Philipps-Universität Marburg

Fachbereich Mathematik und Informatik

03.07.2005

Seminar: Konzepte von Programmiersprachen

Autorin: Carina M. Ringler

Betreuer: Jörn Abels

Modulare Interpreter durch Monadentransformation

1. Kapitel: Inhalt

1. KAPITEL: INHALT 2

2. KAPITEL: EINFÜHRUNG 3

3. KAPITEL: MONADE UND MONADENTRANSFORMATION 4

Monade 4

Monadentransformation 7

DER INTERPRETER 10

Elementare Bausteine 10

Erweiterbare Vereinigungstypen 12

Benötigte Monadentransformationen 13

Lifting Operations 15

4. KAPITEL: FAZIT 17

5. KAPITEL: LITERATUR 18

6. KAPITEL: ANLAGE 19

Bausteine im Programmcode 19

2. Kapitel: Einführung

Diese Arbeit beschäftigt sich mit der Kombination zweier verschiedener Monaden mittels Monadentransformationen, den Nutzen solcher Kombinationen wird wie in der Hintergrundliteratur von S. Liang, P. Hudak und M. Jones [1] auch, anhand der Programmierung eines Interpreters erläutert werden.
Bei diesem Interpreter wird man von wenigen Monaden ausgehen, in denen spezielle Features der Programmiersprache erklärt und deren Interpretation definiert wird, und baut durch mehrfaches anwenden von Monadentransformationen einen kompletten modularen Interpreter auf.
Dazu wird man im Folgenden erst einmal Monaden und Monadentransformationen definieren und erläutern.
Im Anschluss daran werden die für den Interpreter notwendigen elementaren Monaden, die so genannten Elementarbausteine, und die benötigten Monadentransformationen besprochen.
Zum Ende dieser Arbeit wird man dann auf die bei der mehrfachen Transformation von Monaden auftretenden Probleme dargelegt und Lösungsmöglichkeiten angesprochen.
Zur Verdeutlichung der angesprochenen Fragen und Probleme wurde der angesprochene Interpreter in dieser Arbeit zum Teil ausprogrammiert und auch die Definitionen sind im Programmcode als Abbildungen eingefügt worden. Dieser Programmcode wurde in der funktionalen Programmiersprache Haskell verfasst.

3. Kapitel: Monade und Monadentransformation

Monade

Eine Monade ist eine spezielle Konstruktorenklasse, die mit den Operationen unit und bind versehen ist, welche wie in Abbildung 1 definiert sind.

```
class Monade m where
     unit :: a -> m a
     bind :: m a -> (a -> m b) -> m b
```

Abbildung 1: Monadendefinition

Diese Operationen müssen die in Abbildung 2 aufgeführten Gleichungen, auch Monadengesetze genannt, erfüllen.

```
Linke Verträglichkeit:
      bindM (unitM a) k = k a
Rechte Verträglichkeit:
      bindM m unitM = m
Assoziativität:
      bindM m (bindM (\a -> k a) (\b -> h b))
      = bindM (bindM m (\a -> k a)) (\b -> h b)
```

Abbildung 2: Monadengesetze

In der gängigen Literatur ist die Definition der Monade mit den Operationen map, unit und join gebräuchlich, diese müssen wiederum eigene Monadengesetze erfüllen, wie in Abbildung 3 angegeben.

```
class Monade m where
     map :: (a->b)->(Ma -> Mb),
     unit:: a->Ma
     join:: M(Ma)->Ma

     Zughörige Monadengesetze:
     (1) map id = id
     (2) map f . map g = map (f . g)
     (3) unit . f = map f . unit
     (4) join . map (map f) = map f . join
     (5) join . unit = id
```

Abbildung 3: Alternative Monadendefinition

Mit Hilfe der in Abbildung 4 genannten Gleichungen, die von den vier Operationen erfüllt werden, kann die Äquivalenz der beiden Definitionen bewiesen werden, was Wadler [2] in seiner Arbeit ausführlich getan hat.

```
bind m f  = join (map f m)

join z    = bind z (\m → m)
map  f m  = bind m (\a → unit (f a))
```

Abbildung 4: Zusammenhang der Monadendefinitionen

Das gebräuchlichste Beispiel für eine Monade ist die Listenmonade, wie sie in Abbildung 5 beschrieben ist.

```
instance Monade Liste where
     unit x         = [x]
     bind li k      = flatten (map k li)
```

Abbildung 5: Listenmonade

Mit denen in Haskell vordefinierten Funktionen map und flatten. Mit ähnlichen Definitionen kann man auch die Datenstrukturen Baum, Menge und Multimenge als Monaden auffassen.

