Von der Entwurfsspezifikation zum Programm und zurück

Christoph Lüth

24. Juni 2002

The Big Picture

Anforderungs-spezifikation	$\rightsquigarrow$	Entwurfs-spezifikation	$\rightsquigarrow$	Implementation
• Lose		• ausführbar		• Haskell
• Viele Modelle		• monomorph		• C
• Eigenschaften		• Definitionen		

(Anforderungsspezifikation und Entwurfsspezifikation: CASL)

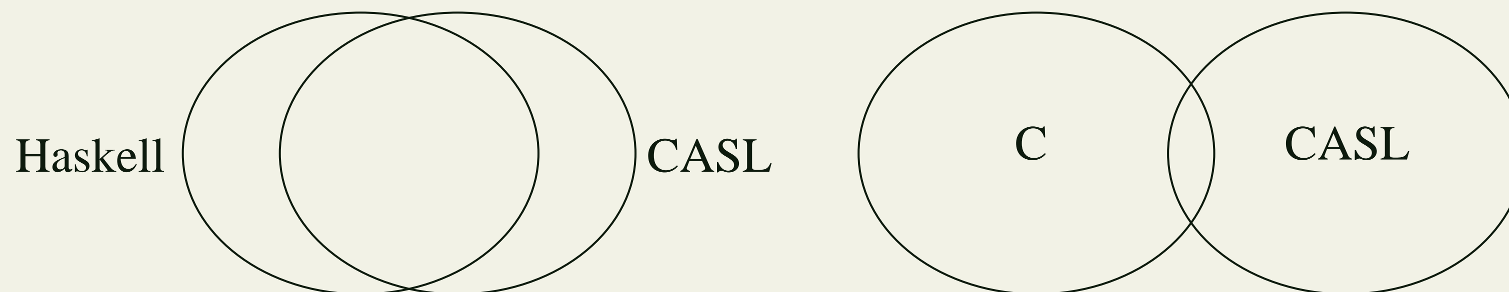
CASL vs. C und Haskell

Nicht in CASL, aber in . . .

- Haskell: Funktionen höherer Ordnung, Polymorphie
- C: Zustand, Variablen

In CASL, aber nicht in . . .

- C: alg. Datentypen
- C und Haskell: Subsorten, Partialität



CASL und Haskell: Das Problem

Haskell kennt

- Polymorphie (Typvariablen a, b) und
- Funktionen höherer Ordnung (Funktion f als Argument):

```
map :: (a-> b)-> [a]-> [b]
map f []     = []
map f (x:l) = f x: map f l
```

CASL kennt das nicht.

Die Lösung

Emulation durch parametrisierte Spezifikationen

- Polymorphe und Funktionen höherer Ordnung durch parametrisierte Spezifikation ersetzen
- Funktionale Argument und Typvariablen werden Parameter
- Funktionen entcurrieren
- Bei Benutzung instantiieren

Beispiel: Aus . . .

```
(.) :: (b-> c)-> (a-> b)-> a-> c
(f. g) x = f (g x)
```

wird. . .

```
spec ComposeArg=
  sorts Elem1,Elem2,Elem3
  ops f:Elem1->Elem2; g:Elem2->Elem3
end
spec Compose[ComposeArg] =  %def
  op comp[g,f]:Elem1->Elem3
  forall x:Elem1. comp[g,f](x)=g(f(x))
end
```

Listen und map

Listen in CASL

- Siehe Basic/StructuredDatatypes

```
spec List [sort Elem]=
  free type List[Elem] ::=
        []
        | __::__ (first:? Elem; rest:? List[Elem])
```

```
spec Map [sorts Elem1,Elem2; op f:Elem1->Elem2] =
  List[sort Elem1] with List[Elem1]
                   |-> SourceList[Elem1]
and
  List[sort Elem2] with List[Elem2]
                   |-> TargetList[Elem2]
then
  op map[f]:SourceList[Elem1]->TargetList[Elem2]
  forall x:Elem1; l:SourceList[Elem1]
  . map[f]([]) = []
  . map[f](x::l) = f(x)::map[f](l)
end
```

```
spec PredArg =
sort Elem
pred p:Elem
end
spec Filter [PredArg] =
  List[sort Elem]
then
  op filter[p]: List[Elem] -> List[Elem]
  forall x:Elem; l:List[Elem]
  . filter[p]([]) = []
  . filter[p](x::l) = (x::[] when p(x) else [])
                          ++ filter[p](l)              end
```

Fazit

- Für jede Funktion eigene Spezifikation (e.g. `map`, `filter`)

Fazit

- Für jede Funktion eigene Spezifikation (e.g. `map`, `filter`)
- Für jede Anwendung und jede Instantiierung eigens instantiierter Import.

