# Ferienakademie 2000

## Kurs 1: Formale Modelle von Programmiersprachen


**Stefan Büttcher**

stuckel@bigfoot.com

# Inhalt

# 1. Einführung

*„The mismatch between the operational semantics of the lambda calculus and the actual behavior of implementations is a major obstacle for compiler writers. They cannot explain the behavior of their evaluators in terms of source level syntax, and they cannot easily compare distinct implementations of different lazy strategies."*
(P. Wadler et al., POPL '95 1/95)

Funktionale Programmiersprachen mit *lazy evaluation* implementieren den Call by Name λ-Kalkül. Die Semantik von Funktionsaufrufen in diesen Programmiersprachen ist die β-Reduktion des λ-Kalküls in *normal order*: Jeder Funktionsaufruf ist äquivalent zum Rumpf der Funktion, in dem jedes Auftreten des formalen Parameters textuell durch den aktuellen Parameter ersetzt wurde (im Gegensatz zu Call by Value, bei dem jedes Auftretenb des formalen Parameters durch den Wert des aktuellen Parameters ersetzt wird).

Sinnvolle Implementierungen funktionaler Programmiersprachen mit *lazy evaluation* umgehen die mit Call by Name verbundene wiederholte Auswertung des selben Parameters, die ja immer den selben Wert als Resultat hat, dadurch, dass sie bei der ersten Auswertung den Wert des Funktionsarguments speichern und später statt einer erneuten Auswertung lediglich auf diesen gespeicherten Wert zurückgreifen. Dieses Verfahren trägt den Namen „Call by Need".

Ich werde im Folgenden zeigen, auf welche Weise Call by Need mit Hilfe von λ-Termen dargestellt werden kann und welche Reduktionsregeln im Gegensatz zu Call by Name und Call by Value zum Einsatz kommen.

Schließlich werde ich eine mögliche (von Friedman und Wise 1976 vorgestellte) Implementierung von Call by Need präsentieren, die das *lazy* Verhalten bereits auf Sourcecode-Ebene charakterisiert.

# 2. Reduktionsstrategie

## 2.1. Makrodefinitionen

Boolesche Werte / IF-Konstrukt

True := λx.λy.x False := λx.λy.y IF [A, B, C] := A B C

Die Anforderungen an IF werden offenbar erfüllt:

IF [True, A, B] = (lx.ly.x) B C → B und IF [False, A, B] = (lx.ly.y) B C → C.

Paare und Listen

Pair [A, B] := λx.IF[x, A, B] = λx.x A B

First[A] := A True Rest[A] := A False

Nil := λx.True Null[A] := A (λx.λy.False)

Listen werden sukzessive in der üblichen Weise aufgebaut:
List [] = Nil, List[A] = Pair[A, Nil], List[A, B] = Pair[A, Pair[B, Nil]], usw.

Durch einsetzen der Definitionen für True und False und Nachrechnen lassen sich folgende Gültigkeiten und damit der Sinn der Definitionen leicht überprüfen:

First[Pair[A, B]] = A, Rest[Pair[A, B]] = B,
(Null[A] = True <=> A = Nil), (Null[A] = False <=> A != Nil).

Natürliche Zahlen

Zero := lx.x IsZero[A] := First[A] = A True

Succ[A] := Pair[False, A] Pred[A] := Rest[A]

Auch der Sinn dieser Definitionen lässt sich durch Nachrechnen leicht überprüfen.

## 2.2. Call by Name vs Call by Value

**Call by Name**
(normal order reduction)

Der Funktionsaufruf erfolgt durch textuelle Ersetzung des formalen Parameters durch den aktuellen Parameter im Rumpf der Funktion und anschließende Auswertung des auf diese Weise gewonnenen neuen Rumpfes.

| | | |
|---|---|---|
| Syntax | Variablen | x, y, z |
| | Werte | V, W ::= x \| λx.M |
| | Terme | L, M, N ::= V \| M N |
| | Kontexte | E ::= [] \| E M |
| Reduktionsregel | (β) | E [ (λx.M) N ] → E [ M [x := N] ] |

