Beispielgetriebene Schemainduktion beim induktiven Programmieren

Martin Hofmann

Kognitive Systeme
Universität Bamberg

11. Februar 2010

Gliederung

Gliederung

Automatisches Programmieren

> *“We’re all lazy and trying to cheat — and let the computer write some piece of code.”*
>
> *Lennart Augustsson*

Automatisches Programmieren fortges.

Die Vision

Dem Computer das Problem beschreiben und ihn ein entsprechendes Programm erstellen lassen.

Beschreibungsmöglichkeiten

- Natürliche Sprache
- Vollständige und formale Spezifikation
  $\rightsquigarrow$ *Deduktive Programmsynthese*.
- Eingabe/Ausgabe (E/A) Beispiele, beisp. Berechnungsspuren
  $\rightsquigarrow$ *Induktive Programmsynthese*.

Automatisches generieren ganzer Softwaresysteme zu ambitioniert.

Halb-automatisches Erzeugen von Modulen, Funktionen und *Programmteilen*.

Automatisches Programmieren fortges.

Die Vision

Dem Computer das Problem beschreiben und ihn ein entsprechendes Programm erstellen lassen.

Beschreibungsmöglichkeiten

- Natürliche Sprache
- Vollständige und formale Spezifikation
  $\rightsquigarrow$ *Deduktive Programmsynthese*.
- Eingabe/Ausgabe (E/A) Beispiele, beisp. Berechnungsspuren
  $\rightsquigarrow$ *Induktive Programmsynthese*.

Automatisches generieren ganzer Softwaresysteme zu ambitioniert.

Halb-automatisches Erzeugen von Modulen, Funktionen und *Programmteilen*.

Induktive Programmsynthese (IPS)

IPS beschäftigt sich mit dem *automatischen Erzeugen (rekursiver) Programme* aus *unvollständigen Spezifikationen* (E/A Beispielen).

Beispiel: last

E/A Beispiele

```
last [a]       = a
last [a,b]     = b
last [a,b,c]   = c
last [a,b,c,d] = d
```

induziertes Programm

```
last [x]    = x
last (x:xs) = last xs
```

Anmerkung zur Syntax

```
   -- gezuckert
[1,2,3,4]
   -- normal infix
(1:2:3:4:[])
   -- normal prefix
((:) 1
     ((:) 2
          ((:) 3
               ((:) 4
                    [])))))
```

Induktive Programmsynthese (IPS)

IPS beschäftigt sich mit dem *automatischen Erzeugen (rekursiver) Programme* aus *unvollständigen Spezifikationen* (E/A Beispielen).

Beispiel: last

E/A Beispiele

```
last [a]       = a
last [a,b]     = b
last [a,b,c]   = c
last [a,b,c,d] = d
```

induziertes Programm

```
last [x]    = x
last (x:xs) = last xs
```

Anmerkung zur Syntax

```
   -- gezuckert
[1,2,3,4]
   -- normal infix
(1:2:3:4:[])
   -- normal prefix
((:) 1
     ((:) 2
          ((:) 3
               ((:) 4
                    [])))))
```

Induktive Programmsynthese (IPS)

IPS ist Suche in einer Klasse von Programmen.

Programmklasse beschrieben durch:

Syntaktische Bausteine

- Grundprimitive, gewöhnlich Datenkonstruktoren
- Hintergrundwissen, zusätzliche, problemspezifische, benutzerdefinierte Primitive
- Unterfunktionen, automatisch erzeugte Hilfsfunktionen

Restriction Bias

Syntaktische Einschränkungen einer deklarativen Sprache.

Ergebnis beeinflusst durch:

Preference (oder Such-) Bias

unterscheidet zwischen syntaktisch verschiedenen Ergebnissen.

Induktive Programmsynthese (IPS)

IPS ist Suche in einer Klasse von Programmen.

Programmklasse beschrieben durch:

Syntaktische Bausteine

- Grundprimitive, gewöhnlich Datenkonstruktoren
- Hintergrundwissen, zusätzliche, problemspezifische, benutzerdefinierte Primitive
- Unterfunktionen, automatisch erzeugte Hilfsfunktionen

Restriction Bias

Syntaktische Einschränkungen einer deklarativen Sprache.