Fazit

- Für jede Funktion eigene Spezifikation (e.g. `map`, `filter`)
- Für jede Anwendung und jede Instantiierung eigens instantiierter Import.
- Keinerlei Typinferenz.

Fazit

- Für jede Funktion eigene Spezifikation (e.g. `map`, `filter`)
- Für jede Anwendung und jede Instantiierung eigens instantiierter Import.
- Keinerlei Typinferenz.
- Möglich, aber sehr umständlich. - ächz! -

Imperative Konzepte in CASL

Konzepte imperativer Programme:

- Zustand und Variablen

Imperative Konzepte in CASL

Konzepte imperativer Programme:

- Zustand und Variablen
- Imperatives Ausführungsparadigma:
  Programm = Zustandstransformation

Imperative Konzepte in CASL

Konzepte imperativer Programme:

- Zustand und Variablen
- Imperatives Ausführungsparadigma:
  Programm = Zustandstransformation
- Datentypen

Imperative Konzepte in CASL

Konzepte imperativer Programme:

- Zustand und Variablen
- Imperatives Ausführungsparadigma:
  Programm = Zustandstransformation
- Datentypen
- Speicherverwaltung

Imperative Konzepte in CASL

Konzepte imperativer Programme:

- Zustand und Variablen
- Imperatives Ausführungsparadigma:
  Programm = Zustandstransformation
- Datentypen
- Speicherverwaltung

Wie das in CASL?

Zustände in deklarativen Sprachen

Problem: zustandsabhängige Berechnungen

- z.B. `getLine : String`
- Bekanntes Problem in funktionalen Sprachen
- Bekannte Lösung: State Transformer
- Zustand wird (explizit oder implizit) “durch die Gegend gereicht“

Zustandstransformationen

Eine Zustandstransformation ist parametrisiert über

- den Zustand `s`
- das Ergebnis `a`

```
data ST s a = ST (s-> (s, a))
```

- `f :: ST s a` ist eine vom Zustand `s` abhängige Berechung mit dem Ergebnistyp `a`.

- ST ist ein Monade:

```
class Monad m a where
   return :: a-> m a
   (>>=)  :: m a-> (a-> m b)-> m b
```

  - Monadenstruktur erzwingt Zustandstransformationsstruktur
  - Zustand wird immer nur weitergegeben

```
instance Monad (ST s) where
     return a     = ST (\s-> (s, a))
     (ST f) >>= g = ST (\s-> let (s1, a)= f s
                                 ST g1  = g a
                             in  g1 s1)
```

Beispiel

Speicher: Liste von Strings, Referenzen: Index in Liste

```
type Mem = ST [String]    -- Zustand
type Ref = Int

newRef :: Mem Ref
newRef = ST (\s-> (s++[""], length s))
readRef :: Ref-> Mem String
readRef r = ST (\s-> (s, s !! r))
writeRef :: Ref-> String-> Mem ()
writeRef r v = ST (\s-> (take r s ++
                           [v]++ drop (r+1) s, ()))
run :: Mem a-> a
run (ST f)= snd (f [])
```

Zusammenfassung ST

- Zustand wird
  - als Parameter übergeben,
  - mit dem Ergebnis zurückgeliefert.
- Zustand ist abstrakt:
  - wird nur mit gegebenen Funktionen manipuliert;
  - zustandsabhängige Berechungen werden nur mit (>>=) komponiert.
- Dadurch: immer genau ein Zustand

Zustände mit CASL

Signatur einer zustandsbehafteten Funktion

$$f : s \times a \longrightarrow s \times b$$

- Eingabewerte vom Typ a
- Rückgabewerte vom Typ b
- Benutzt Zustand vom Typ s

Benutzung:

- Zustand nie verdoppeln oder vergessen
- Immer nur mit vorgegeben Funktionen manipulieren, oder weiterreichen.

