Deskriptive Programmierung

Analyse und Transformation funktionaler Programme

Beispiel: algebraische Eigenschaften von Parserkombinatoren

- Welche der folgenden Äquivalenzen gelten?

```
p ||| failure                    =  p
p ||| q                          =  q ||| p
p ++> yield                      =  p
yield x +++ q                    =  q
p +++ q                          =  q
p ||| yield x                    =  p
yield x ||| p                    =  yield x
failure ++> f                    =  failure
p +++ failure                    =  failure
(p ++> f ) ||| (p ++> g)         =  p ++> (\x → f x ||| g x)
(p ++> f ) ||| (q ++> f)         =  (p ||| q) ++> f
(p +++ q) +++ r                  =  p +++ (q +++ r)
```

- Und wie beweist man solche Äquivalenzen?

Beispiel: Übersetzung und Optimierung von list comprehensions

- Auf die Frage:
  Kann man list comprehensions allgemein durch Aufrufe von filter, map, concat ersetzen?

- … könnte man ja mal schauen, was der Compiler tut:

[e | x ← xs]	↦	map (\x → e) xs
[e | b]	↦	if b then [e] else [ ]
[e | b, Q]	↦	if b then [e | Q] else [ ]
[e | x ← xs, Q]	↦	concat (map (\x → [e | Q]) xs)

- Kein filter? Dafür „umständliche“ Ausdrücke, etwa:

```
concat (map (\x → concat (map (\y → if x `mod` y > 0 then [ (x, y) ] else [ ]) [1 .. x])) [1 .. 100])
```

für:

```
[ (x, y) | x ← [1 .. 100], y ← [1 .. x], x `mod` y > 0 ]
```

- Kann man da was mit Post-Processing tun?

Beispiel: Übersetzung und Optimierung von list comprehensions

- Ja, man kann. Allgemein gilt:

$$\text{concat (map (\textbackslash y} \rightarrow \text{if p y then [f y] else [ ]) ys)}$$

ist äquivalent zu:

$$\text{map f (filter p ys)}$$

- Folglich ist:

$$\text{concat (map (\textbackslash x} \rightarrow \text{concat (map (\textbackslash y} \rightarrow \text{if x `mod` y > 0 then [(x, y)] else [ ]) [1 .. x])) [1 .. 100])}$$

äquivalent zu:

$$\text{concat (map (\textbackslash x} \rightarrow \text{map (\textbackslash y} \rightarrow \text{(x, y)) (filter (\textbackslash y} \rightarrow \text{x `mod` y > 0) [1 .. x])) [1 .. 100])}$$

- Aber, es wäre ratsam, die obige allgemeine Aussage zunächst zu beweisen!

Beispiel: Übersetzung und Optimierung von list comprehensions

- Immer noch im Kontext der Übersetzung von list comprehensions:
  - man erzeugt generell oft Aufrufe der Form:

```
concat (map f xs)
```

  - Der Compiler kann benutzen, dass dies äquivalent ist zu:

```
concatMap f xs
```

  wobei:

```
concatMap :: (a → [b]) → [a] → [b]
concatMap f  =  foldr ((++) . f) [ ]
```

- Aber, ist das wirklich korrekt (und effizienter)?
- Und, hätte man (bzw. ein Compiler) die Definition von concatMap automatisch finden können?

Einige weitere interessante Fragestellungen (1)

- Es gibt mindestens zwei interessante Definitionen der Funktion reverse:

$$\begin{aligned}
&\text{reverse} :: [a] \rightarrow [a]\\
&\text{reverse } [\,] \quad = [\,]\\
&\text{reverse } (x : xs) = \text{reverse } xs \mathbin{+\!\!+} [x]
\end{aligned}$$

vs.

$$\begin{aligned}
&\text{reverse} :: [a] \rightarrow [a]\\
&\text{reverse } xs = \text{reverse}' \; xs \; [\,]\\
&\text{reverse}' \; [\,] \quad ys = ys\\
&\text{reverse}' \; (x : xs) \; ys = \text{reverse}' \; xs \; (x : ys)
\end{aligned}$$

- Sind diese beiden Versionen in geeignetem Sinne wirklich äquivalent?
- Wie kommt man von einer zur anderen?
- Wird irgendetwas einfacher, wenn wir foldr verwenden?

