Vorlesung vom 30.01.06: Techniken der formalen Programmentwicklung

Wo sind wir?

- Teil I: Grundlagen formaler Logik
- Teil II: Modellierung
- Teil III: Spezifikationsformalismen
  - ◦ Axiomatische Spezifikation
  - ◦ Spezifikation mit Z: Grundlagen & Einführung
  - ◦ Z: Schemakalkül
  - ◦ *Formale Programmentwicklung: Endrekursion & Datenverfeinerung*

Formale Programmentwicklung

- Von der *Spezifikation* zum *Code*:

$$SP_1 \rightsquigarrow SP_2 \rightsquigarrow \ldots \rightsquigarrow SP_n$$

- Einzelne *Schritte*:
  - ◦ *Theoriemorphismen*: *Modellverfeinerung*
  - ◦ *Funktionen* implementieren:
    Rekursion ⟶ *Endrekursion* ⟶ Iteration
  - ◦ *Datenverfeinerung*: *Datentypen* implementieren

Endrekursion

- *Def:* Eine Funktion $f : S \Rightarrow T$

  $$f\ x \equiv \mathit{if}\ B\,x\ \mathit{then}\ f(K\,x)\ \mathit{else}\ H\,x$$

  ist *endrekursiv* und entspricht der Iteration

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

- ⟦T⟧: Übersetzung des Typen T

Kriterien für Endrekursivität

- Genau *ein* rekursiver Aufruf (*lineare Rekursion*)
- Genau *eine* Fallunterscheidung
- Rekursiver Aufruf *außen* (unter Fallunterscheidung)

Beispiele

- Berechung des *Modulus*:

```
consts mod :: "nat* nat=> nat"
recdef mod  "measure fst"
  "mod (x,y) = (if x >= y & y > 0 then mod (x-y ,y)
                                  else x)"
```

- Berechnung des *ggT*:

```
consts gcd :: "nat * nat => nat"
recdef gcd  "measure (%(x, y). x + y)"
  "gcd (x, 0) = x"
  "gcd (x, y) = gcd (y, mod(x, y))"
```


Von der Rekursion zur Endrekursion

- *Gegenbeispiel* — nicht endrekursiv:

```
consts fact :: "nat => nat"
recdef fact  less_than
  "fact x = (if x = 0 then 1
                      else x * fact (x - 1))"
```

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

Von der Rekursion zur Endrekursion

- Dazu *Hilfsfunktion*

```
constdefs fact :: "nat => nat"
  "fact x == fact'(x, 1)"

consts fact' :: "(nat, nat) => nat"
recdef fact'  "measure fst"
  "fact' x a = (if x = 0 then a
                         else fact' (x - 1, x*a)"
```

- Technik läßt sich *verallgemeinern*

Überführung in Endrekursion

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

$$\begin{array}{lcl} f & :: & S \Rightarrow T \\ f(x) & = & \mathit{if}\, B(x)\ \mathit{then}\, \phi(f(K(x)), E(x))\ \mathit{else}\ H(x) \end{array}$$

  mit $K : S \Rightarrow S, \phi : T \times R \Rightarrow T, E : S \Rightarrow R, H : S \Rightarrow T$
- Überführung in *endrekursive* Form durch *Hilfsfunktion*

Einführung von Endrekursion

- *Aufrufschema*: nach n Aufrufen

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

- *Voraussetzung*: ϕ *assoziativ*, $\phi(\phi(r,s),t) = \phi(r, \phi(s,t))$
- Dann gilt

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

- Das ist *endrekursiv!*

Transformationsregel

- Für $f : S \Rightarrow T$ mit

$$f(x) = \mathit{if}\, B(x)\ \mathit{then}\, \phi(f(K(x)), E(x))\ \mathit{else}\ H(x)$$

  mit $K : S \Rightarrow S, \phi : T \times T \Rightarrow T, E : S \Rightarrow T, H : S \Rightarrow T,$
- 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:

$$\begin{array}{lcl} f(x) & = & g(x, e) \\ g(x, y) & = & \mathit{if}\, B(x)\ \mathit{then}\, g(K(x), \phi(E(x), y))\ \mathit{else}\ \phi(H(x), y) \end{array}$$

Beispiel: Länge einer Liste

- Rekursive Definition:

```
consts length :: "'a list=> nat"
recdef length measure size
 length []     = 0
 length (x#xs) = 1+ length xs
```

- Endrekursiv:

```
constdefs length ::  "'a list=> nat"
 "length xs == length'(xs, 0)"
consts length' :: "'a list* nat=> nat"
recdef length' "measure fst"
  length' ([], n)   = n
  length' (x#xs, n) = length'(xs, n+1)"
```


Beispiel: Listenreversion

- Rekursive Definition:

```
consts rev :: "'a list=> 'a list"
recdef rev "measure size"
  rev []     = []
  rev (x#xs) = (rev xs)@[x]
```

- Endrekursiv:

```
constdefs rev ::  "'a list=> 'a list"
 "rev xs == rev'(xs, [])"
consts rev' :: "'a list=> 'a list=> 'a list"
recdef rev' "measure size"
  rev'([], ys)   = ys
  rev'(x#xs, ys) = rev'(xs, x#ys)
```


Datenverfeinerung

- Problem: *Abstrakten* Datentyp (`List`, `Set`) implementieren
- Abbildung der *Operationen*
- *Axiome* gelten? *Welche?*
- Verfeinerung auf Datenebene (*Reification*)