- Problem: In CASL keine Tupel als Ergebnistyp. . .
- Lösung: `Pair` aus `Basic/StructuredDatatypes`
- . . . oder explizit:

```
spec Stateful[sort S][sort D] =
   free type St[S,D] ::= st(state: S; data: D)
end
```

Datentypen in CASL und C

	CASL	C
	`Int`	`int` und Freunde
	`Char`	`char`
	`Boolean`	`boolean` oder `int`
	`String`	`char *`
	—	*t* `*`
	`free type ...`	`struct T` ,`union T`
	`Array`	*t* `a[c]`
Notation:	S	$[\![S]\!]_C$
	$[\![\mathtt{t}]\!]_{CASL}$	`t`

Felder in CASL

- Aus Basic/StructuredDatatypes

```
 spec Array
  [ops min, max: Int axiom min <= max %(Cond_nonEmptyIndex)%]
  [sort Elem] given Int
= sort Index = { i : Int . min <= i /\ i <= max }
then {     FiniteMap [sort Index fit sort S|-> Index]
                     [sort Elem fit sort T |-> Elem]
           with
           sort FiniteMap[Index,Elem] |-> Array[Elem],
           ops  [] |-> init
       then
```

```
  ops    __!__:=__ : Array[Elem] * Index *Elem -> Array[Elem];
         __!__ :     Array[Elem] * Index ->? Elem
  forall A: Array[Elem]; i: Index; e:Elem
  . A!i:=e = A[e/i]                     %(assignment_def)%
  . A!i = eval(i,A)                     %(evaluate_def)%
  } reveal
      sort Array[Elem],
      ops  init, __!__, __!__:=__
then %implies
 forall A: Array[Elem]; i,j: Index; e,f: Elem
 . not def init!i
 . def (A!i:=e)!i
 . (A!i:=e)!j = e if i=j
 . (A!i:=e)!j = A!j if not (i=j)
end
```

Übersetzung von algebraischen Datentypen

- Enumerationen (nur nullstellige Konstruktoren):

```
free type T ::= C1 | C2 | ... | Cn
```

$$[\![T]\!]_C = \texttt{t}$$

```
typedef enum {C1, C2, ..., Cn} t;
```

Nur ein Konstruktor:

- `free type T ::= C(sel1: T1, ..., seln: Tn)`

  $$[\![T]\!]_C = \texttt{t *}$$

  ```
  typedef struct { [[T_1]] sel1; ...;
                   [[T_n]] seln; } t;
  ```

- Beispiel:

  ```
  free type Book ::= Book(author: String,
                          title : String)
  ```

  ```
  typedef struct { char *author;
                   char *title; } book;
  ```

- Der allgemeine Fall:

```
free type T ::=
         C1(sel11: T11, ..., sel1n_1: T1n_1)
       | C2(sel21: T21, ..., sel2n_1: T2n_2)
       | ...
       | Cm(selm1: Tm1, ..., selmn_m: Tmn_m)
```

  - Ein `struct` pro Konstruktor
  - Eine `union` für alle Varianten
  - Eine Aufzählung aller Varianten
  - Bei Rekursion Vorwärtsreferenzen beachten

$$[\![T]\!]_C = \texttt{t *}$$

```
typedef enum  {C1, ... Cm} t_var;
typedef struct { [[T11 ]] sel11; ...;
                 [[T1n_1]] sel1n_1; } t_var1;
...
typedef struct { [[Tm1 ]] selm1; ...;
                 [[Tmn_m]] selmn_m; } t_varm;
typedef struct { t_var t_discr;
                 union { t_var1 t_var1; ...;
                         t_varm t_varm;
                       } t_data; } t;
```

- Parameter `sort P` (der Spezifikation):

$$[\![P]\!]_C = \texttt{void *}$$

  - Kann mit jedem algebraischen Typ instantiiert werden;
  - . . . aber nicht mit Basisdatentypen.

