f_1
v_1
...
v_m
%
(n, e, f)
S-Stack
F-Stack
$v_i = (p_i, f_i)$
$p_i = (n_i, e_i)$
Aktuelle Zeiger:
b)
n= n'
e= e'
f= t
Aktuelle Zeiger:
c)
n= n_2
e= e_2
f= f_2
v_2
$p_1 := (n', e')$
Aktuelle Zeiger:
d)
n= n''
e= e''
f= t



Nach der Beendigung der Berechnung von v_1 (Abbildung 5.4 b)) erfolgt der Update von p_1 mit dem aktuellen Wert sowie die Initialisierung der nächsten virtuellen Maschine (Abbildung 5.4 c)). Für das Initiieren der nächsten virtuellen Maschine ist folgende Transformationsregel zuständig:

$$\Big(e, n, \Big[\%.S\Big], \Big[(p_1, f_1).(p_2, f_2).F\Big], I\Big[p_2 = (n_2, e_2)\Big], G, E, t\Big)$$
$$\Longrightarrow \Big(e_2, n_2, \Big[\%.S\Big], \Big[(p_2, f_2).F\Big], I\Big[p_1 = (n, e)\Big], G, E, f_2\Big).$$

So werden sukzessive alle virtuellen Maschinen berechnet, bis der F-Stack nur noch die letzte Update-Information enthält (Abbildung 5.4 d)). Ist dies erreicht, so kann mittels der auf dem F-Stack nachgehaltenen Informationen die ursprüngliche Berechnung fortgesetzt werden (Abbildung 5.4 e)).

Für die Realisierung der Datenstrukturen bzw. primitiven Funktionen steht also die Spezifikation zweier Zustandsübergänge im Vordergrund: zum einen der *Schritt der Gründung von virtuellen Maschinen* und zum anderen der *Schritt zur Beendung der letzten virtuellen Maschine.*

5.2.1 Die δ-Reduktion

Zeigt der CPR auf eine k-stellige primitive Funktion, so befinden sich Verweise auf die Argumente, auf die sie angewendet werden soll, auf dem S-Stack.

Sind mindestens k Argumente vorhanden, so müssen zunächst für die Berechnung der WHNF der Argumente virtuelle Maschinen gegründet werden. Dabei verbleiben die Verweise auf die Argumente auf dem S-Stack, um zum einen die Argumente nach der Berechnung noch referenzieren zu können und zum anderen im Falle der Nichtanwendbarkeit die ursprüngliche Applikationsstruktur wiederherstellen zu können.

Die für die vollständige Spezifizierung der virtuellen Maschinen für die einzelnen Argumente notwendigen Kontrollmodusflags sind nicht auf die WHNF beschränkt, sondern für jede primitive Funktion spezifisch definiert. So wird es möglich, primitive Funktionen in das System zu integrieren, die zur Anwendung auf Argumente diese in einer anderen als der WHNF benötigen. Eine denkbare Anwendung dafür wäre es z.B., das If-Then-Else-Konstrukt als dreistellige Funktion durch die Zuordnung der Flags WHNF|TERMINATED|TERMINATED zu den Argumenten zu handhaben. In der bisherigen Version von $\mathcal{L}i$SA wird diese Möglichkeit jedoch nicht genutzt. Das If-Then-Else-Konstrukt wird durch ein monadisches Conditional realisiert.
Als Zustandsübergang ergibt sich für die initiale Gründung der virtuellen Maschinen:

$$\Big(e, n, S, F, I\Big[p_1 = (n_1, e_1)\Big], G\Big[n = prf^{\,k,0}\Big], E, f\Big) \text{ mit } S = \Big[(p_1, \hat{n_1}).\ldots.(p_k, \hat{n_k}).S'\Big]$$
$$\Longrightarrow \Big(e_1, n_1, \Big[\%.S\Big], \Big[(p_1, f_1).\ldots.(p_k, f_k).\%.(n, e, f).F\Big], I, G, E, f\Big).$$

sowie für die eigentliche δ-Reduktion:

$$\Big(e, n, \Big[\%.(p_1, \hat{n}_1).\ldots.(p_k, \hat{n}_k).S\Big], \Big[(p_k, f).(\%, n_f).F\Big], I, G\Big[n_f{=}prf^{\,k,0}\Big], E, t\Big)$$
$$\Longrightarrow \begin{cases} \Big(e, n', S, F, I\Big[p_k{=}(n, e)\Big], G, E, t\Big) & \text{falls } n'{=}prf(p_1, \ldots, p_k) \\ \Big(e, n_f, \Big[(p_1, \hat{n}_1).\ldots.(p_k, \hat{n}_k).S\Big], F, I\Big[p_k{=}(n, e)\Big], G, E, t\Big) \text{ sonst} \end{cases}.$$

Liegen weniger als k Argumentverweise auf dem S-Stack, so wird wie bei der β-Reduktion ein Abschluß gebildet:

$$\Big(e, n, \Big[(p_1, n_1).\ \ldots\ .(p_l, n_l).S\Big], F, I, G\Big[n = prf^{\,k,0}\Big], E, f\Big) \quad \wedge \quad l < k$$
$$\Longrightarrow \Big(e, n', S, F, I, G\Big[n' = prf^{\,k,l} p_1 \ldots p_l\Big], E, t\Big).$$

5.2.2 Lazy Listen

Der Zustandsübergang bei Lazy Listen hängt entscheidend von der benötigten Normalform und somit dem Kontrollmodusflag ab.

Wird die WNF einer Lazy Liste benötigt, so müssen virtuelle Maschinen zur Berechnung der Unterkomponenten gegründet werden. Da nach der Auswertung der Unterkomponenten diese nicht mehr ausschließlich durch einen Zeiger in den Graphen, sondern durch Indirektionsknoten beschrieben werden, muß für die ausgewertete Lazy Liste ein neues internes Konstrukt ($\overline{\texttt{cons}}$), das zwei Indirektionsknoten als Untergraphen hat, gebildet werden. Um ein zusätzliches Ablegen der Indirektionsknoten auf dem S-Stack zu vermeiden, erfolgt die Bildung des $\overline{\texttt{cons}}$-Knotens bereits bei der initialen Gründung der virtuellen Maschinen:

$$\Big(e, n, S, F, I, G\Big[n = \texttt{cons}\, n_1\, n_2\Big], E, w\Big)$$
$$\Longrightarrow \Big(e, n_1, \Big[\%.S\Big], \Big[(p_1, w).(p_2, w).(\%, n', w).F\Big],$$
$$I\Big[p_1 = (n_1, e), p_2 = (n_2, e)\Big], G\Big[n' = \overline{\texttt{cons}}\, p_1\, p_2\Big], E, w\Big).$$

Dadurch ergibt sich als Zustandsübergang nach der Beendigung der letzten virtuellen Maschine:

$$\Big(e, n, \Big[\%.S\Big], \Big[(p_2, .).(\%, n_f, w).F\Big], I, G\Big[n_f = \overline{\texttt{cons}} p_1, p_2\Big], E, t\Big)$$
$$\Longrightarrow (e, n_f, S, F, P, I\Big[p_2 = (n, e)\Big], G, E, t\Big).$$

Wird jedoch die WHNF einer Lazy Liste benötigt, so werden zwar die Komponenten der Liste instanziiert (Bildung eines $\overline{\texttt{cons}}$ Knotens), ihre Berechnung findet jedoch nicht statt:

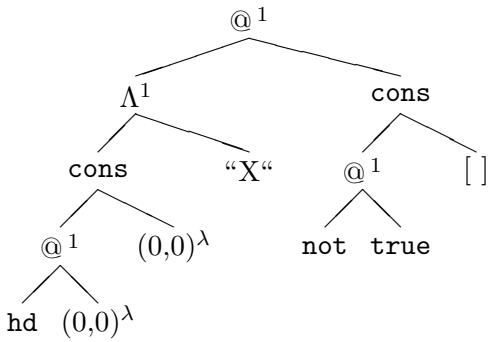
$$\Big(e, n, S, F, I, G\Big[n = \texttt{cons}\, n_1\, n_2\Big], E, h\Big)$$
$$\Longrightarrow \Big(e, n', S, F, I\Big[p_1 = (n_1, e), p_2 = (n_2, e)\Big], G\Big[n' = \overline{\texttt{cons}}\, p_1\, p_2\Big], E, t\Big).$$

Eine Besonderheit der primitiven Funktionen stellen die destrukturierenden Funktionen `hd` und `tl`, die die erste bzw. die zweite Komponente einer Lazy Liste selektieren, dar. Bei ihrer Anwendung treten aufgrund des Indirektionsknotenkonzeptes Probleme mit dem Sharing von Berechnungen auf. Dies läßt sich am besten an einem Beispiel aufzeigen. Der $\mathcal{L}$K*i*R-Ausdruck

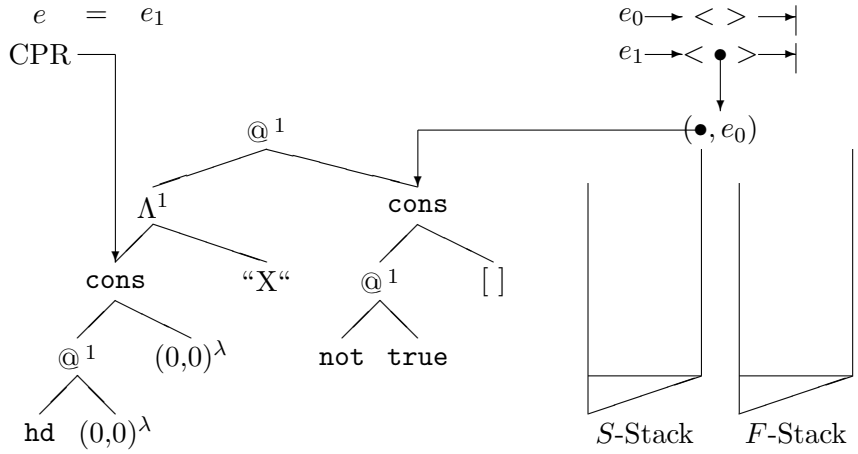
Beispiel 5.2: Das Sharing von Berechnungen bei der Anwendung von `hd`

```
let X = [ not(true) . [] ]
in [ hd(X) . X ]
```

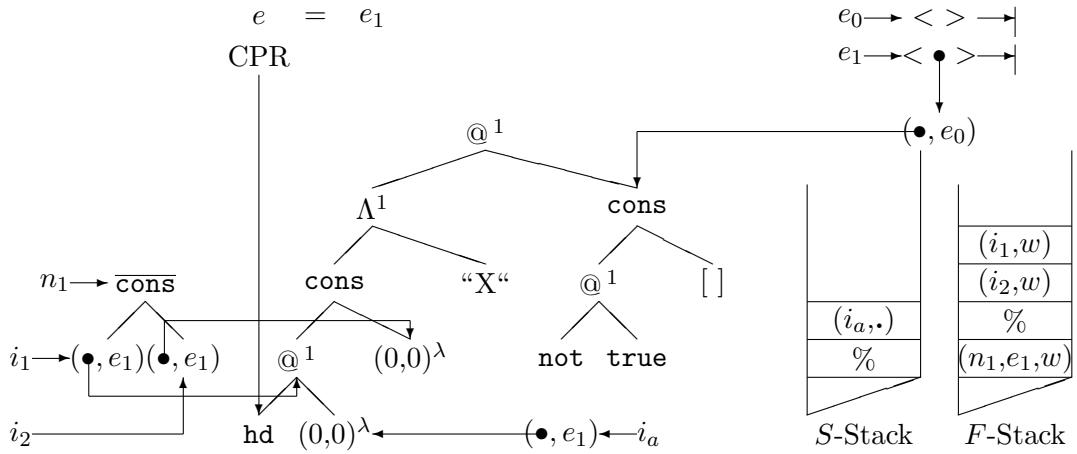
wird intern dargestellt als:



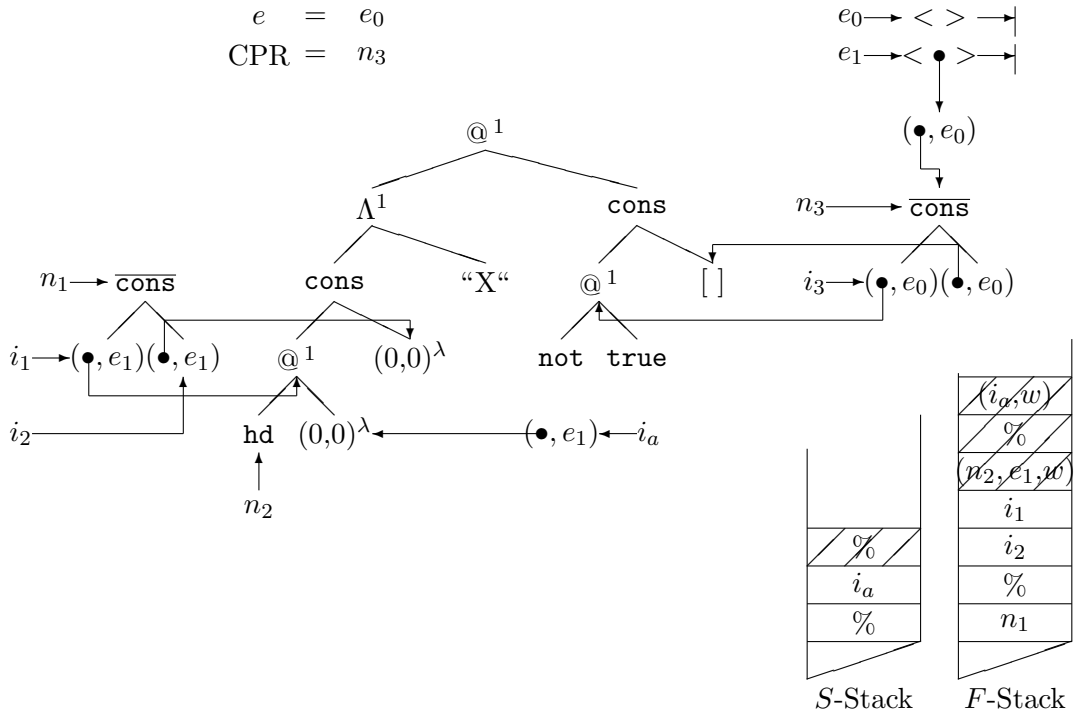
Nach zwei Transformationsschritten zeigt der CPR auf den Rumpf des Let-Konstruktes und im aktuellen Environment e_1 befindet sich eine Instanz der Liste `[ not(true) . [] ]`:



Da das Resultat der gesamten Reduktion aus einer Liste besteht, werden virtuelle Maschinen zur Berechnung der Komponenten gegründet und der $\overline{\texttt{cons}}$-Knoten für das Resultat bereits erzeugt. Anschließend wird eine Instanz des Argumentes der Anwendung von `hd` auf den S-Stack gelegt und der CPR verweist auf die primitive Funktion `hd`. Nach zwei weiteren Transformationsschritten ergibt sich also:



Diese Situation führt wiederum zur Gründung einer virtuellen Maschine für die Berechnung der WHNF des durch i_a repräsentierten Ausdruckes. Da es sich dabei um eine Variable handelt, wird das Sharing der Berechnung des zugehörigen Substitutes in e_1 durch dessen Update mit dem Ergebnis der Berechnung - einem neuen $\overline{\texttt{cons}}$-Knoten - erreicht. Somit liegt bei der eigentlichen Anwendung von `hd` folgende Situation vor:



Die ordnungsgemäße Beendung der durch (i_a,w) repräsentierten virtuellen Maschine führt dazu, daß

1. i_a mit dem Ergebnis (n_3, e_0) überschrieben,
2. die schraffierten Stackelemente entfernt und
3. dabei der CPR auf n_2 gesetzt sowie e_1 zum aktuellen Environment gemacht

werden.

Die Anwendung von `hd` hat die Fortsetzung der Berechnung mit der linken Komponente des $\overline{\texttt{cons}}$-Konstruktes zur Folge. Dies wird dadurch realisiert, daß der CPR auf die Anwendung von `not` auf `true` gesetzt, e_0 zum aktuellen Environment gemacht und i_a vom S-Stack genommen wird. Ein solches Vorgehen alleine führt jedoch dazu, daß nach der Berechnung von `not(true)` zwar i_1 mit einem Verweis auf `false` überschrieben wird, i_3 jedoch unverändert bleibt. Im Verlauf der Berechnung von i_2 (rechte Komponente der Ergebnisliste n_1) kommt es somit zu einer "zweiten" Berechnung von `not(true)`. Um dies zu vermeiden, muß sichergestellt sein, daß nach der Berechnung von `not(true)` nicht nur i_1, sondern auch i_3 mit dem Verweis auf das Ergebnis überschrieben wird. Das läßt sich genauso wie bei Variablenzugriffen mittels einer Update-Marke $(\$, i_3)$ auf dem S-Stack veranlassen.
Allgemein ergibt sich als Transformationsregel:

$$\Big(e, n, \Big[\%.(p_1, \hat{n}_1).S\Big], \Big[(p_1, .).(\%, n_f, f').F\Big], I\Big[p_l{=}(n_l, e_l)\Big], G\Big[n_f = hd\Big], E, t\Big)$$
$$\Longrightarrow \begin{cases} \Big(e_l, n_l, \Big[(\$, p_l).S\Big], F, I\Big[p_1{=}(n, e)\Big], G, E, f'\Big) & \text{falls } G\Big[n{=}\overline{\texttt{cons}}\, p_l\, p_r\Big] \\ \Big(e, n_f, \Big[(p_1, \hat{n}_1).S\Big], F, P, I\Big[p_1{=}(n, e)\Big], G, E, t\Big) & \text{sonst} \end{cases}.$$

In analoger Weise erfolgt die Behandlung von Anwendungen der primitiven Funktion `tl`, die die zweite Komponente einer binären Liste selektiert.

5.2.3 Tupel

Neben den Lazy Listen stellt $\mathcal{L}i\text{SA}$ noch ein striktes Listenkonstrukt, die sogenannten *Tupel* zur Verfügung. Strikt bedeutet in diesem Fall, daß die WHNF des Tupels die WHNF der Unterkomponenten erfordert. Daraus resultiert, daß sowohl zur Berechnung der WHNFals auch zur Berechnung der WNF eine initiale Gründung von virtuellen Maschinen erfolgen muß:

$$\begin{aligned} &\Big(e, n, S, F, I, G\Big[n = \text{tup}^k\, n_1 \ldots n_k\Big], E, f\Big) \\ &\Longrightarrow \Big(e, n_1, \Big[\%.S\Big], \Big[(p_1, f) \ldots (p_k, f).(\%, n', f).F\Big], \\ &\qquad I\Big[p_1 = (n_1, e) \ldots p_k = (n_k, e)\Big], G\Big[n' = \overline{\text{tup}}^k\, p_1 \ldots p_k\Big], E, f\Big). \end{aligned}$$

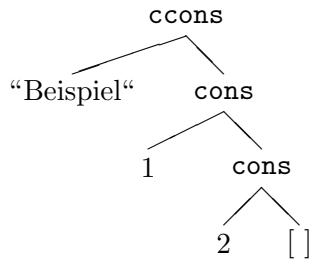
Der Zustandsübergang nach Beendigung der letzten virtuellen Maschine ist vollkommen analog zu dem der Lazy Listen.

Abschließend bleibt anzumerken, daß die primitiven destrukturierenden Funktionen für Tupel (`lselect` und `cut`) keines zusätzlichen Updates bedürfen, da vor der Selektion die betreffenden Listenelemente bereits evaluiert worden sind.

5.2.4 Benutzerdefinierte Konstruktoren

Da auch mit den benutzerdefinierten Konstruktoren konzeptuell unendliche Datenstrukturen konstruierbar sein sollen, erfolgt ihre Darstellung mit Hilfe der Lazy Listen. Um sie dennoch von diesen eindeutig unterscheidbar zu halten, ist das äußerste Listenkonstrukt der benutzerdefinierten Konstruktoren ein eigenständiger Graphknoten (`ccons`). Ein Ausdruck wie z.B. `Beispiel{1,2}` wird intern dargestellt als:

Abbildung 5.5: Interne Darstellung von `Beispiel{1,2}`



Daher ergeben sich für die benutzerdefinierten Konstruktoren vollkommen zu den Lazy Listen analoge Transformationsregeln. Die Zerlegung dieser Datenstrukturen mittels primitiver Funktionen ist im Gegensatz zur π-RED* nicht möglich. Sie kann lediglich mit Hilfe des Patternmatches erfolgen.

5.3 Das Patternmatch

In diesem Abschnitt werden die Konstrukte der internen Darstellung entwickelt, mit deren Hilfe der Matchvorgang von Case-Abstraktionen in $\mathcal{L}i$SA realisiert wird. Wie eine solche interne Darstellung aus einer vorgegebenen Case-Abstraktion abgeleitet wird, wird hier nur insoweit beschrieben, als es für die Entwicklung der Konstrukte sinnvoll scheint; eine detaillierte Darstellung der Erzeugung des Patternmatch-Codes findet sich im nächsten Kapitel.
Eine Case-Abstraktion hat die allgemeine Form:

case
  when $(p_{1,1}, \ldots, p_{1,n})$ guard g_1 do e_1
  $\vdots$
  when $(p_{m,1}, \ldots, p_{m,n})$ guard g_1 do e_m
  otherwise e_o
end

Liegt eine Anwendung auf die Argumente $a_1, \ldots, a_n$ vor, so befinden sich Verweise auf die Argumente auf dem S-Stack. Eine solche Situation erfordert, daß zunächst das kleinste $i \in \{1, \ldots, m\}$ gesucht wird, für das die Pattern $(p_{i,1}, \ldots, p_{i,n})$ auf die Argumente $(a_1, \ldots, a_n)$ passen. Dazu bedarf es eines Mechanismus, der die Argumente bezüglich eines festen i in geordneter Weise untersucht und bei einem nicht passenden Pattern die Argumentverweise wieder auf dem S-Stack hinterläßt. Dies wird mittels des Pm-Stacks realisiert: die Verweise auf Argumente, bei denen ein Patterntest erfolgreich ist, werden vom S-Stack genommen und auf den Pm-Stack gelegt. Sind alle Tests für ein i erfolgreich, so folgt die Berechnung des zugehörigen Guard-Ausdruckes. Schlägt ein Test fehl, so können die Verweise bereits überprüfter Argumente wieder vom Pm-Stack zurück auf den S-Stack gelegt und mit den Patterntests der nächsten π-Abstraktion fortgefahren werden.

Dieser Mechanismus ermöglicht insofern die Realisierung eines baumartigen Ansatzes, als daß es möglich ist, nach dem Fehlschlagen eines Tests nur einen Teil der bereits untersuchten Argumentverweise vom Pm-Stack auf den S-Stack zurückzulegen.

Bei den Pattern-Konstrukten gibt es zwei prinzipiell verschiedene Gruppen: zum einen *überprüfende Pattern*, die eine vorgegebene Argumentstruktur verlangen, und zum anderen *bindende Pattern*, die eine Substitution von Argumenten in den Guard bzw. Rumpf der zugehörigen π-Abstraktion bewirken. Bei den überprüfenden Pattern muß weiterhin zwischen Konstanten und Datenstrukturen unterschieden werden. Während bei Konstanten nur ein Test auf einen Wert erfolgt, erfordern Datenstrukturen eine strukturelle Überprüfung sowie anschließend einen Patterntest auf die Unterkomponenten. Um für diese Tests die gleichen Mechanismen wie für alle Pattern verwenden zu können, wird der oben skizzierte Mechanismus dahingehend erweitert, daß nach dem erfolgreichen Patterntest einer Datenstruktur Verweise auf die Unterkomponenten wie Argumentverweise auf den S-Stack gelegt werden. Diese Vorgehensweise ermöglicht eine gleichartige Behandlung von Teilstrukturen von Argumenten wie von Argumenten selbst. Deshalb soll im weiteren von *Argumentteilen* als Überbegriff gesprochen werden.
Es kristallisieren sich vier für das Patternmatch benötigte Mechanismen

1. zur Substitution von Argumentteilen in den Guard bzw. Rumpf einer π-Abstraktion,
2. zur Überprüfung von Argumentteilen auf Konstanten,
3. zur strukturellen Überprüfung von Argumentteilen auf Datenstrukturen sowie ggf. deren Zerlegung und
4. zur Rekonstruktion von Argumentteilverweisen auf dem S-Stack mit Hilfe des Pm-Stacks nach dem Fehlschlagen eines Tests

heraus.

5.3.1 Bindende Patternkonstrukte

Um die bei π-Abstraktionen durch bindende Patternkonstrukte gebundenen Variablen genauso handhaben zu können wie λ-gebundene Variablen, muß nach erfolgreichem Patterntest das aktuelle Environmentframe kopiert und um die Substitute für die durch das Patternmatch gebundenen Variablen ergänzt werden.