**Call by Value**
(applicative order reduction)

Der Funktionsaufruf erfolgt durch Ersetzung des formalen Parameters durch den *Wert* des aktuellen Parameters im Rumpf der Funktion und anschließende Auswertung des auf diese Weise gewonnenen neuen Rumpfes.

| | | |
|---|---|---|
| Syntax | Variablen | x, y, z |
| | Werte | V, W ::= x \| λx.M |
| | Terme | L, M, N ::= V \| M N |
| | Kontexte | E ::= [] \| E M \| (λx.M) E |
| Reduktionsregel | ($\beta_v$) | E [ (λx.M) V ] → E [ M [x := V] ] |

## Beispiel 1

Call by Name
$(\lambda x.x+x+x)\ ((\lambda z.z)\ 2)$
$\rightarrow_\beta\ ((\lambda z.z)\ 2) + ((\lambda z.z)\ 2) + ((\lambda z.z)\ 2)$
$\rightarrow_\beta\ 2 + ((\lambda z.z)\ 2) + ((\lambda z.z)\ 2)$
$\rightarrow_\beta\ 2 + 2 + ((\lambda z.z)\ 2)$
$\rightarrow_\beta\ 2 + 2 + 2$

Call by Value
$(\lambda x.x+x+x)\ ((\lambda z.z)\ 2)$
$\rightarrow_\beta\ (\lambda x.x+x+x)\ 2$
$\rightarrow_\beta\ 2 + 2 + 2$

In diesem Beispiel können mit Call by Value gegenüber Call by Name zwei Reduktionsschritte gespart werden.

## Beispiel 2

Call by Name
$(\lambda x.\lambda y.IF[IsZero[y], x, 0])\ ((\lambda z.z)\ a)\ 1$
$\rightarrow_\beta\ (\lambda y.IF[IsZero[y], (\lambda z.z)\ a, 0])\ 1$
$\rightarrow_\beta\ IF[IsZero[1], (\lambda z.z)\ a, 0]$
$\rightarrow_\beta^3\ IF[False, (\lambda z.z)\ a, 0]$
$\rightarrow_\beta^2\ 0$

Call by Value
$(\lambda x.\lambda y.IF[IsZero[y], x, 0])\ ((\lambda z.z)\ a)\ 1$
$\rightarrow_\beta\ (\lambda x.\lambda y.IF[IsZero[y], x, 0])\ a\ 1$
$\rightarrow_\beta\ (\lambda y.IF[IsZero[y], a, 0])\ 1$
$\rightarrow_\beta\ IF[IsZero[1], a, 0]$
$\rightarrow_\beta^3\ IF[False, a, 0]$
$\rightarrow_\beta^2\ 0$

Hier benötigt Call by Value wegen der Reduktion von $(\lambda z.z)\ a$ einen Schritt mehr als Call by Name.

## Erkenntnis

Call by Name wertet einen Parameter so oft aus, wie er im Rumpf der Funktion benötigt wird.
Call by Value wertet einen Parameter auch dann aus, wenn sein Wert überhaupt nicht benötigt wird.

Gesucht ist ein effizientes Verfahren, das die Vorzüge von Call by Name und Call by Value miteinander vereint: *Call by Need*.

# 3. Formale Definition von Call by Need

## 3.1. Syntax und Reduktionsregeln

| | | | |
|---|---|---|---|
| Syntax | Variablen | x, y, z | |
| | Werte | V, W ::= x \| λx.M | |
| | Terme | L, M, N ::= V \| M N \| let x = M in N | |
| Reduktionsregeln | (I) | (λx.M) N | → let x = N in M |
| | (V) | let x = V in C[x] | → let x = V in C[V] |
| | (C) | (let x = L in M) N | → let x = L in (M N) |
| | (A) | let y = (let x = L in M) in N | → let x=L in (let y=M in N) |
| | (G) | let x = M in N | → N, wenn x ∉ fv(N) |

Regel I (*introduction*) erzeugt aus einer Anwendung eine let-Bindung: Erzeugen einer Kopie von M, Ersetzen aller x durch eine Referenz auf den Graphen von N.