- Beispiel:

```
free type LTree[Elem] ::=
      Empty | VeryEmpty
    | Cons(hd: Elem,
           tl: LTree[Elem])
    | Node(lt: LTree[Elem],
           data: Elem,
           rt: LTree[Elem])
```

Ergibt:

$[\![LTree]\!]_C =$ `ltree` mit

```
typedef enum { empty, veryEmpty,
               cons, node } ltree_var;
typedef struct { void *hd;
                 struct ltree *tl; } ltree_cons;
typedef struct { struct ltree *lt;
                 void *data;
                 struct ltree *rt; } ltree_node;
typedef struct { ltree_var ltree_discr;
                 union { ltree_cons ltree_cons;
                         ltree_node ltree_node;
                       } ltree_data; } ltree;
```

- Vereinfachungen:
  - Wenn genau eine nullstellige Variante vorhanden ist, diese durch den ausgezeichneten Wert NULL modellieren.
- Beispiel: Listen (Basic/StructuredDatatypes)

```
free type List[Elem]::=
     Empty | Cons (first :? Elem, rest :? Elem)
```

  - In erster Näherung:

```
typedef enum { Empty, Cons } list_var;
typedef struct { void *first;
                 struct list *rest;} list_cons;
typedef struct { list_var list_discr;
                 union { list_cons list_cons;
                       } list_data; } list;
```

- Einfacher: `Empty` fällt weg

```
typedef struct { void* first;
                 struct list*rest; } list;
```

- Diskriminator: `l == NULL`
- Selektoren durch Feldselektion in C:

`l-> first`, `l-> rest`

- Konstruktor:
  - Speicherbereich allozieren (`malloc`)
  - Achtung, Speicherfreigabe (`free`) muß von Hand eingefügt werden (wenn die Liste nicht mehr benutzt wird).

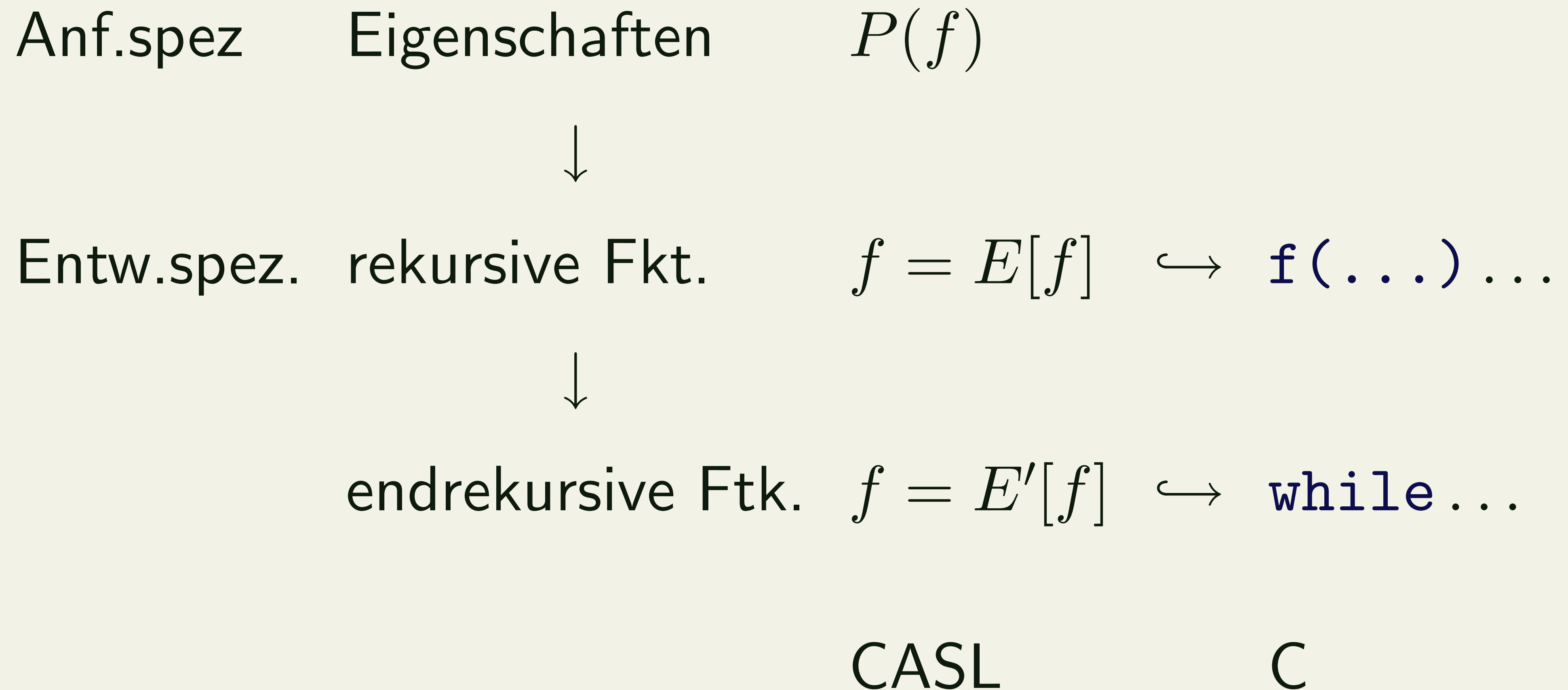
```
list *cons(void *hd, list *tl)
{
  list *l;
  if ((l= malloc(sizeof *l))== NULL) abort();
  l-> first= hd;
  l-> rest= tl;
  return l;
}
```

