B. Operationale Modellierung der Proemialrelation

1 Einführung

Die von Gotthard Günther konzipierte und von Rudolf Kaehr verallgemeinerte *Proemialrelation* stellt eine der grundlegenden *transklassischen* Begriffsbildungen der Polykontexturalitätstheorie dar [1].

Im folgenden wird ein operationales Modell der Proemialrelation vorgeschlagen, das in Anlehnung an Implementierungstechniken funktionaler Programmiersprachen entwickelt wird. Insbesondere wird ein *Proemialkombinator PR* definiert, dessen operationale Semantik mittels einer parallelverarbeitenden Graphreduktionsmaschine angegeben wird.

Diese Modellierung expliziert die Proemialrelation im Kontext der Grundlagenuntersuchungen der Mathematik, Logik und Informatik und weist auf die Relevanz der Polykontexturalitätstheorie für aktuelle Fragestellungen der KI-Forschung hin.

2 Polykontexturale Logik als Distribution und Vermittlung formaler Systeme

Die von G. Günther konzipierte *Polykontexturale Logik* (PKL) ging aus seinen umfangreichen Untersuchungen zur Formalisierung der Dialektik Hegels und der Reflexionstheorie Fichtes[2]

und seinen Arbeiten zur Formalisierung selbstreferentieller Systeme innerhalb der 'Second Order Cybernetics'[3] hervor.

---

[1] Günther, G.: *Erkennen und Wollen.* In: *Gotthard Günther und die Folgen.* Klagenfurt.

Kaehr, R.: *Materialien zur Formalisierung der dialektischen Logik und der Morphogrammatik 1973-1975.* In: Günther, G.: *Idee und Grundri einer nicht-aristotelischen Logik*, Anhang.

Kaehr, R.: *Einschreiben in Zukunft.* In: Hombach, D.(Hg.): *Zeta 01. Zukunft als Gegenwart.* Berlin, Verlag Rotation, 1982.

Kaehr, R.: *Disseminatorik: Zur Logik der 'Second Order Cybernetics'. Von den 'Laws of Form' zur Logik der Reflexionsform.* In: Baecker, D. (Hg.): *Kalkül der Form.*, Frankfurt a. M., Suhrkamp 1994.

[2] Günther, G.: *Grundzüge einer neuen Theorie des Denkens in Hegels Logik.* Hamburg, Felix Meiner Verlag, 1978.,

Günther, G.: *Idee und Grundriß einer nicht-aristotelischen Logik. Die Idee und ihre philosophischen Voraussetzungen.* 2. Auflage, Hamburg, Verlag Felix Meiner, 1978.,

Günther, G.: *Beitrge zur Grundlegung einer operationsfhigen Dialektik. Bd. 1-3, Hamburg, Verlag Felix Meiner, 1976, 1978, 1980* Bd.1.

[3] Günther, G.: *Cybernetic Ontology and Transjunctional Operations*, in Günther (1980), FN 5

Ausgehend von einer an die Philosophie des deutschen Idealismus anschließende Kritik der klassischen Logik, ihrer zugrundeliegenden Ontologie und der von ihr implizierten dualistischen logischen Form[4] entwirft Günther eine polykontexturale Ontologie. Die Ontologie der klassischen Logik reduziert die phänomenologisch beobachtbare Vielzahl der Subjekte auf eine einzige universale Subjektivität, die strikt von der objektiven Welt isoliert ist. Im Gegensatz dazu nimmt Günthers polykontexturale Ontologie „*die Immanenz der Subjektivität in der Welt*“[5], sowie „*die Irreduzibilität von Ich–Subjektivität und Du–Subjektivität aufeinander in einem universalen Subjekt*“[6] an.

Diese grundlegende Ontologie der Subjektivität will Günther in seiner Polykontexturalen Logik formalisieren, deren Anspruch er präzisiert: „*Jedes Einzelsubjekt begreift die Welt mit der selben Logik, aber es begreift sie von einer anderen Stelle im Sein. Die Folge davon ist: insofern, als alle Subjekte die gleiche Logik benutzen, sind ihre Resultate gleich, insofern aber, als die Anwendung von unterschiedlichen Stellen her geschieht, sind ihre Resultate verschieden. [...] Ein logischer Formalismus [hat] nicht einfach zwischen Subjekt und Objekt zu unterscheiden, er muß vielmehr die Distribution der Subjektivität in eine Vielzahl von Ichzentren in Betracht ziehen. Das aber bedeutet, daß das zweiwertige Verhältnis sich in einer Vielzahl von ontologischen Stellen abspielt, die nicht miteinander zur Deckung gebracht werden können.*“[7] Die PKL erkennt also die zweiwertige Logik als Formulierung der ontologischen Stellung eines einzelnen Subjektes zu *seiner* objektiven Welt an. Die Monokontexturalität der klassischen Logik wird nun nicht durch *intrakontexturale* Modifikationen, sondern durch die *extrakontexturale* Ergänzung einer Vielzahl weiterer Kontexturen erweitert.

Die PKL bildet komplexe Systeme ab, indem sie deren Subsysteme über mehrere logische Kontexturen verteilt. Innerhalb dieser lokalen Subsysteme, *intrakontextural*, gilt jeweils eine klassische Logik. Die verteilten Logiken sind durch bestimmte ausserlogische Operationen miteinander verkoppelt, wodurch die Interaktion der das Gesamtsystem konstituierenden Subsysteme modelliert wird.

Die PKL zeichnet sich also durch eine *Distribution* und *Vermittlung* verschiedener logischer Kontexturen aus, wobei innerhalb einer Kontextur — *intra-kontextural* — alle Regeln der klassischen Aussagenlogik ihre vollständige Gültigkeit besitzen. Durch die Vermittlung sind die einzelnen Kontexturen nicht im Sinne einer Typenhierarchie voneinander isoliert, sondern durch besondere *inter-kontexturale* Übergänge miteinander verkoppelt. Da sowohl die Kontexturen in sich, als auch ihre Vermittlung widerspruchsfrei beschreibbar ist, lassen sich somit zirkuläre und selbstreferentielle Strukturen innerhalb der PKL widerspruchsfrei modellieren.

Die innerhalb der monokontexturalen klassischen Logik ausschließlich gelten-

[4] Günther (1933), FN 55; Günther (1978), FN 5
[5] Günther (1980), FN 5, S. 87.
[6] A.a.O.
[7] A.a.O

de Form der Dualität ist in der PKL nur noch lokal (je Kontextur) gültig, nicht mehr jedoch ausschließlich bzw. global (für das Gesamtsystem). Die strukturelle Erweiterung der dualistischen Form der klassischen Logik zur *Reflexionsform* der PKL ergibt sich aus dem *proemiellen* (d.h. simultanen und verkoppelten) Zusammenspiel der Distribution und Vermittlung von Kontexturen und ist auf keinen dieser Aspekte allein reduzierbar. Die Distribution regelt die Verteilung von Logiken über Kontexturen und läßt dabei den interkontexturalen Raum unberücksichtigt. Die Vermittlung beschreibt hingegen die interkontexturale Operativität der PKL, kann aber die intrakontexturalen, logischen Aspekte nicht erfassen, da sie von aller logischen Wertigkeit abstrahieren muß. Die Distribution bildet den Hintergrund der Vermittlung, die Vermittlung den Hintergrund der Distribution. Der formale Aufbau der PKL muß dieser fundamental selbstreferentiellen Architektur Rechnung tragen und das proemielle Zusammenspiel der Distribution und Vermittlung vollständig abbilden.

Es soll im weiteren Verlauf eine operationale Modellierung der Proemialrelation entwickelt werden, da diese, wie gezeigt, die fundamentale Struktur der Polykontexturalen Logik bildet. Im nächsten Abschnitt wird die Proemialrelation zunächst informell charakterisiert.

3 Die Proemialrelation

Die von Günther eingeführte *Proemialrelation „gehört zur Ebene der kenogrammatischen Strukturen“*[8] und beschreibt die in der Kenogrammatik[9] mögliche Notierung von 'Relationalität' und Operativität, die aller dualistischer Aufspaltung in Subjekt-Objekt, Form-Inhalt usw. *vorangeht*[10] *„Es gibt einen deutlichen Unterschied zwischen der symmetrischen Umtauschrelation, wie sie zum Beispiel die Negationstafel in der zweiwertigen Logik darstellt, und dem Umtausch von Relator und Relatum. In der klassischen Symmetrierelation wechseln die beiden Relata lediglich ihre Plätze. Formal ausgedrückt:*

$$R(x, y) \tag{1}$$

wird zu

$$R(y, x) \tag{2}$$

Hierbei ändert sich materiell überhaupt nichts. Wenn dagegen der Relator die Stelle eines Relatums einnimmt, dann ist der Umtausch nicht wechselseitig. Der Relator kann zum Relatum werden, doch nicht in der Relation, für die er zuvor

[8] Günther, G.: *Cognition and Volition*. In: Günther (1978), FN 5

[9] Gr. kenos: leer. Zur Einführung, Formalisierung und Implementierung der Kenogrammatik vgl. Mahler, Th.: *Morphogrammatik in Darstellung, Analyse, Implementierung und Applikation*. Arbeitsbericht Nr. 1 des Forschungsprojektes „Theorie komplexer biologischer Systeme“, Ruhr Universität Bochum, 1993.

[10] Gr. prooimion: Vorspiel

die Beziehung einrichtete, sondern nur relativ zu einem Verhältnis bzw. Relator höherer Ordnung.

Umgekehrt kann das Relatum zum Relator werden, jedoch nicht in Bezug auf das Verhältnis, in dem es als relationales Glied — als Relatum — aufgetreten ist, sondern nur in Bezug auf Relata niedrigerer Ordnung. Wenn

$$R_{i+1}(x_i, y_i)$$

gegeben ist und das Relatum x oder y zum Relator wird, dann erhalten wir

$$R_i(x_{i-1}, y_{i-1})$$

wobei $R_i = x_i$ oder y_i ist. Wenn dagegen der Relator zu einem Relatum wird, dann erhalten wir

$$R_{i+2}(x_{i+1}, y_{i+1})$$

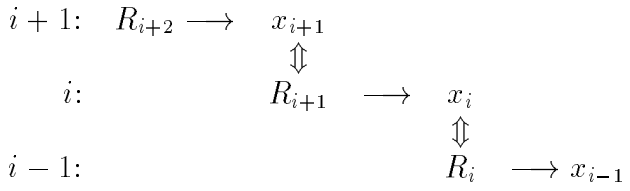
wobei $R_{i+1} = x_{i+1}$ oder y_{i+1} ist. Der Index i bezeichnet höhere oder niedrigere logische Ordnung. Wir nennen diese Verbindung zwischen Relator und Relatum das Proemialverhältnis, *da es der symmetrischen Umtauschrelation und der Ordnungsrelation vorangeht und — wie wir sehen werden — ihre gemeinsame Grundlage bildet.*"([?], op. cit., S.33)

Die Ordnungsrelation bestimmt also immer das hierarchische Verhältnis des Relators R_{i+1} zum Relatum x_i innerhalb einer logischen Stufe i. Die Umtauschrelation bezieht sich auf den Wechsel zwischen dem Relator R_i einer Relation der Stufe $i-1$ und dem Relatum x_i der logischen Stufe i. Dieser Zusammenhang ist im folgenden Diagramm (1) abgebildet. Die Proemialrelation stellt sich somit als ein ineinandergreifender Mechanismus von Umtausch und Ordnung dar und kann in zweifacher Weise interpretiert werden. Zum Einen läßt sich Proemialität als ein Umtausch deuten, der auf Ordnung basiert. Die Ordnung innerhalb einer logischen Stufe i ist jedoch dadurch begründet, daß ein Relator R_i der Stufe $i-1$ durch die Umtauschrelation zum Relatum x_i der Stufe i wird und daß gleichfalls der Relator R_{i+1} durch Umtausch des Relatums x_{i+1} auf die Stufe i versetzt wird. Somit ist zum Anderen Proemialität auch als Ordnung zu verstehen, die auf Umtausch gründet.

„Weder die Umtauschrelation noch die Ordnungsrelation wären uns begreiflich, wenn unsere Subjektivität nicht in der Lage wäre, zwischen einem Relator überhaupt und einem einzelnen Relatum ein Verhältnis herzustellen. Auf diese Weise stellt das Proemialverhältnis eine tiefere Fundierung der Logik bereit, als ein abstraktes Potential, aus dem die klassischen Relationen des symmetrischen Umtauschs und der proportionalen Ordnung hervorgehen.

Dies ist so, weil das Proemialverhältnis jede Relation als solche konstituiert. Es definiert den Unterschied zwischen Relation und Einheit oder — was das gleiche ist — zwischen der Unterscheidung und dem was unterschieden ist — was wiederum das gleiche ist — wie der Unterschied zwischen Subjekt und Objekt."([11])

[11] Günther, FN 13



R_i: Relator R_i der Stufe i
x_i: Relatum x_i der Stufe i
$\longrightarrow$: Ordnungsrelation zwischen einem Relator R_i und einem Relatum x_{i-1}
$\Updownarrow$: Umtauschrelation zwischen einem Relator R_i und einem Relatum x_i

Abbildung 1:
Die Proemialrelation

„*Der von der Proemialrelation bewirkte Umtausch ist einer zwischen höherer und niedrigerer relationaler Ordnung. Wir können beispielsweise ein Atom als Relation zwischen mehreren Elementarpartikeln betrachten, wobei letztere dann den Part der Relata einnehmen. Wir können jedoch auch sagen, daß das Atom ein Relatum in einer komplexeren Ordnung darstellt, die wir als Molekül bezeichnen. Folglich ist ein Atom beides: ein Relator relativ zu den Elementarpartikeln, jedoch kann es diese Eigenschaft mit der eines Relatums vertauschen, wenn wir es innerhalb der umfassenderen Relation (Relator) eines Moleküls betrachten.*“([12])

Die Proemialität läßt sich somit als eine Begriffsbildung verstehen, die es ermöglicht, ein bestimmtes Objekt auf mehrere logische Stufen (Bezugssysteme) verteilt in verschiedenen Funktionalitäten zu erfassen. Der fundamentale Unterschied zwischen der Proemialrelation wie sie von Günther intendiert ist und klassischen Konzepten logischer Stufung in Objekt- und Metaebenen ist die *Simultaneität* der Ebenen innerhalb der Proemialrelation, die sich der klassischen Darstellung entzieht. So ist ja das von Günther angeführte Beispiel des Atoms so zu verstehen, daß das Atom *simultan* Relatum (bzgl. des Moleküls) und Relator (bzgl. seiner Partikel) ist, und sich durch dieses Zusammenwirken erst konstituiert.