Regel V (*value*) ersetzt eine let-gebundene Variable durch einen Wert: Dereferenzieren.

Regel C (*commute)* erlaubt es let-Bindungen, mit Anwendungen zu kommutieren: Herausziehen einer let-Bindung aus dem Funktionsteil einer Anwendung.

Regel A (*associate*) überführt links stehende in rechts stehende let's.

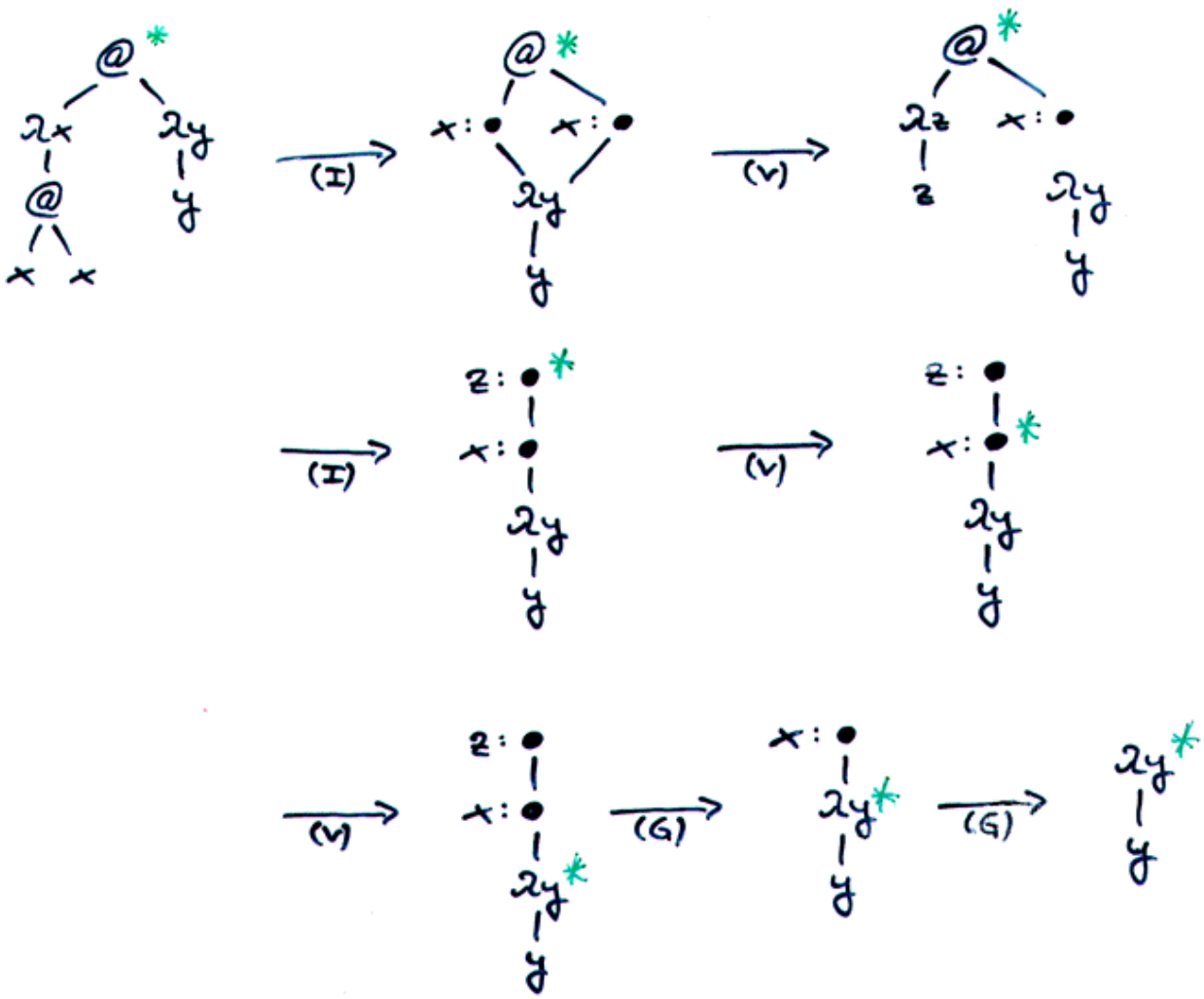
Regel G (*garbage collection*) entfernt eine let-Bindung, deren Variable x nicht mehr im Term N auftaucht.