Von der CASL-Spezifikation zum C-Programm

Zwei Herangehensweisen:

1. Schrittweise Verfeinerung
   - Anforderungsspezifikation schrittweise in Programm überführen
   - Problem: CASL — Rekursion, C — Iteration
   - Lösung: Endrekursion entspricht Iteration
2. Invent & Verify
   - Programm erfinden, dann verifizieren (Hoare-Kalkül)

Implementation durch Verfeinerung



Endrekursive Funktionen

Eine Funktion $f : S \longrightarrow T$ der folgende Form

```
f(x) = f(K(x)) if B(x)
f(x) = H(x)    if not B(x)
```

ist endrekursiv und entspricht der Iteration:

```
[[T]] f([[S]] x)
{
  [[S]] a= x;
  while B(a) { a= K(a); }
  return H(a);
}
```

Kriterien für Endrekursivität

- Genau ein rekursiver Aufruf
- Rekursiver Aufruf außen (unter Fallunterscheidung)
- Eine Fallunterscheidung

Beispiel

- Der Modulus

```
mod : Nat * Nat-> Nat
mod (x, y) = mod (x- y, y) if x >= y
mod (x, y) = x              if x < y
```

- Der größte gemeinsame Teiler:

```
gcd : Nat* Nat-> Nat
gcd(n, m) = gcd (m, mod(n, m)) if not (m= 0)
gcd(n, m) = n                  if m= 0
```

mod iterativ

Nach obiger Transformation:

```
int mod(int x, int y)
{
 int a= x;
 int b= y;

 while (a>= b) { a= a- b; }
 return a;
}
```

Optimierung:

mod iterativ

```
int mod(int x, int y)
{
 while (x>= y) { x= x- y; }
 return x;
}
```

gcd iterativ:

Nach obiger Transformation:

```
int gcd(int x, int y)
{
 int a= x;
 int b= y;

 while (b != 0) { (a, b)= (b, mod(a, b)); }
 return a;
}
```

Noch kein C: Tupelzuweisung!

gcd iterativ:

Tupelzuweisung auflösen.

Dazu Hilfsvariable c.

```
int gcd(int x, int y)
{
 int z;

 while (y != 0) { z= x; x= y; y= mod(z, y); }
 return x;
}
```

Von der Rekursion zur Endrekursion

- Gegenbeispiel:

```
fact : Int-> Int
fact (x) = x* (fact (x-1)) if x > 0
fact (x) = 1               if x <= 0
```

Von der Rekursion zur Endrekursion

- Gegenbeispiel:

```
fact : Int-> Int
fact (x) = x* (fact (x-1)) if x > 0
fact (x) = 1               if x <= 0
```

- Wie von Rekursion zur Endrekursion?
- Endergebniss in weiterem Parameter akkumulieren:

- Dazu Hilfsfunktion `fact'`

```
fact : Int-> Int
fact n = fact' (n, 1)
fact' : Int* Int-> Int
fact'(x, a) = fact'(x-1, x*a)  if x> 0
fact'(x, a) = a                if x<= 0
```

- Technik läßt sich verallgemeinern.