Die *zeitliche* und *räumliche* Simultaneität der logischen Bezugssysteme macht das eigentliche Formalisierungsproblem der Proemialrelation aus. Im klassischen Kalkülkonzept gibt es die Begriffe der Zeitlichkeit und Räumlichkeit formaler Systeme nicht. Daß ein bestimmter formaler Prozeß zu einer bestimmten Zeit an einem bestimmten Ort stattfindet, wird von klassischen Kalkülen nicht grundsätzlich thematisiert, sondern findet allenfalls als nachgeschobene sekundäre Interpretation Berücksichtigung. So wird von der obigen Abbildung

[12]Günther, FN 13, S. 34

1 und den ihr entsprechenden Formalismen zwar die *logische* Struktur der Proemialrelation erfaßt, nicht jedoch deren zeitliche und räumlichen Aspekte. Ein der Güntherschen Konzeption angemessenes operationsfähiges Modell der Proemialrelation muß also sowohl deren logische als auch deren zeitliche und räumliche Struktur abbilden. Dabei kann jedoch nicht auf klassische Raum- und zeitmodelle zurckgegriffen werden, da diese ja wiederum in klassischen Formalismen abgebildet werden. Fr eine angemessene Modellierung sind daher *transklassische* Modellierungen erforderlich.

Dem hier vorgeschlagenen Modell liegt eine parallelverarbeitende Graphreduktionsmaschine zur Implementierung funktionaler Programmiersprachen zugrunde. Zunächst wird eine knappe Einführung in den λ-Kalkül, die grundlegende Theorie funktionaler Programmiersprachen, gegeben (Abschnitt 4). In Abschnitt 5 wird die parallelisierte Graphreduktion als eine Implementierungstechnik λ-Kalkül-basierter Programmiersprachen entwickelt. In Abschnitt 6 wird dann die Proemialrelation durch eine Erweiterung der operationalen Semantik der parallelen Graphreduktion modelliert. Es werden hier außerdem mögliche Anwendungen sowie die Beschränkungen der Modelliering skizziert.

Als Implementierungssprache wurde ML (Meta Language) gewählt, das sich aus mehreren Gründen sehr zur Implementierung der hier entwickelten Formalismen eignet[13].

[13]ML wurde ursprünglich als Spezifikations- und Metasprache für automatische Beweissysteme entwickelt. Aufgrund seiner fortschrittlichen Konzeption, seiner exakten, sicheren, übersichtlichen und effizienten Struktur wird ML inzwischen in vielen Bereichen der Informatik angewandt und neben anderen funktionalen Sprachen (Miranda, Scheme, FP, Haskell) an einer wachsenden Zahl von Universitäten als erste Programmiersprache gelehrt. Zu den fortschrittlichen Konzepten von ML gehören das strikte Typechecking bei Typpolymorphie, Pattern Matching, ein hochentwickeltes Modulkonzept, separate und inkrementelle Compilierung, Interaktivität und Portabilität. Von der Möglichkeit, in ML neben allen Konzepten rein funktionaler Sprachen auch imperative Konstrukte (wie Zuweisungen und verzeigerte Strukturen) zu benutzen, wird bei der Implementierung der Graphreduktionsmaschine ebenfalls Gebrauch gemacht.

In ML lassen sich zu mathematischen Definitionen äquivalente, d.h. lediglich syntaktisch verschiedene Implementierungen erstellen, die einerseits als Programm ausführbar sind, andererseits aber auch gut lesbare Definitionen und Spezifikationen darstellen. Die in dieser Arbeit entwickelten Implementierungen sind daher nicht als effiziente Programme gedacht, die bestimmte fixierte Probleme mit einem minimalen Resourcenaufwand lösen, sondern als formale Definitionen und Notationen in einer formalen Metasprache (eben ML), die in einem gewissen Rahmen auch zu direkten Berechnungen benutzt werden können.

Zur Einarbeitung in die Programmiersprache ML sei das Buch '*ML for the Working Programmer*' Paulson, L.C.: *ML for the Working Programmer*. University of Cambridge, Computer Laboratory, Cambridge University Press, 1991. , das auch Bezugsadressen für kostenlose ML–Implementierungen enthält, besonders empfohlen.

4 Der λ-Kalkül

Turing-Maschinen, rekursive Funktionen und Register Maschinen sind formale Modelle der Berechenbarkeit. Der von Alonzo Church [14] entwickelte λ-Kalkül ist eine der frühesten Theorien der berechenbaren Funktionen und bildet heute die formale Grundlage der funktionalen Programmiersprachen und ihrer Implementierungen.

Im Gegensatz zur naiven Mengenlehre, die in einer eher statischen Sichtweise Funktionen als spezielle Teilmengen kartesischer Produkte auffaßt, interpretiert der λ-Kalkül Funktionen als Berechnungsvorschriften und betont so die dynamischen Aspekte einer von den Argumenten der Funktion ausgehenden Berechnung der Funktionswerte.

4.1 λ-Terme

λ-Terme werden aus Variablen und anderen λ-Termen rekursiv aufgebaut.

Definition 4.1 (λ-Term) *Sei $\mathcal{V}$ eine abzählbare Menge von Variablensymbolen. Dann gilt:*

1. *Jede Variable $x \in \mathcal{V}$ ist ein λ-Term.*
2. *Wenn t ein λ-Term und $x \in \mathcal{V}$ ist, dann ist auch $(\lambda x.t)$ ein λ-Term (Abstraktion).*
3. *Sind t und u λ-Terme, dann ist auch $(t\ u)$ ein λ-Term (Applikation).*

Eine Funktionsabstraktion $(\lambda x.t)$ ist das Modell einer *anonymen Funktion*, x ist die *gebundene Variable* und t der *Rumpf* der Funktion. Jedes Vorkommen von x in t ist durch die Abstraktion *gebunden*. Entsprechend ist das Vorkommen einer Variablen y *frei*, wenn es nicht innerhalb des Rumpfes u einer Abstraktion $(\lambda y.u)$ geschieht. In dem Term $(\lambda z.(\lambda x.(y\ x)))$ ist beispielsweise x gebunden und y frei.

Eine Funktionsapplikation $(t\ u)$ modelliert die Anwendung der Funktion t auf ein Argument u.

λ-Terme werden in der folgenden Implementierung als rekursiver Datentyp `term` repräsentiert:

```
datatype term = Free  of string
              | Bound of string
              | Abs   of string*term
              | Apply of term*term;
```

[14] Church, A.: *The Calculi of Lambda–Conversion*. Princeton University Press, 1951.

Der reine λ-Kalkül enthält keine Konstanten. Zahlen und andere in Programmiersprachen gebräuchliche Konstanten können durch bestimmte λ-Terme modelliert werden. Obwohl verzichtbar, werden im folgenden aus Gründen der Effizienz und Übersichtlichkeit auch Konstanten als Terme zugelassen. Der Ausdruck (+ 1 2) ist dann ein λ-Term, weil er eine Applikation der konstanten Operation + auf die Konstanten 1 und 2 notiert. Diese Erweiterung führt zu folgender ML Definition der λ-Terme:

```
datatype term = Free  of string
              | Bound of string
              | Int of int
              | Op of string
              | Abs   of string*term
              | Apply of term*term;
```

In der weiteren Darstellung werden die gebräuchlichen Konventionen zur Darstellung von λ-Termen benutzt: $x, y, t, \ldots$ sind Variablensymbole. Klammern werden nach Möglichkeit fortgelassen. So wird:

$$(\lambda x.(\lambda y.(\ldots(\lambda z.(E))\ldots)))$$

als $\lambda xyz.E$ geschrieben. Der Ausdruck:

$$(\ldots((E_1 E_2)E_3)\ldots E_n)$$

wird abkürzend notiert als $E_1 E_2 E_3 \ldots E_n$.

4.2 λ-Konversionen

λ-Konversionen sind Regeln zur symbolischen Transformation, nach denen λ-Terme unter Bewahrung ihrer intuitiven Bedeutung umgeformt werden können.

Die α-Konversion benennt die gebundene Variable einer Abstraktion um:

$$(\lambda x.t) \Longrightarrow_\alpha (\lambda y.t_{[y/x]}).$$

Die Abstraktion über x wird in eine Abstraktion über y umgeformt und jedes Vorkommen von x in t durch y ersetzt. Zwei λ-Terme sind äquivalent, wenn sie durch α-Konversionen in identische Terme umgeformt werden können.

Für das Verständnis funktionaler Programmiersprachen am wichtigsten ist die β-Konversion, die eine Funktionsapplikation umformt, indem sie die gebundene Variable durch das Funktionsargument substituiert:

$$((\lambda x.t)u) \Longrightarrow_\beta t_{[u/x]}.$$

Die β-Konversion entspricht der Substitutionsregel für lokale Variablen in Programmiersprachen.

4.3 λ-Reduktion und Normalformen

Ein Reduktionsschritt $t \Longrightarrow u$ formt den λ-Term t in den Term u durch Anwendung einer β-Konversion auf einen beliebigen Unterterm von t um. Wenn ein λ-Term keine weiteren Reduktionen mehr zuläßt, ist er in *Normalform*. Einen Term zu *normalisieren* heißt also, so lange β-Konversionen auf ihn anzuwenden, bis eine Normalform erreicht ist.

Das **erste Church–Rosser Theorem** [15] besagt, daß die Normalformal eines λ-Terms, falls sie existiert, eindeutig ist. Mit anderen Worten: verschiedene Folgen von Reduktionen, die von einem bestimmten λ-Term ausgehen, müssen äquivalente Normalformen erreichen.

Viele λ-Terme besitzen keine Normalform. DD mit $D := (\lambda x.xx)$ ist ein Beispiel für einen solchen Term, da $DD \Longrightarrow_\beta DD$. Ein Term kann eine Normalform besitzen, auch wenn bestimmte Reduktionsfolgen nicht terminieren. Typisches Beispiel ist etwa das Auftreten eines Subtermes u, der keine Normalform besitzt, der jedoch durch bestimmte Reduktionen eliminiert werden kann.

> **Beispiel:** Die Reduktion
>
> $$\underline{(\lambda x.a)(DD)} \Longrightarrow a$$
>
> erreicht sofort eine Normalform und eliminiert den Term (DD). Dies entspricht dem *Call-by-name* Verhalten von Prozeduren: das Argument wird *nicht* ausgewertet, sondern einfach in den Prozedurrumpf hineinsubstituiert. Wird das Argument hingegen ausgewertet, *bevor* es in den Rumpf eingesetzt wird, entspricht dies der *Call-by-value* Technik. Im obigen Beispiel würde diese Auswertungsstrategie keine Normalform erreichen:
>
> $$(\lambda x.a)\underline{(DD)} \Longrightarrow (\lambda x.a)\underline{(DD)} \Longrightarrow \ldots.$$

Wird wie im ersten Fall immer der am weitesten links befindliche Redex reduziert, so wird immer eine Normalform erreicht, sofern eine solche existiert. Eine solche Reduktion ist dann in *Normalordnung* (normal-order) . Besitzt ein Term t eine Normalform u, dann gibt es eine Reduktion in Normalordnung von t nach u (**zweites Church–Rosser Theorem** [16]). Die die normal-order Reduktion findet daher jede existierende Normalform.

5 Implementierung funktionaler Sprachen

Der im vorhergehenden Abschnitt eingeführte λ-Kalkül ist berechnungsuniversell [17] und kann als Termtransformationssystem (*Stringreduktion*) implementiert

[15] Barendregt, H.P.: *The Lambda-calculus. Its Syntax and Semantics.* North-Holland, 1980.
[16] Barendregt, FN 22
[17] Barendregt, FN 22

werden. Ein solches System ist jedoch extrem ineffizient und zur praktischen Anwendung als Programmiersprache kaum anwendbar [18]. Tatsächliche Implementierungen funktionaler Sprachen bedienen sich häufig abstrakter Maschinen[19]. Der hier gewählte Implementierungsansatz bedient sich einer *Graphreduktionsmaschine*.

Die Auswahl dieser modernen Implementierungstechnik soll kurz begründet werden. Die Graphreduktion ermöglicht die Reduktion von Ausdrücken in Normalordnung (alle existierenden Normalformen werden gefunden). Im Gegensatz zur Call-by-name Evaluation der Termersetzungsmethode müssen Argumente von Funktionen jedoch nicht *textuell* in die Funktionsrümpfe kopiert werden (was zu erheblichen Raum und Zeitaufwand führt), sondern können als verzeigerte Objekte gehandhabt werden. Mehrere Instanzen einer Variablen verweisen so auf ein einziges physikalisches Objekt (*node-sharing*). Diese Technik reduziert durch die Verzeigerung den Speicheraufwand und, da ein bestimmter Knoten nur ein einziges Mal evaluiert werden muß, auch den Zeitaufwand einer Berechnung.