Ergebnis beeinflusst durch:

Preference (oder Such-) Bias

unterscheidet zwischen syntaktisch verschiedenen Ergebnissen.

Induktive Programmsynthese (IPS)

Beispiel: reverse

E/A Beispiele

```
reverse  []  =  []          reverse [a,b]   = [b,a]
reverse [a] = [a]         reverse [a,b,c] = [c,b,a]
```

Induziertes funktionale Programm

```
reverse   []    = []
reverse (x:xs) = last (x:xs) : reverse(init (x:xs))
```

Automatisch induzierte Hilfsfunktionen (*umbenannt*)

```
last [x]    = x
last (x:xs) = last xs

init [a]    = []
init (x:xs) = x:(init xs)
```

Verschiedene Ansätze der IPS

	analytisch	*erzeuge & teste*	
		systematisch	*evolutionär*
funktional	IGOR I, IGOR II, THESYS	MAGIC-HASKELLER, G∀ST	ADATE
logisch	DIALOGS, DIALOGS-II, GOLEM	FFOIL, PROGOL	

Analytisch

Erzeuge & Teste: Systematisch

Erzeuge & Teste: Evolutionär

Verschiedene Ansätze der IPS

	analytisch	*erzeuge & teste*	
		systematisch	*evolutionär*
funktional	IGOR I, IGOR II, THESYS	MAGIC-HASKELLER, G∀ST	ADATE
logisch	DIALOGS, DIALOGS-II, GOLEM	FFOIL, PROGOL	

Analytisch

- Rekursive Funktionen berechnen die Ausgabe mit Hilfe des Ergebnisses einer “kleineren” Eingabe.
- Dies zeigt sich in Regularitäten in den E/A Beispielen.
- Regularitäten werden in eine rekursive Definition “gefaltet”.

Verschiedene Ansätze der IPS

	analytisch	*generiere & teste*	
		systematisch	*evolutionär*
funktional	IGOR I, IGOR II, THESYS	MAGIC-HASKELLER, G∀ST	ADATE
logisch	DIALOGS, DIALOGS-II, GOLEM	FFOIL, PROGOL	

Beispiel: last

(Variablen umbenannt)

```
last          (d:[]) = d
last        (c:d:[]) = d
last      (b:c:d:[]) = d
last    (a:b:c:d:[]) = d
```

```
last [x]    = x
last (x:xs) = last xs
```

Verschiedene Ansätze der IPS

	analytisch	*erzeuge & teste*	
		systematisch	*evolutionär*
funktional	IGOR I, IGOR II, THESYS	MAGIC-HASKELLER, G∀ST	ADATE
logisch	DIALOGS, DIALOGS-II, GOLEM	FFOIL, PROGOL	

Erzeuge & Teste: Systematisch

- Zähle alle korrekten Programme systematisch auf.
- Suchbasierte Einschränkungen (Typinformation, Bibliotheken)
- E/A werden nur zum Testen verwendet.

Verschiedene Ansätze der IPS

	analytisch	*erzeuge & teste*	
		systematisch	*evolutionär*
funktional	IGOR I, IGOR II, THESYS	MAGIC-HASKELLER, G∀ST	ADATE
logisch	DIALOGS, DIALOGS-II, GOLEM	FFOIL, PROGOL	

Erzeuge & Teste: Evolutionär

- Programme sind Individuen einer Population.
- *Genetische Operatoren* verändern Individuen (Kreuzung, Mutation, etc.)
- *Gesetz des Stärkeren* bez. Laufzeit, Größe, etc.
- E/A werden nur zum Testen verwendet.

Gliederung

IGOR II ist *funktional*, *analytisch* und *induktiv*

Induktiv

- induziert Programme aus unvollständigen Spezifikationen
- inspiriert durch Summers THESYS
- Nachfolger von IGOR I

Analytisch

- datengetrieben
- findet Rekursionen durch Analyse der E/As
- integrierte *uniforme Kostensuche*

Funktional

- IGOR II erzeugt funktionale Programme
- erster Prototyp von Emanuel Kitzelmann in MAUDE
- reimplementiert und erweitert in HASKELL

IGOR II ist *funktional*, *analytisch* und *induktiv*

Stärken

- **Termination** durch Konstruktion
- beliebige **nutzerdefinierte Datentypen**
- beliebiges **Hintergrundwissen** nutzbar
- **Unterfunktionen**
- **komplexe Aufrufbeziehungen** (Baum-, genestete Rekursion)
- E/As mit **Variablen**
- induziert gleichzeitig mehrere (wechselseitig) rekursive Zielfunktionen