Überführung in Endrekursion

- Voraussetzung: lineare Rekursion
  - d.h. genau ein rekursiver Aufruf
- Gegeben

$$
\begin{array}{lcll}
f & : & S \to T & \\
f(x) & = & \phi(f(K(x)), E(x)) & \mathit{if}\ B(x) \\
f(x) & = & H(x) & \mathit{if}\ \neg\, B(x)
\end{array}
$$

mit $K : S \to S, \phi : T \times R \to T, E : S \to R, H : S \to T$

- Überführung in endrekursive Form durch Hilfsfunktion

- Aufrufschema: nach n Inkarnationen

$$f(x) = \phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_1), a_0)$$
$$a_i = E(K^i(x)), b = H(K^n(x))$$

- Aufrufschema: nach n Inkarnationen

$$f(x) = \phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_1), a_0)$$
$$a_i = E(K^i(x)), b = H(K^n(x))$$

- Wenn es jetzt ϕ assoziative ist:

$$\phi(\phi(r, s), t) = \phi(r, \phi(s, t))$$

dann ist

$$\begin{aligned} f(x) &= \phi(\phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_2), a_1), a_0) \\ &= \phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_2), \phi(a_1, a_0)) \\ &= \phi(\cdots(\phi(b, a_{n-1}), \ldots), \phi(a_2, \phi(a_1, a_0))) \\ &= \phi(b, \phi(a_{n-1}, \ldots \phi(a_2, \phi(a_2, \phi(a_1, a_0))))) \end{aligned}$$

- Das ist endrekursiv!

Transformationsregel

- Für $f : S \longrightarrow T$ mit
$$\begin{array}{lcll} f(x) & = & \phi(f(K(x)), E(x)) & \mathit{if}\ B(x) \\ f(x) & = & H(x) & \mathit{if}\ \neg\ B(x) \end{array}$$
mit $K : S \longrightarrow S, \phi : T \times T \longrightarrow T, E : S \longrightarrow T, H : S \longrightarrow T$,
- Wenn ϕ assoziativ und e neutrales Element:
$$\phi(\phi(r, s), t) = \phi(r, \phi(s, t)) \qquad \phi(r, e) = r$$
- Dann ist die äquivalente endrekursive Formulierung:
$$\begin{array}{lcll} f(x) & = & g(x, e) & \\ g(x, y) & = & g(K(x), \phi(E(x), y)) & \mathit{if}\ B(x) \\ g(x, y) & = & \phi(H(x), y) & \mathit{if}\ \neg\ B(x) \end{array}$$

Allgemeine Transformationsregel

- Für $f : S \to T$, $\phi : T \times R \to T$

$$\begin{array}{rcll} f(x) & = & \phi(f(K(x)), E(x)) & \mathit{if}\ B(x) \\ f(x) & = & H(x) & \mathit{if}\ \neg\ B(x) \end{array}$$

- Gegegebn $\psi : R \times R \to R$ so dass

$$\phi(\phi(r, s), t) = \phi(r, \psi(s, t))$$

- Dann ist die äquivalente endrekursive Formulierung:

$$\begin{array}{rcll} f(x) & = & g(K(x), E(x)) & \mathit{if}\ B(x) \\ f(x) & = & H(x) & \mathit{if}\ \neg\ B(x) \\ g(x, y) & = & g(K(x), \psi(E(x), y)) & \mathit{if}\ B(x) \\ g(x, y) & = & \phi(H(x), y) & \mathit{if}\ \neg\ B(x) \end{array}$$

Sonderfall: Prädikate

- Für `pred p: S` mit
$$\begin{array}{lcll} p(x) & \Leftrightarrow & \phi(p(K(x)), E(x)) & \mathit{if}\ B(x) \\ p(x) & \Leftrightarrow & H(x) & \mathit{if}\ \neg\ B(x) \end{array}$$
mit $K : S \to S$, $\texttt{pred}\ E : S$, $\texttt{pred}\ p : S$, $\texttt{pred}\ H : S$
- Sei ϕ binär, assoziativ und e neutrales Element.
- Dann ist die äquivalente endrekursive Formulierung:
$$\begin{array}{lcll} p(x) & \Leftrightarrow & q(x, e) & \\ q(x, y) & \Leftrightarrow & q(K(x), \phi(E(x), y)) & \mathit{if}\ B(X) \\ q(x, y) & \Leftrightarrow & \phi(H(x), y) & \mathit{if}\ \neg\ B(x) \end{array}$$

Implementation rekursiv definierter Funktionen

1. Pattern matching entfernen, Lexikalik an C anpassen
   - z.B. Konstruktor von `::` nach `cons` umbenennen
2. Umformen in linearrekursive Form, wo möglich;
3. Linearrekursion in Endrekursion, wo möglich;
4. In C auskodieren.