Die Graphreduktion eignet sich desweiteren zu einer einfachen und anschaulichen Implementierung auf Mehrprozess(or)-Systemen. Was ein weiteres Hauptargument für ihre berlegenheit zu klassischen von-Neumann Architekturen ausmacht[20].

Eine Graphreduktion, die direkt auf λ-Ausdrücken operiert, erfordert erheblichen Kopieraufwand, selbst wenn voller Gebrauch von der Node-sharing Technik gemacht wird. Dieser hohe Resourcenaufwand ist auf das Vorhandensein von Variablen zurückzuführen, wobei jede Variablenbindung eine komplette Kopie des Funktionsrumpfes erforderlich macht.[21] Aus diesem Grund werden in der hier gewählten Implementierung λ-Terme zunächst in variablenfreie *Kombinatorausdrücke* übersetzt, für die dann effiziente Graphreduktionsmethoden entwickelt werden.

Die nun folgende Darstellung dieses Verfahrens orientiert sich an den Arbeiten Turners [22] und Davies [23].

---

[18] Vgl. Paulson, FN 20

[19] Abelson und Sussman (Abelson, H., Sussman, G.J., Sussman, J.: *Structure and Interpretation of Computer Programs.* Cambridge, Massachusetts, The MIT Press, 1987.) behandeln die Implementierung von LISP auf einer abstrakten Registermaschine. Die wohl bekannteste abstrakte Maschine zur Implentierung ist die SECD-Maschine (Landin, P.J.: *The Mechanical Evaluation of Expressions.* The Computer Journal, Bd.6, S.308–320, 1964.,

Henderson, P.: *Functional Programming: Application and Implementation.* New York, Prentice Hall, 1980.,

Davies, A.J.T.: *An Introduction to Functional Programming Systems Using Haskell.* Cambridge Computer Science Texts 27, Cambridge University Press, 1992.)

[20] Davis, FN 26, S.1 ff.

[21] Davis, FN 26, S. 154.

[22] Turner, D.A.: *A New Implementation Technique for Applicative Languages.* Software Practise and Experience Bd. 9, 1979.

[23] Davies, A.J.T.: *An Introduction to Functional Programming Systems Using Haskell.* Cambridge Computer Science Texts 27, Cambridge University Press, 1992.

5.1 Übersetzung von λ-Ausdrücken

5.1.1 Kombinatoren

Funktionen wie etwa:

$$S\ x\ y\ z = (x\ z)(y\ z), \tag{3}$$
$$K\ x\ y = x, \tag{4}$$
$$I\ x = x, \tag{5}$$

enthalten keine freien Variablen. Solche Funktionen werden *Kombinatoren* genannt. Der Kombinator S verteilt sein drittes Argument an die beiden ersten. K eliminiert sein zweites Argument und I bildet die Identitätsfunktion. Ausdrücke des λ-Kalküls können mittels einfacher Termtransformationen in Kombinatorausdrücke umgeformt werden, wobei alle Variablen der λ-Terme eliminiert werden. Diese Kombinatorausdrücke können von einer Kombinatormaschine, die die Definitionsgleichungen der Kombinatoren als Reduktionsvorschriften benutzt, evaluiert werden.

5.1.2 Variablen–Abstraktion

Im λ-Kalkül werden Funktionen notiert als:

$$\lambda x.t$$

Die *Abstraktion* der Variablen x im Ausdruck t wird notiert als:

$$[x]t.$$

Der resultierende Ausdruck enthält keine Vorkommen von x, besitzt jedoch die gleiche Normalform wie der ursprüngliche Ausdruck. Im Gegensatz zu einer Run-time Bindung eines Wertes an die Variable x, stellt $[x]$ eine Compile-time Transformation dar. Diese Abstraktion verläuft nach folgenden Regeln[24]:

$$[x]x = I, \tag{6}$$
$$[x]y = K\ y, \tag{7}$$
$$[x](e_1, e_2) = S\ ([x]e_1)([x]e_2). \tag{8}$$

Wobei y entweder eine von x verschiedene Variable oder aber eine Konstante ist. Implementiert wird $[x]$ von der ML-Funktion `abs x`, die auf λ-Termen operiert:

```
exception Doublebind
fun abs x (Free y) = Apply(Op "K", (Free y))
   |abs x (Bound y) = if y=x then (Op "I") else Apply(Op "K",(Bound y))
```

[24] Davis, FN 26, S.155.

```
   |abs x (Int y) = Apply(Op "K",(Int y))
   |abs x (Op y) = Apply("K",(Op y))
   |abs x (Abs(y,body)) = abs x (abs y body)
   |abs x (Apply(a,b)) =
         Apply(Apply(Op "S",abs x a),
                    (abs x b));
```

5.1.3 Compilierung

Der Datentyp `snode` implementiert eine binäre Kombinatorgraphendarstellung:

```
datatype snode = satom of value
               |scomb of comb
               |sapp of (snode*snode);
```

Die Übersetzung von λ-Termen nach Kombinatortermen geschieht nach den folgenden Regeln[25]:

$$c(x) = x, \tag{9}$$

$$c(t\ u) = c(t)\ c(u), \tag{10}$$

$$c(\lambda x.u) = c([x]u). \tag{11}$$

Implementiert wird c durch die ML–Funktion `c`:

```
exception Compile;
fun c (Free a) = scomb(DEF a)
   |c (Bound a) = raise Compile
   |c (Int a) = sapp(scomb K,satom(int a))
   |c (Op k) = sapp(scomb K,scomb(mkComb k))
   |c (Apply(a,b)) = sapp(c a,c b)
   |c (Abs(x,body)) = c (abs x body);

fun mkComb "I" = I
   |mkComb "K" = K
   |mkComb "S" = S
   |mkComb "B" = B
   |mkComb "C" = C
   |mkComb "Y" = Y
   |mkComb "CONS" = CONS
   |mkComb "HD" = HD
   |mkComb "TL" = TL
   |mkComb "+" = PLUS
   |mkComb "-" = MINUS
```

[25] Diller, A.: *Compiling Functional Languages.* New York, John Wiley & Sons, 1988., S.88.

```
   |mkComb "*" = TIMES
   |mkComb "/" = DIV
   |mkComb "IF" = IF
   |mkComb "EQ" = EQ
   |mkComb "AND" = AND
   |mkComb "OR" = OR
   |mkComb "NOT" = NOT
   |mkComb "PR" = PR
   |mkComb str = DEF str;
```

Da mit dieser bersetzungsfunktion selbst kurze Definitionen zu großen, speicheraufwendigen Kombinatorausdrücken führen, schlägt Turner eine Optimierungsfunktion *opt* vor, die auf der Einführung zweier weiterer Kombinatoren B und C beruht:

$$B\ x\ y\ z \quad = \quad x(y\ z), \tag{12}$$

$$C\ x\ y\ z \quad = \quad x\ z\ y. \tag{13}$$

Die Optimierungsregeln:

$$opt(S\ (K\ e_1)(K\ e_2)) \quad = \quad K(e_1\ e_2), \tag{14}$$

$$opt(S\ (K\ e)\ I) \quad = \quad e, \tag{15}$$

$$opt(S\ (K\ e_1)\ e_2) \quad = \quad B\ e_1\ e_2, \tag{16}$$

$$opt(S\ e_1\ (K\ e_2)) \quad = \quad C\ e_1\ e_2, \tag{17}$$

werden von der ML-Funktion `ropt` solange auf die Resultate der Compilierung angewandt, bis alle möglichen Optimierungen `opt` durchgeführt wurden:

```
fun opt (sapp(sapp(scomb S,sapp(scomb K,e)),scomb I)) = (e : snode)
   |opt (sapp(sapp(scomb S,sapp(scomb K,e1)),sapp(scomb K,e2))) =
       sapp(scomb K,sapp(e1,e2))
   |opt (sapp(sapp(scomb S,sapp(scomb K,e1)),e2)) =
       sapp(sapp(scomb B,e1),e2)
   |opt (sapp(sapp(scomb S,e1),sapp(scomb K,e2))) =
       sapp(sapp(scomb C,e1),e2)
   |opt (x : snode) = x;

fun ropt x =
 let
  val y = opt x;
 in
   if y=x then x
   else ropt y
 end;
```

5.2 Reduktion von Kombinatorausdrücken

Um Kombinatorausdrücke auszuwerten, müssen die definierenden Gleichungen der Kombinatoren als Reduktionsregeln benutzt werden. Die Auswertung muß zudem in Normalordnung erfolgen, um das Auffinden aller existierender Normalformen zu gewährleisten. Außer den Kombinatoren S, K, I, B und C müssen auch noch arithmetische und andere grundlegende Operationen definiert werden. Die Reduktionsregeln für einige dieser Operationen lauten:

$$
\begin{aligned}
PLUS\ x\ y &= (\texttt{eval}\ x) + (\texttt{eval}\ y) && (18)\\
MINUS\ x\ y &= (\texttt{eval}\ x) - (\texttt{eval}\ y) && (19)\\
TIMES\ x\ y &= (\texttt{eval}\ x) * (\texttt{eval}\ y) && (20)\\
DIV\ x\ y &= (\texttt{eval}\ x)/(\texttt{eval}\ y) && (21)\\
IF\ \texttt{true}\ x\ y &= x && (22)\\
IF\ \texttt{false}\ x\ y &= y && (23)\\
IF\ exp\ x\ y &= \texttt{if (eval}\ exp) = \texttt{true then}\ x\ \texttt{else}\ y && (24)\\
EQ\ x\ y &= \texttt{if (eval}\ x) = (\texttt{eval}\ y)\ \texttt{then true} \\
&\quad\ \texttt{else false} && (25)\\
CONS\ x\ y &= x :: y && (26)\\
&\ \vdots
\end{aligned}
$$

Wobei `(eval node)` die Normalform von `node` bezeichnet. Die zugrundeliegende Datenstruktur der Kombinatormaschine sind binäre Bäume. Ein Knoten eines Baumes kann entweder atomar sein, oder aber eine Applikation (@) zweier Knoten darstellen. Der Kombinatorausdruck $S\ x\ y\ z$ wird durch folgende Datenstruktur repräsentiert:

```
sappl(sappl(sappl(scomb S, x),
            y),
      z);
```

Die Reduktion dieses Ausdrucks anhand der Definition $S\ x\ y\ z = (x\ z)(y\ z)$ ist in Abbildung (2) dargestellt.

Um die Reduktion der Kombinatorgraphen in Normalordnung durchzuführen, kann folgende Technik angewandt werden. Ausgehend vom Wurzelknoten wird durch Herabsteigen zu den linken Nachfolgern der äußerste linke Kombinator ermittelt. Ist dieser Kombinator *gesättigt*, d.h. sind alle benötigten Argumente vorhanden, wird eine Reduktion dieses Kombinators (entsprechend der oben beschriebenen Methode) ausgeführt. Dieser Ablauf wird so lange wiederholt, bis keine weiteren Reduktionen mehr möglich sind, der Graph also in Normalform vorliegt. Der verbleibende Ausdruck ist das Resultat der Berechnung.

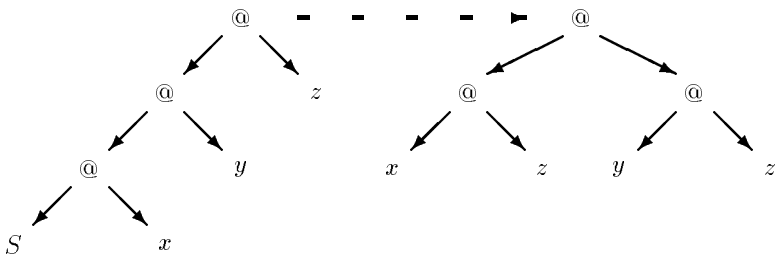


Abbildung 2: Die Auswertung von $S\ x\ y\ z$ als Graphreduktion

Beispiel: Dem ML-Ausdruck

```
let
   fun succ x = 1+x
in
   succ 2
end;
```

entspricht der λ-Ausdruck $(\lambda x.(+\ 1\ x))\ 2$, der zu $C\ I\ 2\ (PLUS\ 1)$ compiliert werden kann. Dieser Kombinatorausdruck wird von der Kombinatormaschine wie folgt reduziert (Abbildung 3):

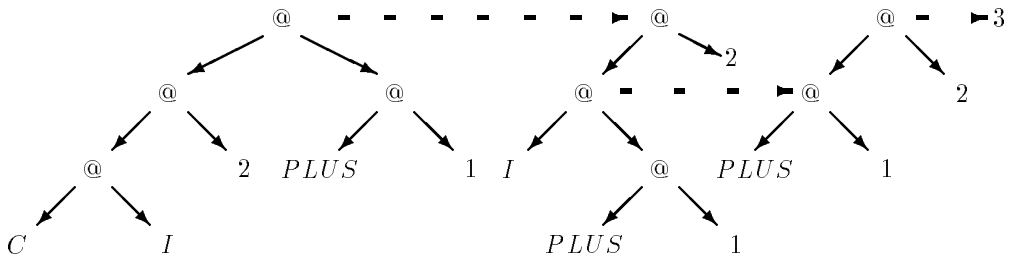


Abbildung 3: Die Graphreduktion von $C\ I\ 2\ (PLUS\ 1)$

5.3 Implementierung der Kombinator–Maschine

Im Vorhergehenden wurde angedeutet, daß die Transformationen der Kombinatorausdrücke von der Kombinatormaschine nicht auf ihrer abstrakten Term-

struktur sondern auf einer verzeigerten Datenstruktur durchgeführt wird, um alle Effizienzvorteile der Graphreduktion auszuschöpfen.