Nun kann es jedoch während eines Matchvorganges vorkommen, daß Tests auf bindende Patternkonstrukte gemacht werden, die zu den Pattern einer π-Abstraktion gehören, die nicht vollständig auf die Argumente paßt. Hierzu ein Beispiel:

Beispiel 5.3: Überflüssige Environment-Ergänzungen während des Matchvorganges

```
ap
  case
    when ( A, B, 1 ) guard true do < A, B>
    when ( [ D. E ], F, 2 ) guard true do < D, E, F >
    when ( [ 1. 7 ], X, 3 ) guard true do X
  end
to [ [ 1. 7 ], 42, 3 ]
```

Hier werden die während der Patterntests ausfindig gemachten Substitute für die Variablen A-F zur Auswertung des Rumpfes nicht mehr benötigt. Um das aufwendige Einrichten nicht benötigter Environments zu ersparen, gibt es in der $\mathcal{L}i$SA einen zweiten für das Patternmatch genutzten Stack, den Bi-Stack. Auf ihm werden während des Matchvorganges Zeiger

auf Argumentteile abgelegt, die nach erfolgreichen Patterntests in die aktuelle Umgebung eingetragen werden müssen.

Es gibt zwei Arten von bindenden Patternkonstrukten: Variablen und Ausdrücke der Form `as` $< Pattern >< Variable >$.

Bei Variablen wird der Verweis auf den zugehörigen Argumentteil auf den Bi-Stack gelegt und der Argumentteil gilt als erfolgreich getestet; d.h. der Verweis darauf gelangt vom S- auf den Pm-Stack.

$$\Big(e, n, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, Bi, I, G\Big[n = \Lambda_b n_0\Big], E, f\Big)$$
$$\Longrightarrow \Big(e, n_0, S, F, \Big[(p_1, \hat{n}_1).Pm\Big], \Big[p_1.Bi\Big], I, G, E, f\Big).$$

Im anderen Fall, realisiert durch das interne Konstrukt Λ_{as}, wird zwar auch der Argumentteilverweis auf den Bi-Stack gelegt, jedoch bedarf der Argumentteil selbst weiterer Überprüfung.

$$\Big(e, n, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, Bi, I, G\Big[n = \Lambda_{as} n_0\Big], E, f\Big)$$
$$\Longrightarrow \Big(e, n_0, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, \Big[p_1.Bi\Big], I, G, E, f\Big).$$

Die Übernahme der im Bi-Stack abgelegten Substitute in das Environment muß noch vor der Beendigung des Matchvorganges erfolgen. Dies ist darin begründet, daß zur Berechnung des Guard-Ausdruckes das Environment benötigt wird. Deshalb ist die Überprüfung des Guard-Ausdruckes folgendermaßen realisiert: an einem internen Konstrukt Λ_π befinden sich als Unterkomponenten

1. eine für das Pre- und Postprocessing benötigte Darstellung des Patterns in seiner ursprünglichen Form,
2. der Guard Ausdruck,
3. der Rumpfausdruck sowie
4. ein Verweis auf den weiter zu verfolgenden Patternmatch Graphen im Falle eines zu false evaluierbaren Guards.

Zeigt der CPR auf ein solches Konstrukt, so wird das notwendige Environment erstellt und eine virtuelle Maschine zur Auswertung des Guards gegründet.

$$\Big(e, n, S, F, Pm, Bi, I, G\Big[{\Lambda_\pi}^k n_p n_g n_b n_f\Big], E\Big[e{=}\langle p'_1 \ldots p'_m\rangle \to \hat{e}\Big], f\Big) \text{ mit } Bi{=}\Big[p_1.\ldots.p_k.Bi'\Big]$$
$$\Rightarrow \Big(e', n_g, \Big[\%.S\Big], \Big[(p, w).\%.(n, e, e', f).F\Big], Pm, Bi, I, G, E\Big[e'{=}\langle p_1 \ldots p_k p'_1 \ldots p'_m\rangle \to \hat{e}\Big], f\Big).$$

Nach dessen Berechnung wird anhand des Ergebnisses entschieden, ob mit der Berechnung des Rumpfes begonnen oder der Matchvorgang weitergeführt werden soll.

$$(e'', n', \Big[\%.S\Big], \Big[(p, w).\%.(n, e, e', f).F\Big], Pm, Bi, I, G\Big[n' = {\Lambda_\pi}^k n_p n_g n_b n_f\Big], E, f\Big)$$
$$\Longrightarrow \begin{cases} \Big(e', n_b, S, F, Pm, Bi, I, G, E, f|\Big) & \text{falls } n = \text{true} \\ \Big(e, n_f, S, F, Pm, Bi, I, G, E, f\Big) & \text{falls } n = \text{false} \end{cases}.$$

5.3.2 Konstanten und Datenstrukturen

Aufgrund der in Kapitel 3.5 festgelegten Bedeutung von π-Abstraktionen ergibt sich, daß die Berechnung eines Argumentteiles nur soweit vorangetrieben wird, wie es notwendig ist, um zu entscheiden, ob das Pattern paßt. Daraus resultiert, daß zu jeder Konstanten bzw. Datenstruktur im Pattern ein zugeordnetes Argumentteil potentiell unausgewertet ist. Deshalb ist vor dem Test auf Gleichheit der betreffende Argumentteil zur WHNF zu reduzieren. Dies wird mittels eines Σ-Konstruktes realisiert, das zunächst zur Gründung einer virtuellen Maschine führt.

$$\begin{aligned}&\Big(e, n, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p_1 = (n_1, e_1)\Big], G\Big[n = \Sigma\ldots\Big], E, f\Big)\\ &\Longrightarrow \Big(e_1, n_1, \Big[\%.(p_1, n').S\Big], \Big[(p_1, w).\%.(n, e, f).F\Big], Pm, Bi, I, G, E, w\Big).\end{aligned}$$

Da für das Patternmatch ein baumartiger Ansatz verfolgt werden soll, ist es sinnvoll, das Konstrukt derart zu gestalten, daß nicht *auf bestimmte Werte getestet*, sondern *je nach Wert selektiert* wird. Eine solche Selektion führt das Σ Konstrukt nach der Evaluation des Argumentteiles aus. Dazu wird zwischen den verschiedenen Typen und den Konstanten **true** sowie **false** unterschieden.

$$\begin{aligned}&(e, n, \Big[\%.(p_1, n').S\Big], \Big[(p, w).\%.(n_f, e_f, f).F\Big], Pm, Bi, I, G\Big[n_f = \Sigma\ldots\Big], E, f\Big) \qquad (*)\\ &\quad \text{mit } G\Big[n_f = \Sigma n_{tup} n_{cons} n_{ccons} n_{nil} n_{true} n_{false} n_{int} n_{string} n_{default}\Big]\\ &\Longrightarrow \begin{cases}\Big(e, n_{tup}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \text{falls } n = \text{tup}^k \wedge k \geq 1\\ \Big(e, n_{cons}, \Big[(p_l, n).(p_r, n).S\Big], F, & \text{falls } n = \overline{\texttt{cons}} p_l p_r\\ \quad \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \\ \Big(e, n_{ccons}, \Big[(p_l, n).(p_r, n).S\Big], F, & \text{falls } n = \overline{\texttt{ccons}} p_l p_r\\ \quad \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \\ \Big(e, n_{nil}, S, F, \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \text{falls } n = \text{tup}^0\\ \Big(e, n_{true}, S, F, \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \text{falls } n = \texttt{bool}^{true}\\ \Big(e, n_{false}, S, F, \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \text{falls } n = \texttt{bool}^{false}\\ \Big(e, n_{int}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \text{falls } n = \texttt{int}^i\\ \Big(e, n_{string}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \text{falls } n = \texttt{string}^{\text{`}s\text{`}}\\ \Big(e, n_{default}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f\Big) & \text{sonst}\end{cases}\end{aligned}$$

Eine differenzierte Selektion zwischen beliebigen Konstanten findet nicht statt, um die Funktionalität des Konstruktes nicht zu überladen. Da jedoch diese für das Vergleichen auf Konstanten unerläßlich ist, gibt es ein σ Konstrukt, das sowohl eine Liste mit den zu vergleichenden Konstanten als auch entsprechend viele Unterausdrücke enthält. Es führt bei Übereinstimmung des aktuellen Wertes mit einem Wert der Liste zur Selektion des zugehörigen Unterausdruckes. Für den Fall, daß keiner der Werte übereinstimmt, führt das zur Wahl eines gesonderten Ausdruckes, des sogenannten Default-Ausdruckes.

$$(e, n, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p_1 = (n_1, e_1)\Big], G\Big[n = \sigma\langle v_1 \ldots v_k\rangle n_1 \ldots n_k n_{def}\Big], E, f\Big)$$
$$\Longrightarrow \begin{cases} \Big(e, n_i, S, F, \Big[(p_1, n').Pm\Big], Bi, I, G, E, f\Big) & \text{falls } n_1 = v_i \wedge \forall j \in \{1, \ldots, (i\text{-}1)\} : v_j \neq n_1 \\ \Big(e, n_{def}, \Big[(p_1, n').S\Big], F, Pm, Bi, I, G, E, f\Big) & \text{sonst} \end{cases}.$$

Betrachten wir noch einmal die Transformationsregel $(*)$. Liegt eine Lazy Liste oder ein benutzerdefinierter Konstruktor vor, so wird direkt durch das Σ-Konstrukt der S-Stack vollständig auf das Patternmatch der Unterkomponenten vorbereitet. Soll ein Test auf ein Tupel erfolgen, ist dies nicht der Fall. Die Ursache dafür liegt darin, daß bei der Spezifikation von Pattern ein sogenanntes Drei-Punkte-Konstrukt zulässig ist. Durch dieses Konstrukt wird es möglich, eine beliebig lange Teilliste eines Tupels beim Match-Vorgang unberücksichtigt zu lassen. Da also gegebenenfalls ein großer Teil der Unterkomponenten eines Tupels gar nicht beim Match-Vorgang getestet werden muß, ist es unneffizient, immer Verweise auf alle Unterkomponenten eines Tupels durch das Σ-Konstrukt auf den S-Stack zu legen. Stattdessen werden mittels neuer Konstrukte $\sigma_{<>}$ und $\sigma_{<.>}$ nur Verweise auf die noch zu untersuchenden Unterkomponenten auf den S-Stack gelegt.

Eine weitere Folge des Drei-Punkte-Konstruktes ist es, daß ein Pattern, das ein solches Konstrukt enthält, auf Tupel unterschiedlicher Länge paßt. Deshalb wird nicht stelligkeitsselektierend vorgegangen, sondern nur auf eine Stelligkeit ($\sigma_{<>}$) bzw. Mindeststelligkeit ($\sigma_{<.>}$) geprüft und bei Erfolg die entsprechenden Unterkomponentenverweise auf den S-Stack gelegt. Der Test auf eine bestimmte Tupellänge wird durch das $\sigma_{<>}$-Konstrukt realisiert:

$$(e, n, \Big[(p_0, n').S\Big], F, Pm, Bi, I\Big[p_0 = (n_0, e_0)\Big], G\Big[n = {\sigma_{<>}}^k n_t n_f, n_0 = \overline{\text{tup}}^l p_1 \ldots p_l\Big], E, f\Big)$$
$$\Longrightarrow \begin{cases} \Big(e, n_t, \Big[(p_1, n_0).\ldots.(p_l, n_0).S\Big], F, \Big[(p_0, n').Pm\Big], Bi, I, G, E, f\Big) & \text{falls } k = l \\ \Big(e, n_f, \Big[(p_0, n').S\Big], F, Pm, Bi, I, G, E, f\Big) & \text{sonst} \end{cases}.$$

Ein Test auf eine Mindestlänge erfolgt durch das $\sigma_{<.>}$-Konstrukt. Da ggf. mittels einer Konstruktion **as ...** $< Variable >$ im Pattern eine Teilliste variabler Länge ebenfalls gebunden werden kann, wird ein entsprechendes Tupel ebenfalls mit auf den S-Stack gelegt. Die Restriktion, maximal ein Drei-Punkte -Konstrukt pro Tupel im Pattern vorliegen zu haben, ermöglicht die statische Bestimmung der für den Matchvorgang benötigten Tupelkomponenten. Zur vollständigen Spezifikation der benötigten Unterkomponenten dienen zwei ganzzahlige Attribute k und o, wobei k die Mindestlänge und o den Offset des Drei-Punkte-Konstruktes innerhalb des Tupels darstellen. Ein Tupel $< e_1, \ldots, e_l >$ wird durch ${\sigma_{<.>}}^{k,o}$ also in folgende $(k+1)$ Ausdrücke zerlegt:

$$\underbrace{e_1, \ldots, e_o}_{o}, < \underbrace{e_{(o+1)}, \ldots, e_{(o+l-k)}}_{l-k} >, \underbrace{e_{(o+l-k+1)}, \ldots, e_l}_{k-o}.$$

Damit ergibt sich als Transformationsregel:

$$(e, n, \Big[(p_0, n').S\Big], F, Pm, Bi, I\Big[p_0 = (n_0, e_0)\Big], G\Big[n = {\sigma_{<.>}}^{k,o} n_t n_f, n_0 = \overline{\text{tup}}^l p_1 \ldots p_l\Big], E, f\Big)$$
$$\Longrightarrow \begin{cases} \Big(e, n_t, S', F, \Big[(p_0, n').Pm\Big], Bi, I\Big[p' = (n'', e)\Big], G', E, f\Big) & \text{falls } k \leq l \\ \quad \text{mit } S' = \Big[(p_1, n_0).\ldots.(p_o, n_0).(p', n_0).(p_{(o+l-k+1)}, n_0).\ldots.(p_l, n_0).S\Big] & \\ \quad \text{und } G' = G\Big[n'' = \overline{\text{tup}}^{(l-k)} p_{(o+1)} \ldots p_{(o+l-k)}\Big] & \\ \Big(e, n_f, \Big[(p_0, n').S\Big], F, Pm, Bi, I, G, E, f\Big) & \text{sonst} \end{cases}.$$

5.3.3 Match-Kontroll-Mechanismen

Bis jetzt sind ausschließlich testende Konstrukte beschrieben worden; es fehlt also noch die Realisierungsbeschreibung des Ausführungsmechanismus für den Fall eines fehlschlagenden Tests. Eine mögliche Realisierung liegt darin, ein Backtracking auf den Matching-Konstrukten vorzunehmen. Dies bringt jedoch Nachteile mit sich: Backtracking erfordert das Nachhalten von sogenannten *Choice-Punkten* - dabei handelt es sich um die Argumentpositionen, in denen sich Patternausdrücke überlappen - und gegebenenfalls das Rückverfolgen mehrerer Konstrukte, um eine korrekte Argumentsituation auf dem S-Stack zu erhalten. Ein solches schrittweises Vorgehen kann zu redundanten Stackoperationen wie z.B. Traversieren von PM nach S und anschließendes Entfernen von S führen.

Zur Vermeidung dieses Aufwandes existiert in der $\mathcal{L}i\text{SA}$ ein Konstrukt Ξ, das es gestattet, mit minimalem Aufwand von Stackoperationen die benötigte Konstellation von Argumentverweisen auf dem $S-, Pm-$ sowie Bi-Stack zu rekonstruieren:

$$\Big(e, n, \Big[s_1 \ldots s_s . S\Big], F, \Big[d_1 \ldots d_p . p_1 \ldots p_m . Pm\Big], \Big[b_1 \ldots b_{bi} . Bi\Big], I, G\Big[n{=}\Xi_{(s,p,m,bi)}\, n_0\Big], E, f\Big)$$
$$\Longrightarrow \Big(e, n_0, \Big[p_m . \ldots . p_1 . S\Big], F, Pm, Bi, I, G, E, f\Big).$$

Sollte kein solcher mehr existieren, so führt ein fehlschlagender Test auf ein Ω Konstrukt, was zu einer Beendung der Berechnung führt:

$$\Big(e, n, S, F, P, G\Big[n = \Omega\, n_0\Big], E, f\Big)$$
$$\Longrightarrow \Big(e, n_0, S, F, P, G, E, t\Big).$$

Der Vollständigkeit halber sei hier das Verhalten des Ausführungsmechanismus bei partiellen Anwendungen bzw. nicht entscheidbaren Matchvorgängen (abgelaufener Reduktionszähler) kurz erläutert. Für diese Fälle wird mittels des $\overline{\Pi}$-Konstruktes ein Abschluß mit allen relevanten Komponenten (aktueller Knoten, aktueller $S-, Pm-, Bi-$Stack) gebildet. Diese Vorgehensweise ermöglicht sowohl eine hochsprachliche Rekonstruktion des verbleibenden Ausdruckes als auch eine problemlose Fortsetzung der Berechnung, falls weitere Argumente zugeführt bzw. eine weitere Berechnung angestrengt wird (*verpointertes Wiederaufsetzen*).

6 Die Schnittstelle zu $\mathcal{L}$KiR

Während die Architektur des Systems eine Transformation von $\mathcal{L}$KiR- Ausdrücken vorgibt, erfolgt intern eine Reduktion von Graphen. Wie bereits in Kapitel 3.1 erwähnt, ist deshalb eine Konvertierung zwischen diesen beiden Darstellungen erforderlich; sie wird durch eine Abbildung $\Psi : \mathcal{L}\mathrm{K}i\mathrm{R} \rightarrow \mathcal{L}_{Graph}$ mit folgenden Eigenschaften realisiert:

$$\begin{array}{c} \forall e \in E_{\mathcal{L}\mathrm{K}i\mathrm{R}} : \Psi^{-1}(\Psi(e)) = e \\ \forall e' \in E_{\mathcal{L}_{Graph}} : \exists e \in E_{\mathcal{L}\mathrm{K}i\mathrm{R}} : \Psi^{-1}(e') = e. \end{array} \quad (1)$$

Um die Transformationsvorgänge besser beschreiben zu können, wird die Abbildung Ψ in vier Teilabbildungen Ψ_1, Ψ_2, Ψ_3 und Ψ_4 zerlegt. Damit ergibt sich als Diagramm:

Abbildung 6.1: Sprachebenen

$$\begin{array}{ccccc} & \mathcal{L}\mathrm{K}i\mathrm{R} & \longrightarrow & \mathcal{L}\mathrm{K}i\mathrm{R} & \\ \Psi_1 & \downarrow & & \uparrow & \Psi_1^{-1} \\ & \mathcal{L}_2 & & \mathcal{L}_2 & \\ \Psi_2 & \downarrow & & \uparrow & \Psi_2^{-1} \\ & \mathcal{L}_3 & & \mathcal{L}_3 & \\ \Psi_3 & \downarrow & & \uparrow & \Psi_3^{-1} \\ & \mathcal{L}_4 & & \mathcal{L}_4 & \\ \Psi_4 & \downarrow & & \uparrow & \Psi_4^{-1} \\ & \mathcal{L}_{Graph} & \longrightarrow & \mathcal{L}_{Graph} & \end{array} .$$

Dabei sind die Abbildungen Ψ_i derart gestaltet, daß für geeignete Sprachen $\mathcal{L}_1 = \mathcal{L}\mathrm{K}i\mathrm{R},\ \mathcal{L}_2,\ \mathcal{L}_3,\ \mathcal{L}_4,\ \mathcal{L}_5 = \mathcal{L}_{Graph}$ und alle $i \in \{1, 2, 3, 4\}$ gilt:

$$\forall e \in \mathcal{L}_i : \Psi_i^{-1}(\Psi_i(e)) = e \quad (*)$$
$$\forall e' \in \mathcal{L}_{(i+1)} : \exists e \in \mathcal{L}_i : \Psi_i^{-1}(e') = e \quad (**)$$

Daraus ergibt sich offenbar für $\Psi = (\Psi_4 \circ \Psi_3 \circ \Psi_2 \circ \Psi_1)$ mit $\Psi^{-1} = (\Psi_1^{-1} \circ \Psi_2^{-1} \circ \Psi_3^{-1} \circ \Psi_4^{-1})$ die geforderte Bedingung (1).

Es erfolgt deshalb in diesem Kapitel ein suksessive Beschreibung der Abbildungen Ψ_i (Preprocessing) sowie ihrer Umkehrabbildungen Ψ_i^{-1}(Postprocessing). Um eine konzise Darstellung zu erreichen, werden Graphen so weit wie möglich pre order linearisiert dargestellt.

6.1 Schritt 1: Sprachvereinfachung

In $\mathcal{L}$KiR gibt es verschiedene Sprachkonstrukte, die eine gleiche Funktionalität haben. Sie dienen dem Benutzer, um Hochsprachenprogramme übersichtlicher und lesbarer zu gestalten (*syntactic sugar*). Um den Ausführungsmechanismus kompakt halten zu können, werden sie vor dem Übersetzen in die interne Darstellung durch bedeutungsgleiche Ausdrücke ersetzt. Dies beschreibt die Funktion $\Psi_1 : \mathcal{L}\mathrm{K}i\mathrm{R} \rightarrow \mathcal{L}\mathrm{K}i\mathrm{R}'$ mit $\mathcal{L}\mathrm{K}i\mathrm{R}' \subset \mathcal{L}\mathrm{K}i\mathrm{R}$:

$\Psi_1[\![fun[e_1,\ldots,e_n]]\!]$ $\mapsto$ ap fun to $[\Psi_1[\![e_1]\!],\ldots,\Psi_1[\![e_n]\!]]$
$\Psi_1[\![primfun[e_1,\ldots,e_n]]\!]$ $\mapsto$ ap $primfun$ to $[\Psi_1[\![e_1]\!],\ldots,\Psi_1[\![e_n]\!]]$
$\Psi_1[\![(e_1e_2e_3)]\!]$ $\mapsto$ ap $\Psi_1[\![e_2]\!]$ to $[\Psi_1[\![e_1]\!],\Psi_1[\![e_3]\!]]$
$\Psi_1[\![$if $e_1$then $e_2$else $e_3]\!]$ $\mapsto$ ap then $\Psi_1[\![e_2]\!]$ else $\Psi_1[\![e_3]\!]$ to $[\Psi_1[\![e_1]\!]]$
$\Psi_1[\![$ let $x_1 = e_1,\ldots,x_n = e_n$ in $e]\!]$ $\mapsto$ ap sub $[x_1\ldots x_n]$ in $\Psi_1[\![e]\!]$ to $[\Psi_1[\![e_1]\!],\ldots,\Psi_1[\![e_n]\!]]$
$\Psi_1[\![fun[x_1,\ldots,x_n]] = e$ $\mapsto$ $fun =$ sub $[x_1,\ldots,x_n]$ in $\Psi_1[\![e]\!]$
$\Psi_1[\![[e_1.\ldots.e_n]]\!]$ $\mapsto$ $[\Psi_1[\![e_1]\!].[\ldots[\Psi_1[\![e_{n-1}]\!].\Psi_1[\![e_n]\!]]\ldots]]$
$\Psi_1[\![$ letp $p_1 = e_1,\ldots,p_n = e_n$ in $e]\!]$ $\mapsto$ ap case when $(\Psi_1[\![p_1]\!],\ldots\Psi_1[\![p_n]\!])$ guard true do $\Psi_1[\![e]\!]$ end to $[\Psi_1[\![e_1]\!],\ldots,\Psi_1[\![e_n]\!]]$
$\Psi_1[\![$ when $(p_1,\ldots,p_n)$ do $e]\!]$ $\mapsto$ $\Psi_1[\![$ when $(p_1,\ldots p_n)$ guard true do $e]\!]$
$\Psi_1[\![$ when $(p_1,\ldots,p_n)$ guard e' do $e]\!]^\dagger$ $\mapsto$ case when $(\Psi_1[\![p_1]\!],\ldots\Psi_1[\![p_n]\!])$ guard $\Psi_1[\![e']\!]$ do $\Psi_1[\![e]\!]$ end

† gilt nur, falls die π-Abstraktion nicht Teilausdruck einer Case-Abstraktion ist.

Für alle übrigen $\mathcal{L}$KiR-Ausdrücke gilt: hat der Ausdruck Unterkomponenten, so wird Ψ_1 rekursiv in diese hineingetrieben; hat er keine, so ist Ψ_1 die Identität.

Mit $\mathcal{L}$KiR$' := \Psi_1(\mathcal{L}$KiR) ist die Abbildung offenbar surjektiv, aber nicht injektiv, da z.B.$\Psi_1[\![$ fac $[5]) = \Psi_1[\![$ ap fac to $[5]]\!]$. Daher läßt sich zwar eine Abbildung Ψ_1^{-1} finden, die $(**)$ erfüllt, nicht jedoch $(*)$. Um dieses Problem zu lösen, und dennoch eine undifferenzierte Behandlung der Ausdrücke durch den Ausführungsmechanismus zu ermöglichen, erhalten die Graphknoten in der Implementierung ein Flag, die sog. *edit-info*, anhand der sich der Originalausdruck rekonstruieren läßt.