Einige weitere interessante Fragestellungen (2)

- Wir hatten für sumsquare eine schön modulare Version:

```
sumsquare :: Int → Int
sumsquare n  =  foldr (+) 0 [ i * i  |  i ← [0 .. n] ]
```

und eine (vermutet) effizientere:

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i – 1)
```

- Sind diese beiden Versionen äquivalent?
- …

Einige weitere interessante Fragestellungen (3)

- Gegeben:

```
data Nat = Zero | Succ Nat
```

und:

```
add :: Nat → Nat → Nat
add Zero      m = m
add (Succ n)  m = Succ (add n m)
```

- Kann man die üblichen algebraischen Zusammenhänge (Kommutativität, Assoziativität, …) beweisen?
- Ist Zero ein neutrales Element für die Addition add?
- Offenbar gilt für alle n :: Nat, add Zero n = n.
- Aber auch für alle n :: Nat, add n Zero = n ?
- Beweisidee: Induktion.

Gleichungsbasiertes Schließen und Induktion für natürliche Zahlen

- Gleichungsbasiertes Schließen: Anwendung von Funktionsgleichungen nicht nur zur Auswertung, sondern auch zum Nachweis von Programmeigenschaften.

- Das mathematische Prinzip der Induktion:
  - wenn ein beliebiges Prädikat P für die Null gilt: P(0),
  - und wenn für jede natürliche Zahl aus Gültigkeit von P auch dessen Gültigkeit für den Nachfolger folgt: $P(n) \Rightarrow P(n + 1)$,
  - dann gilt P für alle natürlichen Zahlen.

- Naheliegende Idee: Übertragung auf

```
data Nat = Zero | Succ Nat
```

  - wenn ein beliebiges Prädikat P für die „Null" gilt: P(Zero),
  - und wenn für jeden Wert vom Typ Nat aus Gültigkeit von P auch dessen Gültigkeit für den „Nachfolger" folgt: $P(n) \Rightarrow P(\text{Succ } n)$,
  - dann gilt P für alle n :: Nat.

Gleichungsbasiertes Schließen und Induktion für natürliche Zahlen

- Naheliegende Idee: Übertragung auf

```
data Nat = Zero | Succ Nat
```

  - wenn ein beliebiges Prädikat P für die „Null“ gilt: P(Zero),
  - und wenn für jeden Wert vom Typ Nat aus Gültigkeit von P auch dessen Gültigkeit für den „Nachfolger“ folgt: P(n) $\Rightarrow$ P(Succ n),
  - dann gilt P für alle n :: Nat.

- Für unsere „Zielaussage“, P(n) $\Leftrightarrow$ add n Zero = n, mit:

```
add :: Nat → Nat → Nat
add Zero      m = m
add (Succ n)  m = Succ (add n m)
```

erhalten wir als Beweisobligationen:

  - P(Zero): add Zero Zero = Zero
  - P(n) $\Rightarrow$ P(Succ n): (add n Zero = n) $\Rightarrow$ (add (Succ n) Zero = Succ n)

Gleichungsbasiertes Schließen und Induktion für natürliche Zahlen

- Für unsere „Zielaussage", P(n) $\Leftrightarrow$ add n Zero = n, mit:

```
add :: Nat → Nat → Nat
add Zero      m = m
add (Succ n)  m = Succ (add n m)
```

  erhalten wir als Beweisobligationen:
  - P(Zero): add Zero Zero = Zero
  - P(n) $\Rightarrow$ P(Succ n): (add n Zero = n) $\Rightarrow$ (add (Succ n) Zero = Succ n)

- Die benötigten Aussagen können wir tatsächlich zeigen:
  - P(Zero): add Zero Zero = Zero

    okay, wegen

    ```
    add Zero      m = m
    ```

  - P(n) $\Rightarrow$ P(Succ n): (add n Zero = n) $\Rightarrow$ (add (Succ n) Zero = Succ n)

    okay, wegen

    ```
    add (Succ n) Zero = Succ (add n Zero) = Succ n
    ```

Ein weiteres Beispiel

- Gegeben nun:

```
add :: Nat → Nat → Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)

mult :: Nat → Nat → Nat
mult m Zero     = Zero
mult m (Succ n) = add m (mult m n)
```

- Wir würden gern zeigen, dass für alle n :: Nat, mult Zero n = Zero.
- Das übliche Vorgehen:
  - P(Zero): mult Zero Zero = Zero, okay
  - P(n) ⇒ P(Succ n): (mult Zero n = Zero) ⇒ (mult Zero (Succ n) = Zero) okay, wegen

mult Zero (Succ n) = add Zero (mult Zero n) = mult Zero n = Zero

Vorsicht: Falle!

- Wir haben soeben bewiesen, dass für:

```
add :: Nat → Nat → Nat
add Zero      m = m
add (Succ n)  m = Succ (add n m)

mult :: Nat → Nat → Nat
mult m Zero       = Zero
mult m (Succ n) = add m (mult m n)
```

gilt: mult Zero n = Zero.

- Was ist mit:

```
infinity :: Nat
infinity = Succ infinity
```

?

- „Überraschenderweise“:

```
mult Zero infinity   =  mult Zero (Succ infinity) = add Zero (mult Zero infinity)
                     =  mult Zero infinity = … = „Endlosschleife“
```

Vorsicht: Falle!

- Was ist schiefgegangen?

- Das Beweisprinzip:
  - wenn P(Zero) und P(n) $\Rightarrow$ P(Succ n),
  - dann für alle n :: Nat, P(n)

  gilt nur für endliche n :: Nat !

- Für mathematische natürliche Zahlen ist das kein Problem, aber

```
data Nat = Zero | Succ Nat
```

  enthält eben auch unendliche oder (partiell) undefinierte Werte.

- Die Rettung des Beweisprinzips (zumindest für Prädikate in Gleichungsform):
  - zusätzliche Beweisobligation: P(undefined) für:

```
undefined :: Nat
undefined = undefined
```

Vorsicht: Falle!

- Rettung der Aussage am konkreten Beispiel:

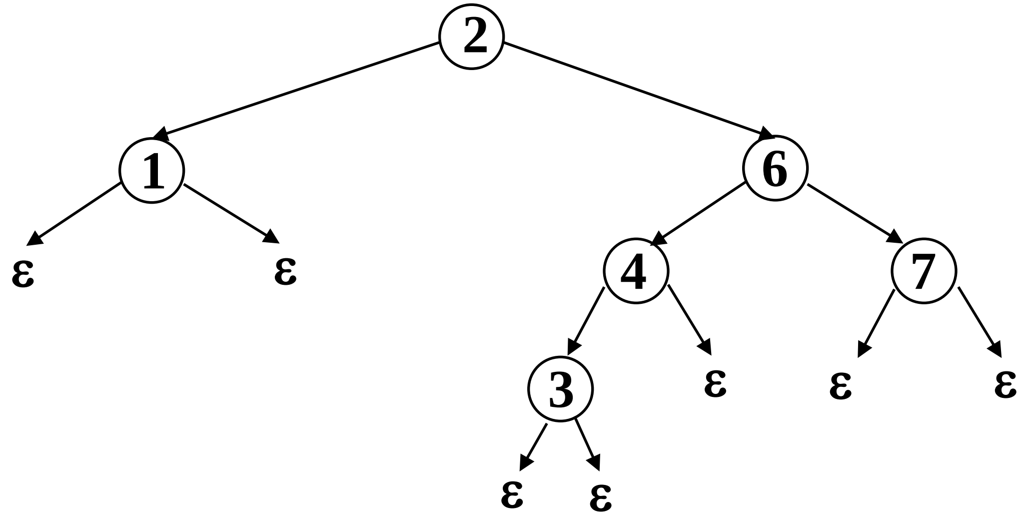
```
add :: Nat → Nat → Nat
add Zero       m = m
add (Succ n)  m = Succ (add n m)