Es existieren aber auch weniger bekannte Monaden wie die Statemonade die in Abbildung 6 zu sehen ist.

```
instance Monade (State s) where
unit a = State (\s ->(s,a))
bind (State m) f = State (\s ->let (s’,a)=m s
                                       State m’=f a
                                    in m’ s’)
```

Abbildung 6: Statemonade

Monadentransformation

Wie oben bereits erwähnt sind Monaden spezielle Konstruktorenklassen. Nun kann man speziellere Klassen definieren indem man zusätzliche Operationen verlangt. Eine Monadentransformation fügt nun einer gegebenen Monaden solche zusätzliche Operationen zu bildet also eine gegebene Monade in eine speziellere Konstruktorenklasse ab, wobei das Bild wieder eine Monade ist.
Dazu definiert man allgemein eine Klasse Monadentransformationen, deren Instanzen nur dann wohldefiniert sind, wenn Monaden auf Monaden abgebildet werden.

```
class (Monade m, Monade (t m))⇒ MonadT t m  where
     lift  :: m a → t m a
```

Abbildung 7: Definition Monadentransformationen

Mit einer solchen Monadentransformation kann man nun das obige Beispiel der Statemonade zu einer ganzen Klasse von Statemonaden erweitern, nämlich der Monaden mit zusätzlicher Zustandsbeschreibung und einer zusätzlichen Operation, die Den Zustand aktualisiert, der Operation *update*. Diese Monadenklasse ist in Abbildung 8 zu sehen.

```
------------------------------
-- State Monad Transformer --
------------------------------
instance Monade m => Monade (StateT s m) where
     unit x         = State (\s  ->     unit (s,x))
     bind (State l) k    = State (\s0 ->     bind (l s0)
(\(s1,a) ->
                                             let  State n = k a
                                             in   n s1      ))
instance (Monade m, Monade (StateT s m)) => MonadT (StateT s)
m where
     lift m = State (\s -> bind m (\x -> unit (s,x)))

class Monade m => StateMonad s m where
     update :: (s -> s) -> m s

instance Monade m => StateMonad s (StateT s m) where
     update f  = State (\s -> unit((f s),s))
```

Abbildung 8: Beispiel Statemonadentransformation

Jede Monadentransformation verfügt, wie in der allgemeinen Definition (Abb. 8) ersichtlich über die Operation lift, die den unten genannten Regeln genügt.

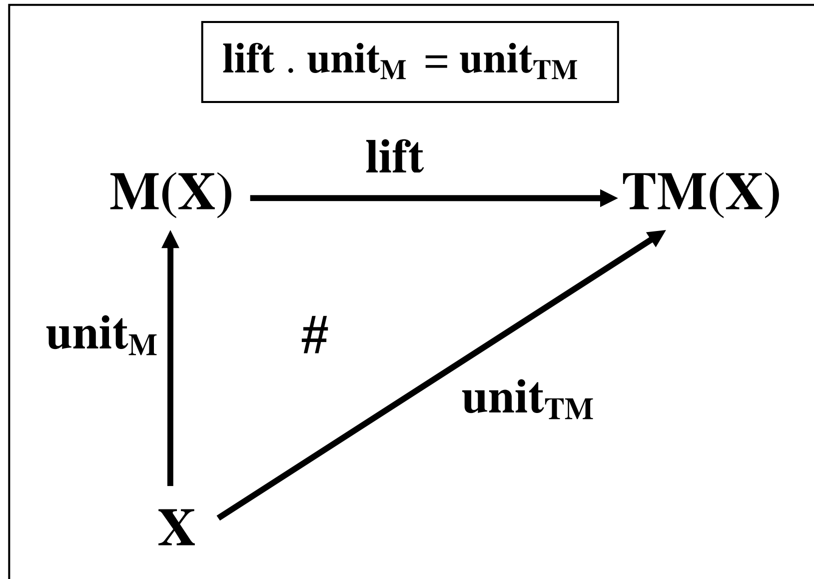

Abbildung 9: 1. Regel lift

Das heißt lift muss mit unit und bind so verträglich sein, dass es muss sowohl mit unit als auch mit bind eine gewisse Kommutativität vorhanden sein, die Reihenfolge der Anwendung von lift und unit bzw. lift und bind darf keine Veränderung des Ergebnisses zur Folge haben.