Für Ausdrücke, die während der Reduktion neu entstehen (durch δ-Reduktion), wird ein Default-Flag generiert, und für solche, die aus anderen Strukturen entstehen (z.B. $\overline{@}$ aus @, $\overline{\texttt{cons}}$ aus `cons`, etc.), wird das Flag dieser Strukturen übernommen.

Auf diese Weise wird die Eigenschaft $(*)$ sichergestellt. Der Einfachheit halber soll im weiteren dieses Kapitels jedoch von den edit-infos abstrahiert werden.

6.2 Schritt 2: Konstruktorerzeugung

Die Abbildung Ψ_2 erzeugt aus $\mathcal{L}$KiR$'$-Ausdrücken eine Graphdarstellung, die isomorph zu dem Syntaxbaum des Programmausdruckes ist. Da diese Darstellung zyklenfrei ist, läßt sie sich durch eine Konstruktorsyntax KS beschreiben und eignet sich als Schnittstelle zu einem syntaxgesteuerten Editor (näheres siehe [Rath91]).

Aufgrund der baumartigen Struktur von KS-Ausdrücken erfolgt eine Graphdarstellung in pre order linearisierter Form. Damit ergibt sich für $\Psi_2 : \mathcal{L}$KiR$' \to KS$ mit $KS := \mathcal{G}^{\mathcal{K}_S}_{\mathcal{K}_S} \subset \mathcal{G}^{\mathcal{K}_{Expr}}_{\mathcal{K}}$ (vergl. Kap 3.1.2 bzw. Anhang C):

$$
\begin{array}{ll}
\Psi_2[\![\langle const\rangle]\!] & \mapsto \langle const\rangle \\
\Psi_2[\![[e_1.e_2]]\!] & \mapsto \texttt{cons}\ \Psi_2[\![e_1]\!]\ \Psi_2[\![e_2]\!] \\
\Psi_2[\![e_1\{\}]\!] & \mapsto \texttt{ccons}\ \Psi_2[\![e_1]\!]\ [] \\
\Psi_2[\![e_1\{e_1,\ldots,e_n\}]\!] & \mapsto \texttt{ccons}\ \Psi_2[\![e_1]\!]\ \Psi_2[\![\Psi_1[\![[e_1.\ldots.e_n.[\ ]]]\!]]\!] \\
\Psi_2[\![< e_1,\ldots,e_n >]\!] & \mapsto \mathrm{tup}^n\ \Psi_2[\![e_1]\!] \ldots \Psi_2[\![e_n]\!] \\
\Psi_2[\![\text{ sub } [x_1,\ldots,x_n] \text{ in } e]\!] & \mapsto \Lambda^n\ \Psi_2[\![e]\!]\ \texttt{name}^{`x_1`} \ldots \texttt{name}^{`x_n`} \\
\Psi_2[\![\text{ ap } e \text{ to } [e_1,\ldots e_n]\!] & \mapsto @^n\ \Psi_2[\![e]\!]\ \Psi_2[\![e_1]\!] \ldots\ \Psi_2[\![e_n]\!] \\
\Psi_2[\![\text{ then } e_1 \text{ else } e_2]\!] & \mapsto \Delta\ \Psi_2[\![e_1]\!]\ \Psi_2[\![e_2]\!] \\
\Psi_2[\![\text{ def } f_1 = e_1,\ldots,f_l = e_l \text{ in } e]\!] & \mapsto \alpha^{k,l}\Psi_2[\![e]\!]\ \Lambda_r\Psi_2[\![e_{rv(1)}]\!] \ldots\ \Lambda_r\Psi_2[\![e_{rv(k)}]\!] \\
 & \qquad \Psi_2[\![e_1]\!] \ldots \Psi_2[\![e_l]\!]\ \texttt{name}^{`f_1`} \ldots \texttt{name}^{`f_l`} \\
 & \quad \text{mit } k = |nrv(\{e_1,\ldots,e_l\})| \text{ und} \\
 & \qquad rv : \{1,\ldots,k\} \to \{1,\ldots,l\} \\
 & \qquad\quad i \mapsto min(\{j | i = |nrv(\{e_1,\ldots e_j\})|\}) \\
\Psi_2[\![\text{ case } p_1,\ldots,p_n \text{ end }]\!] & \mapsto \Pi^n\Box\Psi_2[\![p_1]\!] \ldots \Psi_2[\![p_n]\!]\Omega \\
\Psi_2[\![\text{ case } p_1,\ldots,p_n \text{ otherwise } e \text{ end }]\!] & \mapsto \Pi^n\Box\Psi_2[\![p_1]\!] \ldots \Psi_2[\![p_n]\!]\Psi_2[\![e]\!] \\
\Psi_2[\![\text{ when } (p_1,\ldots p_n) \text{ guard } e_1 \text{ do } e_2]\!] & \mapsto \Lambda_\pi^{|\bigcup_{i=1}^{n} nv(p_i)|} \prec^n \Psi_2[\![p_1]\!] \ldots \Psi_2[\![p_n]\!]\Psi_2[\![e_1]\!]\Psi_2[\![e_2]\!]\Box \\
\Psi_2[\![\text{ when } p \text{ guard } e_1 \text{ do } e_2]\!] & \mapsto \Lambda_\pi^{|nv(p)|}\Psi_2[\![p]\!]\Psi_2[\![e_1]\!]\Psi_2[\![e_2]\!]\Box
\end{array}
$$

Dabei ist nv die Abbildung, die einem Ausdruck aus $\mathcal{L}$KiR$'$ die Menge der in ihm enthaltenen Variablen zuordnet. nrv selektiert aus einer Menge von Ausdrücken die maximale Teilmenge, so daß alle Elemente dieser Teilmenge keine Funktionen sind.$\Box$ ist ein Graphknoten, der keine Unterknoten hat und nur einen später einzufügenden (Schritt 4) Graphen andeutet.

Da die Abbildung eines Ausdruckes lediglich auf der Abbildung der Unterkomponenten beruht, ist es in einem Bottom-Up-Traversiervorgang möglich, einen Ausdruck vollständig zu konvertieren.

Die Umkehrabbildung Ψ_2^{-1} ergibt sich direkt aus der Bijektivität von Ψ_2. Da $KS = \Psi_2(\mathcal{L}\mathrm{K}i\mathrm{R}')$ sind auch für Ψ_2 die Forderungen $(*)$ sowie $(**)$ erfüllt.

6.3 Schritt 3: Variablen und Funktionsumbenennung

Nach Anwendung von Ψ_1 und Ψ_2 auf einen $\mathcal{L}$KiR-Ausdruck liegt dieser bereits in Graphform vor. Als nächstes erfolgt die Ersetzung der angewandten Vorkommen von Variablen und Funktionen durch die für den Ausführungsmechanismus benötigten Darstellungen.

Dazu müssen die zwischen definierendem und angewandtem Vorkommen befindlichen Def-Blöcke sowie λ-Bindungen gezählt und daraus die Indextupel bzw. Graphreferenzen mit Environment-Index abgeleitet werden. Aus der Tatsache, daß die Bindungen somit namenlos werden, ergibt sich, daß *die* Protect-Schlüssel von global freien Variablen entfernt werden müssen, die diese vor der Anwendung von Ψ_3 gegen eine Bindung im Programmausdruck geschützt haben. Sämtliche Modifikationen lassen sich mittels eines Top-Down-Traversiervorganges realisieren, indem rekursiv alle Unterausdrücke mit einer Datenstruktur attributiert werden, welche jeweils die für den Ausdruck bezüglich des Gesamt-Ausdruckes weiter außen gebundenen Variablen enthält. Durch eine geeignete Struktur dieses Attributes (Listen von Listen) lassen sich daraus die benötigten Konstrukte ableiten. Die Beschreibung der Realisierungsfunktion $\Psi_3 : KS \to \mathcal{L}_{Graph}{}'$ mit $\mathcal{L}_{Graph}{}' \subset \mathcal{L}_{Graph}$ erfolgt deshalb mittels der Hilfsfunktion $\Psi_3' : KS \times Bind \to \mathcal{G}_{\mathcal{K}}^{\mathcal{K}_{Expr}}$. Dabei stellt $Bind$ die oben erwähnten Attribute als Listen von Listen dar. Als Einträge in den Listen dienen drei unterschiedliche

Formen von Elementen:

1. (λ, Verweis auf den Namen) für λ-gebundene Variablen;
2. (fun\{Verweis auf das zugehörige $\alpha^{k,l}$-Konstrukt, Verweis auf den zugehörigen Funktionsgraphen, Offset in der Namenliste des $\alpha^{k,l}$-Konstruktes\}, Verweis auf den Namen) für def-gebundene Funktionen;
3. ($recvar$\{Verweis auf das zugehörige $\alpha^{k,l}$-Konstrukt, Offset in der Namenliste des $\alpha^{k,l}$-Konstruktes\}, Verweis auf den Namen) für def-gebundene Variablen.

Mit $\Psi_3(e) := (\Psi_3'[\![e, \langle\langle\rangle\rangle], \langle\rangle{\rightarrow}|)$ gilt:

$$\Psi_3'[\![\texttt{name}^{\text{`}\backslash^k x\text{`}}, bind]\!] \mapsto \text{lookup}(\texttt{name}^{\text{`}x\text{`}},0,0,\text{k},bind)$$

$$\Psi_3'[\![\Lambda^n n_0\, x_1 \ldots x_n, \langle\langle b_1, \ldots, b_r\rangle, l_2, \ldots, l_s\rangle]\!] \mapsto \Lambda^n \Psi_3'[\![n_0, bind']\!] x_1 \ldots x_n$$
$$\text{mit } bind' = \langle\langle(\lambda, x_n), \ldots, (\lambda, x_1), b_1, \ldots, b_r\rangle, \ldots, l_s\rangle$$

$$\Psi_3'[\![\alpha^{k,l} e_0\, v_1 \ldots v_k\, e_1 \ldots e_l\, n_1 \ldots n_l, \langle\langle b_1, \ldots, b_r\rangle, l_2, \ldots, l_s\rangle]\!]$$
$$\mapsto \underbrace{\alpha^{k,l}}_{\alpha_0} \Psi_3'[\![e_0, bind']\!]\, \Psi_3'[\![v_1, bind']\!] \ldots \Psi_3'[\![v_k, bind']\!]\, \Psi_3'[\![e_1, bind']\!] \ldots \Psi_3'[\![e_l, bind']\!]\, n_1 \ldots n_l$$
$$\text{mit } bind' = \langle\langle\rangle, \langle(fl(e_l), n_l), \ldots, (fl(e_1), n_1), b_1, \ldots, b_r\rangle, \ldots, l_s\rangle$$
$$\text{sowie } fl : \{e_1, \ldots, e_l\} \to \{recvar\{\alpha_0, i\}, fun\{\alpha_0, e_i, i\}\}$$

$$\Psi_3'[\![\Pi^n t\, p_1 \ldots p_n\, e_o, bind]\!] \mapsto \Pi^n t \Psi_3'[\![p_1, bind]\!] \ldots \Psi_3'[\![p_n, bind]\!]\, \Psi_3'[\![e_o, bind]\!]$$

$$\Psi_3'[\![\Lambda_\pi{}^n p\, g\, b\, t, \langle\langle b_1, \ldots, b_r\rangle, l_2, \ldots, l_s\rangle]\!] \mapsto \Lambda_\pi{}^n p\, \Psi_3'[\![g, bind']\!]\, \Psi_3'[\![b, bind']\!]\, t$$
$$\text{mit } bind' = \langle\langle(\lambda, v_n), \ldots, (\lambda, v_1), b_1, \ldots, b_r\rangle, \ldots, l_s\rangle \text{ und } v_i \in nv(p)$$

Für alle übrigen Graphknoten gilt: besitzen sie Unterausdrücke, so wird Ψ_3' ohne Modifikation von *bind* rekursiv in diese hereingetrieben; sind sie Blattknoten, so bleiben sie unverändert. Die eigentliche Identifikatorumbenennung nimmt die Funktion "lookup" vor. Es gilt:

$$\text{lookup}(\texttt{name}^{\text{`}x\text{`}}, \text{off}, \text{ind}, \text{prot}, \langle\langle(kind, name\text{`}x\text{`}), b_2, \ldots, b_r\rangle, \ldots, l_s\rangle)$$
$$= \begin{cases} \text{lookup}(\texttt{name}^{\text{`}x\text{`}}, \text{off+1}, \text{ind}, \text{prot-1}, \langle\langle b_2, \ldots, b_r\rangle, \ldots, l_s\rangle) & \text{falls prot} > 0 \\ (\text{ind}, \text{off})^{\lambda} & \text{falls prot} = 0 \wedge kind = \lambda \\ (\text{ind})^{\alpha}_{off_\alpha} e_0\, \alpha_0 & \text{falls prot} = 0 \wedge kind = fun\{\alpha_0, e_0, off_\alpha\} \\ (\text{ind}, \text{off})^{\lambda_r}_{off_\alpha} \alpha_0 & \text{falls prot} = 0 \wedge kind = recvar\{\alpha_0, off_\alpha\} \end{cases}$$

$$\text{lookup}(\texttt{name}^{\text{`}x\text{`}}, \text{off}, \text{ind}, \text{prot}, \langle\langle\rangle\rangle) = \texttt{name}^{\text{`}\backslash^{\text{prot}} x\text{`}}$$

$$\text{lookup}(\texttt{name}^{\text{`}x\text{`}}, \text{off}, \text{ind}, \text{prot}, \langle\langle\rangle, l_2, \ldots, l_s\rangle) = \text{lookup}(\texttt{name}^{\text{`}x\text{`}}, \text{off}, \text{ind+1}, \text{prot}, \langle l_2, \ldots, l_s\rangle)$$

$$\text{lookup}(\texttt{name}^{\text{`}x\text{`}}, \text{off}, \text{ind}, \text{prot}, \langle\langle b_1, \ldots, b_r\rangle, \ldots, l_s\rangle)$$
$$= \text{lookup}(\texttt{name}^{\text{`}x\text{`}}, \text{off+1}, \text{ind}, \text{prot}, \langle\langle b_2, \ldots, b_r\rangle, \ldots, l_s\rangle)$$

Da die von Ψ_3 verursachten Graphmodifikationen ausschließlich Variablendarstellungen betreffen und die Bindungsstruktur unverändert bleibt, sind die Modifikationen reversibel und es gilt $(*)$. Um die Gültigkeit von $(**)$ überprüfen zu können, bedarf es zunächst einer genaueren Definition von $\mathcal{L}_{Graph}{}'$.

Sei $\mathcal{L}_{Graph}{}'$ die Menge aller der Graphen in $\mathcal{L}_{Graph}$, die kein $\overline{\Pi}$-Konstrukt enthalten. Das bedeutet, daß environmentbehaftete Ausdrücke, wie sie durch die naive Substitution des Ausführungsmechanismus entstehen, in $\mathcal{L}_{Graph}{}'$ enthalten sind und Ψ_3^{-1} somit die Vervollständigung der β-Reduktion bereitstellt. Diese Vervollständigung umfaßt das Durchführen während der Processing Phase zurückgestellter:

1. Substitutionen,
2. partieller Anwendungen sowie

3. ggf. erforderlicher Protect-Schlüssel-Ergänzungen bei global freien Variablen.

Die Realisierung erfolgt mittels eines Top-Down-Traversiervorganges, bei dem die Substitute für die Variablen in Form von Environments rekursiv an die Unterausdrücke übergeben werden. Bei partiellen Anwendungen werden die Namen der verbleiben λ-Abstraktionen mit einer speziellen Markierung ($\texttt{name}_\lambda^{\text{`}x\text{`}}$) in die Umgebungen eingetragen. Die Anzahl der bei angewandten Vorkommen solcher Variablen voranzustellenden Protect-Schlüssel ergibt sich direkt aus der Anzahl der gleichnamigen Abstraktionen, in deren Rümpfe ein solches Environment hereingereicht wird. Da durch das Hereinreichen eines Environments in eine Abstraktion jedesmal das Environment um einen derartig markierten Eintrag erweitert wird, läßt sich aus der Anzahl der gleichnamigen Eintragungen auf die Anzahl der benötigten Protect-Schlüssel schließen. Auf gleiche Weise kann die Anzahl von ggf. erforderlichen Protect-Schlüsseln bei den durch $\texttt{name}^{\text{`}x\text{`}}$ dargestellten global freien Variablen abgeleitet werden.

Die Vervollständigung der Substitution erfordert bei angewandten Vorkommen von rekursiven Ausdrücken, die mittels des Def-Konstruktes definiert sind, daß eine instanziierte Kopie des Def-Blockes erzeugt wird. Um das Erzeugen weiterer Kopien bei angewandten Vorkommen innerhalb eines Def-Blockes vermeiden zu können, bedarf es eines Kontextes für Ψ_3^{-1}, aus dem sich ableiten läßt, welche angewandten Vorkommen solcher Ausdrücke keine Kopie des Def-Blockes benötigen. Dies wird wie bei Ψ_3 mittels eines Attributes in Form von Listen von Listen von Verweisen auf Namen und einer Hilfsfunktion $\Psi_3'' : \mathcal{L}_{Graph}{}' \times Prot \to KS$ realisiert. Mit $\Psi_3^{-1}(n,e) := \Psi_3''[\![n, e, \langle\langle\rangle\rangle]\!]$ gilt:

$$\Psi_3''[\![\overline{\texttt{cons}}\,(n_l,e_l)\,(n_r,e_r), env, prot]\!] \quad \mapsto \texttt{cons}\;\Psi_3''[\![n_l,e_l,\langle\langle\rangle\rangle]\!]\;\Psi_3''[\![n_r,e_r,\langle\langle\rangle\rangle]\!]$$
$$\Psi_3''[\![\overline{\texttt{ccons}}\,(n_l,e_l)\,(n_r,e_r), env, prot]\!] \quad \mapsto \texttt{ccons}\;\Psi_3''[\![n_l,e_l,\langle\langle\rangle\rangle]\!]\;\Psi_3''[\![n_r,e_r,\langle\langle\rangle\rangle]\!]$$
$$\Psi_3''[\![\overline{\text{tup}}^k\,(n_1,e_1)\ldots(n_k,e_k), env, prot]\!] \mapsto \text{tup}^k\;\Psi_3''[\![n_1,e_1,\langle\langle\rangle\rangle]\!]\ldots\Psi_3''[\![n_k,e_k,\langle\langle\rangle\rangle]\!]$$
$$\Psi_3''[\![prf^{\,k,l}(n_1,e_1)\ldots(n_l,e_l), env, prot]\!] \mapsto @^{\,l}\,prf^{(k+l),0}\;\Psi_3''[\![n_1,e_1,\langle\langle\rangle\rangle]\!]\ldots\Psi_3''[\![n_l,e_l,\langle\langle\rangle\rangle]\!]$$
$$\Psi_3''[\![\texttt{name}^{\text{`}x\text{`}}, e_1 \to \ldots \to e_n, prot]\!] \quad \mapsto \texttt{name}^{\text{`}\backslash^k x}$$

mit $\texttt{name}_\lambda^{\text{`}x\text{`}}$ ist in $\bigcup_{i=1}^{n} e_i$ genau k-mal enthalten.

$$\Psi_3''[\![\overline{@}^{\,k}\,n_0\,(n_1,e_1)\ldots(n_k,e_k), env, prot]\!]$$
$$\mapsto @^{\,k}\;\Psi_3''[\![n_0, env, prot]\!]\;\Psi_3''[\![n_1,e_1,\langle\langle\rangle\rangle]\!]\ldots\Psi_3''[\![n_k,e_k,\langle\langle\rangle\rangle]\!]$$
$$\Psi_3''[\![\Lambda^k\,n_0\,n_1\ldots n_k, \langle s_1,\ldots,s_l\rangle \to env, \langle\langle p_1,\ldots,p_r\rangle, l_2,\ldots,l_s\rangle]\!]$$

mit $n_i = \texttt{name}^{\text{`}x_i\text{`}}$ für $i \in \{1,\ldots,k\}$

$$\mapsto \Lambda^k \Psi_3''[\![n_0, env', prot']\!]\;n_1\ldots n_k$$

mit $env' = \langle\langle n'_k,\ldots n'_1, s_1,\ldots,s_l\rangle\rangle \to env$ und $n'_i = \texttt{name}_\lambda^{\text{`}x_i\text{`}}$ für $i \in \{1,\ldots,k\}$

sowie $prot' = \langle\langle n_k,\ldots n_1, p_1,\ldots,p_r\rangle, l_2,\ldots,l_s\rangle$

$$\Psi_3''[\![\overline{\Lambda}^k\,n_0\,n_{sub}, \langle s_1,\ldots,s_l\rangle \to env, \langle\langle p_1,\ldots,p_r\rangle, l_2,\ldots,l_s\rangle]\!]$$

mit $n_{sub} = \Lambda^n n_0\,n_1\ldots n_n$ und $n_i = \texttt{name}^{\text{`}x_i\text{`}}$ für $i \in \{1,\ldots,n\}$

$$\mapsto \Lambda^k \Psi_3''[\![n_0, env', prot']\!]\;n_{(n-k)}\ldots n_n$$

mit $env' = \langle\langle n'_n,\ldots n'_{(n-k)}, s_1,\ldots,s_l\rangle\rangle \to env'$,

und $n'_i = \texttt{name}_\lambda^{\text{`}x_i\text{`}}$ für $i \in \{(n-k),\ldots,n\}$

sowie $prot' = \langle\langle n_n,\ldots n_{(n-k)}, p_1,\ldots,p_r\rangle, l_2,\ldots,l_s\rangle$

$$\Psi_3''[\![(i,j)^\lambda, e_0 \to \ldots \to e_i = \langle s_1,\ldots,s_n\rangle \to e_{(i+1)}, prot]\!]$$
$$\mapsto \begin{cases} \texttt{name}^{\text{`}\backslash^k x\text{`}} & \text{falls } (s_j = \texttt{name}^{\text{`}x\text{`}} \;\vee\; s_j = \texttt{name}_\lambda^{\text{`}x\text{`}}) \\ & \text{und } \texttt{name}_\lambda^{\text{`}x\text{`}} \text{ ist in } \bigcup_{k=1}^{(i-1)} e_k \cup \{s_1,\ldots,s_{(j-1)}\} \text{ genau k-mal enthalten.} \\ \Psi_3''[\![n,e,\langle\langle\rangle\rangle]\!] & \text{falls } s_j = (n,e) \end{cases}$$

$\Psi_3''[\![\alpha^{k,l}\, n_0\, v_1 \ldots v_k\, b_1 \ldots b_l\, n_1 \ldots n_l, \langle s_1, \ldots, s_m\rangle \to env, \langle\langle p_1, \ldots p_r\rangle, l_2, \ldots, l_s\rangle]\!]$
mit $n_i = \texttt{name}^{\text{`}x_i\text{`}}$ für $i \in \{1, \ldots, n\}$
$\mapsto \alpha^{k,l}\, \Psi_3''[\![n_0, env', prot']\!]\, v_1 \ldots v_k\, \Psi_3''[\![b_1, env', prot']\!] \ldots \Psi_3''[\![b_l, env', prot']\!]\, n_1 \ldots n_l$
mit $env' = \langle\rangle \to \langle \texttt{name}^{\text{`}x_{rv(k)}\text{`}}, \ldots, \texttt{name}^{\text{`}x_{rv(1)}\text{`}}, s_1, \ldots, s_m\rangle \to env$
und $prot' = \langle\langle\rangle, \langle n_l, \ldots, n_1, p_1, \ldots, p_r\rangle, l_2, \ldots, l_s\rangle$
$\Psi_3''[\![(i)^{\alpha}_{off}\alpha_0\, e_0, env, \langle\langle p_{1,1}, \ldots p_{1,r_1}\rangle, \ldots \langle p_{s,1}, \ldots p_{s,r_s}\rangle\rangle]\!]$
mit $\alpha_0 = \alpha^{k,l}\, n_0\, v_1 \ldots v_k\, b_1 \ldots b_l\, n_1 \ldots n_l$
und $n_i = \texttt{name}^{\text{`}x_i\text{`}}$ für $i \in \{1, \ldots, n\}$

$$\mapsto \begin{cases} \texttt{name}^{\text{`}\backslash^k x_{off}\text{`}} \text{ falls } i \le s \\ \qquad \text{und } \texttt{name}_{\lambda}^{\text{`}x_{off}\text{`}} \text{ ist genau k-mal in } \bigcup_{j=1}^{(i-1)} \{p_{j,1}, \ldots, p_{j,r_j}\} \cup \{p_{i,1}, \ldots, p_{i,off}\} \\ \qquad \text{enthalten.} \\ \alpha^{k,l}\, \Psi_3''[\![n_0, env', prot']\!]\, v_1, \ldots, v_k\, \Psi_3''[\![b_1, env', prot']\!] \ldots \Psi_3''[\![b_l, env', prot']\!]\, n_1 \ldots n_l \\ \qquad \text{falls } i > s \end{cases}$$

$\Psi_3''[\![(i,j)^{\lambda_r}_{off}\alpha_0, env, \langle\langle p_{1,1}, \ldots p_{1,r_1}\rangle, \ldots \langle p_{s,1}, \ldots p_{s,r_s}\rangle\rangle]\!]$
mit $\alpha_0 = \alpha^{k,l}\, n_0\, v_1 \ldots v_k\, b_1 \ldots b_l\, n_1 \ldots n_l$
und $n_i = \texttt{name}^{\text{`}x_i\text{`}}$ für $i \in \{1, \ldots, n\}$

$$\mapsto \begin{cases} \texttt{name}^{\text{`}\backslash^k x_{off}\text{`}} \text{ falls } i \le s \\ \qquad \text{und } \texttt{name}_{\lambda}^{\text{`}x_{off}\text{`}} \text{ ist genau k-mal in } \bigcup_{(j=1)}^{(i-1)} \{p_{j,1}, \ldots, p_{j,r_j}\} \cup \{p_{i,1}, \ldots, p_{i,off}\} \\ \qquad \text{enthalten.} \\ \alpha^{k,l}\, \Psi_3''[\![n_0, env', prot']\!]\, v_1, \ldots, v_k\, \Psi_3''[\![b_1, env', prot']\!] \ldots \Psi_3''[\![b_l, env', prot']\!]\, n_1 \ldots n_l \\ \qquad \text{falls } i > s \end{cases}$$

$\Psi_3''[\![\Pi^n\, t\, p_1 \ldots p_n$

$\mapsto \Pi^n \Box\, \Psi_3''[\![p_1, env, prot]\!] \ldots \Psi_3''[\![p_n, env, prot]\!]\, n_o$
$\Psi_3''[\![\Lambda_\pi{}^n\, p\, g\, b\, t, \langle s_1, \ldots, s_l\rangle \to env, \langle\langle p_1 \ldots, p_r\rangle, l_2, \ldots, l_s\rangle]\!]$
mit $v_i = \texttt{name}^{\text{`}x_i\text{`}}$ und $x_i \in nv(p)$ für $i \in \{1, \ldots, n\}$.
$\mapsto \Lambda_\pi{}^n\, p\, \Psi_3''[\![g, env', prot']\!]\, \Psi_3''[\![b, env', prot']\!]\, \Box$
mit $env' = \langle v_n', \ldots v_1', s_1, \ldots, s_l\rangle \to env$ und $v_i' = \texttt{name}_{\lambda}^{\text{`}x_i\text{`}}$ für $i \in \{1, \ldots, n\}$
sowie $prot' = \langle\langle v_n, \ldots v_1, p_1, \ldots, p_r\rangle, l_2, \ldots, l_s\rangle$

Für alle übrigen Graphknoten gilt: besitzen sie Unterausdrücke, so wird Ψ_3'' ohne Modifikation von *env* bzw. *prot* rekursiv in diese hereingetrieben; sind sie Blattknoten, so bleiben sie unverändert.