mult :: Nat → Nat → Nat
mult Zero n        = Zero
mult m Zero        = Zero
mult m (Succ n) = add m (mult m n)
```

- Nun gilt offensichtlich mult Zero n = Zero für alle n :: Nat.
- Auch der Beweis geht durch (was im konkreten Fall hier natürlich schon trivial ist).
- Im Allgemeinen allerdings oft größere Komplikationen durch Berücksichtigung von Unendlichkeit/Nichtdefiniertheit.
- Daher im Folgenden meist Beschränkung auf Aussagen über endliche und total definierte Strukturen.

Beispiel: Induktion über Bäumen

- Zur Verallgemeinerung, eine interessante (und repräsentative) rekursive Datenstruktur: Binärbäume



Jeder Knoten hat zwei Nachfolger, die auch leer sein können (oben repräsentiert durch das Symbol ε).

```
data BinTree a  = Empty | Node (BinTree a) a (BinTree a)
```

Beispiel: Induktion über Bäumen

- Zwei Operationen auf Binärbäumen: Größe und Tiefe

$$
\begin{array}{ll}
\text{size, depth :: BinTree a} \to \text{Int} & \\
& \\
\text{size Empty} & = 0 \\
\text{size (Node } t_1 \text{ a } t_2) & = \text{size } t_1 + 1 + \text{size } t_2 \\
& \\
\text{depth Empty} & = 0 \\
\text{depth (Node } t_1 \text{ a } t_2) & = 1 + \text{depth } t_1 \text{ `max` depth } t_2
\end{array}
$$

- Möglicherweise interessante Zusammenhänge:

$$\textit{depth } t \leq \textit{size } t \leq 2\hat{\ }(\textit{depth } t) - 1$$

bzw.

$$\textit{ld}(\textit{size } t + 1) \leq \textit{depth } t \leq \textit{size } t$$

Beispiel: Induktion über Bäumen

Aufgrund der komplexeren rekursiven Struktur wird das Induktionsschema angepasst:

1) Induktionsanfang: leerer Baum, zu zeigen: P(Empty)

2) Induktionsschritt: Verzweigung, zu zeigen: $P(t_1) \wedge P(t_2) \Rightarrow \forall a.\ P(\text{Node } t_1 \text{ a } t_2)$

Beispiel:

P(t)

$$\textit{depth } t \leq \textit{size } t \leq 2 \wedge \textit{depth } t - 1$$

bzw.

$$\textit{ld}(\textit{size } t + 1) \leq \textit{depth } t \leq \textit{size } t$$

Induktionsanfang: size Empty = 2 ^ depth Empty – 1 = 0

Induktionsschritt: size (Node t_1 a t_2)

= size t_1 + 1 + size t_2
≤ 2 ^ depth t_1 – 1 + 1 + 2 ^ depth t_2 – 1
≤ 2 * (2 ^ depth t_1 \`max\` 2 ^ depth t_2) – 1
= 2 ^ 1 * 2 ^ (depth t_1 \`max\` depth t_2) – 1
= 2 ^ (1 + depth t_1 \`max\` depth t_2) – 1
= 2 ^ depth (Node t_1 a t_2) – 1

Zurück zu einigen der konkreten Fragestellungen über Listen …

- Behauptet wurde, dass allgemein gilt:

  concat (map (\y → if p y then [f y] else [ ]) ys)

  ist äquivalent zu:

  map f (filter p ys)

- Natürlich bietet sich nun zum Beweis eine Induktion über ys an.

- Die zu beweisenden Aussagen wären also, wegen

  data [a] = [ ] | (:) a [a]

  - P([ ]):

    concat (map (\y → if p y then [f y] else [ ]) [ ]) = map f (filter p [ ])

  - P(ys) ⇒ ∀y. P((:) y ys):

    concat (map (\y → if p y then [f y] else [ ]) ys) = map f (filter p ys)

    ⇒ concat (map (\y → if p y then [f y] else [ ]) (y : ys)) = map f (filter p (y : ys))

  - (und eventuell P(undefined))

Zurück zu einigen der konkreten Fragestellungen über Listen …

Der interessante Fall:

- P(ys) ⇒ ∀y. P((:) y ys):

concat (map (\y → if p y then [f y] else [ ]) ys) = map f (filter p ys)

⇒ concat (map (\y → if p y then [f y] else [ ]) (y : ys)) = map f (filter p (y : ys))