So repräsentieren die breiten gestrichelten Pfeile in den beiden vorhergehenden Abbildungen nicht nur die Termumformung des Kombinatorausdrucks, sondern auch, daß der Wurzelknoten des jeweiligen Redexes durch den resultierenden Ausdruck *überschrieben* wird[26]. Im Fall der Reduktion von $S\ x\ y\ z$ bleibt also der oberste Knoten erhalten und erhält neue Verweise auf die dynamisch generierten Repräsentationen von $(x\ z)$ und $(y\ z)$. x, y und z selbst brauchen jedoch nicht neu kopiert zu werden, da sie als verzeigerte Objekte vorliegen. Die beiden in einer Stringreduktion textuell verschiedenen Vorkommen von z sind somit physikalisch an einer Speicheraddresse lokalisiert, so daß z, falls es im im weiteren Verlauf der Auswertung benötigt wird, nur einmal berechnet wird.

Eine verzeigerte Repräsentierung der Kombinatorgraphen ist durch die Definition des Typs `node` gegeben:

```
datatype Emark = Eval|Busy|Ready;

datatype node = atom of (value * Emark ref * (node ref list) ref)
              |comb of (comb  * Emark ref * (node ref list) ref)
              |app of  ((node ref * node ref) * Emark ref *
                                          (node ref list) ref);
```

Die `(Emark ref)`-Verweise der Knoten enthalten spezielle Markierungen, die erst für die Einführung einer parallelisierten Evaluation benötigt werden und deren Erläuterung weiter unten geschieht.

Mittels der Funktion `alloc` werden abstrakte Termdarstellungen der Kombinatorausdrücke vom Typ `snode` in verzeigerte Objekte umgewandelt und so im Speicher alloziert:

```
fun alloc x =
 let
   fun allocate (satom val) = atom(val,ref Ready, ref [])
      |allocate (scomb com) = comb(com,ref Ready, ref [])
      |allocate (sapp(a,b)) = app((ref (allocate a),ref (allocate b)),
                                  ref Eval, ref [])
 in
   ref(allocate x)
 end;
```

Die oben beschriebene Reduktion in Normalordnung kann mit einem *left-ancestors-stack* (Stack der linken Nachfolge-Knoten) realisiert werden. Der Stack enthält zunächst nur einen Eintrag, einen Zeiger zum auszuwertenden Knoten. So lange, wie der oberste Knoten eine Applikation `app((l,r),_,_)` ist,

[26] Bei den beiden Knoten rechts und links neben einem solchen gestrichelten Pfeil handelt es sich also um den selben physikalischen Knoten.

wird ein neuer Knoten `l` auf den Stack geschoben. Auf diese Weise wird eine Kette von Knoten zum linken *spine* des obersten Knotens erzeugt. Die Funktion `spine node` ermittelt den Spine des Knotens `node`, sowie den Stack der linken Nachfolger von `node`:

```
fun spine (ref(atom(a,m,q))) stack = (atom(a,m,q),stack)
   |spine (ref(comb(c,m,q))) stack = (comb(c,m,q),stack)
   |spine (node as (ref(app((l,r),_,_)))) stack =
         spine l (node::stack);
```

Der Spine des Knotens `node` enthält den äußersten anwendbaren Kombinator, der nun direkt auf seine auf dem Stack gesammelten Argumente angewendet werden kann. Diese Anwendung des Kombinators `k` auf die in `stack` gesammelten Argumente geschieht mittels der Funktion `apply (k,stack)`:

```
fun apply (I,(node as ref(app((_,ref x),_,_)))::_) =
        node := x
   |apply (K,ref(app((_,ref x),_,_))::node::_) =
        node := x
   |apply (S,(ref(app((_,x),_,_)))::(ref(app((_,y),_,_)))
              ::(node as (ref(app((_,z),_,_))))::_) =
        node := app((ref(app((x,z),ref Eval,ref [])),
                     ref(app((y,z),ref Eval,ref []))),
                    ref Eval,ref [])
   |apply (PLUS,ref(app((_,ref(atom(int x,_,_))),_,_))::(node as
               ref(app((_,ref(atom(int y,_,_))),_,_)))::_) =
        node := atom(int(x+y),ref Ready,ref [])
   |apply (PLUS,(stack as ref(app((_,x),_,_))::
                          ref(app((_,y),_,_))::_)) =
        node := atom(int((eval x)+(eval y)),ref Ready,ref [])
```

Die schrittweise Durchführung der Reduktion eines Knotens `node` geschieht mittels der Funktion `step node`:

```
fun step node =
   let
    val (c,stack) = (spine node []);
   in
    if is_atom c then ()
    else let val comb(k,_,_)= c
         in apply (k,stack)
         end
   end;
```

5.4 Parallelisierung

Die Lazy–Evaluation besitzt mehrere nützliche Eigenschaften. Da sie die Reduktion in Normalordnung nachbildet, terminiert sie immer, wenn eine Normalform existiert (im Gegensatz zur Eager–Evaluation). Sie erlaubt außerdem die Verwendung sehr großer, potentiell unendlicher Datenstrukturen, da sie immer nur diejenigen Teile der Strukturen berechnet, die von anderen Teilen der Gesamtberechnung benötigt werden. Das Ziel von parallelisierten Evaluationsmechanismen sollte es offenbar sein, die Semantik der Lazy Evaluation zu bewahren und ihre Effizienz durch Verwendung von Mehr-prozess(or)-Techniken zu optimieren.

Im Gegensatz zu den *nicht strikten* Funktionen (wie etwa den Kombinatoren S, K, I, B, C und $CONS$, die ihre Argumente vor der Applikation *nicht* auswerten, erwarten *strikte* Operationen (wie etwa die arithmetischen und booleschen Operationen) stets vollständig evaluierte Argumente. In den Reduktionsregeln (18) bis (26) ist diese Evaluierung eines Argumentes x einer strikten Funktion durch (`eval` x) ausgedrückt. Die Evaluation des Terms $(+\ x\ y)$ muß also zunächst die Ausdrücke x und y auswerten und kann erst dann die resultierenden Werte addieren (vgl. Gleichung 18). Dies spiegelt sich in der Implementierung des Kombinators $PLUS$ in der Funktion `apply` direkt wider (vgl. Unterstreichung):

```
|apply (PLUS,(stack as ref(app((_,x),_,_))::
                        ref(app((_,y),_,_))::_)) =
     node := atom(int((eval x)+(eval y)),ref Ready,ref [])
```

x und y müssen also auch im Fall der Lazy Evaluation immer ausgewertet werden. Da die Argumente strikter Operationen immer evaluiert werden müssen, können durch *Striktheitsanalyse* diejenigen Teile von Berechnungen ermittelt werden, die unabhängig von der Auswertungsstrategie unbedingt berechnet werden müssen. Strikte Operationen bieten daher die Möglichkeit zur Parallelisierung der Lazy Evaluation[27]. Im Folgenden wird eine an [Dav92] orientierte Implementierung einer parallesierten Kombinatormaschine entwickelt, die die oben geschilderten Eigenschaften strikter Operationen ausnützt.

Die Reduktionsregeln der nicht-strikten Kombinatoren sind von der Parallelisierung nicht betroffen und operieren weiterhin wie oben definiert. Die Reduktion strikter Kombinatoren geschieht wie am Beispiel der Additionsoperation `PLUS` gezeigt:

```
|apply (PLUS,ref(app((_,ref(atom(int x,_,_))),_,_))::(node as
            ref(app((_,ref(atom(int y,_,_))),_,_))::_) =
     node := atom(int(x+y),ref Ready,ref [])
|apply (PLUS,(stack as ref(app((_,x),_,_))::
                       ref(app((_,y),_,_))::_)) =
     (subEval (last stack,x);
```

[27] Davis, FN 26, S. 201.ff.

```
        subEval (last stack,y); ())
```

Falls die Argumente bereits ausgewertet vorliegen (im Falle der Addition als Integer Zahlen), kann direkt operiert werden (erster Teil der Definition), ansonsten müßen zunächst die Argumente `x` und `y` ausgewertet werden (zweiter Teil). Die Funktion `subEval` wertet die Argumente nicht selbst ausm, was wieder eine sequentielle Bearbeitung von `x` und `y` zur Folge hätte, sondern erzeugt neue Subprozesse, die dem Schedulingmechanismus übergeben werden:

```
fun subEval (root,node) =
 let
   val emark = get_mark node;
   val wq = get_q node;
 in
  if (! emark = Ready) then ()
  else if (! emark = Busy) then
   (make_wait root;
    wq := root::(! wq))
  else
   (make_wait root;
    emark := Busy;
    wq := [root];
    newTask node)
 end;
```

Falls die Evaluationsmarkierung des Subknotens `node` auf `Ready` gesetzt ist, wurde der Knoten bereits ausgewertet und der `subEval` Prozess kann sofort beendet werden. Falls die Markierung `Busy` lautet, zeigt dies an, daß bereits ein anderer Prozeß an der Evaluierung des Knotens arbeitet. Es muß daher kein neuer Prozeß zur Evaluierung von `node` gestartet werden, jedoch muß der Wurzelknoten von `node`, `root`, so lange in den Wartemodus gesetzt werden, bis die Reduktion abgeschlossen ist. Dies geschieht durch die Funktion `(make_wait root)`:

```
fun make_wait node =
 let
  val (k,(w::_)) = spine node [];
  val ref(app((ref rator,rand),_,_)) = w;
 in
  w := app((ref(app((ref(comb(WAIT,ref Eval,ref [])),
                     ref rator),ref Eval,ref [])),
            rand),E,Q)
 end;
```

Diese Funktion ersetzt den Spine Kombinator von `root` durch den Kombinator $WAIT$. Die Reduktionsregeln für $WAIT$ lautet:

$$WAIT\ x \quad = \quad WAIT1\ x \tag{27}$$
$$WAIT1\ x \quad = \quad WAIT\ x \tag{28}$$

Diese beiden speziellen Kombinatoren erzeugen eine Endlosschleife, die den Evaluationsvorgang von `root` solange suspendiert, bis durch Terminierung der Subprozesse der $WAIT$ Kombinator wieder entfernt wird:

```
fun make_unwait node =
 let
  val (_,w::_) = spine node [];
  val ref(app((_,ref k),_,_)) = w;
 in
  w := k
 end;
```

Damit der Subknoten `node` bei der Beendung seiner Reduktion auch den Prozess `root`, der ihn ja nicht gestartet hat, reaktivieren kann, muß `root` in die *Waitequeue* von `node` aufgenommen werden. Diese Warteschlange enthält alle Knoten, deren Evaluation auf die Auswertung von `node` wartet. Ist die Evaluation von `node` abgeschlossen, werden alle diese Prozesse reaktiviert.

Falls die Markierung jedoch `Eval` lautet, muß `node` evaluiert werden. `node` wird in den `Busy` Status gesetzt, was anderen Prozessen signalisiert, daß `node` gerade bearbeitet wird. Durch die Zuweisung `wq := [root]` wird die Warteschlange von `node` initialisiert, zunächst wartet nur der Knoten `root` auf die Auswertung von `node`. Dann wird durch `(newTask node)` ein neuer Prozeß, nämlich die Evaluation von node, gestartet.

```
fun newTask task =
   Tasks := (task::(! Tasks));
```

Die globale Variable `Tasks` enthält einen Zeiger auf eine Liste aller zu evaluierender Knoten und bildet den *Task-pool* der Kombinatormaschine.

Der Schedulingmechanismus eines parallelverarbeitenden Systems ist für die optimale Verteilung der Tasks auf die physikalischen Prozessoren verantwortlich. Da es hier lediglich um die *Modellierung* einer parallelen Architektur geht, wird ein rudimentärer stochastischer Scheduler benutzt, der das Verhalten eines Mehrprozessorsystems simuliert:

```
val Seed = ref 4.34793;

fun Scheduler () =
 let
  fun rnd () =
   let
```

```
    val x = (! Seed)* 1475.37697;
    val res = x-real(floor x);
   in
    (Seed := res; res)
   end;
  fun intrand n = floor(rnd()*(real n));
 in
  if (!Tasks) = [] then ()
  else
     (evalstep (nth(!Tasks,intrand (length (!Tasks))));
      Scheduler())
 end;
```

Der Scheduler wählt zufällig einen Knoten aus dem Task-pool `!Tasks` aus und führt für ihn einen Evaluationsschritt aus. Diesen Vorgang wiederholt er so lange, bis alle Knoten vollständig evaluiert sind und wieder aus dem Task-pool entfernt wurden. Ein einzelner Evaluationsschritt wird durch die Funktion `evalstep node` ausgeführt:

```
fun evalstep (node as ref(atom(_,_,_))) = remTask node
   |evalstep (node as ref(comb(_,_,_))) = remTask node
   |evalstep (node as ref(app((ref rator,ref rand),Emark,WQ))) =
     let
      val old = copy node
     in
      (step node;
       if ((equal old node) orelse
           (!Emark = Ready) orelse
           (not (is_app (!node))))
        then (wakeup WQ;
              remTask node;
              Emark := Ready)
        else ())
     end;
```

Falls `node` atomar, d.h vom Typ `value` oder `comb` ist, evaluiert der Knoten zu sich selbst und die Evaluierung kann beendet werden (`remTask node`).

```
fun remTask task =
   Tasks := (remove task (! Tasks));
```

Andernfalls wird ein Evaluationsschritt `(step node)` ausgeführt. Danach wird überprüft, ob die Normalform bereits gefunden wurde. Dies geschieht durch den Vergleich des Zustandes vor dem `step` mit dem neuen. Falls sie gleich sind, ist eine Normalform gefunden und die Reduktion somit abgeschlossen. Die auf

das Ergebnis wartenden Prozesse können mit `(wakeup WQ)` wieder reaktiviert werden:

```
fun wakeup waitQ =
  (map (fn task => (make_unwait task))
       (! waitQ);
   waitQ := []);
```

Dann wird `node` aus dem Task-pool entfernt und die Evaluationsmarkierung auf `Ready` gesetzt. Falls noch keine Normalform gefunden wurde, ist der Evaluationsschritt direkt beendet. Anschließend kann der Scheduler mit seiner Bearbeitung des Task-pools fortfahren.