## 3.2. Beispielreduktion

Darstellung durch $\lambda_{Need}$-Terme

$(\lambda x.xx)(\lambda y.y)$ $\rightarrow_I$ let x = ly.y in x x
$\rightarrow_V$ let x = ly.y in (lz.z) x
$\rightarrow_I$ let x = ly.y in let z = x in z
$\rightarrow_V$ let x = ly.y in let z = x in x
$\rightarrow_V$ let x = ly.y in let z = x in ly.y
$\rightarrow_G^2$ ly.y

Darstellung durch Graphen




Der mit * gekennzeichnete Knoten sei in der aktuellen Betrachtung jeweils die Wurzel des Graphen (*marked reduction*).

## 3.3. Notwendigkeit von let-A

“Naive” Anwendung von Let-V:

let x = (let y = M in V) in C[x]

→ let x = (let y = M in V) in C[let y = M in V]

bewirkt den Verlust des Sharings:

let f = (let z = II) in λw.zw) in fI (fI)

→ let f = (let z = II in λw.zw) in (let z = II in λw.zw) I (fI).

Der Redex II muss doppelt reduziert werden. Außerdem lässt die Grammatik mit der vorliegenden Definition von let-V eine solche Reduktion gar nicht zu.

Die Lösung des Problems erfolgt durch let-A:

let f = (let z = II in lw.zw) in fI (fI)

→ let z = II in let f = λw.zw in fI (fI)

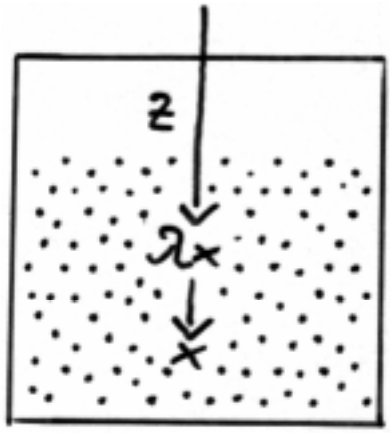
→ let z = II in let f = λw.zw in (λw.zw)I (fI).

Obwohl f zweimal im auftaucht, findet nur eine einzige Reduktion statt!

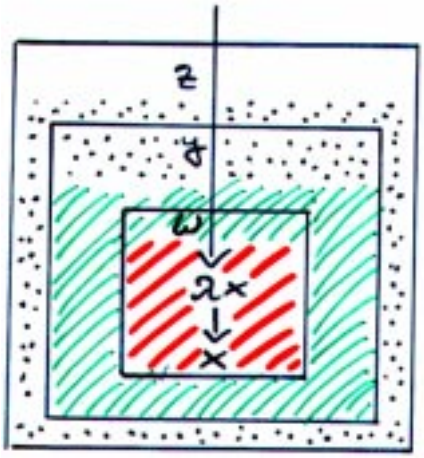
Entsprechende Anwendung findet die *commute*-Regel let-C.