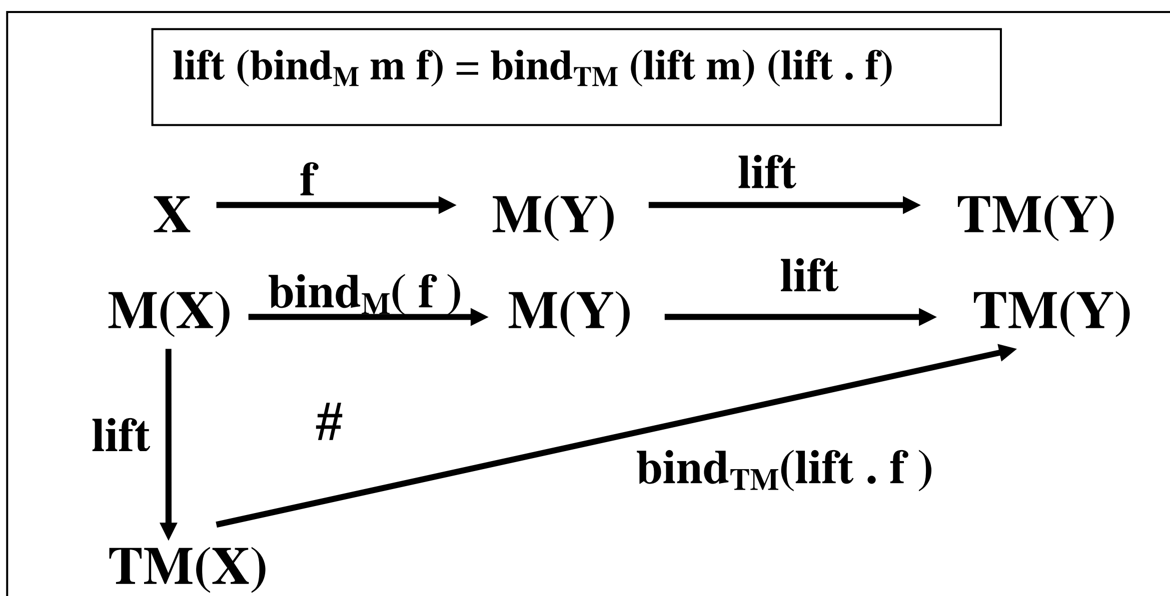

Abbildung 10: 2. Regel lift

Der Interpreter

Elementare Bausteine

Wie in der Einführung bereits erwähnt benötigt man zum Bau eines modularen Interpreters „atomare" Bausteine in denen spezielle Features der Programmiersprache und deren Interpretertion definiert werden. Im Ganzen benötigen wir davon sieben für die Interpretation von

- Arithmetischen Ausdrücken
- Funktionen
- Referenzen und Verweisen
- Lazy Evaluation
- Program Tracing
- Continuation
- Nichtdeterminismus.

Diese Monaden beinhalten jeweils Operationen die für den Interpreter benötigt werden (s. Tabelle 1).

Feature	Operation	Baustein
Error Handling	err:: String -> InterpM a	Arithmetischer
Nicht-determinismus	merge:: [InterpM a] -> InterpM a	Nicht-determismus
Environment	rdEnv:: InterpM Env inEnv:: Env -> InterpM a -> InterpM a	Funktionen
Store	allocLoc::InterpM Int lookupLoc:: Int -> InterpM a updateLoc:: (Int, InterpM Value) -> InterpM Int	Referenzen und Verweise
Stringausgabe	write:: String -> InterpM()	Program Tracing
Continuations	callcc:: ((a -> InterpM b) -> InterpM a) -> InterpM a	Continuation

Tabelle 1: Bausteinoperationen

Diese Bausteine sind in der Anlage im Programmcode zu sehen, sie werden darin zu Instanzen einer Klasse InterpC deklariert, die ihrerseits wieder eine Instanz der Interpretermonade InterpM (Abb. 11) ist und speziell für die Interpretertion von Termen angelegt wurde.

```
type InterpM   = StateT Store
          { EnvT Env
          { ContT Answer
          { StateT String
          { ErrorT
            List
            }}}}
```