```
  map f (filter p (y : ys))
= map f (y : filter p ys)
= f y : map f (filter p ys)
= f y : concat (map (\y → if p y then [ f y ] else [ ]) ys)
= [ f y ] ++ concat (map (\y → if p y then [ f y ] else [ ]) ys)
= concat ([ f y ] : map (\y → if p y then [ f y ] else [ ]) ys)
= concat ((if p y then [ f y ] else [ ]) : map (\y → if p y then [ f y ] else [ ]) ys)
= concat (map (\y → if p y then [ f y ] else [ ]) (y : ys))
```

Unter Annahme, dass p y gilt!
Der andere Unterfall ist ähnlich.

Zurück zu: Äquivalenz verschiedener Versionen von reverse

- Für die Äquivalenz von effizientem und ineffizientem reverse gilt es zu zeigen, dass:

$$\text{reverse xs} = \text{reverse}' \text{ xs } [\,]$$

wobei:

$$\begin{array}{lcl} \text{reverse} :: [a] \to [a] \\ \text{reverse } [\,] & = & [\,] \\ \text{reverse } (x : xs) & = & \text{reverse xs} \mathbin{++} [x] \end{array}$$

$$\begin{array}{l} \text{reverse}' :: [a] \to [a] \to [a] \\ \text{reverse}' \; [\,] \quad \text{ys} = \text{ys} \\ \text{reverse}' \; (x : xs) \; \text{ys} = \text{reverse}' \; \text{xs} \; (x : \text{ys}) \end{array}$$

- Der Basisfall, $\text{reverse } [\,] = \text{reverse}' \; [\,] \; [\,]$, ist sehr einfach.

- Der induktive Fall,

$$\text{reverse xs} = \text{reverse}' \text{ xs } [\,] \quad \Rightarrow \quad \text{reverse } (x : xs) = \text{reverse}' \; (x : xs) \; [\,]$$

bereitet Kopfzerbrechen…

Zurück zu: Äquivalenz verschiedener Versionen von reverse

$$\begin{aligned} &\text{reverse} :: [a] \to [a] \\ &\text{reverse}\ [\,] = [\,] \\ &\text{reverse}\ (x : xs) = \text{reverse}\ xs \mathbin{{+}{+}} [x] \end{aligned}$$

$$\begin{aligned} &\text{reverse}' :: [a] \to [a] \to [a] \\ &\text{reverse}'\ [\,]\ ys = ys \\ &\text{reverse}'\ (x : xs)\ ys = \text{reverse}'\ xs\ (x : ys) \end{aligned}$$

- Beweisversuch für:

$$\text{reverse}\ xs = \text{reverse}'\ xs\ [\,] \quad \Rightarrow \quad \text{reverse}\ (x : xs) = \text{reverse}'\ (x : xs)\ [\,]$$

$$\begin{aligned} &\ \ \text{reverse}\ (x : xs) \\ &= \text{reverse}\ xs \mathbin{{+}{+}} [x] \\ &= \text{reverse}'\ xs\ [\,] \mathbin{{+}{+}} [x] \\ &= \ ??? \\ &= \text{reverse}'\ xs\ [x] \\ &= \text{reverse}'\ (x : xs)\ [\,] \end{aligned}$$

Beweis der Äquivalenz verschiedener Versionen von reverse: ein Problem!

$$\begin{aligned}&\text{reverse} :: [a] \to [a]\\&\text{reverse}\ [\,] = [\,]\\&\text{reverse}\ (x : xs) = \text{reverse}\ xs \mathbin{++} [x]\end{aligned}$$