## 3.4. Darstellung von let-Ausdrücken durch Graphen


let x = M in N

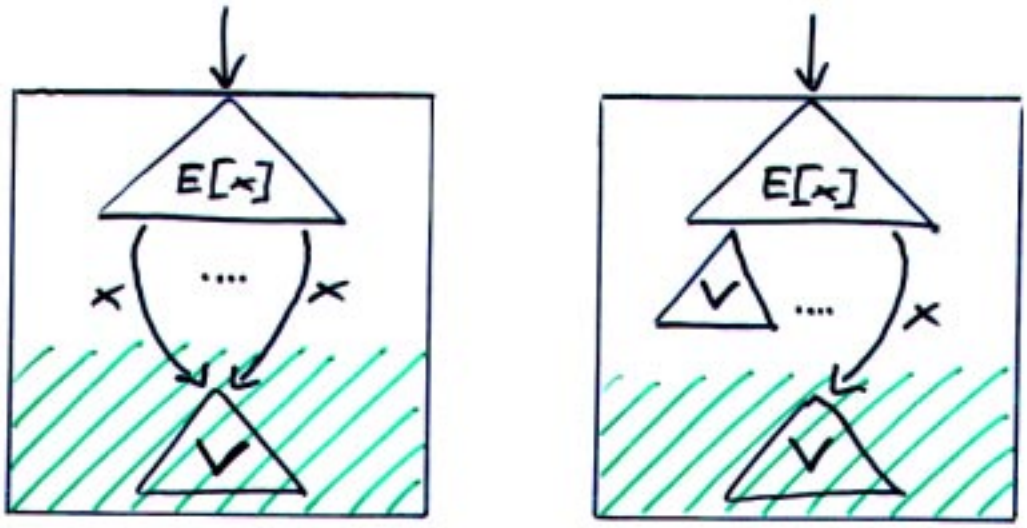



let z = λx.x in z



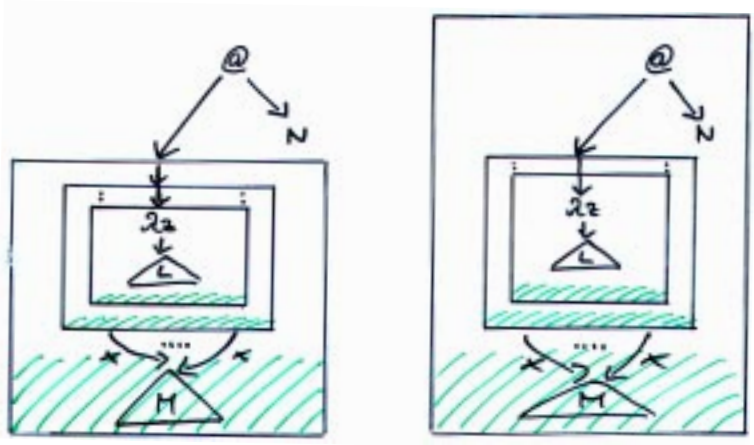

let z = let y = let w = λx.x in w in y in z

**let-V:**



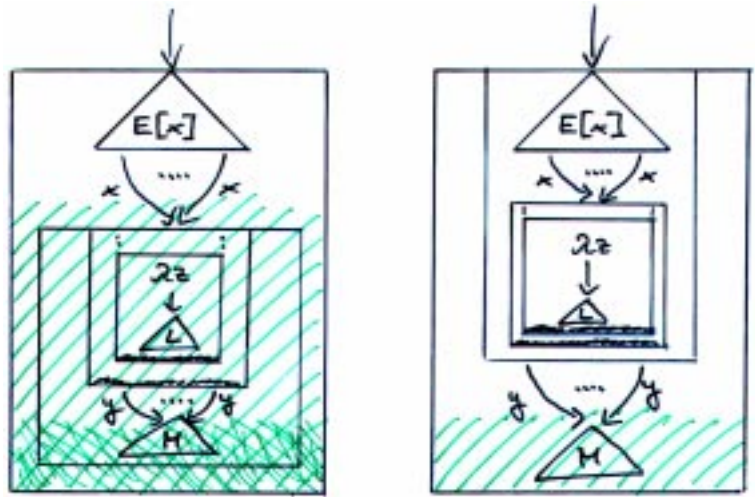

let x = V in E[x] → let x = V in E[V]

**let-C :**




(let x = M in P) N) → let x = M in (P N)

**let-A:**




let y = (let x = L in M) in N → let x = L in (let y = M in N)

## 3.5. Eigenschaften des $\lambda_{Need}$-Kalküls

∀ Terme M:

M →* M′ Λ M →* M″ => ∃ N: M′ →* N Λ M″ →* N.

(Konfluenzeigenschaft)


Im Übrigen lässt sich zeigen, dass die Auswertung eines Terms nach Call by Need genau dann konvergiert, wenn sie es nach Call by Name tut, und auch zum selben Wert. Es liegt also Äquivalenz vor.