Abbildung 11: Die Interpretermonade InterpM

Erweiterbare Vereinigungstypen

Um den Interpreter zusammenzusetzen, wird es außerdem nötig sein zwei disjunkte Datentypen zu einem zu vereinigen. Ausprogrammiert wurde dies in Abbildung 12. Dies ist eine sehr elegante Methode komplizierte Datentypen Baumförmig aufzubauen.

```
-- ====================== --
-- extensible union types ---
-- ====================== --

data OR a b  = L a |R b

class SubType sub sup where
        inj :: sub -> sup
        prj :: sup -> Maybe sub

instance SubType a (OR a b) where
        inj             = L
        prj (L x)       = Just x
        prj _           = Nothing

instance SubType a b => SubType a (OR c b) where
        inj             = R . inj
        prj (R a)       = prj a
        prj _           = Nothing
```

Abbildung 12: Erweiterbare Vereinigungstypen

Benötigte Monadentransformationen

Für den modularen Interpreter benötigen wir die folgenden Monadentransformationen:

- Statemonadentransformation
- Errormonadentransformation
- Environmentmonadentransformation
- Continuationmonadentransformation

Außerdem benötigen wir lediglich die in Abbildung 5 bereits vorgestellte Listenmonade.

Die Statemonadentransformation haben wir oben bereits gesehen, deshalb werde ich mich hier auf das Beispiel der Environmentmonadentransformation konzentrieren und diese erläutern.

```
---------------------------------------------
-- Environment Monad Transformer --
---------------------------------------------

instance (Monade m, Monade (EnvT r m)) => MonadT (EnvT r) m
where
        lift m = Env (\r -> m )

class Monade m => EnvMonad env m where
        inEnv :: env -> m a ->m a
        rdEnv :: m env

instance Monade m => Monade (EnvT r m) where
        unit a          = Env (\r -> unit a)
        bind (Env l) k          = Env (\r -> bind (l r) (\a ->          let
        Env n  = k a
                                                         in       n r
                                                         ))

instance Monade m => EnvMonad r (EnvT r m) where
        inEnv r (Env m)       = Env (\_ -> m r)
        rdEnv                 = Env (\r -> unit r)
```

Abbildung 13: Die Environmentmonadentransformation im Programmcode

Wie allgemein bereits in Kapitel 3 beschrieben wird hier die übergebene Monade m

in eine Environmentmonade umgewandelt, so dass den vorher schon zur Verfügung stehenden Monadenopertionen nun auch noch eine Umgebung mitgegeben wird in der gearbeitet werden kann. Dazu wird zuerst die Environmentmonadentransformation definiert, anschließend die Klasse der Environmentmonaden und deren speziellen Operationen definiert. In den letzten beiden Abschnitten in Abbildung 13 wird die transformierte Monade als Instanz der Klassen der Monaden und der Environmentmonaden deklariert und die entsprechenden Operationen für diese Monade definiert.
Die anderen Transformationen sind analog zu definieren. Dann kann man den Interpreter wie in Abbildung 14 zusammensetzen.

```
type InterpM   = StateT Store
          { EnvT Env
          { ContT Answer
          { StateT String
          { ErrorT
            List
            }}}}

type Term = OR TermA
          { OR TermF
          { OR TermR
          { OR TermL
          { OR TermT
          { OR TermC
               TermN
               }}}}}

type Value     = OR Int (OR Fun ())
```

Abbildung 14: Modularer Interpreter

Lifting Operations

In Abbildung 14 wird deutlich, wie wichtig es ist verschiedene Transformationen hintereinander ausführen zu können. Nun stellt sich das Problem, was zum Beispiel mit der Operation *update* passiert, wenn man eine Monadentransformation auf eine Statemonade anwendet.

Leider ist es bei diesem Problem, dem so genannten *Liften von Operationen,* nicht möglich eine für alle möglichen Fälle gültige Lösung anzugeben, da in ungünstigen Fällen der Rechenaufwand zu groß oder sonstige, derartige Schwierigkeiten auftreten können.