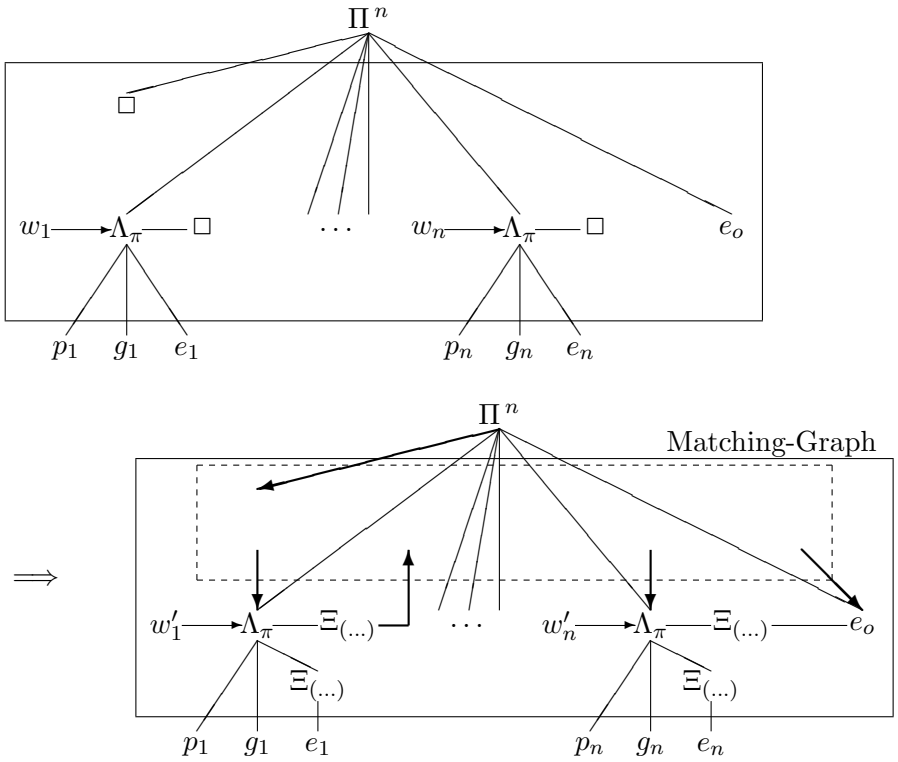
Da alle Graphknoten aus $\Psi_3''(\mathcal{L}_{Graph}{}')$ in $\mathcal{K}_S$ liegen, gilt $\Psi_3^{-1}(\mathcal{L}_{Graph}{}') \subset \mathcal{G}_{\mathcal{K}_S}^{\mathcal{K}_S} = KS$. Somit gilt auch für Ψ_3/Ψ_3^{-1} die Forderung $(**)$.

6.4 Schritt 4: Erzeugung des Patternmatchcodes

Genauso wie die Variablen/Funktionen aus Effizienzgründen intern eine besondere Darstellung erfordern, wird für die Ausführung des Patternmatches ebenfalls eine gesonderte Darstellung - im weiteren mit *Matching-Graph* bezeichnet - verwendet. Die Konstruktion des Matching-Graphen erfolgt mittels der in Kapitel 5.3 abgeleiteten Konstrukte. Nach sener Generierung wird der Graph in das gesamte Programm folgendermaßen integriert: der Wurzelknoten des Matching-Graphen wird zum linken Unterknoten des Π-Konstruktes (interne Darstellung der Case-Abstraktion) und seine "Blätter", die π-Abstraktionen (Λ_π in der internen Darstellung), ersetzen die ursprünglichen Λ_π-Konstrukte des Programm-Graphen. Abbildung 6.2 veranschaulicht diesen Vorgang. Dabei deutet der gestrichelte Kasten den

pattern-abhängigen Teil des Matching-Graphen an. Die fett gezeichneten Pfeile stehen für Verweise in diesen Teilgraphen hinein bzw. aus ihm heraus.

Abbildung 6.2: Die Einfügung des Matching-Graphen in den Gesamt-Graphen



Eine formale Beschreibung erfolgt mittels der Funktion $\Psi_4 : \mathcal{L}_{Graph}{}' \to \mathcal{L}_{Graph}$. Die Konstruktion des Matching-Graphen aus den Pattern der π-Abstraktionen wird durch eine Hilfsfunktion Ψ_{MG} beschrieben. Für eine korrekte Handhabung der Stacks S, Pm und Bi während des Matching-Vorganges bedarf es Ξ-Konstrukten, um Stackmodifikationen vornehmen zu können (vergl. Kapitel 5.3). Eine korrekte Erzeugung dieser erfordert Kenntnis über die Struktur des bis dahin erzeugten Matching-Graphen. Die Verwendung von zwei Attributen *Alt* und *Clear* bei der Generierung des Matching-Graphen durch Ψ_{MG} ermöglicht einen Zugriff auf die benötigten Informationen ohne eine Inspektion von irgendwelchen Teilen des Matching-Graphen. Damit ergibt sich als Signatur für

$$\Psi_{MG} : (Patlist \times Bodyref)^n \times Alt \times Clear \to \mathcal{G}^{\mathcal{K}_P}_{\mathcal{K}_P \cup \mathcal{K}_{P'}}.$$

und für Ψ_4 läßt sich definieren:

$$\Psi_4[\![\Pi^n \,\square\, w_1 \ldots w_n\, e_o]\!] \quad \text{mit } w_i = {\Lambda_\pi}^{n_i} p_i\, g_i\, e_i\, \square \text{ für } i \in \{1, \ldots, n\}$$
$$\mapsto \begin{cases} \Pi^n\, \Psi_{MG}[\![\langle (p_1, w_1), \ldots, (p_n, w_n)\rangle, \Xi_{(0,1,0,1)}\, e_o, \Xi_{(0,1,0,1)}]\!]\, w'_1 \ldots w'_n\, e_o & \text{falls } e_o = \Omega \\ \Pi^n\, \Psi_{MG}[\![\langle (p_1, w_1), \ldots, (p_n, w_n)\rangle, \Xi_{(1,1,0,1)}\, e_o, \Xi_{(1,1,0,1)}]\!]\, w'_1 \ldots w'_n\, e_o & \text{sonst} \end{cases}$$

Für alle übrigen Graphknoten wird Ψ_4 in die Unterausdrücke rekursiv hereingetragen bzw. wirkt wie die Identität. Offengeblieben ist somit lediglich die Frage, welcher Patternmatch-Graph einer Case-Abstraktion konkret zugeordnet werden soll, d.h. die Spezifikation von Ψ_{MG}.

Dazu betrachten wir zunächst eine Untermenge der in $\mathcal{L}$K*i*R spezifizierbaren Case-Abstraktionen. Sei Pm_1 die Menge der Case-Abstraktionen, die sich durch

$$
\begin{array}{lll}
\langle case\_expr\rangle & \Longrightarrow & \text{case } \langle v\_pat\_expr\rangle\Big(,\langle v\_pat\_expr\rangle\Big)^* \text{ otherwise } \langle expr\rangle \text{ end} \quad | \\
& & \text{case } \langle a\_pat\_expr\rangle\Big(,\langle a\_pat\_expr\rangle\Big)^* \text{ otherwise } \langle expr\rangle \text{ end} \quad | \\
& & \text{case } \langle k\_pat\_expr\rangle\Big(,\langle k\_pat\_expr\rangle\Big)^* \text{ otherwise } \langle expr\rangle \text{ end} \\
\langle v\_pat\_expr\rangle & \Longrightarrow & \text{when } \langle var\rangle \text{ guard } \langle expr\rangle \text{ do } \langle expr\rangle \\
\langle a\_pat\_expr\rangle & \Longrightarrow & \text{when as } \langle var\rangle\ \langle var\rangle \text{ guard } \langle expr\rangle \text{ do } \langle expr\rangle \\
\langle k\_pat\_expr\rangle & \Longrightarrow & \text{when } \langle k\_pat\rangle \text{ guard } \langle expr\rangle \text{ do } \langle expr\rangle \\
\langle k\_pat\rangle & \Longrightarrow & [\langle var\rangle.\langle var\rangle] \quad | \quad \langle var\rangle\{\langle var\rangle\} \quad | \quad [] \quad | \quad \text{true} \quad | \quad \text{false} \quad | \\
& & \langle integer\_number\rangle \quad | \quad \langle string\rangle \quad | \quad <\langle var\rangle\Big(,\langle var\rangle\Big)^*> \quad | \\
& & <\Big(\langle var\rangle,\Big)^* \ldots \Big(,\langle var\rangle\Big)^*>
\end{array}
$$

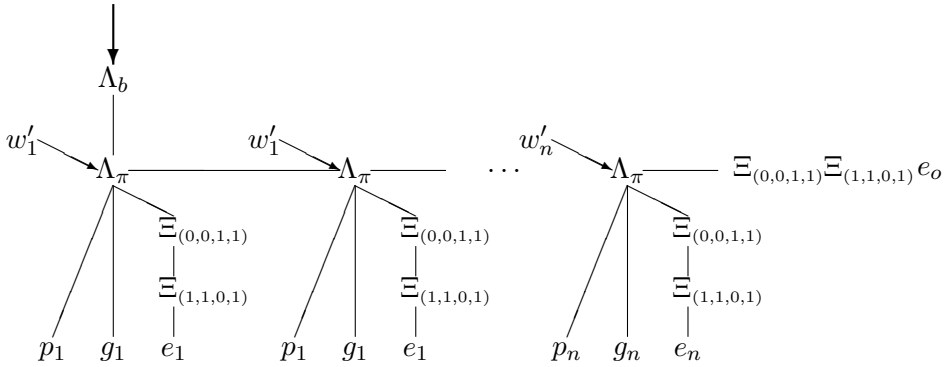
ableiten läßt. Für derartige Case-Abstraktionen läßt sich auf einfache Weise ein Matching-Graph konstruieren. Ursache dafür ist, daß die Pattern der einzelnen π-Abstraktionen entweder vollständig überlappend oder aber vollkommen disjunkt sind. Dadurch können während des Match-Vorganges keine Choice-Punkte (vergl. Kapitel 5.3) entstehen und ein Backtracking wird nicht erforderlich.

Je nach verwendeter Ableitungsregel sind drei Fälle ("v","a" und "k") - im weiteren *Kategorien* genannt - zu unterscheiden. Sei K eine Abbildung, die jeder Case-Abstraktion aus Pm_1 seine Kategorie zuordnet. Dann gilt für einen Ausdruck $expr$ der Form $\Pi\,\Box w_1 \ldots w_n e_o$ mit $w_i = {\Lambda_\pi}^{k_i} p_i g_i e_i \Box$ für $i \in \{1,\ldots,n\}$:

Fall I: $K(expr) =$ "v"; d.h. es handelt sich ausschließlich um Variablen als Pattern.

Dann gilt:
$\Psi_{MG}(\langle(p_1,w_1),\ldots,(p_n,w_n)\rangle, alt, clear)$
$\mapsto\ \Lambda_b\,\Psi_{MG}[\![\langle(\Box,w_1)\ldots,(\Box,w_n)\rangle, \Xi_{(0,0,1,1)}\,alt, \Xi_{(0,0,1,1)}\,clear]\!]$.

Mit der allgemeinen Regel
$\Psi_{MG}[\![\langle(\Box,w_1)\ldots,(\Box,w_n)\rangle, alt, clear]\!] \mapsto\ \Lambda_\pi p_1\, g_1\ clear\, e_1\ \ldots \Lambda_\pi p_n\, g_n\ clear\, e_n\ alt$
ergibt sich als Matching-Graph:



Fall II: $K(expr) =$ "a"; d.h. es handelt sich ausschließlich um "as"-Pattern.

Dann gilt:
$\Psi_{MG}(\langle(p_1,w_1),\ldots,(p_n,w_n)\rangle, alt, clear)$ mit $p_i = \prec_{as} a_i\, v_i$ für $i \in \{1,\ldots,n\}$
$\mapsto\ \Lambda_{as}\,\Psi_{MG}(\langle(v_1,w_1),\ldots,(v_n,w_n)\rangle, \Xi_{(0,0,0,1)}\,alt, \Xi_{(0,0,0,1)}\,clear)$.

Durch die in Fall I spezifizierten Regeln ergibt sich ein vollständiger Matching-Graph.

Fall III: $K(expr) =$ "k"; d.h. es handelt sich um Konstanten oder Datenstrukturen. In diesem Fall können die einzelnen π-Abstraktionen nach verschiedenen Typen bzw. Werten geordnet werden, ohne daß sich die Bedeutung des Ausdruckes ändert, da ein Argument auf maximal einen Wert/Typ paßt. Es kann also selektiv vorgegangen werden, ohne daß eine Choice-Punkt entsteht. Sind mehrere syntaktisch gleiche Pattern vorhanden, so darf ihre Reihenfolge nicht vertauscht werden. Damit ergibt sich:

$$\Psi_{MG}(\underbrace{\langle(p_1,w_1),\ldots,(p_n,w_n)\rangle}_{patlist}, alt, clear)$$
$$\mapsto\ \Sigma\, MG_{tup}\, MG_{cons}\, MG_{ccons},\ MG_{[]}\, MG_{true}\, MG_{false}\, MG_{int}\, MG_{str}\, alt$$

mit
$$\begin{aligned}
MG_{tup} &:= \Psi_{tup}[\![sel_{tup}(patlist), alt, clear]\!]\\
MG_{cons} &:= \Psi_{cons}[\![sel_{cons}(patlist), \Xi_{(2,0,1,0)}\, alt, \Xi_{(2,0,1,0)}\, clear]\!]\\
MG_{ccons} &:= \Psi_{cons}[\![sel_{ccons}(patlist), \Xi_{(2,0,1,0)}\, alt, \Xi_{(2,0,1,0)}\, clear]\!]\\
MG_{[]} &:= \Psi_{const}[\![sel_{[]}(patlist), \Xi_{(0,0,1,0)}\, alt, \Xi_{(0,0,1,0)}\, clear]\!]\\
MG_{true} &:= \Psi_{const}[\![sel_{true}(patlist), \Xi_{(0,0,1,0)}\, alt, \Xi_{(0,0,1,0)}\, clear]\!]\\
MG_{false} &:= \Psi_{const}[\![sel_{false}(patlist), \Xi_{(0,0,1,0)}\, alt, \Xi_{(0,0,1,0)}\, clear]\!]\\
MG_{int} &:= \sigma\langle val_1,\ldots,val_k\rangle\ \Psi_{const}[\![sel_{intval_1}(patlist), \Xi_{(0,0,1,0)}\, alt, \Xi_{(0,0,1,0)}\, clear]\!]\\
&\quad \ldots\Psi_{const}[\![sel_{intval_k}(patlist), \Xi_{(0,0,1,0)}\, alt, \Xi_{(0,0,1,0)}\, clear]\!]\ alt\\
MG_{str} &:= \sigma\langle val_1,\ldots,val_k\rangle\ \Psi_{const}[\![sel_{strval_1}(patlist), \Xi_{(0,0,1,0)}\, alt, \Xi_{(0,0,1,0)}\, clear]\!]\\
&\quad \ldots\Psi_{const}[\![sel_{strval_k}(patlist), \Xi_{(0,0,1,0)}\, alt, \Xi_{(0,0,1,0)}\, clear]\!]\ alt
\end{aligned}$$

sowie $\Psi_{tup}[\![\langle(p_1,w_1),\ldots,(p_n,w_n)\rangle, alt, clear]\!]$

$$\mapsto \begin{cases}
{\sigma_{<>}}^{k}\,\Psi_{MG}[\![\langle(<v_1,\ldots,v_k>,w_1)\rangle, \Xi_{(k,0,1,0)}MG_{alt}, \Xi_{(k,0,1,0)}clear]\!]\, MG_{alt}\\
\quad \text{mit } MG_{alt} := \Psi_{tup}[\![\langle(p_2,w_2),\ldots,(p_n,w_n)\rangle, alt, clear]\!]\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{falls } p_1 = tup^k\, v_1 \ldots v_k\\
{\sigma_{<.>}}^{k,l}\,\Psi_{MG}[\![\langle(<v_1,\ldots,v_{(k+1)}>,w_1)\rangle, \Xi_{(k+1,0,1,0)}MG_{alt}, \Xi_{(k+1,0,1,0)}clear]\!]\\
MG_{alt} \text{ mit } MG_{alt} := \Psi_{tup}[\![\langle(p_2,w_2),\ldots,(p_n,w_n)\rangle, alt, clear]\!]\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{falls } p_1 = tup^{(k+1)}\, v_1 \ldots v_l\, name^{\text{“}\cdots\text{“}}\, v_{(l+2)} \ldots v_{(k+1)}
\end{cases}$$

$$\begin{aligned}
&\Psi_{cons}[\![\langle(p_1,w_1),\ldots,(p_k,w_k)\rangle, alt, clear]\!] \text{ mit } p_i = \texttt{cons}\, pl_i\, pr_i \text{ für } i \in \{1,\ldots,k\}\\
&\mapsto\ \Psi_{MG}[\![\langle(<pl_1,pr_1>,w_1),\ldots,(<pl_k,pr_k>,w_k)\rangle, alt, clear]\!]\\
&\Psi_{const}[\![\langle(p_1,w_1),\ldots,(p_k,w_k)\rangle, alt, clear]\!]\\
&\mapsto\ \Psi_{MG}[\![\langle(\square,w_1),\ldots,(\square,w_k)\rangle, alt, clear]\!]
\end{aligned}$$

Die Abbildungen sel_{kind} selektieren aus $patlist$ die Tupel heraus, deren Pattern dem Typ/Wert $kind$ entsprechen; dabei bleibt die Reihenfolge dieser erhalten. Der Übersicht halber fehlen in der obigen Darstellung einige Fälle. So führt für jedes $kind \in \{tup, cons, ccons, ...\}$ $sel_{kind}(patlist) = \langle\rangle$ dazu, daß der entsprechende Unterbaum nicht generiert und dies im Σ-Konstrukt vermerkt wird. Im Falle einer Anwendung auf ein Argument dieses Types erfolgt eine Selektion des Default-Unterbaumes (letzte Komponente des Σ-Konstruktes).

Somit ist Ψ_{MG} für Ausdrücke aus Pm_1 vollständig beschrieben.