# 4. Eine mögliche Implementierung von Call by Need

## 4.1. Eine Implementierung mit Call by Value

**Ausgangspunkt** 5 strikte Basis-Funktionen: atom, car, cdr, cons, eq

Def.: Eine Funktion heiße „strikt in ihrem i-ten Parameter", wenn die Divergenz des i-ten Arguments die Divergenz der Funktion mit diesem Argument impliziert.

Def.: Eine Funktion heiße „strikt", wenn die Funktion in allen ihren Parametern strikt ist.

Eine strikte Funktion kann ausgewertet werden, indem alle ihre Argumente ausgewertet werden, bevor ihre Definition interpretiert wird.
In einer Umgebung von ausschließlich strikten Funktionen entspricht das Verhalten dem von Call by Value.

**Vorstellung der Basis-Funktionen**

(atom x) überprüft, ob es sich bei x um ein Atom handelt und liefert True bzw. False (entspricht (not (pair? x)) ).

(cons x y) liefert eine Referenz auf das Paar (x, y).

(car z) liefert das erste Element eines mit *cons* erzeugten Paares:
(car (cons x y)) = x.

(cdr z) liefert das zweite Element eines mit *cons* erzeugten Paares;
(cdr (cons x y)) = y.

(eq x y) überprüft x und y auf Gleichheit (Identität) und liefert True bzw. False.

## Aufrufkonvention

Außer den Basis-Funktionen besitze jede Funktion höchstens einen Parameter. Dieser Parameter sei eine Liste der eigentlichen Parameter.

## Beispiele

```
(f x)        →  (f (cons x Nil))

(g x y z)    →  (g (cons x (cons y (cons z Nil))))

(define (f x)
      (succ x)
)              →
(define (f l)
      (succ (car l))
)

(define (g z y z)
      (add x (mul y z))
)              →
(define (g l)
      (add (car l)
      (mul (car (cdr l))
            (car (cdr (cdr l)))
)     ))
```

## Beobachtung

*cons* wertet seine Argumente sofort aus, da es per definitionem eine strikte Funktion ist; jede Funktion erhält ihre Argumente durch eine von *cons* erzeugte Liste.

Folglich werden bei jedem Funktionsaufruf alle Argumente sofort ausgewertet, es liegt also Call by Value vor.

## 4.2. Modifikation zu „Call by delayed Value”

**Ziel:** Es sollen immer nur die Elemente einer Liste ausgewertet werden, deren Wert für die Ausführung tatsächlich benötigt werden.

Def.: Ein unausgewerteter Ausdruck heiße „Form”.

Def.: Ein „Environment” sei eine Funktion, die formalen Parametern ihre Werte zuweist.

Def.: Eine „Suspension” sei eine nur dem Interpreter zugängliche Datenstruktur, bestehend aus einer Form und einen Environment, für die eventuelle Auswertung der Form.

Die Funktion *suspend*: F x E $\rightarrow$ S erhalte eine Form und ein Environment als Argumente und erzeuge aus ihnen eine Suspension, deren Referenz sie als Funktionswert zurückliefert.

Die Selektorfunktionen *form* und *env* liefern das entsprechende Feld einer Suspension. Die Funktion *suspended*: T $\rightarrow$ Boolean prüft, ob es sich bei ihrem Argument um eine Suspension handelt.

Die Funktion *eval*: F x E $\rightarrow$ V erhält als Argumente eine Form und ein Enviroment uind liefert als Ergebnis den Wert der Form im angegebenen Environment.