Beispiel: Enthaltensein

- Originaldefinition:

```
not (x eps [])
x eps (x::L)
(x eps (y::L) <=> x eps L  if not (x= y)
```

- Ohne Pattern-Matching:

```
not (x eps xs)      if xs = empty
x eps xs            if not (xs= empty)
                       /\ first xs= x
x eps xs <=> x eps (rest xs)
      if not (xs= emty) /\ not (first xs= x)
```

- Linearekursiv (ersten beiden Fälle zusammenfassen)

```
(xs = empty => not x eps xs) \/
(not (xs= empty) /\
 (first xs= x) => x eps xs)
                  if xs= empty \/ first xs= x

x eps xs <=> x eps (rest xs)
        if not (xs= empty) /\ not (first xs= x)
```

Benutzt: $A \vee (\neg A \wedge B) \Leftrightarrow (A \vee \neg A) \wedge (A \vee B) \Leftrightarrow (A \vee B)$

Ist schon endrekursiv!

- Implementation in C:

```
bool eps(void *a, list *l)
{
 void *x= a; list *m= l;
 while (!(m == NULL) && (m-> first!= x)) {
     m= m-> rest;
     }
 if (m == NULL) return FALSE;
 if (!(m == NULL) && (*m-> first)== *x)
    return TRUE;
}
```

- Optimierung:
  - Lokale Variablen unnötig;
  - Nach Schleifendurchlauf gilt Abbruchbedingung.
  - Immer noch ein Problem: Gleichheit der Elemente (hier: gleiche Referenz — nicht optimal).

```
bool eps(void *a, list *l)
{
 while (!(l == NULL) && (l-> first!= a)) {
      l= l-> rest;
      }
 return l != NULL;
}
```

Beispiel: Länge einer Liste

- Rekursive Definition:

```
length : List[Elem]-> Nat
length xs = 0                   if isEmpty(xs)
length xs = 1+ length (rest xs)
                                if not(isEmpty(xs))
```

- Endrekursiv:

```
length xs  = len xs 0
len(xs, n) = n                  if isEmpty(xs)
len(xs, n) = len(rest xs, n+1)
                                if not (isEmpty(xs))
```

Beispiel: Listenreversion

- Rekursive Definition:

```
rev xs = xs                     if isEmpty(xs)
rev xs = rev (rest xs) ++ ((first xs)::[])
                                if not (isEmpty(xs))
```

Beispiel: Listenreversion

- Rekursive Definition:

```
rev xs = xs                          if isEmpty(xs)
rev xs = rev (rest xs) ++ ((first xs)::[])
                                     if not (isEmpty(xs))
```

- Endrekursiv:

```
rev xs          = rev'(xs, [])
rev'(xs, ys) = ys                    if isEmpty(xs)
rev'(xs, ys) = rev'(rest xs, (first xs)::ys)
                                     if not (isEmpty(xs))
```

- In C codiert (optimierte Version):

```
list *rev(list *xs)
{
  list *ys;

  while (xs != NULL) {
    ys= cons(xs-> first, ys);
    xs= xs-> rest;
    }
  return ys;
}
```

Zusammenfassung

Von der CASL-Spezifikation zum C-Programm

- Zustand als expliziten Parameter (wenn nötig)
- Datentypen systematisch übersetzen
- Implementation durch schrittweise Verfeinerung:
  - Rekursion in Endrekursion überführen, wo möglich;
  - Endrekursion durch Iteration implementieren.
- **oder** Invent & Verify:
  - Funktion spezifiziert mit Vor- und Nachbedingung
  - Frei implementieren
  - Verifikation mit dem Hoare-Kalkül

Bewertung

Von der CASL-Spezifikation zum C-Programm

- Vorteile
  - systematischer Übergang
  - schematisch (potentiell automatisierbar)

Bewertung

Von der CASL-Spezifikation zum C-Programm

- Vorteile
  - systematischer Übergang
  - schematisch (potentiell automatisierbar)

- Nachteile
  - Aufwändig
  - Fehleranfällig: Zustandstransformationen, Endrekursion
  - Mangelnde Features in CASL:
    Polymorphie, Funktionen höherer Ordnung
  - Mangelnde Werkzeugunterstützung