Wie bereits in Kapitel 5.3 erläutert, bewirkt das $\Xi_{(s,p,m,b)}$-Konstrukt ein Entfernen von s Elementen vom S-Stack, p Elementen vom Pm-Stack, b Elementen vom Bi-Stack sowie anschließend ein Traversieren von m Elementen vom Pm- auf den S-Stack. Daher liegt es nahe, aufeinander folgende Ξ-Konstrukte so weit wie möglich zusammenzufassen. Es gilt:

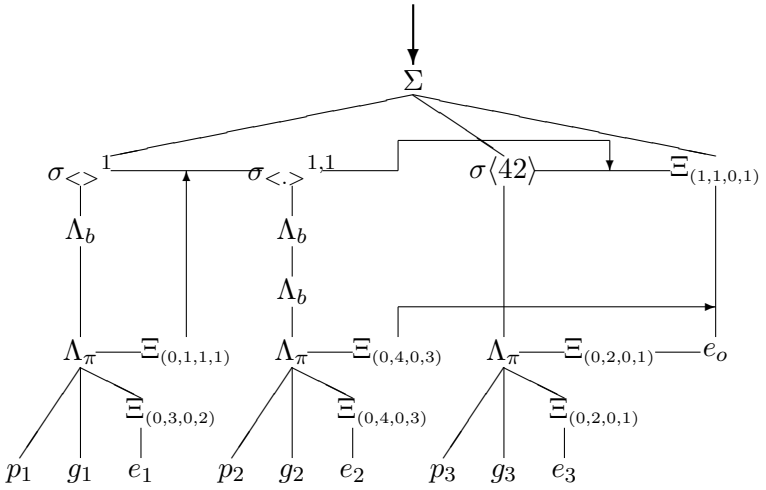
$$\Xi_{(s,p,m,b)}\,\Xi_{(s2,p2,m2,b2)}$$
$$\Rightarrow \begin{cases} \Xi_{(s+s2-m,\ p+p2+m,\ m2,\ b+b2)} & \text{falls } s2 \geq m \\ \Xi_{(s,\ p,\ m+m2,\ b+b2)} & \text{falls } s2 < m \ \wedge\ s2 = 0 \ \wedge\ p2 = 0 \\ \Xi_{(s,\ p,\ m-s2,\ b+b2)}\,\Xi_{(0,\ s2+p2,\ m2,\ 0)} & \text{sonst} \end{cases}.$$

Zur Verdeutlichung der bisher beschriebenen Funktionalität von Ψ_{MG} ein Beispiel:

Beispiel 6.1: Eine einfache Case-Abstraktion aus Pm_1:

```
case
  when <A> guard true do 1,
  when <A,...> guard true do 2,
  when 42 guard true do 3
  otherwise 4
end
```

Als Matching-Graph entsteht dazu:



Als nächstes soll nun die Menge der Case-Abstraktionen dergestalt erweitert werden, daß Konstanten, Datenstrukturen, Variablen und "as"-Pattern gemischt auftreten und somit Choice-Punkte entstehen können. Für Ausdrücke aus Pm_2 mit $Pm_1 \subset Pm_2 \subset \mathcal{L}Ki$R gilt:

$$\langle case\_expr\rangle \Longrightarrow \text{case } \langle pat\_expr\rangle \Big(, \langle pat\_expr\rangle\Big)^* \text{ otherwise } \langle expr\rangle \text{ end}$$
$$\langle pat\_expr\rangle \Longrightarrow \langle v\_pat\_expr\rangle \quad | \quad \langle a\_pat\_expr\rangle \quad | \quad \langle k\_pat\_expr\rangle$$

Die Generierung des Matching-Graphen für Ausdrücke aus Pm_2 läßt sich auf die Generierung des Matching-Graphen für Pm_1-Ausdrücke zurückführen. Dazu werden die π-Abstraktionen in möglichst große, aufeinander folgende *Gruppen* eingeteilt, so daß innnerhalb dieser Gruppen nur Pattern *einer* Kategorie sind. Die zugrunde liegende Idee ist es, eine Case-Abstraktion aus Pm_2 der allgemeinen Form:

case
when p_1 guard g_1 do e_1,
$\vdots$
when p_k guard g_k do e_k,
when $p_{(k+1)}$ guard $g_{(k+1)}$ do $e_{(k+1)}$,
$\vdots$
when p_n guard g_n do e_n,
otherwise o
end

mit $K(p_i) = K(p_1)$ für $i \in \{1, \ldots k\}$

durch sukzessive Transformationen in verschachtelte Case-Abstraktionen aus Pm_1 zu überführen:

case
when p_1 guard g_1 do e_1,
$\vdots$
when p_k guard g_k do e_k,
otherwise case
when $p_{(k+1)}$ guard $g_{(k+1)}$ do $e_{(k+1)}$,
$\vdots$
when p_n guard g_n do e_n
otherwise e_0
end
end

Durch eine geeignete Schachtelung von Ψ_{MG} kann diese Umformung so erfolgen, daß keine vollständigen Case-Abstraktionen gebildet werden müssen:

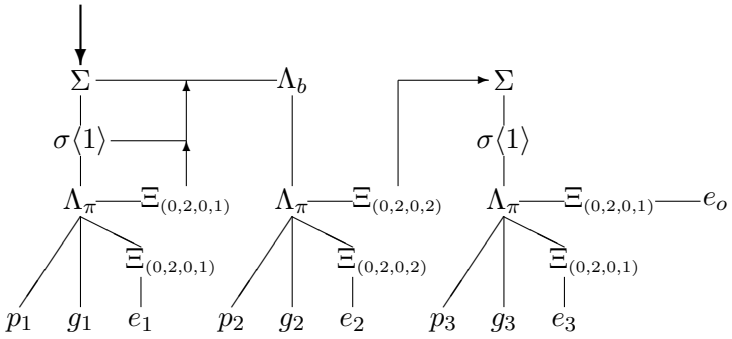
$\Psi_{MG}[\![\langle (p_1, w_1), \ldots, (p_k, w_k)\rangle, alt, clear]\!]$
$\mapsto \Psi_{MG}[\![\langle (p_1, w_1), \ldots, (p_k, w_k), \Psi_{MG}[\![\langle (p_{(k+1)}, w_{(k+1)}), \ldots, (p_n, w_n)\rangle, alt, clear]\!], clear]\!]$

Diese Vorgehensweise ist zwar pragmatisch, kann jedoch zu redundanten Tests führen. Sie können nur dann entstehen, wenn eine Case-Abstraktion folgender Struktur vorliegt: es existieren mindestens drei Gruppen von Pattern; die erste und dritte dieser Gruppen besteht aus Pattern der Kategorie “k“, die mittlere Gruppe enthält Variablen oder “as“-Pattern. Die Ursache der Redundanzen liegen darin, daß beim Testen der Pattern der ersten Gruppe gewonnenen Informationen bei den Patterntests der dritten Gruppe nicht genutzt werden. Betrachten wir dazu als Beispiel:

Beispiel 6.2:

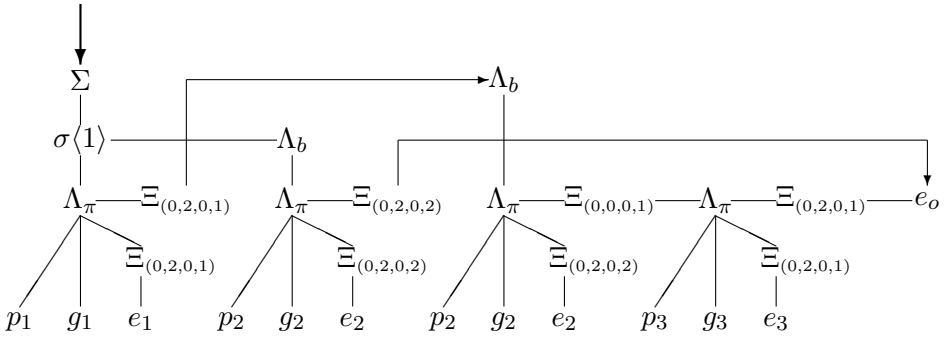
case
when 1 guard g_1 do e_1,
when A guard g_2 do e_2,
when 1 guard g_3 do e_3
otherwise e_o
end

Daraus ergibt sich als Matching-Graph:



Wird dieser Ausdruck auf ein Argument angewendet, das sich nicht zu eins evaluieren läßt und berechnet sich dann g_2 zu false, so wird das Argument überflüssigerweise nochmals auf eins getestet. Selbst für den Fall, das das Argument zu eins reduziert wird, kommt es zu zwei Tests auf den Wert eins, falls g_1 und g_2 beide false ergeben. Sollen die redundanten Tests auf eins vermieden werden, so ist je nach Ergebnis des "ersten" Tests auf eins unterschiedlich zu verfahren, falls sich g_2 zu false reduzieren läßt. Dies kann nur dadurch erreicht werden, daß eine Kopie des Λ_b bzw. Λ_π-Konstruktes erzeugt wird. Als Matching-Graph ergibt sich:

Abbildung 6.3: Graph zu Beispiel 5.2 ohne redundante Tests



Jede Erweiterung von Beispiel 5.2 durch Hinzufügen eines konstanten Patterns bedeutet somit entweder eine weitere Kopie des Λ_b bzw. Λ_π-Konstruktes oder aber das Entstehen von redundanten Tests durch Σ bzw. σ-Konstrukte bei geeigneter Argumentwahl.

Sei

kG :=Anzahl der Gruppen von konstanten Pattern und

kP := Anzahl der konstanten Pattern

für eine beliebige Case-Abstraktion. Dann läßt sich das beschriebene Verfahren der Konstruktion von Matching-Graphen mit redundanten Tests einem Verfahren ohne redundante Tests folgendermaßen gegenüberstellen:

- **Für den Laufzeitaufwand während der Processing Phase gilt:**

  Bei einer Konstruktion des Matching-Graphen mit redundanten Tests werden maximal $(kG-1)$ Σ/σ-Konstrukte durchlaufen, obwohl dies nicht erforderlich wäre.

  Ein Verfahren ohne redundante Tests garantiert, daß die minimale Anzahl von Patternmatch-Konstrukten durchlaufen wird.

- **Für den Platzaufwand gilt:**

  Das Verfahren mit redundanten Tests erzeugt pro Gruppe ein Baum gemäß $\Psi_{MG}|_{Pm_1}$.

  Bei einer Vermeidung von redundanten Tests wird ein Baum gemäß $\Psi_{MG}|_{Pm_1}$ für alle konstanten Pattern erzeugt. Weiterhin entsteht pro Var/"as"-Gruppe ein Baum gemäß $\Psi_{MG}|_{Pm_1}$ und dieser wird maximal kP oft kopiert.

- **Für den Laufzeitaufwand während der Preprocessing Phase gilt:**

  Die Grapherzeugung mit redundanten Tests läßt sich direkt auf $\Psi_{MG}|_{Pm_1}$ zurückführen (siehe oben).

  Um einen Matching-Graphen ohne redundante Tests erzeugen zu können, ist ein Nachhalten der für die einzelnen Pfade gültigen Informationen über das Argument erforderlich. Vor der Konstruktion des Graphen müssen zunächst alle Pattern inspiziert werden.

Ausgehend von der Hypothese, daß die Anzahl der Gruppen von konstanten Pattern (kG) im Verhältnis zur Anzahl der insgesamt vorliegenden konstanten Pattern (kP) bei den meisten Applikationen gering ist, ergibt sich aus der obigen Gegenüberstellung für das Verfahren der Matching-Graph-Erzeugung mit redundanten Tests, daß die Laufzeitnachteile der Processing Phase gegenüber den Vorteilen im Bezug auf den Platzbedarf und die Laufzeit der Preprocessing Phase vernachlässigbar gering sind. Deshalb findet dieses Verfahren bei $\mathcal{L}i$ISA Verwendung.

Schließlich soll die betrachtete Menge der Case-Abstraktionen auf den vollen Umfang von $\mathcal{L}$KiR erweitert werden. Das bedeutet die Hinzunahme von geschachtelten sowie mehrstelligen Pattern. Aufgrund der in Kapitel 3.5 festgelegten Reihenfolge der einzelnen Patterntests einer π-Abstraktion ("von links nach rechts") lassen sich sämtliche Case-Abstraktionen aus $\mathcal{L}$KiR mittels der folgenden Transformationsregeln auf Case-Abstraktionen aus Pm_2 zurückführen:

when $(p_1, \ldots, p_n)$ guard g do e⇒when p_1 do when $(p_2, \ldots, p_n)$ guard g do e
when $[p_1.p_2]$ guard g do e ⇒when $[v_1.v_2]$ do ap when (p_1, p_2) guard g do e to $[v_1, v_2]$.

Auch diese Transformationen lassen sich durch eine Erweiterung von Ψ_{MG} so realisieren, daß keine zusätzlichen Π^n- bzw. ${\Lambda_\pi}^n$-Konstrukte entstehen. Dies wird dadurch erreicht, daß der Parameter *patlist* auf Listen von Pattern erweitert wird und die bisherigen Regeln sich jeweils nur auf die ersten Elemente der Listen beziehen. Transformationen, die zu einem rekursiven Aufruf mit dem $\Box$-Konstrukt anstelle des Patterns führen, werden so modifiziert, daß die rekursiven Aufrufe mit Patternlisten erfolgen, die um das erste Element verkürzt sind. Die den Konstruktionsvorgang eines Matching-Graphen abschließende Regel

$\Psi_{MG}[\![\langle(\Box, w_1)\ldots,(\Box, w_n)\rangle, alt, clear]\!] \mapsto\ \Lambda_\pi p_1\, g_1\ clear\, e_1\ \ldots \Lambda_\pi p_n\, g_n\ clear\, e_n\ alt$
wird somit zu
$\Psi_{MG}[\![\langle(\langle\rangle, w_1)\ldots,(\langle\rangle, w_n)\rangle, alt, clear]\!] \mapsto\ \Lambda_\pi p_1\, g_1\ clear\, e_1\ \ldots \Lambda_\pi p_n\, g_n\ clear\, e_n\ alt.$

Da der Matching-Graph in den Programm-Graphen eingefügt wird und diesen nur gering modifiziert (vergl. Abbildung 6.2), läßt sich die Forderung ($*$) leicht erfüllen. So gilt:

$\Psi_4^{-1}[\![\Pi^n\, mg\, w_1 \ldots w_n\, e_0]\!]$ mit $w_i = {\Lambda_\pi}^{n_i} p_i\, g_i\, e_i\, t_i\ \wedge\ e_i = \Xi_{(\ldots)}\, e_i'$ für $i \in \{1, \ldots, n\}$
$\mapsto\ \Pi^n \Box\, w_1 \ldots w_n\, e_0$ mit $w_i = {\Lambda_\pi}^{n_i} p_i\, g_i\, e_i'\, \Box$

Um die Forderung (∗∗) erfüllen zu können, müssen bei der Reduktion entstehende $\overline{\Pi}$-Konstrukte durch Π-Konstrukte ersetzt werden, da $\mathcal{L}_{Graph}{}'$ keine $\overline{\Pi}$-Konstrukte enthält (vergl. Kapitel 6.3).

Ein $\overline{\Pi}$-Konstrukt kann aus zwei Gründen entstehen. Entweder liegt ein Argument vor, für das nicht entschieden werden kann, ob das Pattern paßt, oder es sind nicht genügend Argumente vorhanden. In beiden Fällen können vorher erfolgreiche Patterntest stattgefunden haben, so daß eine *partielle Anwendung der Case-Abstraktion* vorliegt. Für die hochsprachliche Darstellung einer solchen Anwendung gibt es prinzipiell zwei Möglichkeiten. Zum einen kann eine Applikation der Case-Abstraktion auf die Argumente rekonstruiert werden, zum anderen ist es möglich, die partielle Anwendung durch eine Modifikation der Case-Abstraktion explizit sichtbar zu machen. Ein Ausdruck der Form

Beispiel 6.3: Partielle Anwendung einer Case-Abstraktion

```
ap case
     when (A,B,1) do 1
     when (A,1,2) do 2,
     when (1,2,3) do 3
   end
to [1,2]
```

könnte transformiert werden zu:

```
case
  when 1 do 1
  otherwise ap case
                when (1,2) do 2
                otherwise ap case
                              when (1,2,3) do 3
                            end
                          to [1,2]
              end
            to [2]
end
```

Schon an diesem kleinen Beispiel zeigt sich, daß der Bezug solcher Darstellungen zu den unreduzierten Ausdrücken nicht unmittelbar ersichtlich und damit der Nutzen dieser Darstellungsform für die Anwendungs-Programmierung fragwürdig ist. Daher wird bei $\mathcal{L}i$SA eine Darstellung durch Rekonstruktion der Applikation verfolgt:

$$\Psi_4^{-1}[\![\overline{\Pi}^{k,l,m}\, n_0\, s_1 \ldots s_k\, pm_1 \ldots pm_l\, bi_1 \ldots bi_m]\!] \text{ mit } find_{\Lambda_\pi}(n_0) = \langle w_1, \ldots, w_n\rangle,$$
$$w_i = {\Lambda_\pi}^{n_i} p_i\, g_i\, e_i\, t_i \ \wedge\ e_i = \Xi_{(\ldots)}\, e_i' \text{ für } i \in \{1, \ldots, n\} \text{ sowie}$$
$$getargs(\langle pm_l, \ldots, pm_1, s_1, \ldots, s_k\rangle) = \langle a_1, \ldots, a_q\rangle$$
$$\mapsto\ @^q\ \Pi^n \Box\, w_1 \ldots w_n\, e_0\ a_1 \ldots a_q \qquad \text{mit } w_i = {\Lambda_\pi}^{n_i} p_i\, g_i\, e_i' \Box.$$

Dabei liefert $find_{\Lambda_\pi}$ eine Liste von Verweisen auf die Λ_π-Konstrukte, die über den Matching-Graphen erreichbar sind, und *getargs* selektiert aus den vor der Bildung des $\overline{\Pi}$-Konstruktes gültigen Stackeinträgen des S- bzw. Pm-Stacks die Argumentverweise. Somit ist auch für Ψ_4/Ψ_4^{-1} die Forderung (∗∗) erfüllt.

7 Zusammenfassung und Bewertung

In der vorliegenden Arbeit wird die Realisierung des Lazy Evaluators $\mathcal{L}i$SA beschrieben, der eine schrittweise Transformation von $\mathcal{L}$KiR-Ausdrücken ermöglicht. Die Verwendung der Lazy Evaluation erschließt dem Anwender die Vollständigkeit der Normal-Order-Strategie und stellt somit gegenüber dem auf der Applicative Order Strategie basierenden System π-RED* eine echte Erweiterung der Menge der zur Normalform reduzierbaren Ausdrücke dar. Dadurch wird es möglich, die in Kapitel 2.3 beschriebenen Programmier-Techniken zu nutzen, die auf der Verwendung von zyklischen Datenstrukturen beruhen. Sie gestatten elegante und konzise Programmspezifikationen für Problemstellungen, die in irgend einer Weise Erzeuger-Verbraucher-Abhängigkeiten enthalten wie z.B. der in [John85] vorgestellte λ-Lifting Algorithmus.

7.1 Der Ausführungsmechanismus

Dem bei Verwendung der Normal Order Strategie anfallenden Mehraufwand, der durch das Kopieren von Redizes in Argumentposition entsteht, wird mittels Lazy Evaluation durch ein Sharing von Berechnungen begegnet. So werden überflüssige Berechnungen auf Kosten eines Verwaltungsaufwandes für die Realisierung des Sharing vermieden. Um diesen Aufwand (Ersparnis) sowie die Auswirkungen der Verwendung unterschiedlicher Sprachkonstrukte wie Tupel, Listen und Patternmatch quantifizieren zu können, bedarf es einiger Laufzeitmessungen an ausgewählten Beispielen.

Da für $\mathcal{L}i$SA ausschließlich die Processing Phase in der Programmiersprache C codiert ist, sind Vergleiche zur π-RED* nur für diesen Teil der Ausführungsphase sinnvoll. Deshalb beschränken sich alle Messungen auf diese Phase.

Bei Vergleichen zur π-RED* ist weiterhin zu berücksichtigen, daß die Art der Speicherverwaltung bei den beiden Systemen unterschiedlich ist. π-RED* benutzt eine auf *Reference-Counting* basierende Speicherverwaltung und sorgt somit dafür, daß die Speicherbereiche von Datenstrukturen freigegeben werden, sobald diese nicht mehr benötigt werden. Die Speicherverwaltung von $\mathcal{L}i$SA stellt diesen Verwaltungsaufwand jedoch so lange wie möglich zurück; d.h. der Speicherbereich unbenötigter Datenstrukturen wird mittels einer sog. *Garbage-Collection* erst dann freigegeben, wenn für eine neu anzulegende Datenstruktur kein Speicherbereich mehr vorhanden ist. Diese Vorgehensweise hat zur Folge, daß in $\mathcal{L}i$SA die Anzahl solcher Speicherfreigaben und somit der Verwaltungsaufwand für den Speicher in starkem Maße von dem insgesamt zur Verfügung stehenden Speicherplatz abhängt. Dieser Einfluß auf die Laufzeiten von $\mathcal{L}i$SA soll hier nicht betrachtet werden; er wird in [Rath91] ausführlich untersucht. Zur Eliminierung dieses Einflusses werden hier ausschließlich Laufzeiten ohne den Zeitaufwand für eventuell stattfindende Garbage-Collection-Vorgänge betrachtet. Dies führt zwar zu einer "Verfälschung" der absoluten Laufzeiten von $\mathcal{L}i$SA, ermöglicht jedoch qualifiziertere Aussagen beim Vergleich der Laufzeiten unterschiedlicher Programme bzw. Probleminstanzen.

Alle Messungen werden auf einer SPARC-SUN IPC mit 24MB Hauptspeicher unter dem Betriebssystem SunOS-Release4.1.1 vorgenommen. Die Laufzeitmessungen erfolgen mittels des durch die Laufzeit-Bibliothek zur Verfügung gestellten timer-Kommandos und werden aus drei Messungen arithmetisch gemittelt.

Die Platzbedarf-Angaben bezeichnen den maximal benötigten Speicherbedarf für Deskriptoren (aufgerundet auf volle Tausend) bzw. Heapsegmente (aufgerundet auf volle kByte). Für $\mathcal{L}i$SA werden die Werte dadurch gewonnen, daß nach jeweils 1000 Transformationsschrit-

ten des Ausführungsmechanismus eine Garbage Collection veranlaßt und anschließend das Maximum der dabei ermittelten Werte gebildet wird.

Zunächst sollen Laufzeitvergleiche für die β-Reduktion und die primitive Rekursion erfolgen. Dazu bedarf es Programmen, die ausschließlich diese "Grundoperationen" nutzen. Da es sich bei beiden System um rücktransformierende Systeme handelt, ist dies dergestalt möglich, daß "Endlos-Rekursionen" spezifiziert werden und die Berechnung mittels des Reduktionszählers beendet wird.

Für die Messung von β-Reduktionen liegt es daher nahe, die Anwendung des Y-Kombinators auf die Identität zu betrachten.

Beispiel 7.1: Rekursion mittels β-Reduktion (simple_y):

```
let A = sub [ X, Y ]
        in Y [ X [ X, Y ] ]
in A [ A, sub [ X ]
          in X ]
```

Dies stellt sich jedoch insofern als ungünstiges Vergleichsbeispiel heraus, als daß aufgrund der Applicative Order Strategie eine Berechnung von großen Probleminstanzen (50000 Reduktionsschritte oder mehr) mit π-RED* an einem zu hohen Platzbedarf bzw. zu großen Postprocessing-Zeiten scheitert. Deshalb soll als weiteres Beispiel für die β-Reduktion folgender, sich reproduzierender Ausdruck untersucht werden:

Beispiel 7.2: Ein Ausdruck, der sich reproduziert (omega):

```
ap sub [ X ]
   in X [ X ]
to [ sub [ X ]
     in X [ X ] ]
```

Zur Messung der primitiven Rekursion finden drei weitere Beispiele Verwendung. Sie demonstrieren die Abhängigkeit der Laufzeiten bei primitiven Rekursionen von der Anzahl der Parameter bzw. der Verwendung von relativ freien Variablen als aktuellen Parametern.

Beispiel 7.3: Primitive Rekursion mit einem Paramter (y1):

```
def
  F [ X ] = F [ X ]
in F [ 3 ]
```

Beispiel 7.4: Primitive Rekursion mit zwei Paramtern (y2):

```
def
  F [ X, Y ] = F [ X, Y ]
in F [ 3, 42 ]
```

Beispiel 7.5: Primitive Rekursion mit Anwendung auf relativ freie Variablen (y3):

```
let A = 1,
    B = 1
in def
     F [ X, Y ] = F [ A, B ]
   in F [ 3, 42 ]
```

Als Meßwerte ergeben sich:

Beispiel	Instanzgröße [Reduktionsschritte]	Laufzeit $\mathcal{L}i$SA [sec]	Laufzeit π-RED* [sec]	Laufzeitquotient $\mathcal{L}i$SA/π-RED*
simple_y	10000	0.75	0.84	0.9
	50000	3.90	–	–
	100000	7.82	–	–
omega	10000	0.63	0.39	1.62
	50000	3.16	2.00	1.58
	100000	6.31	3.85	1.64
y1	10000	0.62	0.33	1.88
	50000	3.16	1.64	1.93
	100000	6.04	3.26	1.85
y2	10000	0.74	0.44	1.68
	50000	3.66	2.19	1.67
	100000	7.40	4.37	1.69
y3	10000	0.73	0.44	1.66
	50000	3.68	2.20	1.67
	100000	7.36	4.38	1.68

Bei der Interpretation der Meßwerte muß zwischen simple_y und den übrigen Beispielen unterschieden werden, da bei diesem Beispiel als einzigem je nach verwendeter Reduktionsstrategie unterschiedliche Redizes reduziert werden. Dies führt zu den oben bereits erwähnten Problemen bei großen Probleminstanzen.

Bei den übrigen Beispielen (omega, y1, y2 und y3) ist die Berechnungsreihenfolge von der Reduktionsstrategie unabhängig. Es handelt sich also im Bezug auf Lazy Evaluation um "Worst-Case-Beispiele". Aus den Meßwerten dieser Beispiele ist ersichtlich, daß das Verhältnis der Laufzeiten unabhängig von der Probleminstanz sowohl bei β-Reduktionen als auch bei primitiven Rekursionen ziemlich konstant 1.7 beträgt. Weiterhin ist zu beobachten, daß die Erweiterung eines rekusiven Aufrufes um einen Parameter bei $\mathcal{L}i$SA einen geringeren Laufzeitzuwachs zur Folge hat, als bei π-RED*. Die Verwendung von relativ freien Variablen als aktuellen Parametern hat in beiden Systemen keine Laufzeitauswirkungen.