Zur Modifikation auf Call by delayed Value müssen nun die Basis-Funktionen in bestimmter Weise abgeändert werden:

Die neue Funktion *cons'* wird implizit definiert. Sie erzeuge aus ihren Parametern x und y eine Liste und übergebe diese zusammen mit dem für die spätere Auswertung notwendigen Environment an eine Funktion *scons'*:

```
(define (scons' args environment)
      (cons (suspend (car args) environment)
            (suspend (car (cdr args)) environment)
)     )
```

In entsprechender Weise müssen nun die Funktionen *car'* und *cdr'* definiert werden:

```
(define (car' x)
      (eval (form (car x)) (env (car x)) )
)
```

```
(define (cdr' x)
      (eval (form (cdr x)) (env (cdr x)) )
)
```

Die durch *eval* erzwungene Auswertung des ersten bzw. zweiten Elements des von *cons'* erzeugten Paares innerhalb von *car'* bzw. *cdr'* stoppt, sobald ein Atom oder eine Anwendung von *cons'* erreicht wird (vgl. Definition von *eval*, S. 18).

Die auf diese Weise durch die Ersetzung von cons, car und cdr durch cons', car' und cdr' gewonnene Umgebung entspricht weitgehend Call by Name. Während Call by Name jedoch immer eine vollständige Auswertung eines bestimmten Parameters verlangt (i.e. wird ein Parameter überhaupt ausgewertet, so wird er vollständig ausgewertet), wird in dieser Umgebung ein Parameter immer nur so weit ausgewertet, wie es unbedingt nötig ist.

**Beispiele**

```
(car' (cons' (cons' (f x) (g y)) (h z)))
```

liefert eine Referenz auf (cons' (f x) (g y)), ohne (f x), (g y) oder (h z) auszuwerten.

Die Folge $a_n := 1/n^2$: 1/1, 1/4, 1/9, 1/16, ... kann nun durch einen Funktionsaufruf (a 1) erzeugt werden:

```
(define (a n)
      (cons' (reciprocal (square n))
            (a (succ n))
)     )
```

Der Aufruf (car' (cdr' (cdr' (a 1)))) liefert erwartungsgemäß den Wert 1/9.

Leider ist das vorgestellte Schema für den praktischen Gebrauch ungeeignet: Die Asuwertung einer Suspension wird bei jedem Auftreten des Parameters im Funktionsrumpf immer wieder erzwungen, nur um jedes Mal zum selben Ergebnis zu kommen. („Suicidal Suspensions")

## 4.3. Modifikation zu Call by Need

Durch eine Änderung der Selektorfunktionen *car'* und *cdr'* soll nun wirkliches Call by Need erreicht werden.

Es sei *rplacliba* („**r**e**plac**e and **lib**erate **a**") eine Funktion mit zwei Argumenten x und y. x ist ein von *cons'* allokierter Knoten, y ein beliebiger Wert im Sinne der Grammatik von Call by Need.

Funktionsweise von *rplacliba*:

- Zwischenspeichern der Referenz im A-Feld von x in eine lokale Variable,
- Speichern der Referenz von y im A-Feld von x,
- Freigeben des von *cons'* allokierten Speichers des ursprünglich im A-Feld referenzierten Objekts,
- Zurückgeben von y als Funktionswert.

In entsprechender Weise sei *rplaclibd* für das D-Feld definiert.

Beide Funktionen *rplacliba* und *rplaclibd* sind für den Benutzer unsichtbar. Das Argument x ist in jedem Fall eine Suspension, das ersetzte Feld Referenz eines Atoms oder *cons'*-Knotens.

Mit der Definition der Replace/Liberate-Funktionen resultieren jetzt die folgenden Definitionen für *car''* und *cdr''*.

Aus Gründen der Übersichtlichkeit wurde von der üblichen Schreibweise abgewichen und in **Rot** then und else als Hilfen eingefügt.

```
(define (car'' x)
      (if (suspended (car x))
      then (rplacliba x (eval (form (car x)) (env (car x)) ))
      else (car x)
)     )
```

```
(define (cdr'' x)
      (if (suspended (cdr x))
      then (rplaclibd x (eval (form (cdr x)) (env (cdr x)) ))
      else (cdr x)
)     )
```

**Ergebnis**

Die durch die Ersetzung von *cons*, *car* und *cdr* durch *cons'*, *car''* und *cdr''* gewonnene Umgebung erfüllt die Forderungen von Call by Need (wie sich aus den ab S. 18 zu findenden Quellen ersehen lässt):

Funktionsparameter werden nur dann ausgewertet, wenn es nötig ist (z.B. der Bedingung eines IF-Konstrukts), und auch dann nur so weit, wie es nötig ist.