In dem oben beschriebenen Fall ist es jedoch ohne Schwierigkeiten möglich das neue *update* zu definieren, denn wie in Abbildung 13 zu sehen, genügt es die alte *update* der übergebenen Statemonade mit dem *lift* der Monadentransformation zu verketten.

```
instance (StateMonad m, MonadT t m) => StateMonad
(t m) where
     update    = lift . update
```

Abbildung 15: Natürliches Lifting

Glücklicherweise ist diese Lösung, auch *natürliches Lifting* genannt, für die meisten Fälle möglich.

Problematische Fälle, also die bei denen das natürliche Liften nicht möglich ist, hat bis auf wenige Ausnahmen bereits Moggi [3] in seiner Arbeit gelöst.

Eine dieser Ausnahmen ist der Fall der Anwendung einer Environmentmonadentransformation oder einer Statemonadentransformation auf eine Environmentmonade, dann ist es notwendig ein solch umständliches Lifting, wie in Abbildung 16 zu sehen, durchzuführen.

```
instance (MonadT (EnvT r') m, EnvMonad r m) =>
               EnvMonad r (EnvT r' m) where
     inEnv r m  = \ r' → inEnv r (m r')
     rdEnv       = lift rdEnv

instance (MonadT (StateT s) m, EnvMonad r m) =>
               EnvMonad r (StateT s m) where
     inEnv r m  = \ s → inEnv r (m s)
     rdEnv       = lift rdEnv
```

Abbildung 16: Unnatürliches Liften

Beim Liften ist grundsätzlich zu beachten, dass das Korrektheitskriterium, welches schematisch in Abbildung 15 dargestellt wird, nicht verletzt und letztendlich in Formeln fasst, was vom Liften erwartet wird.

$$\ell_\tau :: \tau \rightarrow \lceil \tau \rceil_t$$

(1) $\ell_A = id$

(2) $\ell_a = id$

(3) $\ell_{\tau_1 \rightarrow \tau_2} = \backslash f \rightarrow f' \, mit \, f'.\, \ell_{\tau_1} = \ell_{\tau_2} \cdot f$

(4) $\ell_{(\tau_1,\tau_2)} = \backslash \ (a,b) \rightarrow (\ell_{\tau_1} a, \ell_{\tau_2} b)$

(5) $\ell_{m\,\tau} = lift \,.\, (map\ \ell_\tau)$

Abbildung 17: Das Korrektheitskriterium

Bei diesem Korrektheitskriterium ist ℓ_τ die Funktion, die den Datentyp τ auf den transformierten Datentyp $\lceil \tau \rceil_t$ abbildet. Das Problem beim Liften steckt hierbei in der Gleichung (3), den es ist nicht offensichtlich, das zu einem gegebenen f ein f' existiert, welches die Gleichung (3) erfüllt.

4. Kapitel: Fazit

Der in Abbildung 14 gezeigte Interpreter ist ein gutes Beispiel für eine Anwendung von Monaden und der Kombination von Monaden durch Monadentransformationen, da durch die Monadenstruktur eine Isolation der Semantik einzelner Bausteine möglich wird, was zu mehr Übersicht und Verständlichkeit und zu einer Vereinfachung der Implementierung führt. Außerdem wird die Ergänzung weiterer Features vereinfacht.

Die Mächtigkeit dieser Datenkonstrukte zeigt sich vor allem darin, dass man neben den Grundbausteinen nur die Listenmonade und vier Monadentransformationen brauchte um diesen Interpreter zu bauen.

Leider blieben im Bereich der Theorie ein paar Fragen offen, da der Bericht, der dieser Arbeit zugrunde liegt hier sehr kurzgefasst ist, dies trifft besonders auf das Liften von Operationen zu, was bedauerlich ist, da die Autoren dies selbst zu einem ihrer Kernpunkte erklärt haben.

5. Kapitel: Literatur

[1] Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters.
POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, January 1995.

[2] Philip Wadler. The essence of functional programming.
In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 1–14, January 1992.