$$\begin{aligned}&\text{reverse}' :: [a] \to [a] \to [a]\\&\text{reverse}'\ [\,]\ ys = ys\\&\text{reverse}'\ (x : xs)\ ys = \text{reverse}'\ xs\ (x : ys)\end{aligned}$$

- Dann versuchen wir eben als Nächstes, induktiv zu beweisen:

$$\text{reverse}'\ xs\ [\,] \mathbin{++} [x] = \text{reverse}'\ xs\ [x]$$

- Der interessante Fall:

$$\text{reverse}'\ xs\ [\,] \mathbin{++} [x] = \text{reverse}'\ xs\ [x] \;\Rightarrow\; \text{reverse}'\ (y : xs)\ [\,] \mathbin{++} [x] = \text{reverse}'\ (y : xs)\ [x]$$

$$\begin{aligned}&\ \text{reverse}'\ (y : xs)\ [\,] \mathbin{++} [x]\\&= \text{reverse}'\ xs\ [y] \mathbin{++} [x]\\&= \ ???\\&= \text{reverse}'\ xs\ [y, x]\\&= \text{reverse}'\ (y : xs)\ [x]\end{aligned}$$

Das scheint nicht wirklich irgendwohin zu führen…

Äquivalenz verschiedener Versionen von reverse: ein Problem!

$$\begin{aligned} &\mathrm{reverse} :: [a] \to [a] \\ &\mathrm{reverse}\ [\,] = [\,] \\ &\mathrm{reverse}\ (x : xs) = \mathrm{reverse}\ xs \mathbin{++} [x] \end{aligned}$$

$$\begin{aligned} &\mathrm{reverse}' :: [a] \to [a] \to [a] \\ &\mathrm{reverse}'\ [\,]\ ys = ys \\ &\mathrm{reverse}'\ (x : xs)\ ys = \mathrm{reverse}'\ xs\ (x : ys) \end{aligned}$$

- Statt:

$$\mathrm{reverse}\ xs = \mathrm{reverse}'\ xs\ [\,]$$

- oder:

$$\mathrm{reverse}'\ xs\ [\,] \mathbin{++} [x] = \mathrm{reverse}'\ xs\ [x]$$

- oder:

$$\mathrm{reverse}'\ xs\ [y] \mathbin{++} [x] = \mathrm{reverse}'\ xs\ [y, x]$$

- beweise man die allgemeinere Aussage:

$$\mathrm{reverse}\ xs \mathbin{++} ys = \mathrm{reverse}'\ xs\ ys$$

Äquivalenz verschiedener Versionen von reverse: die Lösung!

$$\begin{aligned}
&\text{reverse} :: [a] \rightarrow [a]\\
&\text{reverse}\ [\,] \quad = [\,]\\
&\text{reverse}\ (x : xs) = \text{reverse}\ xs \mathbin{++} [x]
\end{aligned}$$

$$\begin{aligned}
&\text{reverse}' :: [a] \rightarrow [a] \rightarrow [a]\\
&\text{reverse}'\ [\,] \quad ys = ys\\
&\text{reverse}'\ (x : xs)\ ys = \text{reverse}'\ xs\ (x : ys)
\end{aligned}$$

- Der interessante Fall:

$$\text{reverse}\ xs \mathbin{++} ys = \text{reverse}'\ xs\ ys \quad \Rightarrow \quad \text{reverse}\ (x : xs) \mathbin{++} ys = \text{reverse}'\ (x : xs)\ ys$$

$$\begin{aligned}
&\text{reverse}\ (x : xs) \mathbin{++} ys\\
&= (\text{reverse}\ xs \mathbin{++} [x]) \mathbin{++} ys\\
&= \text{reverse}\ xs \mathbin{++} ([x] \mathbin{++} ys)\\
&= \text{reverse}\ xs \mathbin{++} (x : ys)\\
&= \text{reverse}'\ xs\ (x : ys)\\
&= \text{reverse}'\ (x : xs)\ ys
\end{aligned}$$

- So weit, so gut. Aber hätten wir auch irgendwie automatisch von reverse zu reverse′ kommen können?