Ein einzelner Ausdruck wird höchstens einmal ausgewertet.

## 4.4. Der vorgestellte Interpreter im Quelltext

```
(scons ab env) =
      (:cons (suspend (car ab) env))
             (suspend (car (cdr ab)) env)).

(car x) =
      (if suspended (:car x))
            then (rplacliba x (eval (form :car x)) (env (:car x)) ))
            else (:car x) ).

(cdr x) =
      (if suspended (:cdr x))
            then (rplaclibd x (eval (form :cdr x)) (env (:cdr x)) ))
            else (:cdr x) ).

(eq x y) =
      (:eq x y).

(atom x) =
      (:atom x).

(same sexp atm) =
      (if (atom sexp) then (eq sexp atm) else NIL).

(eval form env) =
      (cond if (atom form) then (assoc form env)
      elseif (atom (car form)) then (cond
            if (eq (car form) QUOTE) then (car (cdr form))
            elseif (eq (car form) CONS) then (scons (cdr form) env)
            elseif (eq (car form) COND) then (evcon (cdr form) env)
            else (apply (car form) (evlis (cdr form) env) env))
      else (apply (car form) (evlis (cdr form) env) )

(apply fn args env) =
      (cond if (atom fn) then (cond
            if (eq fn CAR) then (car (:car args))
            elseif (eq fn CDR) then (cdr (:car args))
            elseif (eq fn EQ) then (eq (:car args) (:car (:cdr args)))
            elseif (eq fn ATOM) then (atom (:car args))
            elseif (eq fn NIL) then NIL
            else (apply (eval fn env) args env) )
      elseif (same (car fn) LAMBDA) then (eval (caddr fn) (pairlis (car (cdr fn)) args env))
      elseif (same (car fn) LABEL) then
                  (apply (caddr fn) args) (:cons (:cons (cadr fn) (caddr fn)) env))
      elseif (anynull args) then NIL
      else (cons (apply (car fn) (carlis args) env) (apply (cdr fn) (cdrlis args) env)) )
```

```
(pairlis fpl apl env) =
      (cond if (atom fpl) then (cond
            if (eq fpl NIL) then env
            else (:cons (:cons fpl apl) env))
      else (pairlis (car fpl) (:car apl) (pairlis (cdr fpl) (:cdr apl) env)) )

(assoc at env) =
      (cond if (eq (:car (:car env)) at)
            then (:cdr (:car env))
            else (assoc at (:cdr env)) )

(evlis unargs env) =
      (cond if (atom unargs) then NIL
            else (:cons (eval (car unargs) env) (evlis (cdr unargs) env)) )

(evcon tail env) =
      (cond if (atom tail) then NIL
            elseif (atom (cdr tail)) then (eval (car tail) env)
            elseif (eval (car tail) env) then (eval (car (cdr (tail)) env)
            else (evcon (cdr (cdr tail)) env) )

(anynull list) =
      (cond if (atom lis) then FALSE
            elseif (same (:car lis) NIL) then TRUE
            else (anynull (:cdr lis)) )

(carlist mtx) =
      (cond if (atom mtx) then NIL
            else (cons (car (:car mtx)) (carlis (:cdr mtx))) )

(cdrlis mtx) =
      (cond if (atom mtx) then NIL
            else (cond (cdr (:car mtx)) (cdrlisr (:cdr mtx))) )
```

# 5. Literaturangaben

[1] Z. Ariola, P. Wadler et al.: „A Call-By-Need Lambda Calculus”, Principles of Programming Languages (PoPL) ‘95, 1/95

[2] J. Maraist, M. Odersky, P. Wadler: „The Call-by-Need Lambda Calculus” Journal of Functional Programming 1998

[3] D. Friedman, D. Wise: „CONS should not evaluate its arguments”, Int. Conference on Automata, Languages and Programming (ICALP) ‘76

[4] R. Stansifer: „Theorie und Entwicklung von Programmiersprachen”, Prentice-Hall 1995