Beispiel: Das Verhalten der parallelisierten Kombinatormaschine soll an der Auswertung des Ausdrucks $x = (ADD(ADD\ 1\ 2)(ADD\ 3\ 4))$ veranschaulicht werden. Der Ausdruck wird zunächst übersetzt und alloziert:

```
- val x = alloc(ropt(c(r "ADD (ADD 1 2)(ADD 3 4)")));
```

Der Task-pool `Tasks` ist vor der Berechnung leer, durch `(newTask x)` wird x als erster Prozess in den Taskpool gesetzt:

`Tasks:` $\boxed{ADD(ADD\ 1\ 2)(ADD\ 3\ 4)}$

Dann wird mit `Scheduler ()` der Scheduler gestartet. Da x der einzige Prozess ist, wird er für die Durchführung eines `evalstep` ausgewählt, was durch die Unterstreichung markiert ist:

$$\boxed{\underline{ADD(ADD\ 1\ 2)(ADD\ 3\ 4)}}$$

Die Auswertung des äußersten ADD Kombinators erzeugt zwei neue Prozesse $(ADD\ 1\ 2)$ und $(ADD\ 3\ 4)$ und setzt die äußere Berechnung so lange in den Wartemodus, bis die Subprozesse terminieren (Die parallelen Prozesse in `Tasks` sind jeweils durch vertikale Doppelstriche $\|$ von einander getrennt):

$$\boxed{\underline{WAIT(WAIT(ADD(ADD\ 1\ 2)(ADD\ 3\ 4)))} \parallel ADD\ 1\ 2 \parallel ADD34}$$

Wählt der Scheduler nun zufällig den ersten Knoten zur Auswertung aus, so wird $WAIT$ durch $WAIT1$ ersetzt. Erneute Reduktionen dieses Knotens würden $WAIT1$ wiederum durch $WAIT$ ersetzen.

Dieser Wechsel zwischen den beiden Kombinatoren wird bei jedem Reduktionsschritt wiederholt. Der Evaluationsprozess des Knoten bleibt daher so lange in einer Warteschleife, bis der Wartekombinator durch Reaktivierung des Prozesses wieder entfernt wird. Wird jedoch der zweite Knoten ausgewählt:

$$WAIT1(WAIT(ADD(ADD\ 1\ 2)(ADD\ 3\ 4))) \parallel \underline{ADD\ 1\ 2} \parallel ADD34$$

so stellt die Funktion `Apply` fest, daß die Argumente der Addition bereits atomar (d.h. vollständig ausgewertet) sind und kann eine direkte Addition ausführen. Durch die Verzeigerung des Graphen führt diese Reduktion dazu, daß auch an der Stelle '$(ADD\ 1\ 2)$' des ersten Knotens nun das Ergebnis '3' steht. Da durch die einmalige Anwendung von `evalstep` bereits eine Normalform gefunden wurde, kann der zweite Knoten nun aus dem Taskpool entfernt werden und der wartetende Prozess reaktiviert werden. Da der erste Prozess nu noch auf einen Prozess wartet, wird er nur noch von einem $WAIT$ blockiert. Im Pool befinden sich jetzt nur noch zwei auszuwertende Knoten:

$$WAIT1(ADD\ 3\ (ADD\ 3\ 4)) \parallel \underline{ADD\ 3\ 4}$$

Wird nun als nächstes der verbleibende zweite Knoten ausgewertet, wird wie zuvor sofort eine Normalform, nämlich $3+4=7$ gefunden. Der Prozess wird entfernt, der wartende Prozess reaktiviert. Somit verbleibt nur noch ein Knoten im Pool:

$$\underline{ADD\ 3\ 7}$$

Hier kann ebenfalls mit einem Berechnungsschritt das Ergebnis $3+7=10$ bestimmt werden. Dann wird auch dieser letzte Knoten aus dem Taskpool entfernt. Daraufhin beendet der Scheduler seine Arbeit und der Knoten `x` enthält das Berechnungsergebnis.

```
> x;
- val it = value(int 10,ref Ready,ref []) : node
```

Der hier verwendete Scheduler simuliert die parallele Auswertung verschiedener Knoten durch die wiederholte Ausführung eines einzelnen Reduktionsschrittes auf einen zufällig aus dem Taskpool ausgewählten Knoten. Dieses *quasi–parallele* Auswertungsschema entspricht der Arbeitsweise eines Einprozessor Multitasking Betriebssystems. Ein solches System stellt abwechselnd jedem

Prozess eine bestimmte Prozessorzeit zur Verfügung. Das Hin- und Herschalten zwischen den einzelnen Prozessen erfolgt dabei in so schneller Folge, daß für den Benutzer der Eindruck einer simultanen Bearbeitung der Prozesse entsteht. Würde anstelle des Schedulers eine Mehrprozessormaschine zur Bearbeitung der Prozesse im Taskpool verwendet, könnten diese tatsächlich simultan (d.h. zeitgleich) ausgeführt werden.

Das im vorhergehenden Beispiel demonstrierte Auswertungsschema kann zu der Funktion `eval` zusammengefasst werden. Nachdem ein λ-Term compiliert und alloziert wurde, kann er von `eval` zu seiner Normalform reduziert werden:

```
fun eval node =
 (newTask node;
  Scheduler();
  node);

val ENV = ref [] : (string * node ref) list ref;

fun define name value =
   ENV := (name, alloc(ropt(c(r value)))):::(!ENV);

fun run exp = eval(alloc(ropt(c(r exp))));

- define "fac" "%n.IF (EQ n 0) 1 (MUL n (fac (SUB n 1)))";
> () : unit
- run "fac 7";
> val it = value(int 5040,ref Ready,ref []) : node
```

Hiermit ist die Implementierung der parallelisierten Graphreduktionsmaschine abgeschlossen. Im den folgenden Abschnitten wird die operationale Semantik dieser abstrakten Maschine modifiziert und erweitert, um den Rahmen der klassisch funktionalen Sprachen um proemielle Berechnungen zu erweitern.

6 Modellierung der Proemialrelation

Die Einführung einer parallelisierten Lazy-Evaluation funktionaler Programmiersprachen hatte es zur Aufgabe, die Vorteile einer Multi-prozess-Architektur mit denen der Lazy-Evaluation zu vereinen. Oberstes Gebot war hierbei die Bewahrung der Normal-order-Reduktion und der Semantik der funktionalen Programme.

Aus diesem Grund wurden keine Sprachkonstrukte für eine explizite Handhabung der parallelen Architektur durch funktionale Programme eingeführt, sondern die Parallelisierung nur implizit für die Auswertung strikter Operationen, unter strenger Beachtung der Synchronisation der beteiligten Prozesse eingeführt. Daher muß ein Prozeß, der neue Unterprozesse startet, so lange

im Wartemodus verbleiben, bis alle Subprozesse terminieren und ihn reaktivieren. Ein solcher Parallelisierungsmechanismus erfüllt zwar die an ihn gestellten Anforderungen (Einhaltung der Lazy-Evaluation Semantik), deckt jedoch nur einen sehr begrenzten Bereich der parallelen Prozesse ab, die auf einer wie oben modellierten Architektur möglich wären. Im Folgenden soll ein weiteres Parallelitätskonzept entwickelt werden, daß eine Modellierung der kenogrammatischen Proemialrelation darstellt.

In Abschnitt 3 wurde die Proemialrelation als das simultane Wechselverhältnis von Ordnungs- und Umtauschbeziehungen zwischen Objekten verschiedener logischer Stufen charakterisiert. Aufgrund der speziellen Eigenschaften der Proemialrelation und der Begrenztheit klassischer Kalküle müßte eine algebraische Darstellung der Proemialrelation selbstreferentiell, innerhalb klassischer Formalismen also paradoxal und antinomisch strukturiert sein. Wegen dieser grundlegenden Schwierigkeiten bei der Formalisierung der Proemialrelation wird hier versucht, zumindest eine *operationale* Modellierung der Proemialrelation zu entwickeln. Hierzu wird die operationale Semantik der abstrakten Kombinatormaschine um einen proemiellen Kombinator PR erweitert.

Die in Abschnitt 3 eingeführte Proemialrelation beschreibt das Zusammenspiel einer Ordnungsrelation, $\longrightarrow$, und einer Umtauschrelation, $\Updownarrow$, zwischen je zwei Operatoren und Operanden.

$$PR\Big(R_{i+1}, R_i, x_i, x_{i-1}\Big) :=$$

$$\begin{array}{lcl} R_{i+1} & \longrightarrow & x_i \\ & & \Updownarrow \\ & & R_i \quad \longrightarrow \quad x_{i-1}. \end{array}$$

Kaehr unterscheidet die *offene* und *geschlossene* Form der Proemialrelation[28]. Die geschlossene Proemialrelation ist zyklisch:

$$\begin{array}{ccc} R_{i+1} & \longrightarrow & x_i \\ \Updownarrow & & \Updownarrow \\ x_{i-1} & \longleftarrow & R_i \end{array}$$

Es gilt also $PR(PR^i) = PR^i$. Für die offene Proemialrelation gilt hingegen: $PR(PR^i) = PR^{(i+1)}$. Sie hat die folgende Gestalt:

$$\begin{array}{lcccccl} i+1\text{:} & R_{i+2} \longrightarrow & x_{i+1} & & & & \\ & & \Updownarrow & & & & \\ i\text{:} & & R_{i+1} & \longrightarrow & x_i & & \\ & & & & \Updownarrow & & \\ i-1\text{:} & & & & R_i & \longrightarrow & x_{i-1} \end{array}$$

[28] Kaehr (1978), FN 1, S. 5 f.

PR wurde bisher nur informell beschrieben als eine Relation, die ein gegebenes Objekt als *simultan* auf mehrere logische Ebenen verteilt bestimmt. Diese Eigenschaft erinnert an die Möglichkeit des λ-Kalküls, *identische* λ-Terme sowohl als Operatoren als auch als Operanden zu verwenden. Die simultane Verteilung des *selben* λ-Terms über mehrere logische Stufen läßt sich jedoch im λ-Kalkül nicht modellieren.

Der λ-Term $(f\ x)(x\ f)$ verwendet f und x innerhalb eines Ausdrucks sowohl als Operator als auch als Operand. Innerhalb des λ-Kalküls selbst kann, wie in allen semiotisch fundierten Kalkülen, zwar die *Identität* von Termen, nicht jedoch deren *Selbigkeit* als einmalig realisierte Objekte ausgedrückt werden. Die semiotische Gleichheit des Zeichen 'f' in '$(f\ x)$' mit dem 'f' in '$(x\ f)$' besagt nichts darüber, ob diese beiden identischen Terme innerhalb einer Reduktion des λ-Terms auch als physikalisch gleich, d.h als das *selbe* Objekt behandelt werden. Der Grund hierfür liegt in der Token–Type–Relation der Semiotik, die den λ-Kalkül fundiert.[29] Die Token–Type–Relation subsumiert alle physikalisch verschiedenen, jedoch *gleichgestalteten* Token unter einem Type. Da mit diesem Schritt von der physikalischen Realisierung der Token abstrahiert wird, kann die physikalische Identität oder Differenz verschiedener Zeichenreihenvorkommen nicht aus den Termen eines semiotisch fundierten Kalküls abgeleitet werden. Ein Kalkül operiert nur mit Types und da die Typegleichheit (oder auch Gestaltgleichheit $\equiv_{sem}$) nicht auch Tokengleichheit (d.h. physikalische oder Zeigergleicheit $\equiv_z$) impliziert, kann er gar keinen Begriff der Selbigkeit verwenden[30].

[29] Alle klassischen Kalküle und Formalismen sind dem klassischen Kalkülbegriff untergeordnet, der wiederum auf der Zeichen- und Zeichenreihenkonzeption der *Semiotik* basiert.

Die *Ausdrücke* (oder Terme) eines solchen Kalküls sind *Zeichenreihen*, die durch *lineares Aneinanderreihen* von jeweils endlich vielen *Grundzeichen* entstehen, die einem Alphabet von *wohlunterschiedenen* (d.h. mit sich selbst identischen und von allen anderen verschiedenen) Atomzeichen entstammen.

Der Begriff der *Gleichheit* von Zeichenreihen abstrahiert von der physikalischen Realisierung und benutzt die idealisierte Zeichenreihen*gestalt* als Gleichheitskriterium: Zwei Zeichenreihen sind gleich, wenn sie die gleiche Gestalt haben. Die Zeichenreihengestalt oder kurz Zeichengestalt meint also nicht nur eine individuelle Realisation einer Zeichenreihe, sondern stets die gesamte Klasse aller Zeichenreihen, die mit dieser Zeichenreihe *gleichgestaltet* sind (aber physikalisch durchaus unterschiedlich realisiert sein können). Dieses Verhältnis zwischen den konkreten Zeichenreihenvorkommnissen (*Token*) und der idealisierten Zeichenreihengestalt (*Type*) ist als *Token–Type–Relation* bekannt. Das Abstraktum aller gleichen Zeichenreihen–Tokens ist der Zeichenreihen–Type. Zwei Token sind genau dann gleich, wenn sie dem selben Type angehören.