Nach diesen sehr speziellen Messungen sollen nun Messungen an Beispielen folgen, die möglichst viele verschiedene Sprachkonstrukte nutzen. Da die Verwendung von Lazy Evaluation andere Programmiertechniken erschließt (vergl. Kapitel 2.3), liegt es nahe, die Auswirkungen der Verwendung solcher Techniken auf die Laufzeit bzw. den Speicherbedarf zu untersuchen. Für derartige Betrachtungen eignet sich das sog. "Mintree-Beispiel", bei dem alle Blätter eines binären Baumes durch das Minimum der Blätter ersetzt wird. Der zu modifizierende Baum der Tiefe `N` wird mittels einer Funktion `Gentree` erzeugt. Eine Problemlösung, deren Berechnung auf beiden Systemen zu der gewünschten Normalform führt, ist z.B.:

Beispiel 7.6: "Mintree-Beispiel" mit zwei Traversiervorgängen (apmintree):

```
let N = ...
in let Baum = def
                Gentree [ N, L ] =
                  if (N eq 1)
                  then Leaf{L}
                  else Node{Gentree[(N - 1), (L + 1)], Gentree[(N - 1), (L - 1)]}
              in Gentree [N,0]
   in def
        Min [ ] =
          case
            when Node{L,R} guard true do let L = Min[L],
                                             R = Min[R]
                                         in min(L, R),
            when Leaf{L} guard true do L
          end,
        Subtree [ M ] =
          case
            when Node{L,R} guard true do Node{Subtree[M, L], Subtree[M, R]},
            when Leaf{.} guard true do Leaf{M}
          end,
        Mintree [ Baum ] = Subtree[Min[Baum], Baum]
      in Mintree[Baum]
```

Die Verwendung einer rekursiven Variablen gestattet in einem auf Lazy Evaluation basierenden System eine konzisere Problemlösung in folgender Form:

Beispiel 7.7: "Mintree-Beispiel" mit einem Traversiervorgang (lamintree):

```
let N = ...
in let Baum = def
                Gentree [ N, L ] =
                  if (N eq 1)
                  then Leaf{L}
                  else Node{Gentree[(N - 1), (L + 1)], Gentree[(N - 1), (L - 1)]}
              in Gentree[N, 0]
   in def
        Submintree [ M ] =
          case
            when Node{L,R} guard true do letp [Lt.Lm] = Submintree[M, L],
                                              [Rt.Rm] = Submintree[M, R]
                                          in [ Node{Lt, Rt}. min(Lm, Rm) ],
            when Leaf{L} guard true do [ Leaf{M}. L ]
          end,
        Mintree [ Baum ] =
          hd ( def  Result [ ] = Submintree[tl(Result), Baum]
               in Result )
      in Mintree[Baum]
```

In beiden bisher vorgestellten "Mintree-Beispielen" muß sowohl bei π-RED* als auch bei $\mathcal{L}i$SA der vollständig modifizierte Baum generiert werden, da er das Ergebnis der Berechnung darstellt. Um zu demonstrieren, daß die Verwendung von Lazy Evaluation durch die Vermeidung nicht zum Resultat beitragender Berechnungen zu Laufzeitvorteilen führt, soll

noch ein "Mintree-Beispiel" betrachtet werden, bei dem nicht der gesamte Baum benötigt wird. Ein solches Beispiel entsteht aus Beispiel 7.6 durch Anwendung der Funktion `Leftmost` (siehe Beispiel 7.8) auf den modifizierten Baum; dabei wird der am weitesten links liegende Blattknoten des modifizierten Baumes selektiert.

Beispiel 7.8: Die Funktion `Leftmost`:

```
Leftmost [ ] = case
                 when Node{L,R} guard true do Leftmost[L],
                 when Leaf{L} guard true do L
               end
```

Als Meßwerte ergeben sich für π-RED*:

Beispiel	Instanzgröße [Baumtiefe]	Laufzeit [sec]	Deskriptoren [Tsd.Stck.]	Heap [kByte]	Speicher [kByte]
apmintree/	10	1.15	3	44	92
leftmin	12	4.36	11	167	343
	14	17.90	41	658	1314

und für $\mathcal{L}i$SA:

Beispiel	Instanzgröße [Baumtiefe]	Laufzeit [sec]	Deskriptoren [Tsd.Stck.]	Heap [kByte]	Speicher [kByte]	$\mathcal{L}i$SA/π-RED* [Zeit|Platz]
apmintree	10	1.50	11	31	207	1.30|2.25
	12	5.84	47	133	885	1.34|2.58
	14	23.81	187	523	3515	1.33|2.68
lamintree	10	1.21	13	65	273	1.05|2.97
	12	4.88	57	265	1177	1.12|3.43
	14	19.58	238	1075	4883	1.09|3.71
leftmin	10	0.95	6	18	114	0.83|1.21
	12	3.83	25	68	468	0.88|1.36
	14	15.25	99	265	1849	0.85|1.41

Bei den hier dargestellten Meßwerten handelt es sich um eine Auswahl aus den insgesamt generierten Meßwerten. Die aus Übersichtsgründen in der Darstellung weggelassenen Werte fügen sich dergestalt ein, daß die Laufzeit- bzw. Speicherbedarf-Quotienten im Rahmen der Genauigkeit den übrigen Werten entsprechen.

Erwartungsgemäß zeigt das Beispiel mit einem Traversiervorgang gegenüber dem Beispiel mit zweien Laufzeitvorteile, hat jedoch aufgrund des größeren Umfanges der während der Berechnung zurückgestellten Strukturen einen höheren Speicherbedarf. Das Produkt von Laufzeit- und Platzbedarf-Quotient ist für beide Meßserien annähernd konstant.

Die Meßergebnisse für das nur ein Blatt des Baumes selektierende Beispiel (leftmin) zeigen deutlich die Leistungs-Vorteile von Lazy Evaluation bei nur teilweise benötigten Argumenten: während für π-RED* die Ergebnisse denen von Beispiel 7.6 (apmin) entsprechen, benötigt $\mathcal{L}i$SA im Vergleich zum Beispiel 7.6 nur etwa 2/3 der Laufzeit bei ungefähr halbem Speicherbedarf.

Schließlich soll anhand des Quicksort-Algorithmus ein Vergleich der Verwendung von Tupeln gegenüber Lazy Listen sowie die Auswirkungen der Verwendung von Patternmatch-Konstrukten untersucht werden. Dazu wird zunächst der Quicksort-Algorithmus unter Verwendung von Tupeln und Patternmatch-Konstrukten codiert (vergl.Beispiel 7.9). Eine der-

artige Modifikation, daß alle Tupel-Konstrukte durch Lazy-Listen ersetzt werden, führt zu einem weiteren, als "liquick" referenzierten Beispiel. Schließlich wird noch eine das If-Then-Else-Konstrukt statt des Patternmatch verwendende Variante untersucht (Beispiel 7.10).

Beispiel 7.9: Quicksort-Programm mit Patternmatch und Tupeln (tupquick):

```
let N = ...
in let Liste = def
                 Gen_list [ N, M ] =
                   if (N eq 0)
                   then <>
                   else (<(N * M)> ++ Gen_list[(N - 1), neg(M)])
               in Gen_list[N, 1]
   in def
        QuickSort [ ] =
          def
            Split [ ] =
              case
                when (A,<>,L1,L2) guard true do <L1, L2>,
                when (A,<B,as...L>,L1,L2) guard (B le A)
                                          do Split[A, L, (<B> ++ L1), L2],
                when (A,<B,as...L>,L1,L2) guard true
                                          do Split[A, L, L1, (<B> ++ L2)]
                end
          in case
               when <> guard true do <>,
               when <A,as...L> guard true
                               do letp <L1,L2> = Split [A, L, <>, <>]
                                  in (QuickSort[L1] ++ (<A> ++ QuickSort[L2]))
             end
      in QuickSort [ Liste ]
```

Beispiel 7.10: Quicksort-Programm ohne Patternmatch (ifquick):

```
let N = ...
in let Liste = def
                 Gen_list [ N, M ] =
                   if (N eq 0)
                   then <>
                   else (<(N * M)> ++ Gen_list[(N - 1), neg(M)])
               in Gen_list [ N, 1 ]
   in def
        QuickSort [ L ] =
          def
            Split [ A, L, L1, L2 ] =
              if empty(L)
              then <L1, L2>
              else let B = (1 | L)
                   in if (B le A)
                      then Split[A, lcut(1, L), (<B> ++ L1), L2]
                      else Split[A, lcut(1, L), L1, (<B> ++ L2)]
          in if empty( L )
             then <>
             else let A = (1 | L)
```

```
                in let Erg = Split [A, lcut( 1 , L ), <>, <>]
                   in (QuickSort[(1 | Erg)] ++ (<A> ++ QuickSort[(2 | Erg)]))
in QuickSort[Liste]
```

Als Meßwerte ergeben sich für π-RED*:

Beispiel	Instanzgröße [Listenlänge]	Laufzeit [sec]	Deskriptoren [Tsd.Stck.]	Heap [kByte]	Speicher [kByte]
tupquick	200	9.98	1	85	91
	400	45.51	2	327	359
	600	115.73	3	730	778
ifquick	200	7.30	1	27	43
	400	34.27	1	92	108
	600	90.33	2	197	229

und für $\mathcal{L}i$SA:

Beispiel	Instanzgröße [Listenlänge]	Laufzeit [sec]	Deskriptoren [Tsd.Stck.]	Heap [kByte]	Speicher [kByte]	$\mathcal{L}i$SA/π-RED* [Zeit|Platz]
tupquick	200	8.42	7	30	142	0.84|1.56
	400	34.61	24	98	482	0.76|1.34
	600	84.15	51	205	1021	0.73|1.31
liquick	200	6.70	41	120	776	0.67|8.53
	400	26.40	153	436	2884	0.58 |8.03
	600	48.37	337	941	6333	0.54 |8.14
ifquick	200	10.02	7	30	142	1.37|3.30
	400	41.60	24	99	483	1.21|4.47
	600	98.06	51	207	1023	1.08|4.47

Auch bei diesen Meßwerten handelt es sich um eine repräsentative Auswahl.

Der Quicksort-Algorithmus erfordert eine häufige Komposition bzw. Dekomposition von Listen. Daraus erwächst im Vergleich zu den vorherigen Beispielen ein hoher Zeitaufwand zur Speicherverwaltung. Da dieser bei den Laufzeiten von $\mathcal{L}i$SA unberücksichtigt bleibt (siehe oben), sind alle Laufzeitquotienten verhältnismäßig niedrig.

Der Vergleich der strikten Tupel gegenüber den Lazy Listen zeigt, daß durch die unaufwendige Komposition von binären Listen das liquick-Beispiel zwar erheblich schneller reduziert wird (zwischen Faktor 0.8 und Faktor 0.57), jedoch auch einen größeren Speicherbedarf hat (zwischen Faktor 5.46 und Faktor 6.20). Die Ursache dafür ist darin zu suchen, daß bei den Lazy Listen während der Ausführung viele verschieden instanziierte und in der Berechnung zurückgestellte Ausdrücke entstehen.

Als besonders interessant stellt sich der Vergleich der Quicksort-Programme mit und ohne Verwendung des Patternmatch auf beiden Systemen heraus. Während $\mathcal{L}i$SA durch die Verwendung des Patternmatch bei gleichem Speicherbedarf geringere Laufzeiten aufweist (zwischen Faktor 0.83 und Faktor 0.86), benötigt π-RED* bei mehr als doppelt so hohem Speicherbedarf höhere Laufzeiten (zwischen Faktor 1.37 und Faktor 1.28). Eine Betrachtung des Ausführungsmechanismus von $\mathcal{L}i$SA zeigt: beim Patternmatch wird für die Analyse und Dekomposition eines Argumentes nur *eine* Virtuelle Maschine gegründet; bei dem ifquick-Beispiel sind dafür zwei Gründungen je einer virtuellen Maschine erforderlich. Außerdem werden durch die Realisierung eines baumartigen Ansatzes beim Patternmatch überflüssige Argumentuntersuchungen vermieden. Bei π-RED* entstehen durch das Patternmatch keine derartigen Ersparungen. Stattdessen bleiben bei der Realisierung des Patternmatch in

π-RED* überlappende Pattern unberücksichtigt, und es kommt somit durch das sukzessive Testen der vollständigen Pattern zu Redundanzen.

Eine Zusammenfassende Betrachtung aller gemessenen Beispiele ergibt, daß die Laufzeiten von $\mathcal{L}i$SA (ohne Berücksichtigung der Garbage Collection!) alle deutlich unter Faktor zwei zu denen von π-RED* liegen. Bei Beispielen, die durch die Verwendung von Lazy Evaluation für eine Reduktion zur Normalform erheblich weniger Reduktionsschritte als π-RED* benötigen, sinkt dieser Faktor erwartungsgemäß unter eins.

Allerdings zeigt sich insgesamt ein verhältnismäßig hoher Speicherbedarf bei Anwendungen von $\mathcal{L}i$SA. Zu einem großen Teil ist dieser sicherlich durch die Verwendung von Lazy Evaluation problememanent, die bisherige Implementierung der Indirektionsknoten als Deskriptoren trägt jedoch auch dazu bei; durch Verwendung einer eigenständigen Datenstruktur für diese Graphknoten ist es möglich, den Platzbedarf dieser auf 50% des bisherigen zu reduzieren. Bei einem durchschnittlich gemessenen Deskriptor-Anteil von 70% an Indirektionsknoten kann eine solche Modifikation also eine Platzersparnis von 35% bedeuten. Da die bisherige Implementierung bereits durch die Verwendung von geeigneten Macros auf eine solche Umstellung vorbereitet ist, bietet sich an dieser Stelle eine günstige Optimierungsmöglichkeit.

7.2 Die Implementierungs-Konzeption

Wie in Kapitel 4 beschrieben, erfolgt die Implementierung unter Verwendung der Bootstrap-Technik und einer zunächst vorgenommenen, vollständig funktionalen Spezifikation. Diese Vorgehensweise hat zur Folge, daß sowohl während der Preprocessing Phase, als auch während der Postprocessing Phase durch den Bootstrap-Vorgang erzeugte Graphen in der internen Darstellung durch den Ausführungskern interprtiert werden müssen. Daraus resultiert im Vergleich zur π-RED* ein sehr hoher Zeitaufwand für diese Phasen der Berechnung (ca. Faktor 300!). Aus mehreren Gründen scheint dies jedoch akzeptabel:

- Die Preprocessing- bzw. Postprocessing-Zeiten sind unabhängig von der Probleminstanz.
- $\mathcal{L}i$SA gestattet es, Ausdrücke nach der Preprocessing Phase als interne Darstellung in Form von ASCII-Files abzulegen und anschließend ohne eine Preprocessing Phase zu reduzieren.
- Die absoluten für die Preprocessing- und Postprocessing Phase benötigten Laufzeiten halten sich in einem vertretbaren Rahmen; für das oben gemessene Beispiel 7.6 (apmintree) benötigt $\mathcal{L}i$SA für die Preprocessing Phase 6 Sekunden.

Dafür erweist sich die Implementierungs-Konzeption in allen Phasen der Entwicklung von $\mathcal{L}i$SA als hilfreich:

- Während der Entwurfsphase ist es mit verhältnismäßig geringem Recodieraufwand möglich, unterschiedliche Realisierungsansätze zu überprüfen bzw. konzeptuelle Probleme aufzudecken.
- Nach einer Implementierung des Ausführungsmechanismus in der Programmiersprache C stehen durch den in KiR spezifizierten Preprocessor bereits Beispiele in der internen Darstellung zur Verfügung. Vergleiche im Ausführungsverhalten der funktionalen Spezifikation des Processors mit der in C codierten Version erleichtern die Fehlersuche erheblich.

- Die Bootstrap-Vorgänge ermöglichen ein effizientes Debugging des Preprocessors in K*i*R sowie des Ausführungsmechanismus in C:

  Die erste Selbstanwendung des Preprocessors mittels π-RED* erfordert zunächst eine dahingehende Fehlerfreiheit, daß es möglich ist, die K*i*R-Darstellung des Preprocessors in eine interne Darstellung zu transformieren. Aufgrund des Umfanges dieser internen Darstellung ist ihre Korrektheit im Sinne der Abbildung Ψ aus Kapitel 6 nicht ohne weiteres feststellbar. Deshalb können Fehler bei der Anwendung dieser internen Darstellung auf die K*i*R-Darstellung des Preprocessors mittels des in C spezifizierten Ausführungskernes zwei Ursachen haben: entweder eine unkorrekte interne Darstellung oder eine fehlerhafte Implementierung des Ausführungsmechanismus. In beiden Fällen handelt es sich jedoch um Fehler in der Realisierung des Gesamt-Systems $\mathcal{L}i$SA. Für andere Testbeispiele existieren ebenso diese zwei potentiellen Fehlerquellen, Fehler im Testbeispiel tragen jedoch nicht zum Debugging von $\mathcal{L}i$SA bei. Ergibt die zweite Selbstanwendung des Preprocessors schließlich wiederum eine interne Darstellung des Preprocessors, so läßt sich die Korrektheit dieser durch einen Vergleich mit der durch π-RED* erzeugten internen Darstellung nachweisen. Das Debugging des in C codierten Ausführungskernes anhand der Selbstanwendung des Preprocessors bewirkt also zugleich ein Debugging des Preprocessors selbst. Ein Indiz dafür, daß der erfolgreiche Bootstrap ein verhältnismäßig fehlerfreies System garantiert, zeigt sich darin, daß die Anwendung des Preprocessors auf den in K*i*R spezifizierten Postprocessor schon beim ersten Versuch sowohl in π-RED*, als auch in $\mathcal{L}i$SA die gleiche interne Darstellung des Postprocessors liefert.

- Nach erfolgreichen Bootstrap-Vorgängen steht ein System zur Verfügung, für dessen Portierung auf andere Systeme lediglich eine Recompilation von C-Sourcen im Umfang von ca. 200 kByte (System ohne Debug-Tool) bzw. 650 kByte (System mit Debug-Tool) sowie der Transfer von ca. 500 kByte Text in Form von ASCII-Dateien, die den Pre- und Postprocessor in der internen Darstellung enthalten, erforderlich sind. Eine anschließende Selbstanwendung der transferierten Graphdarstellung für den Preprocessor stellt in gleicher Weise wie bei der Ersterstellung von $\mathcal{L}i$SA ein sehr geeignetes Testbeispiel dar.

  Weiterhin zeigt es sich als vorteilhaft, daß der Pre- und Postprocessor als $\mathcal{L}$K*i*R-Programm vorliegen, da dadurch Erweiterungen des Sprachumfanges von mit im Verhältnis zu einer C-Implementierung geringem Codieraufwand möglich werden.

7.3 Erweiterungs-Ansätze

Aus der Entwicklung von $\mathcal{L}i$SA ergeben sich einige vorstellbare Ansatzpunkte für eine Erweiterung des Systems.

Wie bereits erwähnt, eröffnet die funktionale Spezifikation der Pre- und Postprocessing Phase unaufwendige Spracherweiterungen. Besonders geeignet dafür sind Sprachkonstrukte wie list-comprehensions (ZF-expressions) [PJ87], die nicht unbedingt eine Erweiterung der Processing Phase erfordern. Jedoch auch die Hinzunahme anderer Sprachkonstrukte wie z.B Real-Zahlen, Digit-Strings oder höhere Datenstrukturen (Vektoren, Matritzen etc.) wird durch die funktionale Spezifikation erleichtert.

Aufgrund des Umfanges der funktionalen Spezifikation (ca. 400kByte Programm-Text) zeigt sich sowohl für π-RED* als auch für $\mathcal{L}i$SA der Bedarf für ein Modulkonzept; es sollte die Reduktion von in Bibliotheken befindlichen Funktionen für den Anwender transparent

halten, um damit beim Debugging von Programmen durch die Verringerung der dargestellen Programm-Komponenten das Auffinden von Fehlern zu erleichtern.

Die Laufzeitmessungen zeigen, daß die Verwendung unterschiedlicher Sprachkonstrukte sowie die Art der Speicherverwaltung einen erheblichen Einfluß auf das Laufzeitverhalten von $\mathcal{L}i$SA haben. Eine detaillierte Untersuchung dieser Zusammenhänge verspricht daher neue Ansatzpunkte für eine Optimierung des Systems.

Weitere denkbare Ergänzungen des Systems wären z.B. eine Reduktion zur Normalform (statt zur Weak-Normalform) von Ausdrücken oder die Integration von Ein-/Ausgabe-Kanälen des Betriebssystems in Form von Lazy-Listen.

Zusamenfassend läßt sich sagen, daß mit $\mathcal{L}i$SA ein Reduktions-System zur bedeutungserhaltenden Transformation von Ausdrücken auf der Basis von Lazy Evaluation vorliegt, das im wesentlichen die gleichen syntaktischen Konstrukte wie π-RED* zur Verfügung stellt und zugleich aufgrund seiner Beschaffenheit diverse Ansatzpunkte für weitere Untersuchungen bietet.

A Die Syntax von $\mathcal{L}$KiR

$$
\begin{array}{lll}
\langle expr\rangle & \Longrightarrow & \langle const\rangle \mid \langle var\rangle \mid \backslash^{+}\langle var\rangle \mid \langle fun\rangle \mid \langle primfun\rangle \mid \\
& & [\langle expr\rangle\big(.\langle expr\rangle\big)^{+}] \mid [\,] \mid \\
& & <\langle expr\rangle\big(,\langle expr\rangle\big)^{*}> \mid <> \mid \\
& & \langle name\rangle\{\} \mid \langle name\rangle\{\langle expr\rangle\big(,\langle expr\rangle\big)^{*}\} \mid \\
& & (\langle expr\rangle\langle expr\rangle\langle expr\rangle) \mid \\
& & \langle primfun\rangle(\langle expr\rangle\big(,\langle expr\rangle\big)^{*}) \mid \langle fun\rangle(\langle expr\rangle\big(,\langle expr\rangle\big)^{*}) \mid \\
& & \text{ap } \langle expr\rangle \text{ to } [\langle expr\rangle\big(,\langle expr\rangle\big)^{*}] \mid \\
& & \text{sub } [\langle var\rangle\big(,\langle var\rangle\big)^{*}] \text{ in } \langle expr\rangle \mid \\
& & \text{let } \langle var\_def\rangle\big(,\langle var\_def\rangle\big)^{*} \text{ in } \langle expr\rangle \mid \\
& & \text{letp } \langle pat\_def\rangle\big(,\langle pat\_def\rangle\big)^{*} \text{ in } \langle expr\rangle \mid \\
& & \text{def } \langle fun\_def\rangle\big(,\langle fun\_def\rangle\big)^{*} \text{ in } \langle expr\rangle \mid \\
& & \text{if } \langle expr\rangle \text{ then } \langle expr\rangle \text{ else } \langle expr\rangle \mid \\
& & \text{then } \langle expr\rangle \text{ else } \langle expr\rangle \mid \\
& & \langle case\_expr\rangle \mid \langle pattern\_expr\rangle \\
\langle const\rangle & \Longrightarrow & \text{true} \mid \text{false} \mid \\
& & \langle integer\_number\rangle \mid \tilde{}\,\langle integer\_number\rangle \mid \langle string\rangle \\
\langle string\rangle & \Longrightarrow & \text{`}\langle string\_elem\rangle^{*}\text{`} \\
\langle string\_elem\rangle & \Longrightarrow & \langle char\rangle \mid \langle string\rangle \\
\langle var\rangle & \Longrightarrow & \langle name\rangle \\
\langle fun\rangle & \Longrightarrow & \langle name\rangle \\
\langle name\rangle & \Longrightarrow & \langle alphanumeric\rangle^{*} \mid \_\langle alphanumeric\rangle^{*} \\
\langle var\_def\rangle & \Longrightarrow & \langle var\rangle{=}\langle expr\rangle \\
\langle pat\_def\rangle & \Longrightarrow & \langle pattern\rangle{=}\langle expr\rangle \\
\langle fun\_def\rangle & \Longrightarrow & \langle fun\rangle[\langle var\rangle\big(,\langle var\rangle\big)^{*}]{=}\langle expr\rangle \mid \langle fun\rangle[]{=}\langle expr\rangle \\
\langle case\_expr\rangle & \Longrightarrow & \text{case } \langle pattern\_expr\rangle\big(,\langle pattern\_expr\rangle\big)^{*} \text{ otherwise } \langle expr\rangle \text{ end} \mid \\
& & \text{case } \langle pattern\_expr\rangle\big(,\langle pattern\_expr\rangle\big)^{*} \text{ end} \\
\langle pattern\_expr\rangle & \Longrightarrow & \text{when } \langle patterns\rangle \text{ do } \langle expr\rangle \mid \\
& & \text{when } \langle patterns\rangle \text{ guard } \langle expr\rangle \text{ do } \langle expr\rangle \\
\langle patterns\rangle & \Longrightarrow & \langle pattern\rangle \mid (\langle pattern\rangle\big(,\langle pattern\rangle\big)^{+}) \\
\langle pattern\rangle & \Longrightarrow & \langle konst\rangle \mid \langle var\rangle \mid \text{.} \mid \text{as } \langle pattern\rangle\langle var\rangle \mid \\
& & [\langle pattern\rangle\big(.\langle pattern\rangle\big)^{+}] \mid \\
& & \langle Name\rangle\{\} \mid \langle name\rangle\{\langle pattern\rangle\big(.\langle pattern\rangle\big)^{*}\} \mid \\
& & <\langle pattern\rangle\big(,\langle pattern\rangle\big)^{*}> \mid <\big(\langle pattern\rangle,\big)^{*}\ldots\big(,\langle pattern\rangle\big)^{*}> \mid \\
& & <\big(\langle pattern\rangle,\big)^{*}\text{as}\ldots\langle var\rangle\big(,\langle pattern\rangle\big)^{*}>
\end{array}
$$

B Die primitiven Funktionen in $\mathcal{L}$KiR

B.1 Die Signatur

Für die zulässigen Typen werden folgende Abkürzungen verwendet:

- *int* – integer Wert,
- *bool* – boolean Wert,
- *cons* – Lazy-Liste/Nil,
- *tupel* – n-stellige Liste/Nil,
- *str* – genesteter String,
- *expr* – beliebiger Ausdruck.