[3] Eugenio Moggi. An abstract view of programming languages.
Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

6. Kapitel: Anlage

Bausteine im Programmcode

```
-- ====================================== --
-- The Interpreter Building Blocks ---
-- ====================================== --

class InterpC t where
      interp :: t -> InterpM Value

instance (InterpC t1, InterpC t2)=> InterpC (OR t1 t2) where
      interp (L t) = interp t
      interp (R t) = interp t

------------------------------------
-- The Arithmetic Building Block ---
------------------------------------
data TermA  = Num Int
            |Add Term Term

instance InterpC TermA where
      interp (Num x)  = unitInj x
      interp (Add x y)= interp (bindPrj x (\i ->
                           interp (bindPrj y (\j->
                             unitInj ((i+j)::Int)))))

unitInj     = unit .inj
bindPrj m k = bind m (\a-> case (prj a) of
                        Just x  -> k x
                        Nothing -> err "run-time type error")

---------------------------------------
--- The Continuation Building Block ---
---------------------------------------

data TermC      = CallCC

callcc :: ((a -> InterpM b) -> InterpM a) -> InterpM a

instance InterpC TermC where
      interp CallCC   = unitInj (\f -> bindPrj f (\f' ->
                        callcc (\k -> (f' (unitInj (\a -> bind a k )))))

-----------------------------------------
--- The Nondeterminism Building Block ---
-----------------------------------------

data TermN      = Amb [Term]

merge :: [InterpM a] -> InterpM a

instance InterpC TermN where
      interp (Amb t)  = merge (map interp t)
```

```
-----------------------------------------------------
--- The References and Assignment Building Block ---
-----------------------------------------------------

data TermR          = Ref       Term
                    | Deref     Term
                    | Assign    Term  Term

type Loc          = Int

allocLoc          :: InterpM Loc
lookupLoc :: Loc -> InterpM Value
updateLoc :: (Loc, InterpM Value) -> InterpM ()

instance InterpC TermR where
      interp (Ref x)          = bind (interp x) (\val ->
                                     bind allocLoc (\loc ->
                                      bind updateLoc (loc, unit val) (\_ ->
                                       unitInj loc)))

      interp (Deref x)        = bindPrj (interp x) (\loc -> lookupLoc loc)

      interp (Assign lhs rhs) = bindPrj (interp lhs) (\loc ->
                                     bind (interp rhs) (\val ->
                                       bind (updateLoc (loc, unit val) (\_ ->
                                        unit val)))

------------------------------------------
--- The Lazy Evaluation Building Block ---
------------------------------------------

data TermL        = LambdaL Name Term

instance InterpC TermL where
      interp (LambdaL s t)=
            bind rdEnv (\env -> unitInj (\arg ->
                          bind allocLoc (\loc ->
                            let thunk   = bind arg (\v ->
                                          bind  (updateLoc (loc, unit v))
                                                (\_ -> unit v)
                            in
                             bind updateLoc (loc, thunk)
                              (\_-> inEnv (extendEnv (s, lookupLoc loc) env)
                                                        (interp t))))))


------------------------------------------
--- The Program Tracing Building Block ---
------------------------------------------

data TermT        = Trace String Term

write :: STring -> InterpM ()

instance InterpC TermT where
      interp (Trace l t) = bind (write ("enter" ++ l))
                          (bind (interp t) (\v->
                              bind  (write ("leave "++l++" with:"++show v))
                                    (\_ -> unit v))))
```

```
----------------------------------
-- The Function Building Block ---
----------------------------------
data TermF        = Var       Name
                  | LambdaN   Name Term
                  | LambdaV   Name Term
                  | App       Term Term

type Name          = String

lookupEnv       :: Name -> Env -> Maybe(InterpM Value)
extendEnv       :: (Name, InterpM Value) -> Env -> Env
rdEnv           :: InterpM Env
inEnv           :: Env -> InterpM a -> InterpM a

instance InterpC TermF where
      interp (Var v)     = bind rdEnv (\env ->
                              case (lookupEnv v env) of
                                 Just val     -> val
                                 Nothing      ->err ("unbound variable:"++v)

      interp (LambdaN s t)    = bind rdEnv (\env ->
                                 unitInj (\arg ->
                                    inEnv (extendEnv  (s,arg) env)
                                                      (interp t))))

      interp (LambdaV s t)    = bind rdEnv (\env ->
                                       unitInj (\arg -> bind arg (\v->
                                         inEnv   (extendEnv (s,unit v) env)
                                                 (interp t))))

      interp (App e1 e2)      = interp (bindPrj e1 (\f ->
                                         bind rdEnv (\env ->
                                         f (inEnv env (interp e2)))))
```

Abbildung 18: Die Grundbausteine im Programmcode