Das lineare Aneinandereihen von Zeichengestalten geschieht in der Zeichentheorie durch eine im algebraischen Sinne assoziative Verkettungsoperation, die jedem Paar z_1, z_2 von Zeichengestalten eine eindeutig bestimmte Zeichengestalt $z_3 = z_1 z_2$ zuordnet.

[30] Quine formuliert diese der klassischen Semiotik zugrundeliegende Konzeption wie folgt: „*Um was für Dinge handelt es sich denn, genaugenommen, bei diesen Ausdrücken [den Termen eines klassischen formalen Systems]? Es sind Typen, keine Vorkommnisse. Jeder Ausdruck ist also, so könnte man annehmen, die Menge aller seiner Vorkommnisse. Das heißt, jeder Ausdruck ist eine Menge von Inschriften, die sich an verschiedenen Orten des Raum-Zeit-Kontinuums befinden, die aber Aufgrund ähnlichen Aussehens zusammengefasst werden.*“

Aus der syntaktischen Struktur von λ-Termen kann desweiteren auch nicht abgeleitet werden, nach welchem Reduktionsverfahren sie auszuwerten sind. Aus dem obigen Ausdruck geht nicht hervor, ob — und in welcher Reihenfolge — $(f\ x)$ und $(x\ f)$ sequentiell, oder simultan ausgewertet werden sollen. Die Frage der Selbigkeit von Repräsentationen von λ-Termen bleibt ebenso wie die Wahl der Auswertungsstrategie allein Thema der Implementierungstechniken des λ-Kalküls, da sie die (semiotisch fundierte) Semantik der Terme nicht betrifft.

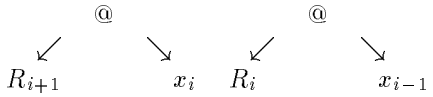
In der Proemialrelation ist nun aber gerade die im λ-Kalkül nicht abbildbare simultane Verteilung des selben Objektes über mehrere Bezugssysteme gemeint. Das hier vorgeschlagene Modell der Proemialrelation versucht nun, gerade dieses Verhalten operational abzubilden. Dazu geht es von Günthers Konzeption der Kenogramme als Leerstellen, an denen semiotische Prozesse eingeschrieben werden können, aus. Die Selbigkeit eines Terms (die auf semiotischer Ebene nicht definiert werden kann) wird dann durch die Selbigkeit des Kenogramms, in das er eingeschrieben wird, bestimmt. Dieser in einem und dem selben Kenogramm realisierte Term kann nun simultan innerhalb verschiedener semiotischer Prozesse sowohl als Operator als auch als Operand dienen[31].

6.1 Implementierung des Proemialkombinators PR

Von dieser grundlegenden Idee der innerhalb der Kenogrammatik notierten Selbigkeit semiotischer Prozesse ausgehend soll nun die operationale Semantik des Proemialkombinators $PR(R_{i+1}, R_i, x_i, x_{i-1})$ mittels der oben entwickelten Kombinatormaschine bestimmt werden.

Dieses Modell nutzt die Homogenität von Programmen und Daten der Graphrepräsentation der Kombinatormaschine aus. So kann ein bestimmter Knoten z, der als physikalisches Objekt an einer bestimmten Adresse im Arbeitsspeicher realisiert ist, innerhalb verschiedener Applikationsknoten sowohl als Operator als auch als Operand dienen. Aufgrund der parallelen Architektur der Kombinatormaschine kann dieser Rollenumtausch (Operator $\Longleftrightarrow$ Operand) des selben Knotens z simultan ausgeführt werden.

R_i, R_{i+1}, x_i und x_{i-1} können beliebige Knoten des Kombinatorgraphen sein. Die Ordnungsrelation der Proemialrelation, $\longrightarrow$, ist hier die Applikation `app(rator,rand)`, die stets eine eindeutige Unterscheidung von Operator und Operand gewährleistet:

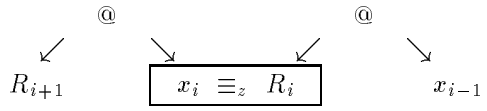


Zwei solche Applikationsknoten können an beliebigen Stellen innerhalb eines Kombinatorgraphen vorkommen. Innerhalb eines solchen Knotens ist durch die

(Quine, W.V.: *Ontologische Relativität und andere Schriften.* Stuttgart, Reclam, 1975.).

[31] Zum Verhältnis von Semiotik und Kenogrammatik vgl. Kaehr (1982), FN 3; Mahler (1993), FN 14

Applikationsstruktur stets determiniert, ob ein Subknoten als Operator oder als Operand dient. Durch die Verzeigerung des Kombinatorgraphen ist es nun möglich, daß x_i und R_i das *selbe* physikalische Objekt z bezeichnen. Innerhalb von $\texttt{app}(R_{i+1}, x_i)$ fungiert z dann als Operand und in $\texttt{app}(R_i, x_{i-1})$ als Operator:



Die Zeigergleichheit von x_i und R_i, $x_i \equiv_z R_i$, bewirkt also, daß z in verschiedenen Applikationen (Ordnungsverhältnissen) als Operator und als Operand fungiert. Dieser Positionswechsel innerhalb der Ordnungsrelation `app(rator,rand)` dient als Modell für die Umtauschrelation der Proemialrelation, $\Leftrightarrow$. Somit erfüllen R_i, R_{i+1}, x_i und x_{i-1}, mit $R_i \equiv_z x_i$, das Schema der Proemialrelation:

$$\begin{array}{lll} R_{i+1} & \longrightarrow & x_i \\ & & \Updownarrow \\ & & R_i \quad \longrightarrow \quad x_{i-1}. \end{array}$$

Die in der informellen Spezifikation der Proemialrelation geforderte Simultaneität der logischen Ebenen, innerhalb derer z an verschiedenen Stellen der Ordnungsrelation `app(rand,rator)` steht, wird nun dadurch modelliert, daß die Applikationen $\texttt{app}(R_{i+1}, x_i)$ und $\texttt{app}(R_i, x_{i-1})$ simultan ausgewertet werden. Das physikalische Objekt z dient dann simultan (im Sinne der jeweiligen verwendeten parallelen Architektur) innerhalb verschiedener Applikationen sowohl als Operator als auch als Operand. Diese theoretische Konzeption führt zu der folgenden Implementierung des Proemialkombinators PR:

```
|apply (PR, (stack as ( ref(app((_,R1),_,_))::
                       ref(app((_,R2),_,_))::
                       ref(app((_,x1),_,_))::(node as
                      (ref(app((_,x2),_,_))))::_))) =
    let
     val first =  ref(app((R1,x1),ref Eval,ref []));
     val second = ref(app((R2,x2),ref Eval,ref []));
    in
     (node := app((ref(app((ref(comb(CONS,ref Ready,ref [])),
                           first),ref Ready,ref [])),
                  second),ref Ready,ref [];
      parEval (last stack, first);
      parEval (last stack, second))
    end
```

```
fun parEval (root,node) =
 let
  val emark = get_mark node;
  val wq = get_q node;
 in
  if (! emark = Ready) then ()
  else if (! emark = Busy) then
   (make_wait root;
    wq := root::(! wq))
  else
   (emark := Busy;
    newTask node)
  end;
```

Die Reduktion des Kombinators `PR` erwartet vier Argumente, `R1, R2, x1` und `x2`. Aus diesen werden zwei Applikationen `first` = (`app(R1,x1)`) und `second` = (`app(R2,x2)`) konstruiert. Diese beiden Knoten werden zunächst zu einem Cons-objekt `CONS(first,second)` zusammengefügt, dadurch bleiben etwaige Ergebnisse der Applikationen `first` und `second` erhalten und können später inspiziert werden (für die eigentliche Semantik von `PR` ist dies jedoch nicht notwendig). Anschließend werden durch `(parEval first)` und `(parEval second)` die zwei Applikationen als parallele Prozesse dem Scheduling-mechanismus übergeben.

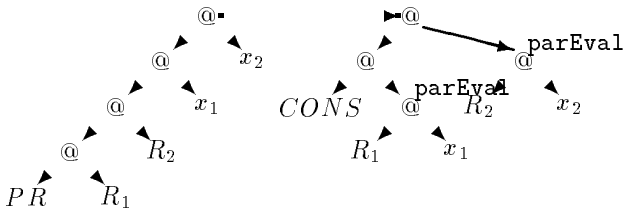


Abbildung 4: Die Graphreduktion von $PR(R_1, R_2, x_1, x_2)$

Dieses Reduktionsschema ist in Abbildung (4) dargestellt. Die Markierung `parEval` an den Knoten $@(R_1, x_1)$ und $@(R_2, x_2)$ sollen hierbei andeuten, daß die Reduktion weder strikt noch nicht-strikt verläuft: Die Knoten werden nicht ausgewertet, bevor sie zu einem $CONS$-Objekt zusammengefügt werden. Im Gegensatz aber zur Reduktion nicht-strikter Operationen werden sie durch die Funktion `parEval` der parallelen Auswertung durch den Scheduler übergeben.

Die Funktion `parEval` ähnelt der Funktion `subEval`, im Gegensatz zu dieser werden jedoch die beteiligten Prozesse nicht synchronisiert. Die Synchronisa-

tion mittels (`make_wait root; wq := [root]`) kann entfallen, da `first` und `second` nicht Subprozesse eines übergeordneten strikten Operators sind, der auf ihre Berechnungsergebnisse wartet, sondern autonome, lediglich proemiell verknüpfte Prozesse sind.

Sind nun R_2 und x_1 als das *selbe* physikalische Objekt z realisiert, so ergibt sich offensichtlich folgendes Reduktionsschema (Abbildung 5).

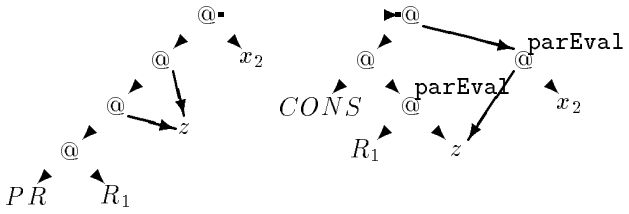


Abbildung 5: Die Graphreduktion von $PR(R_1, z, z, x_2)$

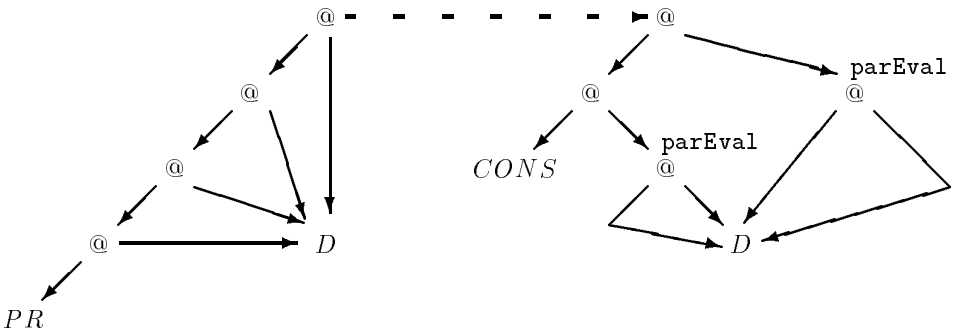
Bezüglich R_1 ist z hier Operand, bezüglich x_2 Operator. Da die beiden Applikationen $@(R_1, z)$ und $@(z, x_2)$ parallel ausgertet werden, erfüllt diese Reduktion das simultane Umtauschverhältnis der Proemialrelation.

Beispiel: Als Beispielberechnung soll hier der selbstreproduzierende λ-Term DD mit $D = (\lambda x.xx)$ proemiell verteilt werden:

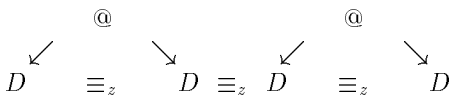
```
- define "D" "%x.xx";
> () : unit

- run "(%x.PR x x x x) D";
```

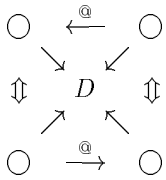
Das Reduktionsschema ist also:



Für diese offensichtlich nichtterminierende Berechnung ist die Struktur des Kombinatorgraphen wie folgt ($\equiv_z$ bezeichnet hierbei die Zeigergleichheit von Objekten):



Ein einziges physikalisches Objekt ist somit über die vier Stellen der Proemialrelation verteilt:



Die Kreise ◯ symbolisieren hier die vier Stellen der Proemialrelation, die jeweils einen Verweis auf das Objekt D enthalten. Die Verweise sind jeweils durch einfache Pfeile notiert. Die Pfeile $\overset{@}{\longleftarrow}$ bezeichnen die Ordnungsrelation der Proemialrelation, die hier durch die Trennung von Operator und Operand *innerhalb* der Applikation @ realisiert ist. Die Umtauschrelation der Proemialrelation, ⇳, steht für den Funktionswechsel von Operator zu Operand (und umgekehrt) der ◯-Objekte *zwischen* den verschiedenen Applikationen @.

Allgemein gilt für $\texttt{PR}(f, g, x, y)$: falls $x \equiv_z g$, dann sind die beiden Berechnungen $(f\ x)$ und $(g\ y)$ (offen) proemiell verkoppelt. Falls außerdem $y \equiv_z f$ liegt die geschlossene Proemialrelation vor. Ist wie im obigen Beispiel $f \equiv_z g \equiv_z x \equiv_z y$, so ist die Struktur *autoproemiell geschlossen*.