Funktion	**Symbol**	**Signatur** $\rightarrow$
Addition	$+$	$int \times int \rightarrow int$
Subtraktion	$-$	$int \times int \rightarrow int$
Multiplikation	$*$	$int \times int \rightarrow int$
Division	$/$	$int \times int \rightarrow int$
Modulo	`mod`	$int \times int \rightarrow int$
Absolutwert	`abs`	$int \rightarrow int$
Negation	`neg`	$int \rightarrow int$
Maximum	`max`	$int \times int \rightarrow int$
Minimum	`min`	$int \times int \rightarrow int$
log. Und	`and`	$bool \times bool \rightarrow bool$
log. Oder	`or`	$bool \times bool \rightarrow bool$
log. Exklusiv Oder	`xor`	$bool \times bool \rightarrow bool$
log. Negation	`not`	$bool \times bool \rightarrow bool$
gleich	`eq`	$(bool \cup int \cup str) \times (bool \cup int \cup str) \rightarrow bool$
ungleich	`ne`	$(bool \cup int \cup str) \times (bool \cup int \cup str) \rightarrow bool$
kleiner	`lt`	$(bool \cup int \cup str) \times (bool \cup int \cup str) \rightarrow bool$
kleiner/gleich	`le`	$(bool \cup int \cup str) \times (bool \cup int \cup str) \rightarrow bool$
größer	`gt`	$(bool \cup int \cup str) \times (bool \cup int \cup str) \rightarrow bool$
größer/gleich	`ge`	$(bool \cup int \cup str) \times (bool \cup int \cup str) \rightarrow bool$
Selektion	`select`	$int \times (tupel \cup str) \rightarrow (expr \cup str)$
Vereinigung	`unite`	$(tupel \cup str) \times (tupel \cup str) \rightarrow (tupel \cup str)$
abschneiden	`cut`	$int \times (tupel \cup str) \rightarrow (tupel \cup str$
Ersetzung	`replace`	$int \times (expr \cup str) \times (tupel \cup str) \rightarrow (tupel \cup str)$
Umkehrung	`reverse`	$(tupel \cup str) \rightarrow (tupel \cup str)$
Länge	`dim`	$(tupel \cup str) \rightarrow int$
Leereliste	`empty`	$(tupel \cup str \cup cons) \rightarrow bool$
Listenkopf	`hd`	$cons \rightarrow expr$
Listenrest	`tl`	$cons \rightarrow cons$
Präfix	`prefix`	$str \times str \rightarrow str$
Suffix	`suffix`	$str \times str \rightarrow str$

Alternative Schreibweisen:

- `unite` = ++ = `lunite`
- `select` = | = `lselect`
- `dim` = `ldim`
- `cut` = `lcut`
- `replace` = `lreplace`

B.2 Die Funktionalität

Die Funktionalität der primitiven Funktionen, die sich auch in K*i*R finden, entspricht der in π-RED* realisierten. Deshalb beschränkt sich die Beschreibung hier auf die neu hinzukommenden Funktionen:

1. Die Funktionen für Lazy Listen:

   (a) `hd` : $\begin{cases} [A.B] & \mapsto \quad A \\ [\,] & \mapsto \quad \texttt{hd}([\,]) \end{cases}$

   (b) `tl` : $\begin{cases} [A.B] & \mapsto \quad B \\ [\,] & \mapsto \quad \texttt{tl}([\,]) \end{cases}$

   (c) `empty` : $\begin{cases} [A.B] & \mapsto \texttt{false} \\ [\,] & \mapsto \texttt{true} \end{cases}$

2. Funktionen auf Strings:

   `prefix` und `suffix` selektieren in Abhängigkeit des ersten Argumentes einen Teilstring des zweiten Argumentes. Das erste Argument stellt eine Zeichenmenge, auch *Suchmenge* genannt, in Form eines Strings dar. `prefix` bewirkt die Selektion des größtmöglichen Prefix des zweiten Argumentes, so daß dieses ausschließlich aus Zeichen der Suchmenge besteht; `suffix` selektiert das Komplement von `prefix`. Bei der Spezifikation der Zeichenmenge gibt es zwei Besonderheiten; zum einen bedeutet ein `!` zu Beginn des Strings - falls dieser mindestens zwei Zeichen umfaßt - , daß das Komplement der spezifizierten Zeichenmenge als Suchmenge fungieren soll; zum anderen steht `''` für beliebige genestete Strings. So gilt zum Beispiel:

   `prefix('ab', 'bbacab')`$\rightarrow_\delta$`'bba`,

   `prefix('!ab', 'xyz bx')`$\rightarrow_\delta$`'xyz `,

   `suffix('ab', 'bbacab')`$\rightarrow_\delta$`'cab`,

   `suffix('!''', 'ba'xyab`ba`')`$\rightarrow_\delta$`''xyab`ba`.

C Die Graph-Darstellung

Die Menge $\mathcal{K}$ aller bei $\mathcal{L}i\text{SA}$ verwendeten Graphknoten wird mit Hilfe mehrerer Teilmengen beschrieben. $\mathcal{K}_S := \mathcal{K}_E \cup \mathcal{K}_K \cup \{\Box\}$ umfaßt alle die Graphknoten, die direkt aus dem Syntaxbaum eines $\mathcal{L}\text{K}i\text{R}$-Ausdruckes ableitbar sind ($\Psi_1 \circ \Psi_2$ aus Kapitel 6). $\mathcal{K}_V$ enthält die in der Preprocessing Phase hinzukommenden Variablen- bzw. Funktionsdarstellungen (Ψ_3 aus Kapitel 6) und $\mathcal{K}_P$ sowie $\mathcal{K}_{P'}$ die für den Patternmatch Code erforderlichen Konstrukte. Die für die β-Reduktion der Processing Phase benötigten Graphknoten sind in $\mathcal{K}_I$ bzw. $\mathcal{K}_{E'}$ zusammengefaßt. Somit ergibt sich $\mathcal{K} := \mathcal{K}_S \cup \mathcal{K}_V \cup \mathcal{K}_P \cup \mathcal{K}_{P'} \cup \mathcal{K}_I \cup \mathcal{K}_{E'}$.

Da die für die Processing Phase erforderliche interne Darstellung $\mathcal{L}_{Graph}$ environmentbehaftet ist, wird sie folgendermaßen definiert:
$\mathcal{L}_{Graph} := \mathcal{G}_{\mathcal{K}}^{\mathcal{K}_{Expr} \times env}$ mit $\mathcal{K}_{Expr} := \mathcal{K} \backslash \{\mathcal{K}_P \cup \mathcal{K}_{P'} \cup \mathcal{K}_I\}$ (zulässige Wurzelknoten).

$k \in \mathcal{K}_E$	$\nu_{anz}(k)$	$\nu_k(m)$	
$\alpha^{k,l}$	$(2 * l + k + 1)$	$\mathcal{K}_{Expr}$	für $m = 1$
		$\{\Lambda_r\}$	für $m \in \{2, \ldots, (k+1)\}$
		$\{\Lambda^n, \Lambda_r\}$	für $m \in \{(k+2), \ldots, (k+l+1)\}$
		$\{\texttt{name}\}$	für $m \in \{(k+l+1), \ldots, (2*l+k+1)\}$
$@^n$	$(n+1)$	$\mathcal{K}_{Expr}$	für $m \in \{1, \ldots, (n+1)\}$
Λ^n	$(n+1)$	$\mathcal{K}_{Expr}$	für $m = 1$
		$\{\texttt{name}\}$	für $m \in \{2, \ldots, (n+1)\}$
Π^n	$(n+1)$	$\{\Lambda_\pi\}$	für $m = 1$
		$\mathcal{K}_P$	für $m \in \{2, \ldots, (n+1)\}$
Λ_π	4	$\{\prec^n\} \cup \mathcal{K}_K$	für $m = 1$
		$\mathcal{K}_{Expr}$	für $m \in \{2, 3\}$
		$\mathcal{K}_P$	für $m = 4$
Λ_r	1	$\mathcal{K}_{Expr}$	für $m = 1$

$k \in \mathcal{K}_K$	$\nu_{anz}(k)$	$\nu_k(m)$	
$\texttt{int}^i$	0		
$\texttt{bool}^b$	0		
$\texttt{string}^{`s`}$	0		
$\texttt{name}^{`x`}$	0		
$prf^{k,0}$	0		
cons	2	$\mathcal{K}_{Expr}$	für $m \in \{1, 2\}$
ccons	2	$\{\texttt{name}^{`x`}\}$	für $m = 1$
		$\{\texttt{cons}\}$	für $m = 2$
tup^n	n	$\mathcal{K}_{Expr}$	für $m \in \{1, \ldots, n\}$

$k \in \mathcal{K}_V$	$\nu_{anz}(k)$	$\nu_k(m)$	
$(i,j)^\lambda$	0		
$(i)^\alpha_{off}$	2	$\{\Lambda^n\}$	für $m = 1$
		$\{\alpha^{k,l}\}$	für $m = 2$
$(i,j)^{\lambda_r}_{off}$	1	$\{\alpha^{k,l}\}$	für $m = 1$

$k \in \mathcal{K}_I$	$\nu_{anz}(k)$	$\nu_k(m)$	
$inode$	2	$\mathcal{K}_{Expr}$	für $m = 1$
		$\{env\}$	für $m = 2$
env	n	$\{inode\}$	für $m \in \{1, \ldots, (n-1)\}$
		$\{env\}$	für $m = n$

$k \in \mathcal{K}_{E'}$	$\nu_{anz}(k)$	$\nu_k(m)$	
$\overline{@}^n$	$(n+1)$	$\mathcal{K}_{Expr}$	für $m = 1$
		$\{inode\}$	für $m \in \{2, \ldots, (n+1)\}$
$\overline{\Lambda}^n$	2	$\mathcal{K}_{Expr}$	für $m = 1$
		$\{sub^k\}$	für $m = 2$
$\overline{\Pi}^{k,l,m}$	$(k+l+m+1)$	$\mathcal{K}_P$	für $m = 1$
		$\{inode\}$	für $m \in \{2, \ldots, (k+l+m+1)\}$
$prf^{k,l}$	l	$\{inode\}$	für $m \in \{1, \ldots, l\}$
$\overline{\texttt{cons}}$	2	$\{inode\}$	für $m \in \{1, 2\}$
$\overline{\texttt{ccons}}$	2	$\{inode\}$	für $m \in \{1, 2\}$
$\overline{\text{tup}}^n$	n	$\{inode\}$	für $m \in \{1, \ldots, n\}$

$k \in \mathcal{K}_P$	$\nu_{anz}(k)$	$\nu_k(m)$	
Σ	9	$\{\sigma_{<>}, \sigma_{<.>}\}$	für $m = 1$
		$\mathcal{K}_P$	für $m \in \{2, 3, 4, 5, 6, 9\}$
		$\{\sigma\}$	für $m \in \{7, 8\}$
Λ_b	1	$\mathcal{K}_P$	für $m = 1$
Λ_{as}	1	$\mathcal{K}_P$	für $m = 1$
Ξ	1	$\mathcal{K}_P$	für $m = 1$
Ω	1	$\{\Pi\}$	für $m = 1$

$k \in \mathcal{K}_{P'}$	$\nu_{anz}(k)$	$\nu_k(m)$	
$\sigma_{<>}$	2	$\mathcal{K}_P$	für $m = 1$
		$\{\sigma_{<>}, \sigma_{<.>}\} \cup \mathcal{K}_P$	für $m = 2$
$\sigma_{<.>}$	2	$\mathcal{K}_P$	für $m = 1$
		$\{\sigma_{<>}, \sigma_{<.>}\} \cup \mathcal{K}_P$	für $m = 2$
σ^k	$(k+1)$	$\mathcal{K}_P$	für $m \in \{1, \ldots, (k+1)\}$
$\prec^n$	n	$\mathcal{K}_K$	für $m \in \{1, \ldots, n\}$
$\prec_{as}$	2	$\mathcal{K}_P$	für $m = 1$
		$\{\texttt{name}^{\text{`}x\text{`}}\}$	für $m = 2$
$\Box$	0		

D Transformationsregeln

D.1 Aktive Berechnung

D.1.1 β-Reduktion und Rekursion

(1) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = @^{\,k} n_0 \dots n_k\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, \Big[(p_1, n).\ \dots\ .(p_k, n).S\Big], F, Pm, Bi, I\Big[p_1 = (n_1, e), \dots, p_k = (n_k, e)\Big], G, E, f, Rc\Big)$

(2) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \overline{@}^{\,k} n_0 p_1 \dots p_k\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, \Big[(p_1, n).\ \dots\ .(p_k, n).S\Big], F, Pm, Bi, I, G, E, f, Rc\Big)$

(3) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \Lambda^k n_0\Big], E, f, Rc\Big)$
$\Rightarrow \Theta_S\Big(k, e', n, S, F, Pm, Bi, I, G, E\Big[e' = e\Big], f, Rc\Big)$

mit $\Theta_S\Big(0, e, n, S, F, Pm, Bi, I, G, E, f, < B, D, Y, M, S >\Big)$
$\Rightarrow \Big(e, n, S, F, Pm, Bi, I, G, E, f, < (B-1), D, Y, M, (S+1) >\Big)$
$\Theta_S\Big(k, e, n, \Big[(p, \hat{n}).S\Big], F, Pm, Bi, I, G, E\Big[e = \langle p_0 \dots p_m\rangle\Big], f, Rc\Big)$
$\Rightarrow \Theta_S\Big((k-1), e, n, S, F, Pm, Bi, I, G, E\Big[e = \langle p, p_0 \dots p_m\rangle\Big], f, Rc\Big)$
$\Theta_S\Big(k, e, n, \Big[(\$, p).S\Big], F, Pm, Bi, I, G\Big[\Lambda^{k'}\, n_0\Big], E, f, Rc\Big)$
$\Rightarrow \Theta_S\Big(k, e, n, S, F, Pm, Bi, I\Big[p = (n', e)\Big], G\Big[n' = \Lambda^k\, n_0\Big], E, f, Rc\Big)$
$\Theta_S\Big(k, e, n, \Big[\%.S\Big], F, Pm, Bi, I, G, E, f, < B, D, Y, M, S >\Big)$
$\Rightarrow \Big(e, n', \Big[\%.S\Big], F, Pm, Bi, I, G\Big[n' = \overline{\Lambda}^k\, n_0\, n\Big], E, t, < (B-1), D, Y, M, (S+1) >\Big)$

(4) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \overline{\Lambda}^k n_0\, n'\Big], E, f, Rc\Big)$
$\Rightarrow \Theta_S\Big(k, e', n, S, F, Pm, Bi, I, G, E\Big[e' = e\Big], f, Rc\Big)$

(5) $\Big(e, n, S, F, Pm, Bi, I\Big[p_j = (n_j, e_j)\Big], G\Big[n = (i, j)^\lambda\Big], E\Big[e \to e_1 \to \dots \to e_i = \langle p_0 \dots p_j \dots p_m\rangle\Big], f, Rc\Big)$
$\Rightarrow \begin{cases} \Big(e_j, n_j, \Big[(\$, p_j).S\Big], F, Pm, Bi, I, G, E, f, Rc\Big) & \text{falls } n_j \in \{@\,, \alpha, \Lambda_r, \texttt{cons}, \overline{\texttt{cons}}, \texttt{ccons}, \overline{\texttt{ccons}}\} \\ \Big(e_j, n_j, S, F, Pm, Bi, I, G, E, f, Rc\Big) & \text{sonst} \end{cases}$

(6) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \alpha^{k,l} n' \Lambda_{r1} \dots \Lambda_{rk} f_1 \dots f_l\Big], E\Big[e = \langle p_1 \dots p_m\rangle \to \hat{e}\Big], f, Rc\Big)$
$\Rightarrow \Big(e', n', S, F, Pm, Bi, I\Big[p'_1 = (\Lambda_{r1}, e) \dots p'_k = (\Lambda_{rm}, e)\Big], G, E\Big[e' = \langle\rangle \to \langle p'_1 \dots p'_k, p_1 \dots p_m\rangle \to \hat{e}\Big], f, Rc\Big).$

(7) $\Big(e, n, S, F, I, G\Big[n = (i)^\alpha_{off} n_0 n_\alpha\Big], E\Big[e \to e_1 \to \dots \to e_j\Big], f, < B, D, Y, S >\Big)$
$\Rightarrow \Big(e_j, n_0, S, F, I, G, E, f, < B, D, (Y+1), (S+1) >\Big).$

(8) $\Big(e, n, S, F, Pm, Bi, I\Big[p_j = (n_j, e_j)\Big], G\Big[n = (i, j)^{\Lambda_r}_{off} n_\alpha\Big], E\Big[e \to e_1 \to \dots \to e_i = \langle p_0 \dots p_j \dots p_m\rangle\Big], f, Rc\Big)$
$\Rightarrow \begin{cases} \Big(e, n_0, \Big[\%.S\Big], \Big[(p_j, h).(\%, n_j, f).F\Big], Pm, Bi, I, G, E, w, Rc\Big) & \text{falls } n_j = \Lambda_r n_0 \\ \Big(e_j, n_j, S, F, Pm, Bi, I, G, E, h, Rc\Big) & \text{sonst} \end{cases}$

D.1.2 δ-Reduktion und Datenstrukturen

(9) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \texttt{int}^{i}/\texttt{bool}^{b}/\texttt{string}^{`s`}/\texttt{name}^{`x`}/\mathrm{tup}^{0}/[]\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n, S, F, Pm, Bi, I, G, E, t, Rc\Big)$

(10) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = prf^{\,k,0}\Big], E, f, Rc\Big)$
$\Rightarrow \Theta_P\Big(k, \langle\rangle, e, n, S, \Big[(\%, n, f).F\Big], Pm, Bi, I, G, E, f, Rc\Big)$

mit $\Theta_P\Big(0, \langle(p_1, \hat{n}_1), \ldots, (p_m, \hat{n}_m)\rangle, e, n, S, F, Pm, Bi, I\Big[p_1 = (n', e')\Big], G, E, f, Rc\Big)$
$\Rightarrow \Big(e', n', \Big[\%.(p_1, \hat{n}_1).\ldots.(p_m, \hat{n}_m).S\Big], \Big[p_1.\ldots.p_m.F\Big], Pm, Bi, I, G, E, f, Rc\Big)$
$\Theta_P\Big(k, \langle(p_1, \hat{n}_1), \ldots, (p_m, \hat{n}_m)\rangle, e, n, \Big[(p, \hat{n}).S\Big], F, Pm, Bi, I, G, E, f, Rc\Big)$
$\Rightarrow \Theta_P\Big((k-1), \langle(p_1, \hat{n}_1), \ldots, (p_m, \hat{n}_m), (p, \hat{n})\rangle, e, n, S, F, Pm, Bi, I, G, E, f, Rc\Big)$
$\Theta_P\Big(k, \langle(p_1, \hat{n}_1), \ldots, (p_l, \hat{n}_l)\rangle, e, n, \Big[(\$, p).S\Big], F, Pm, Bi, I, G, E, f, Rc\Big)$
$\Rightarrow \Theta_P\Big(k, \langle\ldots\rangle, e, n, S, F, Pm, Bi, I\Big[p = (n', e)\Big], G\Big[n' = prf^{\,k,l}\, p_1 \ldots p_l\Big], E, f, Rc\Big)$
$\Theta_P\Big(k, \langle(p_1, \hat{n}_1), \ldots, (p_l, \hat{n}_l)\rangle, e, n, \Big[\%.S\Big], F, Pm, Bi, I, G, E, f, Rc\Big)$
$\Rightarrow \Big(e, n', \Big[\%.S\Big], F, Pm, Bi, I, G\Big[n' = prf^{\,k,l}\, p_1 \ldots p_l\Big], E, t, Rc\Big)$

(11) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = prf^{\,k,l} p_1 \ldots p_l\Big], E, f, Rc\Big)$
$\Rightarrow \Theta_P\Big(k, \langle(p_1, n) \ldots (p_l, n)\rangle, e, n, S, \Big[(\%, n, f).F\Big], Pm, Bi, I, G, E, f, Rc\Big)$

(12) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \Delta n_t n_e\Big], E, f, Rc\Big)$
$\Rightarrow \Theta_P\Big(1, \langle\rangle, e, n, S, \Big[(\%, n, e, f).F\Big], Pm, Bi, I, G, E, f, Rc\Big)$

(13) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \texttt{cons}\, n_1\, n_2\Big], E, h, Rc\Big)$
$\Rightarrow \Big(e, n', S, F, P, I\Big[p_1 = (n_1, e), p_2 = (n_2, e)\Big], G\Big[n' = \overline{\texttt{cons}}\, p_1\, p_2\Big], E, t, Rc\Big)$

(14) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \overline{\texttt{cons}}\, p_1\, p_2\Big], E, h, Rc\Big)$
$\Rightarrow \Big(e, n, S, F, Pm, Bi, I, G, E, t, Rc\Big)$

(15) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \texttt{cons}\, n_1\, n_2\Big], E, w, Rc\Big)$
$\Rightarrow \Big(e, n_1, \Big[\%.S\Big], \Big[(p_1, w).(p_2, w).(\%, n', w).F\Big], Pm, Bi,$
$I\Big[p_1 = (n_1, e), p_2 = (n_2, e)\Big], G\Big[n' = \overline{\texttt{cons}}\, p_1\, p_2\Big], E, w, Rc\Big)$

(16) $\Big(e, n, S, F, Pm, Bi, I\Big[p_1 = (n_1, e_1)\Big], G\Big[n = \overline{\texttt{cons}}\, p_1\, p_2\Big], E, w, Rc\Big)$
$\Rightarrow \Big(e_1, n_1, \Big[\%.S\Big], \Big[(p_1, w).(p_2, w).(\%, n, w).F\Big], Pm, Bi, I, G, E, w, Rc\Big)$

(17) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \mathrm{tup}\, n_1 \ldots n_k\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_1, \Big[\%.S\Big], \Big[(p_1, f) \ldots (p_k, f).(\%, n', f).F\Big], Pm, Bi,$
$I\Big[p_1 = (n_1, e) \ldots p_k = (n_k, e)\Big], G\Big[n' = \overline{\mathrm{tup}}\, p_1 \ldots p_k\Big], E, f, Rc\Big)$

(18) $\Big(e, n, S, F, Pm, Bi, I\Big[p_1 = (n_1, e_1)\Big], G\Big[n = \overline{\mathrm{tup}} p_1 \ldots p_m\Big], E, f_1, Rc\Big)$

$$\Rightarrow \begin{cases} \Big(e_1, n_1, \Big[\%.S\Big], \Big[(p_1, w) \ldots (p_m, w).(\%, n, w).F\Big], Pm, Bi, I, G\Big[n = \overline{\mathrm{tup}^k} p_1 \ldots p_m\Big], E, w, Rc\Big) \\ \qquad\qquad \text{falls } f = h \wedge f_1 = w \\ \Big(e, n, S, F, Pm, Bi, I, G, E, t, Rc\Big) \qquad \text{sonst} \end{cases}$$

Die Transformationsregeln für `ccons`- bzw. $\overline{\texttt{ccons}}$-Knoten sind zu den Regeln (13)-(16) vollkommen analog.