Datentypen (Z)

- *Def (*Datentyp*)*: $\mathcal{X} = \langle X, xi, \{xop_\omega\}_{\omega \in \Sigma}\rangle$ mit
  - *Globaler* Zustandsraum G
  - *Lokaler* Zustandsraum X
  - *Initialisierung* $xi : G \leftrightarrow X$ (*total*)
  - *Operationen* $xop_\omega : X \times G \leftrightarrow X \times G$ (*partiell*)
- $\mathcal{C}$ *verfeinert* $\mathcal{A}$, wenn für *alle Programme* $P(\mathcal{X})$ gilt $P(\mathcal{C}) \subseteq P(\mathcal{A})$ (*globaler Zustandsraum* gleich)
- *Idee:* $\mathcal{C}$ ist *determinierter* und *totaler* als $\mathcal{A}$
- *Problem:* Wie beweisen?

Simulationen

*Def (*Simulation*)*: Sei $\mathcal{A} = \langle A, ai, \{aop_i\}_{i\in\Sigma}\rangle$ and $\mathcal{C} = \langle C, ci, \{cop_i\}_{i\in\Sigma}\rangle$. $\mathcal{C}$ *simuliert* $\mathcal{A}$ wenn es $R : A \leftrightarrow C$ gibt so dass

$$\begin{array}{ll} \textit{(Init)} & \forall\, c' \in C, g \in G \cdot (g, c') \in ci \\ & \Longrightarrow \exists\, a' \in A \cdot (g, a') \in ai \wedge (a', c') \in R \\ \textit{(Appl)} & \forall\, i \in \Omega\, \forall\, a \in A, c \in C, g \in G \cdot \\ & \quad (a, g) \in \textit{dom}\, aop_i \wedge (a, c) \in R \\ & \quad \Longrightarrow (c, g) \in \textit{dom}\, cop_i \\ \textit{(Corr)} & \forall\, i \in \Omega\, \forall\, a \in A, c, c' \in C, g, g' \in G \cdot \\ & \quad (a, g) \in \textit{dom}\, aop_i \wedge (a, c) \in R \wedge ((c, g), (c', g')) \in cop_i \\ & \quad \Longrightarrow \exists\, a' \in A \cdot ((a, g), (a', g')) \in aop_i \wedge (a', c') \in R \end{array}$$

Satz: $\mathcal{C}$ simuliert $\mathcal{A}$, dann $P(\mathcal{C}) \subseteq P(\mathcal{A})$

Beispiel

- Stacks mit $\Sigma = \{push, pop, top\}$

  `types globalstate= "int* int"`

- *Abstrakt:* Stack als *Liste*
- *Konkret:* Stack als *Feld mit Zeiger*
- Problem: keine *direkte* Verfeinerung
  - Axiom a2 gilt *nicht*
  - Aber: Gleichheit auf *lokalem Zustandsraum* nicht *sichtbar*
  - Trotzdem *sinnvolle Verfeinerung* $\rightsquigarrow$ Datenverfeinerung

Stacks durch Listen

```
types lstate = int list
constdefs
 lempty :: "(globalstate* lstate) set"
"lempty == {(g, l). l= []}"
 lpush :: "((globalstate* lstate)* (globalstate* lstate)) set'
"lpush == {(((i, out), l), ((i', out'), l')).
           (i', out')= (i, out) & (l'= i # l)}"
 lpop  :: "((globalstate* lstate)* (globalstate* lstate)) set'
"lpop  == {((g, l), (g', l')).
   l ~= [] & g'= g & l'= tl l}"
 ltop  :: "((globalstate* lstate)* (globalstate* lstate)) set'
"ltop  == {(((i, out), l), ((i', out'), l')).
           l ~= [] & i'= i & out'= hd l & l= l'}"
```


Stacks durch Felder

```
types astate = "nat* (nat ~=> int)"
constdefs
 aempty :: "(globalstate* astate) set"
"aempty == {(g, l). l= (0, empty)}"
 apush :: "((globalstate* astate)* (globalstate* astate)) set'
"apush == {(((i, out), (p, a)), ((i', out'), (p', a'))).
           (i', out')= (i, out) & p'= p+ 1 & a'= a(p |-> i)}"
 apop  :: "((globalstate* astate)* (globalstate* astate)) set'
"apop  == {((g, (p, a)), (g', (p', a'))).
           g'= g & p'= p- 1 & a'= a}"
 atop  :: "((globalstate* astate)* (globalstate* astate)) set'
"atop  == {(((i, out), (p, a)), ((i', out'), (p', a'))).
           i'= i & out'= the (a (p - 1)) & p'= p & a'= a}"
```


Verfeinerung

- Die *Verfeinerungsrelation*:

```
constdefs
  r :: "(lstate* astate) set"
 "r == {(l, (p, a)). p= length l &
                    (! i. 0<= i & i< p -->
                          l ! i = the (a (p-i- 1)))}"
```

- Damit die drei Bedingungen zeigen.

Zusammenfassung

- Schrittweise *Verfeinerung*: von Spezifikation zum Programm
- Rekursion $\longrightarrow$ lineare Rekursion $\longrightarrow$ Endrekursion
- *Transformationsregeln*: Schema für *Entwurfsschritte*
- *Eigenschaftserhaltende Wechsel* von *Datentypen*:
  - *Datenverfeinerung*
  - Verhaltensgleichheit (*observational equivalence*)