6.2 Anwendungsmöglichkeiten

6.2.1 Meta-level Architekturen und Reflektionale Programmierung

Die Definition des Kombinators PR erweitert den Sprachrahmen funktionaler Programmiersprachen um die Modellierung proemieller Konstruktionen. Um die Nützlichkeit einer solchen Erweiterung zu demonstrieren, wird hier kurz ein mögliches Anwendungsgebiet gestreift.

Unter den Schlagworten *Computational Reflection* (CR) und *Meta-level-Architectures* versammeln sich in der aktuellen Grundlagenforschung der Informatik Bemühungen, den klassischen Berechnungsbegriff, wie er etwa im λ-Kalkül formuliert wird, zu erweitern. Insbesondere geht es darum, Berechnungssysteme zu entwickeln, die über ihre Berechnungen 'reflektieren', d.h. wiederum Berechnungen anstellen können.

Nach Maes[32] zeichnet sich eine reflektive Programmiersprache dadurch aus, daß sie Methoden zur Handhabung reflektiver Berechnungen explizit bereitstellt. Konkret heißt dies, daß:

1. Der Interpreter einer solchen Sprache es jedem auszuwertenden Programm ermöglichen muß, auf die Datenstrukturen, die das Programm selbst (oder bestimmte Aspekte des Programms) repräsentieren, zuzugreifen. Ein solches Programm hat dann die Möglichkeit diese Daten, d.h. seine eigene Repräsentation zu manipulieren (Meta–berechnung).

2. Der Interpreter außerdem gewährleisten muß, daß eine kausale Verbindung (*causal connection*) zwischen diesen Daten und den Aspekten des Programmes, die sie repräsentieren, besteht. Die Modifikation, die ein Reflektives Programm an seiner Repräsentation vornimmt, modifizieren also auch den Zustand und weiteren Ablauf des Programmes (der Objektberechnung). In diesem Sinne werden die Metaberechnungen in der Objektberechnung reflektiert[33].

In einem solchen System können Repräsentationen von Berechnungsvorschriften einerseits auf der Ebene der Objektberechnungen *als Programm* evaluiert werden, oder zum anderen — z.B. im Falle eines Fehlers — auf einer Meta-berechnungsebene *als Daten* einer Metaberechnung dienen, die beispielsweise den Fehler korrigieren soll. Diese Struktur ist im nachfolgenden Schema (6) dargestellt.

$$\begin{array}{lcccc} \text{metacomp.:} & PRG_2 & \longrightarrow & DAT_1 & \\ & & & \Updownarrow & \\ \text{objectcomp.:} & & & PRG_1 & \longrightarrow \quad DAT_0 \end{array}$$

$\longrightarrow$: program PRG_i is applied to data DAT_{i-1}.
$\Updownarrow$: PRG_i becomes DAT_i redarding the metacomputation PRG_{i+1} and vice versa.

Abbildung 6:
Locigal levels of Computational Reflection

[32] Maes, P., Nardi, D.: *Meta-Level Architectures and Reflection.* Amsterdam, North–Holland, 1988., S.22 ff.

[33] Das Konzept der computational reflection kann als spezielle Untermenge der Meta–level–Architekturen verstanden werden. Eine Meta–level–Architektur ermöglicht Metaberechnungen, d.h. das Schlußfolgern (Agieren) eines Berechnungssystems über (auf) ein zweites Berechnungssystem. Eine Meta–Level–Architektur erlaubt jedoch nicht notwendigerweise auch reflektive Berechnungen. Dies ist zum Beispiel nicht der Fall, wenn die Metaberechnung nur statischen Zugriff auf das Objektsystem hat, wenn also keine kausale Verknüpfung zwischen den beiden existiert.

Der Umtausch des Operators PRG_i der Objektberechnung zum Operanden DAT_i der Metaberechnung kann von der Umtauschrelation der Proemialrelation $\Updownarrow$ beschrieben werden. Die Unterscheidung von Programm (Operator) und Datum (Operand) innerhalb einer Berechnungsebene entspricht der Ordnungsrelation der Proemialrelation. Das Strukturschema eines reflektiven Berechnungssystems entspricht daher exakt dem der Proemialrelation.

Da der Proemialkombinator $PR\,x_i$ und R_i als ein einziges physikalisches Objet handhabt, wirken sich alle Modifikationen von x_i direkt auch auf R_i aus[34]. Die Zeigergleichheit $x_i \equiv_z R_i$ gewährleistet somit die kausale Verknüpfung von Meta– und Objektebene. Der Proemialkombinator PR ist daher zur Modellierung von reflektiven Systemen im Sinne von Maes Definition geeignet.

In den bislang bekannten reflektiven Systemen (z.B. 3LISP [35]) sind Meta– und Objektebenen nicht als simultane Prozesse realisiert, sondern treten nur rein sequentiell in Aktion. Daher beeinflussen Metaberechnungen PRG_{i+1}, die die Repräsentation des Objektprogramms als Datum DAT_i manipulieren, nicht gleichzeitig die Objektberechnung $(PRG_i \longrightarrow DAT_{i-1})$. Sie werden erst dann wirksam, wenn die Metaberechnung terminiert und wieder die Objektebene aktiviert, indem sie dem Interpreter die veränderte Objektberechnungsvorschrift übergibt. Zu einem bestimmten Zeitpunkt der Gesamtberechnung wird also entweder auf einer Metaebene oder auf der Objektebene evaluiert. Ob eine Berechnungsvorschrift als Programm (Operator) oder als Datum (Operand) dient, ist stets eindeutig determiniert.

Der Proemialkombinator PR und auf ihm basierende Programmierkonstrukte ermöglichen im Gegensatz zu diesen Modellen gerade die *simultane Verkoppelung* von Objekt und Metaberechnungen. Damit bietet PR ein parallelisiertes Modellierungskonzept für reflektive Systeme.

Es kann hier nicht ermittelt werden, inwieweit die entwickelte Konstruktion zur Lösung grundlegender und praktischer Probleme der Computational Reflection Forschung beitragen kann. Da jedoch die kenogrammatische Proemialrelation die selben strukturellen Probleme thematisiert wie die Computational Reflection, und darüber hinaus ein Konzept zur Modellierung der kausalen Verkoppelung paralleler Prozesse bietet, scheinen weitere fruchtbare Forschungen in dieser Richtung möglich[36].

[34] In rein funktionalen Sprachen sind Reduktionen, die keine semantische Veränderung bewirken, die einzig möglichen Modifikationen. Sind jedoch — wie die Zuweisung := in ML — auch imperative Konstrukte zugelassen, lassen sich Modifikationen erzeugen, die auch die Semantik einer Variablen verändern. Um semantische Modifikationen der Objektberechnung zu erzeugen, wird daher streng genommen noch ein Zuweisungsoperator $SET\ x\ value$ benötigt, der der Variablen x den Wert $value$ zuweist.

[35] Smith, B.C.: *Reflection and Semantics in a Procedural Language*. LCS Technical Report TR–272, MIT, Massachusets, 1982.

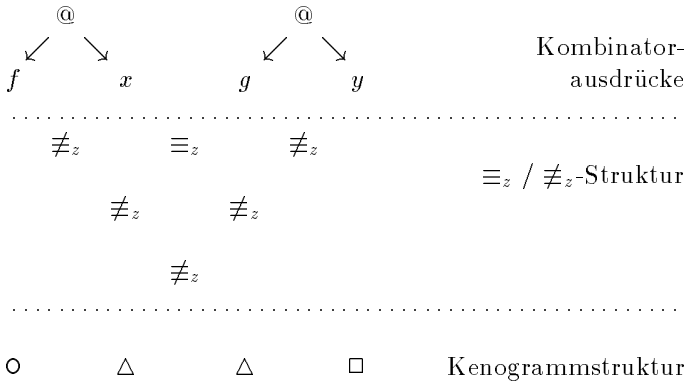
[36] Bestimmte Probleme, wie z.B. das 'Interfacing' von Prozessen (z.B. innerhalb von Betriebssystemen und Benutzeroberflächen) lassen sich in einem sequentiellen CR–System (wie 3LISP) nicht zufriedenstellend lösen. Die vom Kombinator PR gewährleistete simultane Ausführung der beteiligten Prozesse, läßt ihn zur Implementierung solcher Systeme geeignet erscheinen.

6.2.2 Verallgemeinerung des Parallelitätsbegriffes

Die Definition des Proemialkombinators PR beruhte auf der physikalischen Verkoppelung paralleler Berechnungen. Dieser Modellierungsansatz soll hier zu einer kenogrammatisch fundierten Notation paralleler Prozesse ausgeweitet werden. Hierzu wird zunächst die Struktur paralleler Prozesse in dem verwendeten Modell untersucht.

Eine Berechnung $PR\ f\ g\ x\ y$ weist die Form der offenen Proemialrelation auf, wenn $g \equiv_z x$ und $f \not\equiv_z y$. Ist $g \equiv_z x$ und $f \equiv_z y$, handelt es sich um die geschlossene Form.

Welche Art der Proemialrelation bei der parallelen Auswertung zweier Kombinatorausdrücke $(f\ x)$ und $(g\ y)$ vorliegt, kann nicht aus der Termstruktur sondern nur aus der Struktur von Zeigergleichheit ($\equiv_z$) und Zeigerverschiedenheit ($\not\equiv_z$) von f, x, g und y ermittelt werden.



Die $\equiv_z$ / $\not\equiv_z$–Struktur ist strukturisomorph zu der ϵ/ν–Struktur[37] von Kenogrammkomplexionen[38]. Aufgrund dieser Isomorphie kann die obige $\equiv_z$ / $\not\equiv_z$–Struktur von f, x, g, y als Kenogrammsequenz ○△△□ dargestellt werden.

Da hier $x \equiv_z g$ und $f \not\equiv_z y$, entspricht ○△△□ der offenen Proemialrelation. Die Kenogrammsequenz ○△△○ bezeichnet die geschlossene Proemialrelation, da hier $x \equiv_z g$ und $f \equiv_z y$. Die Kenogrammsequenz ○○○○ kennzeichnet eine Situation, in der alle vier Argumente der Proemialrelation auf das selbe Objekt verweisen, diese somit autoproemiell geschlossen ist. Auch die Sequenzen ○○○△ und ○△△△ bezeichnen proemielle Verhältnisse, da für sie ebenfalls $x \equiv_z g$ gilt. Die restlichen zehn der insgesamt fünfzehn Kenogrammsequenzen der Länge 4 bezeichnen jedoch Verhältnisse von Zeigergleichheit und Verschie-

[37] Vgl. Mahler (1993), FN 14

[38] Eine strukturisomorphe Abbildung zwischen $\equiv_z$ / $\not\equiv_z$–Struktur und ϵ/ν–Struktur kann etwa durch den Isomorphismus $\mathcal{I} : (\equiv_z, \not\equiv_z) \longrightarrow (\epsilon, \nu)$ mit $\mathcal{I}(\equiv_z) = \epsilon$ und $\mathcal{I}(\not\equiv_z) = \nu$ definiert werden.

denheit, die nicht der Proemialrelation genügen (da $x \not\equiv_z g$ liegt kein Umtauschverhältnis vor).

Da die proemiellen Verhältnisse nur einen Teilbereich der kombinatorisch möglichen Kenogrammsequenzen abdecken, liegt es nahe, die Proemialrelation als Spezialfall eines Parallelitätsbegriffes zu verstehen, der alle möglichen $\equiv_z$ / $\not\equiv_z$–Strukturen zuläßt. Aufgrund der Isomorphie zwischen $\equiv_z$ / $\not\equiv_z$– und ϵ/ν–Struktur kann die Struktur der parallelen Berechnungen die durch den PR-Kombinator ermöglicht werden, von der Kenogrammatik beschrieben werden. Physikalische Verkoppelung und Wechselwirkung paralleler Prozesse kann dann allgemein mittels kenogrammatischer Operationen formal dargestellt werden.

Soll beispielsweise den beiden parallelen Berechnungen $(f\ x)$ und $(g\ y)$ mit der (geschlossen proemiellen) Struktur ○△△○ eine weitere Berechnung $(h\ z)$ mit der Strukur ○△ hinzugefügt werden, sind für die resultierende Gesamtberechnung $(f\ x)\|(g\ y)\|(h\ z)$ verschiedene ϵ/ν–Strukturen möglich:

f	x	g	y	h	z
ν	ϵ	ν	$?_1$	ν	
ν	ν	$?_2$	$?_5$		
ν	$?_3$	$?_6$			
$?_4$	$?_7$				
$?_8$					

Die entstehenden acht Freiheiten[39] $?_i$ können mit ϵ oder ν belegt werden. Die möglichen Kenogrammsequenzen sind dann:

$$\begin{aligned} ○△△○ \mathbin{@} ○△ \quad = \quad & \{○△△○○△, ○△△○○□, \\ & ○△△○△○, ○△△○△□, \\ & ○△△○□○, ○△△○□△, \\ & ○△△○□\star\}. \end{aligned}$$

Die indizierte Verkettung $@_i$ bestimmt einzelne Elemente dieser Menge. Sie kann also dazu verwendet werden, genau spezifizierte Verkoppelungen paralleler Prozesse zu bestimmen.