D.1.3 Patternmatch

(19) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \Pi^{\,k} n_0 \dots n_k\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, S, F, Pm, Bi, I, G, E, f, Rc\Big)$

(20) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \overline{\Pi}^{k,l,m} n_0\, s_1 \dots s_k\, p_1 \dots p_l\, b_1 \dots b_m\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, \Big[s_1.\dots.s_k.S\Big], F, \Big[p_1.\dots.p_l.\%.Pm\Big], \Big[b_1.\dots.b_m.\%.Bm\Big], I, G, E, f, Rc\Big)$

(21) $\Big(e, n, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, Bi, I, G\Big[n = \Lambda_b n_0\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, S, F, \Big[(p_1, \hat{n}_1).Pm\Big], \Big[p_1.Bi\Big], I, G, E, f, Rc\Big).$

(22) $\Big(e, n, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, Bi, I, G\Big[n = \Lambda_{as} n_0\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, \Big[p_1.Bi\Big], I, G, E, f, Rc\Big).$

(23) $\Big(e, n, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p_1 = (n_1, e_1)\Big], G\Big[n = \Sigma \dots\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e_1, n_1, \Big[\%.(p_1, n').S\Big], \Big[(p_1, w).\%.(n, e, f).F\Big], Pm, Bi, I, G, E, w, Rc\Big).$

(24) $(e, n, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p_1 = (n_1, e_1)\Big], G\Big[n = \sigma\langle v_1 \dots v_k\rangle n_1 \dots n_k n_{def}\Big], E, f, Rc\Big)$
$\Rightarrow \begin{cases} \Big(e, n_i, S, F, \Big[(p_1, n').Pm\Big], Bi, I, G, E, f, Rc\Big) & \text{falls } n_1 = v_i \wedge \forall j \in \{1, \dots, (i\text{-}1)\} : v_j \neq n_1 \\ \Big(e, n_{def}, \Big[(p_1, n').S\Big], F, Pm, Bi, I, G, E, f, Rc\Big) & \text{sonst} \end{cases}$

(25) $(e, n, \Big[(p_0, n').S\Big], F, Pm, Bi, I\Big[p_0 = (n_0, e_0)\Big], G\Big[n = \sigma_{<>}{}^k n_t n_f, n_0 = \overline{\text{tup}}^l p_1 \dots p_l\Big], E, f, Rc\Big)$
$\Rightarrow \begin{cases} \Big(e, n_t, \Big[(p_1, n_0).\dots.(p_l, n_0).S\Big], F, \Big[(p_0, n').Pm\Big], Bi, I, G, E, f, Rc\Big) & \text{falls } k = l \\ \Big(e, n_f, \Big[(p_0, n').S\Big], F, Pm, Bi, I, G, E, f, Rc\Big) & \text{sonst} \end{cases}.$

(26) $(e, n, \Big[(p_0, n').S\Big], F, Pm, Bi, I\Big[p_0 = (n_0, e_0)\Big], G\Big[n = \sigma_{<.>}{}^{k,o} n_t n_f, n_0 = \overline{\text{tup}}^l p_1 \dots p_l\Big], E, f, Rc\Big)$
$\Rightarrow \begin{cases} \Big(e, n_t, S', F, \Big[(p_0, n').Pm\Big], Bi, I\Big[p' = (n'', e)\Big], G', E, f, Rc\Big) & \text{falls } k \leq l \\ \quad \text{mit } S' = \Big[(p_1, n_0).\dots.(p_o, n_0).(p', n_0).(p_{(o+l-k+1)}, n_0).\dots.(p_l, n_0).S\Big] & \\ \quad \text{und } G' = G\Big[n'' = \overline{\text{tup}}^{(l-k)} p_{(o+1)} \dots p_{(o+l-k)}\Big] & \\ \Big(e, n_f, \Big[(p_0, n').S\Big], F, Pm, Bi, I, G, E, f, Rc\Big) & \text{sonst} \end{cases}.$

(27) $\Big(e, n, S, F, Pm, Bi, I, G\Big[n = \Lambda_\pi{}^k n_p n_g n_b n_f\Big], E\Big[e{=}\langle p'_1 \dots p'_m\rangle \to \hat{e}\Big], f, Rc\Big)$ mit $Bi{=}\Big[p_1.\dots.p_k.Bi'\Big]$
$\Rightarrow \Big(e', n_g, \Big[\%.S\Big], \Big[(p, w).\%.(n, e, e', f).F\Big], Pm, Bi, I, G, E\Big[e'{=}\langle p_1 \dots p_k p'_1 \dots p'_m\rangle \to \hat{e}\Big], f, Rc\Big).$

(28) $\Big(e, n, \Big[s_1 \dots s_s.S\Big], F, \Big[d_1 \dots d_p.p_1 \dots p_m.Pm\Big], \Big[b_1 \dots b_{bi}.Bi\Big], I, G\Big[n{=}\Xi_{(s,p,m,bi)}\, n_0\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, \Big[p_m.\dots.p_1.S\Big], F, Pm, Bi, I, G, E, f, Rc\Big).$

(29) $\Big(e, n, S, F, P, G\Big[n = \Omega\, n_0\Big], E, f, Rc\Big)$
$\Rightarrow \Big(e, n_0, S, F, P, G, E, t, Rc\Big).$

D.2 Abgeschlossene Berechnung

D.2.1 β-Reduktion und Rekursion

(30) $\Big(e, n, \Big[(\$, p).S\Big], F, Pm, Bi, I\Big[p = (n', e')\Big], G, E, t, Rc\Big)$
$\Rightarrow \Big(e, n, S, F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, t, Rc\Big)$

(31) $\Big(e, n, \Big[(p_1, \hat{n}).\ldots.(p_k, \hat{n}).S\Big], F, Pm, Bi, I, G, E, t, Rc\Big) \quad \wedge \quad top(S) \neq (., \hat{n})$
$\Rightarrow \Big(e, n', S, F, Pm, Bi, G\Big[n' = \overline{@}^{\,k} n, p_1 \ldots p_k\Big], E, t, Rc\Big)$

(32) $\Big(e, n, \Big[\%.S\Big], \Big[(p_1, f_1).(p_2, f_2).F\Big], Pm, Bi, I\Big[p_2 = (n_2, e_2)\Big], G, E, t, Rc\Big)$
$\Rightarrow \Big(e_2, n_2, \Big[\%.S\Big], \Big[(p_2, f_2).F\Big], Pm, Bi, I\Big[p_1 = (n, e)\Big], G, E, f_2\Big)$

(33) $\Big(e, n, \Big[\%.S\Big], \Big[(p_1, .).(\%, n_f, f).F\Big], Pm, Bi, I, G\Big[n_f = \Lambda_r n_0\Big], E, t, Rc\Big)$
$\Rightarrow (e, n, S, F, Pm, Bi, I\Big[p_1 = (n, e)\Big], G, E, f\Big)$

D.2.2 δ-Reduktion und Datenstrukturen

(34) $\Big(e, n, \Big[\%.(p_1, \hat{n}_1).S\Big], \Big[(p_1, .).(\%, n_f, e_f, f).F\Big], Pm, Bi, I, G\Big[n_f = \Delta n_t, n_e\Big], E, t, Rc\Big)$

$$\Rightarrow \begin{cases} (e_f, n_t, S, F, Pm, Bi, I\Big[p_1 = (n, e)\Big], G, E, f\Big) & \text{falls } G\Big[n = \texttt{bool}^{true}\Big] \\ (e_f, n_e, S, F, Pm, Bi, I\Big[p_1 = (n, e)\Big], G, E, f\Big) & \text{falls } G\Big[n = \texttt{bool}^{false}\Big] \\ (e, n_f, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, Bi, I\Big[p_1 = (n, e)\Big], G, E, t, Rc\Big) & \text{sonst} \end{cases}$$

(35) $\Big(e, n, \Big[\%.(p_1, \hat{n}_1).S\Big], \Big[(p_1, .).(\%, n_f, f').F\Big], Pm, Bi, I\Big[p_l{=}(n_l, e_l)\Big], G\Big[n_f = hd\Big], E, t, Rc\Big)$

$$\Rightarrow \begin{cases} \Big(e_l, n_l, \Big[(\$, p_l).S\Big], F, Pm, Bi, I\Big[p_1{=}(n, e)\Big], G, E, f'\Big) & \text{falls } G\Big[n{=}\overline{\texttt{cons}}\, p_l\, p_r\Big] \\ \Big(e, n_f, \Big[(p_1, \hat{n}_1).S\Big], F, Pm, Bi, I\Big[p_1{=}(n, e)\Big], G, E, t, Rc\Big) & \text{sonst} \end{cases}$$

(36) $\Big(e, n, \Big[\%.(p_1, \hat{n}_1).\ldots.(p_k, \hat{n}_k).S\Big], \Big[(p_k, f).(\%, n_f).F\Big], Pm, Bi, I, G\Big[n_f{=}prf^{\,k,l} p_1 \ldots p_l\Big], E, t, Rc\Big)$

$$\Rightarrow \begin{cases} \Big(e, n', S, F, Pm, Bi, I\Big[p_k{=}(n, e)\Big], G, E, t, Rc\Big) & \text{falls } n'{=}prf^{\,k,0}(p_1, \ldots, p_k) \\ \Big(e, n_f, \Big[(p_1, \hat{n}_1).\ldots.(p_k, \hat{n}_k).S\Big], F, Pm, Bi, I\Big[p_k{=}(n, e)\Big], G, E, t, Rc\Big) & \text{sonst} \end{cases}$$

(37) $\Big(e, n, \Big[\%.S\Big], \Big[(p_2, .).(\%, n_f, w).F\Big], Pm, Bi, I, G\Big[n_f = \overline{\texttt{cons}}\, p_1\, p_2\Big], E, t, Rc\Big)$
$\Rightarrow (e, n_f, S, F, Pm, Bi, I\Big[p_2 = (n, e)\Big], G, E, t, Rc\Big)$

(38) $\Big(e, n, \Big[\%.S\Big], \Big[(p_2, .).(\%, n_f, w).F\Big], Pm, Bi, I, G\Big[n_f = \overline{\texttt{ccons}}\, p_1\, p_2\Big], E, t, Rc\Big)$
$\Rightarrow (e, n_f, S, F, Pm, Bi, I\Big[p_2 = (n, e)\Big], G, E, t, Rc\Big)$

(39) $\Big(e, n, \Big[\%.S\Big], \Big[(p_m, .).(\%, n_f, f').F\Big], Pm, Bi, I, G\Big[n_f = \overline{\text{tup}}\, p_1 \ldots p_m\Big], E, t, Rc\Big)$
$\Rightarrow (e, n_f, S, F, Pm, Bi, I\Big[p_m = (n, e)\Big], G, E, t, Rc\Big)$

D.2.3 Patternmatch

(40) $\Big(e, n, \Big[\%.(p_1, \hat{n}_1).S\Big], \Big[(p_1, .).(\%, n_f, e_0, f).F\Big], Pm, Bi, I, G\Big[n_f{=}\Lambda_\pi{}^{(k+4)} n_p n_g n_b n_n\Big], E, t, Rc\Big)$

$$\Rightarrow \begin{cases} \Big(e, n_b, S, F, Pm, Bi, I, G, E, t, < B, D, Y, (M+1), (S+1) >\Big) & \text{falls } G\Big[n = true\Big] \\ \Big(e_0, n_n, S, F, Pm, Bi, I, G, E, t, Rc\Big) & \text{falls } G\Big[n = false\Big] \\ \Big(e, n', shrink(S), F, shrink(Pm), shrink(Bi), I, & \\ \quad G\Big[n' = \overline{\Pi}^{|get(S)|,|get(Pm)|,|get(Bi)|} get(S)\, get(Pm)\, get(Bi)\Big], E, t, Rc\Big) & \text{sonst} \end{cases}$$

mit $shrink(\Big[\%.S\Big]) \Rightarrow \Big[\%.S\Big]$

$shrink(\Big[A.S\Big]) \Rightarrow shrink(S)$

und $get(\Big[\%.S\Big]) \Rightarrow \{\}$

$get(\Big[A.S\Big]) \Rightarrow \{A\} \cup get(S)$

(41) $\Big(e, n, \Big[\%.(p_1, n').S\Big], \Big[(p, w).\%.(n_f, e_f, f).F\Big], Pm, Bi, I, G\Big[n_f = \Sigma \ldots\Big], E, f, Rc\Big)$

mit $G\Big[n_f = \Sigma n_{tup} n_{cons} n_{ccons} n_{nil} n_{true} n_{false} n_{int} n_{string} n_{default}\Big]$

$$\Rightarrow \begin{cases} \Big(e, n_{tup}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \text{falls } n = \text{tup}^k \wedge k \geq 1 \\ \Big(e, n_{cons}, \Big[(p_l, n).(p_r, n).S\Big], F, & \text{falls } n = \overline{\texttt{cons}} p_l p_r \\ \quad \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \\ \Big(e, n_{ccons}, \Big[(p_l, n).(p_r, n).S\Big], F, & \text{falls } n = \overline{\texttt{ccons}} p_l p_r \\ \quad \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \\ \Big(e, n_{nil}, S, F, \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \text{falls } n = \text{tup}^0 \\ \Big(e, n_{true}, S, F, \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \text{falls } n = \texttt{bool}^{true} \\ \Big(e, n_{false}, S, F, \Big[(p_1, n').Pm\Big], Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \text{falls } n = \texttt{bool}^{false} \\ \Big(e, n_{int}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \text{falls } n = \texttt{int}^i \\ \Big(e, n_{string}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \text{falls } n = \texttt{string}^{`s`} \\ \Big(e, n_{default}, \Big[(p_1, n').S\Big], F, Pm, Bi, I\Big[p = (n, e)\Big], G, E, f, Rc\Big) & \text{sonst} \end{cases}$$

E Bedienungsanleitung

NAME

lisa, glisa, xlisa, nlisa — Lazy Evaluator

SYNTAX

lisa [*options*] *file ...*
glisa [*options*] *file ...*
xlisa [*options*] *file ...*
nlisa [*options*] *file ...*

ALLGEMEIN

lisa stellt den Ausführungsmechanismus des Lazy Evaluators dar, wie er in dieser Arbeit beschrieben wurde. Die Implementierung in C der **lisa** existiert in vier verschiedenen Ausbaustufen:

(1) **nlisa** ist der elementare Ausführungsmechanismus der Eingaben nur aus einem Eingabefile oder von der Commandozeile entgegen nehmen kann.

(2) **xlisa** ist äquivalent zur **nlisa**, jedoch um eine X–Windows Oberfäche erweitert, so daß ein Menu zur Verfügung gestellt wird, wie es auch in einer interaktiven Version der $\mathcal{L}i$sa mit Editor erforderlich wäre.

(3) **glisa** ist zusätzlich um einer graphische Fensteroberfäche, basierend auf X–Windows, erweitert, die die interne Reduktionstätigkeit darstellt (Zustandstransformationen). Dazu gehört die Anzeige der Stacks und des zu reduzierenden Ausdrucks mit Enviroment und ggf. einiger Deskriptoren.

(4) **lisa** stellt eine Debugoberfläche zur Verfügung, die dem Benutzer ein Maximum an Eingriffsmöglichkeiten in den Reduktionsprozeß ermöglichen. So kann z.B. der Inhalt der Stacks oder Graphknoten manipuliert werden.

Als Eingabe werden grundsätzlich Dateien genommen, deren Inhalt interpretiert, ausgewertet und ev. modifiziert wird. Alternativ ist als Eingabedatei auch die Console möglich **stdin** . Die X–Windows–Oberfäche nutzt das Interface des Editors AnnE [Seeg91], der Routinen für Fensterverwaltung und Menubehandlung zur Verfügung stellt.

OPTIONEN

Alle angegebenen Dateien werden in gleicher Weise bearbeitet. Der Arbeitsmodus wird über die Angabe von Optionen beim Aufruf festgelegt. Standardmäßig werden die angegebenen Files als preprocesste Graphen beliebiger Ausdrücke geladen, in der **lisa** und **glisa** kann bei Bedarf reduziert werden. Durch Optionen kann von der Voreinstellung abgewichen werden. Die Optionen sind für alle vier Versionen gleich:

–h zeigt die möglichen Optionen beim Aufruf an.

–#*dbug_parm* setzt die Parameter für das Dbug-Toolkit, sofern dies bei der Compilation dazugebunden wurde.

–i*initfile*	definiert ein optionales Konfigurationsfile, aus dem die Dimensionierung des Reduktionssystems gelesen wird.
–p	interpretiert das Eingabefile als Asciifile, das vor der Reduktion durch den gebootstrapten Preprocessor bearbeitet wird.
–r	reduziert das Eingabefile.
–o[gsp]	schreibt das Ergebnis in eine Datei. **–og** oder **–o** gibt das Resultat im Format eines preprocessten Graphen aus. **–os** schreibt den gesamten Zustand der Maschine mit dem Stacksystem in die Datei, **–op** nimmt den Graphen des Resultats als Eingabe für den gebootstrapten Postprocessor, um das Ergebnis als Asciidatei zu erhalten.
–x	erweitert den Namen der Ausgabedateien um ein Appendix **.out** , um ggf. Überschneidungen von Eingabe– und Ausgabefile zu vermeiden.

DATEIFORMATE

lisa (steht als Synonym für (**n** | **x** | **glisa**) stellt drei verschiedene Dateiklassen zur Verfügung: Ascii, Graph und Zustand. Für jede Klasse existieren eigene **Libraries**, so daß die Files in verschiedenen Directories abgelegt werden können, standardmäßig sind dies **pp/** , **lg/** und **st/** ausgehend von aktuellen Directory. Ebenso hat jede Klasse ein eigenes Appendix, standardmäßig sind dies **.pp** , **.lg** und **.st**. Vor dem Laden einer Datei wird das entsprechende Appendix ergänzt, wenn keines angegeben wurde, ebenso wird der Pfadname der Library ergänzt.

KONFIGURATIONSDATEI

Vor Beginn der Auswertung lädt **lisa** einen Konfigurationsfile in dem die Größen des Graphspeichers, der Stacks und die Anzahl der Reduktionsschritte festgelegt werden. Weiterhin werden hier alternative Libraries und Appendices definiert. Die Einträge haben das Format von Zuweisungen an Shellvariablen. Folgende Eintäge sind vorgesehen (in Klammern sind die Defaultwerte angegeben):

redcnt_beta	(10^8) Anzahl der β–Reduktionen
redcnt_delta	(10^8) Anzahl der δ–Reduktionen
redcnt_alpha	(10^8) Anzahl der Rekursionen (α–Reduktionen)
redcnt_pi	(10^8) Anzahl der Patternmatches (π–Reduktionen)
redcnt	(10^8) Anzahl der Reduktionsschritte insgesamt
steps	(10^8) Anzahl der Transformationsschritte
mode	(WNF) Normalform
sstacksize	(50000) Größe des S–Stacks in Worten
fstacksize	(50000) Größe des F–Stacks in Worten
pmstacksize	(20000) Größe des Pm–Stacks in Worten
bistacksize	(10000) Größe des Bi–Stacks in Worten
heapsize	(1 MByte) Größe des Heaps oder Graphen
heapdes	(30000) Anzahl der Graphknoten incl. der Environments und Indirektionsknoten

lowdesc	(10) Hochwassergrenze für Garbagecollection
lowheap	(1000) Hochwassergrenze für Garbagecollection und Heapcompaction
lowdescload	(500) Hochwassergrenze für Garbagecollection vor Ladeoperation
ascii_lib	(“pp/“) Library für Ascii-Programme
ascii_ext	(“.pp“) Appendix für Ascii-Programme
graph_lib	(“lg/“) Library für preprocesste Programmgraphen
graph_ext	(“.lg“) Appendix für preprocesste Programmgraphen
state_lib	(“st/“) Library für vollständige Zustände
state_ext	(“.st“) Appendix für vollständige Zustände
output_ext	(“.out“) Appendix für **-x** Option

FILES

lisa.init	Konfigurationsfile
large.init	Konfigurationsfile für Pre– und Postprocessor
lg/TMP.lg	Zwischenspeicher für Pre– und Postproceesor

BEISPIEL

Es existiert eine Datei **pp/fac.pp** in der die Definition der Fakultätsfunktion in $\mathcal{L}$KiR–Noatation steht. Diese Datei läßt sich als Eingabe für $\mathcal{L}i$SA verwenden.

Beispielsession für Fakultät

```
haegar% cat pp/fac.pp
def
   fac[n] =if (n le 1)
           then 1
           else (n * fac[(n - 1)])
in  fac[5]

haegar% lisa -popx fac
nlisa version 1.15a
loading pp/fac.pp ...ready.
loading lg/Preprocessor.lg ...ready.
reduction completed      [des Preprocessings]
reducing ...
reduction completed      [des Auswertung von fac[5]]
loading lg/Postprocessor.lg ...ready.
reduction completed      [des Postprocessings]
saving pp/fac.shd.out ...ready.
...
haegar% cat pp/fac.pp.out
120
haegar%
```

In dem **lisa**–Lauf sind diverse Ausgaben weggelassen worden, da sie nicht zum Verständnis beitragen. Hervorzuheben sind nur die Zeitangaben nach jedem abgeschlossenen Reduktionszyklus, die detailiert Auskunft über die benötigte Reduktionszeit, Ladezeit und Zeit für Garbagecollections sowie Heapcompactions geben. Hätte die Option **-x** gefehlt, wäre das Resultat 120 in die Datei **pp/fac.pp** geschrieben worden und hätte die Quelldatei vernichtet. Temporär angelegte Dateien erhalten immer eindeutige Namen.

F Literaturverzeichnis

[Aho88] A.V. Aho; R.Sethi; J.D. Ullmann. **Compilerbau** Addison Wesley, 1988.

[Augu85] L. Augustsson. **Compiling Pattern Matching.** FPCA, Nancy, *LNCS 201*, pp. 301-324, 1985.

[Back78] J. Backus. **Can programming be liberated from the von Neumann style ?** *A functional style and its algebra of programs CACM*, 1978, Vol. 21, No. 8, pp. 613-641.

[Bare84] H.P. Barendregt. **The lambda calculus** Studies in Logic, Vol. 103; North-Holland, 1984.

[Bark89] H. Barkowski. **Implementierung von Patternmatching Funktionen in ein Reduktionssystem** *Diplomarbeit, Universität Kiel*, 1989.

[Berk76] K.J.Berkling. **A Symmetric Complement to the Lambda Calculus** Internal Report GMD ISF-76-7, Sankt Augustin, September 1976.

[Bird84] R.S. Bird. **Using Circular Programs to Eliminate Multiple Traversals of Data** *Acta Informatica 21*, pp.239-250, 1984.

[Bird88] R.S. Bird; P.L. Wadler. **Functional Programming** Prentice Hall, 1988.

[Bloe86] H. Blödorn. **Entwicklung einer Systemumgebung für eine Heap-unterstützte Datentyp-Reduktionsmaschine** *Diplomarbeit, Universität Bonn*, 1986

[Bloe89] H. Blödorn. K*i*R**- The Kiel Reduction Language Users Guide** *internes Skript der Universität Kiel*, 1989.

[Burg75] W.H. Burge; T.J. Watson. **Recursive programming techniques** Addison Wesley 1975

[Card85] P. Cardelli; L. Wegner. **On understanding types, data abstractions and polymorphisms.** *Computing Surveys*, 17(4):pp.471-522, 1985.

[Chur41] A. Church. **The Calculi of Lambda Conversion** Annals of Mathematic Studies, No. 6, Princeton University Press, New Jersey, 1941.

[Curr58] H.B. Curry; R. Feys. **Combinatory logic** North Holland; 1958

[DeBr72] N.G. De Bruijn. **Lambda–Calculus Notation with Nameless Dummies. A Tool for Automatic Formula Manipulation with Application to the Church–Rosser–Theorem** *Indigationes Matematicae 34*, 381–392, 1972.

[Fair] S. Fairbairn, J. Wray. **Tim: A simple, lazy abstract machine to execute supercombinators.** Technical report, University Of Cambridge.

[Fiel88] A.J. Field; P.G. Harrison. **Functional Programming** Addison–Wesley, 1988.

[Haud89] P. Haudak. **Conception, evolution and application of functional programming languages. ACM Computing Surveys**, 21(3):359–411, 1989.

[Hask90] P. Haudak et al. **Report On The Programming Language Haskell**, 1990.

[Hend80] P. Henderson. **Functional Programming - Application and Implementation** Prentice Hall; 1980.

[Hind86] J.R. Hindley; J.P. Seldin. **Introduction to combinators and λ-calculus** London Mathematical Society, 1986.

[John84] T. Johnson. **Efficient compilation of lazy evaluation** *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction* , pp. 58–69, Montreal 1984.

[John85] T. Johnson. **Lambda Lifting: Transforming Programs into Recursive Equations** FPLCA, *LNCS 201*, Nancy, September 1985.

[John86] T. Johnson. **Taget Code Generation from G–Machine Code** Graph Reduction, *LNCS 279*, 1986.

[Klug86] W. Kluge; C. Schmittgen. **Reduction Languages and Reduction Systems** LNCS 272, pp. 153-184, 1986.

[Kogg91] P.M. Kogge. **The Architecture Of Symbolic Computers** Mc Graw Hill, 1991.

[Lavi87] A. Laville. **Lazy Pattern Matching in the ML Language.** INRIA-Report 664, Universite de Reims, 1987.

[Leic89] B. Leick. **Realisierung eines neuartigen Bindungskonzeptes in einem heap-unterstützten Reduktionssystem unter besonderer Berücksichtigung von Pattern-Matching-Funktionen** *Diplomarbeit, Universität Kiel*, 1989.

[Miln78] R. Milner. **A theory of type polymorphism in programming.** *Journal Of Computer And System Science*, 17, 1978.

[Mira87] Research-Software. **Miranda System Manual**, 1987.

[ML87] D. M. Queen et al. **Functional Programming In ML**, 1987.

[PJ87] S. L. Peyton Jones. **The Implementation Of Functional Programming Languages**. Prentice Hall, London, 1987.

[Rath91] C. Rathsack. **$\mathcal{L}$isa- Realisierung eines interaktiven Lazy Evaluators** *Diplomarbeit, Universität Kiel*, 1991.

[Sche86] J. Schepers; R. Zimmer. **Emulation einer Datentyp Reduktionsmaschine mit Heapkonzept** *Diplomarbeit, Universität Bonn*, 1986.

[Seeg91] M. Seeger. **Enwurf und Entwicklung eines syntaxgesteuerten graphikorientierten Editors** *Diplomarbeit, Universität Kiel*, 1991.

[Stra89] U. Strack. **Realisierung eines neuartigen Bindungskonzeptes in einem heap-unterstützten Reduktionssystem unter besonderer Berücksichtigung rekursiver Funktionen** *Diplomarbeit, Universität Kiel*, 1989.

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 30.September 1991