> **Beispiel:** Ein Erzeuger/Verbraucher–System werde durch zwei Berechnungen `(produce list)` und `(consume list)` modelliert. `produce` fügt `list` jeweils neue Elemente hinzu, `consume` arbeitet `list` sequentiell ab. Damit die beiden parallelen Berechnungen wie gewünscht zusammenwirken, müssen die von `produce` neu erzeugten Elemente von `list` auch der Verbraucherberechnung `consume` verfügbar sein. Dies ist nur möglich, wenn beide Prozesse auf dem selben Objekt operieren. Die Kenogrammstruktur von (*PR* `produce consume list list`) muß also ○△□△ lauten. Als

[39] Mahler (1993), FN 14

indizierte Verkettung der Sequenzen $\circ\triangle$ und $\circ\triangle$, die die Struktur der Berechnungen `(produce list)` und `(consume list)` darstellen, ergibt sich daher:

$$\circ\triangle \ @_{[(1,1,\nu),(2,2,\epsilon)]} \ \circ\triangle = \circ\triangle\square\triangle.$$

Da die Kombinatormaschine alle Vorkommen einer Variablen x in einer Abstraktion $\lambda x.\cdots x\cdots x\cdots$ als Instanzen eines einzigen Objektes behandelt (node–sharing), unterschiedliche Variablen jedoch unterschiedlichen Objekten zuordnet, läßt sich die benötigte ϵ/ν–Struktur des Erzeuger/Verbraucher–Systems realisieren durch:

```
- run "(%l. PR produce consume l l) list";
```

6.3 Grenzen des Modells

Das Tao, das genannt werden kann, ist nicht das Tao.
Lao Tse

Das im Vorhergehenden entwickelte operationale Modell der Proemialrelation beschreibt die simultane Verwendung eines gegebenen Objektes innerhalb verschiedener Bezugssysteme und Funktionalitäten. Wie gezeigt, erfüllt das Modell die in 3 eingeführte Bestimmung der Proemialrelation als ein simultanes Wechselverhältnis von Ordnungs- und Umtauschbeziehungen zwischen Objekten verschiedener logischer Stufen. In diesem Abschnitt soll gezeigt werden, welche Aspekte der Güntherschen Konzeption der Proemialrelation durch dieses Modell nicht erfasst werden.

Die simultane Auswertung der proemiell verkoppelten Prozesse wird durch den Schedulermechanismus sichergestellt. Gegenüber den einzelnen Prozessen ist der Scheduler ein Metasystem, daß die scheinbare Vielzahl autonomer Prozesse zu einem globalen (allerdings nicht-deterministischen) System zusammenfasst. Unabhängig davon, ob ein bestimmtes Objekt innerhalb eines Prozesses als Operator und innerhalb eines anderen als Operand fungiert, ist es einfach ein Datenobjekt, das von einem Programm (dem Scheduler und den von ihm kontrollierten Prozessen) bearbeitet wird. Diese globale Analyse des Kontrollflusses zeigt deutlich, daß der aus der Sicht der Prozesse beobachtete simultane Funktionswechsel Operator:Operand nur eine sekundäre Interpretation des eigentlichen Vorganges ist.

Daß das vorgestellte Modell die simultane Umtauschbeziehung der Proemialrelation abbildet, gilt somit nur, wenn es aus der Perspektive der lokalen Prozesse interpretiert wird. Das Gesamtsystem ist jedoch ein klassisches formales System, weshalb es ja auch in einer klassischen Programmiersprache formuliert werden kann.

Eine weitere Einschränkung betrifft den Begriff des *Objekts* und die *Selbigkeit* von Objekten. Die Selbigkeit eines Objektes ist im vorgeschlagenen Modell über die physikalische Gleichheit bzw. Zeigergleichheit definiert. D.h. ein Objekt ist nur dann ein und dasselbe, wenn es tatsächlich an der selben physikalischen Adresse des Hauptspeichers liegt. Objekte, die die *gleiche* Termstruktur aufweisen aber an verschiedenen Speicheradressen liegen, sind zwar *identisch* (bzgl. der semiotisch fundierten Termgleichheit), jedoch nicht selbig. Diese Differenzierung von Gleichheit und Selbigkeit entspricht dem Unterschied der LISP-Funktionen `eq` und `equal`: `eq` testet Zeigergleichheit, `equal` Termgleichheit.

Die Modellierung des simultanen Umtauschs eines selben Objektes innerhalb verschiedener Bezugssysteme stützt sich auf diesen Selbigkeitsbegriff: Das vorgegebene selbige Objekt kann von verschiedenen Prozessen simultan sowohl als Operator als auch als Operand einer Applikation *interpretiert* werden. Wie bereits erwähnt erscheint also die Proemialität hier nur als abgeleitete *a posteriori*-Interpretation eines *a priori* gegebenen Objektes.

Günthers Idee der Proemialrelation beabsichtigt jedoch gerade, eine der dualistischen Aufspaltung Operator:Operand vorangehende Begrifflichkeit zu entwickeln: „*Wir nennen diese Verbindung zwischen Relator [Operator] und Relatum [Operand] das* Proemialverhältnis*, da es der symmetrischen Umtauschrelation und der Ordnungsrelation vorangeht und [...] ihre gemeinsame Grundlage bildet. [...] Dies ist so, weil das Proemialverhältnis jede Relation als solche konstituiert. Es definiert den Unterschied zwischen Relation und Einheit oder — was das gleiche ist — [...] wie der Unterschied zwischen Subjekt und Objekt.*“[40]

Im Gegensatz zu dem obigen Modell, das die selbigen Objekte zur Definition des simultanen Umtauschs voraussetzen muß, kann in Günthers Ansatz der Objektbegriff (und insbesondere die Selbigkeit) nicht vorausgesetzt werden, da er, im Gegensatz zu klassischen Kalkülen, auf eine a priori Unterscheidung Subjekt:Objekt verzichtet. Die Proemialrelation soll vielmehr gerade erst die Unterscheidbarkeit von Subjekt- und Objektkategorien ermöglichen. Die Selbigkeit von Objekten ergibt sich dann als ein *abgeleitetes* Phänomen und nicht als eine fundierende Kategorie.

In dem hier entwickelten Modell tauchen die transklassischen Aspekte der Proemialrelation, wie gezeigt, lediglich aus der Perspektive einer bestimmten Interpretation des Proemialkombinators PR als emergentes Oberflächenphänomen auf, sind jedoch nicht bereits der Architektur der Kombinatormaschine inhärent. Es kann daher eingewandt werden, das hier keine transklassische Modellierung, sondern lediglich eine bestimmte Applikation und Interpretatin einer klassischen Formalisierung vorliegt. Diese Einschränkung ergibt sich zwangsläufig, da das Modell ja im Sprachrahmen klassischer formaler Systeme und Programmiersprachen — *positivsprachlich* — formuliert wurde.

Der eigentliche Sinn des vorgeschlagenen Modells besteht somit primr nicht darin, die Proemialrelation zu operationalisieren, sondern darin, auf sie zu *deu-*

[40] Günther (1980), FN 5, S. 33.

ten als auf etwas, das *da ist*, aber eben — positivsprachlich — nicht genannt werden kann.

7 Ausblick

Der Kombinator PR ermöglicht die formale Beschreibung und Implementierung paralleler Prozesse unter genauer Spezifizierung der von mehreren Prozessen gleichzeitig verwendeten Objekte. Der Sprachrahmen funktionaler Programmiersprachen kann durch die Einführung dieses Konstruktes um die Komposition verkoppelter paralleler Prozesse erweitert werden. Aus dieser Erweiterung ergeben sich eine Fülle von Anwendungsmöglichkeiten, von denen schon einige angedeutet wurden. Es sollen hier noch stichpunktartig weitere Anregungen für zukünftige Untersuchungen gegeben werden.

Ein naheliegender Schritt wäre es, das vorgestellte Berechnungsmodell zu einer vollständigen Programmiersprache auszubauen oder in bestehende Systeme zu integrieren. Diese Programmiersprachen können dann zur Implementierung verkoppelter Parallelität (insbesondere der Proemialrelation) verwendet werden. Mittels dieser um wenige Konstrukte erweiterten funktionalen Sprachen können dann formale Modelle von Prozeßkommunikation und –interaktion komplexer Systeme (z.b Betriebssysteme) erstellt werden.

Außer diesem klassischen Anwendungsfeld bieten sie aber auch die Möglichkeit, die Gültigkeit des hier entwickelten Modells der Proemialrelation anhand der Modellierung und Simulation proemieller, selbstreferentieller und verteilter Systeme zu verifizieren. Sollte sich die Adäquatheit des Modells erweisen, müßte es möglich sein, transklassische Konzeptionen wie polykontexturale Logik oder polykontexturale Arithmetik mittels dieser Programmiersprachen zu implementieren.

Als grundsätzliche Darstellungsbarriere der Proemialrelation erwiesen sich der Objekt-, Zeichen-, und Identitätsbegriff der klassischen Semiotik und der auf ihr basierenden positivsprachlichen Zeichensysteme. Von besonderem Interesse für eine adäquate Formalisierung der Proemialrelation ist daher eine Kritik und Neufassung der hier vorgeschlagenen Modellierung aus der Sicht der Güntherschen Theorie der *Negativsprachen* [41].

[41] Ditterich, J., Kaehr, R.: *Einübung in eine andere Lektüre. Diagramm einer Rekonstruktion der Güntherschen Theorie der Negativsprachen.* In: Philosophisches Jahrbuch, 86. Jg., 2. Halbband, Freiburg und München, S. 385-408, 1979.

Literatur

[Bar80] Barendregt, H.P.: *The Lambda-calculus. Its Syntax and Semantics.* Amsterdam, North-Holland, 1980.

[Cur69] Curry, H.B., Feys, R.: *Combinatory Logic.* Amsterdam, North-Holland, 1969.

[Dav92] Davies, A.J.T.: *An Introduction to Functional Programming Systems Using Haskell.* Cambridge Computer Science Texts 27, Cambridge University Press, 1992.

[Dil88] Diller, A.: *Compiling Functional Languages.* New York, John Wiley & Sons, 1988.

[Foe76] Von Foerster, H.: *Objects: Tokens for (Eigen)-behaviors.* In: ASC Cybernetics Forum VIII (3,4), S. 91-96, 1976.

[Gun78] Günther, G.: *Idee und Grundriß einer nicht-aristotelischen Logik. Die Idee und ihre philosophischen Voraussetzungen.* 2. Auflage, Hamburg, Verlag Felix Meiner, 1978.

[Gun79] Günther, G.: *Identität, Gegenidentität und Negativsprache.* In: *Hegeljahrbuch 1979*, Pahl-Rugenstein, pp.22-28.

[Gun80a] Günther, G.: *Beiträge zur Grundlegung einer operationsfähigen Dialektik.* Bd. 1-3, Hamburg, Verlag Felix Meiner, 1976-1980.

[Gun80b] Günther, G.: *Cognition and Volition. A Contribution to a Cybernetic Theory of Subjectivity.* In: [Gun80a] Bd.2.

[Gun80c] Günther, G.: *Natural Numbers in Trans-Classic Systems. Part I, II.* In: [Gun80a] Bd.2.

[Gun95] Günther, G.: *Number and Logos. Unforgettable hours with Warren St. McCulloch.* In: *Jahrbuch für Selbstorganisation Band 5: Realitäten und Rationalitäten.* Kaehr, R., Ziemke, A. (Eds.), Berlin, Huncker & Dumblot, 1995

[Har91] Van Harmelen, Frank: *Meta-level Inference Systems.* London, Pitman, 1991.

[Kae78] Kaehr, R.: *Materialien zur Formalisierung der dialektischen Logik und der Morphogrammatik 1973-1975.* In: [Gun78], Anhang.

[Kae81] Kaehr, R.: *Das graphematische Problem einer Formalisierung der transklassischen Logik.* In: Beyer, W.R.(Ed.): *Die Logik des Wissens und das Problem der Erziehung.* Hamburg, Felix Meiner Verlag, 1981

[Kae82] Kaehr, R.: *Einschreiben in Zukunft.* In: Hombach, D.(Ed.): *Zeta 01. Zukunft als Gegenwart.* Berlin, Verlag Rotation, 1982.

[Kae88] Kaehr, R., Goldammer, von E.: *Again Computers and the Brain.* In: *Journal of Molecular Electronics.* Vol. 4, 1988, p. 31-37

[Kae92] Kaehr, R.: *Disseminatorik: Zur Logik der 'Second Order Cybernetics'. Von den 'Laws of Form' zur Logik derReflexionsform.* In: *Kalkül der Form.*, Baecker, D. (Ed.), Frankfurt a. M., Suhrkamp Verlag, 1993.

[Kae95] Kaehr, R., Mahler, Th.: *Proömik und Disseminatorik. I. Abbreviaturen transklassischen Denkens, II. Operationale Modellierung der Proemialrelation.* In:*Jahrbuch für Selbstorganisation Bd.5: Realitäten und Rationalitäten.* Kaehr, R., Ziemke, A. (Eds.), Berlin, Huncker & Dumblot, 1995

[Mae88] Maes, P., Nardi, D.: *Meta-Level Architectures and Reflection.* Amsterdam, North–Holland, 1988.

[Mah93] Mahler, Th., Kaehr, R.: *Morphogrammatik. Eine Einführung in die Theorie der Form.* Klagenfurter Beiträge, 1994.

[Pfa91] Pfalzgraf, J.: *Logical Fiberings and polycontextural systems.* In: *Fundamentals of Artificial Intelligence Research*, Ph. Jorrand, J. Kelemen (Eds.), pp. 170-184, New York, Springer, 1991

[Smi82] Smith, B.C.: *Reflection and Semantics in a Procedural Language.* LCS Technical Report TR–272, MIT, Massachusets, 1982.

[Tur79] Turner, D.A.: *A New Implementation Technique for Applicative Languages.* Software Practise and Experience Bd. 9, 1979.