Eingabe

Datentypdefinitionen

```
data [a] = [] | a:[a]
```

Zielfunktion

```
reverse :: [a] → [a]
reverse       [] =       []
reverse      [a] =      [a]
reverse    [a,b] =    [b,a]
reverse [a,b,c] = [c,b,a]
```

Hintergrundwissen

```
snoc :: [a] → a → [a]
snoc      [] x =      [x]
snoc     [x] y =    [x,y]
snoc   [x,y] z =  [x,y,z]
```

- Die ersten *n* E/A Beispiele müssen gegeben werden.
- Hintergrundwissen ist *optional*.

Ausgabe

Eine Menge von Gleichungen die die Beispiele modellieren.

reverse Lösung

```
reverse    []     = []
reverse (x:xs) = snoc (reverse xs) x
```

Restriction Bias

- Untermenge von HASKELL
- Fallunterscheidung durch *“pattern matching”*.
- Syntaktisch Einschränkungen: Patterns dürfen nicht unifizieren.

Preference Bias

- Minimale Anzahl von Fallunterscheidungen werden bevorzugt.

Grundidee

- Für eine (Unter-) Menge von Beispielen wird *eine* Regel gesucht, die alle erklärt.
- Initiale Hypothese ist die *"least general generalisation"* der Beispiele.

Beispielgleichungen

```
reverse [a]   = [a]
reverse [a,b] = [b,a]
```

Initiale Hypothese

```
reverse (x:xs) = (y:ys)
```

Hypothese enthält *ungebundene Variablen*!

Grundidee

- Für eine (Unter-) Menge von Beispielen wird *eine* Regel gesucht, die alle erklärt.
- Initiale Hypothese ist die *"least general generalisation"* der Beispiele.

Beispielgleichungen

```
reverse [a]   = [a]
reverse [a,b] = [b,a]
```

Initiale Hypothese

```
reverse (x:xs) = (y:ys)
```

Hypothese enthält *ungebundene Variablen*!

Grundidee

Initiale Hypothese

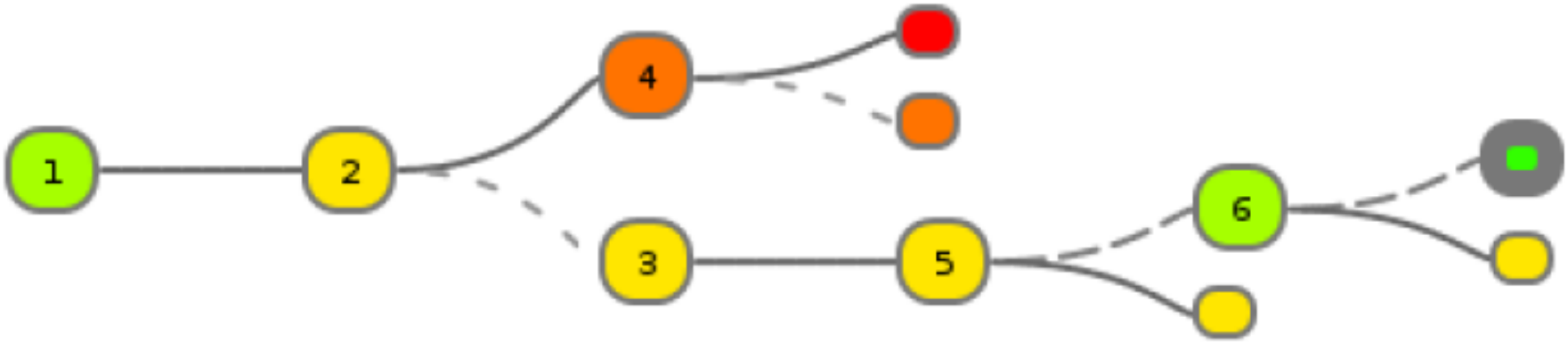
```
reverse (x:xs) = (y:ys)
```

Ungebundene Variablen sind der *Induktionshinweis*!

Drei Induktionsoperatoren gleichzeitig anzuwenden

1. **Partitionierung** der Beispieles
   $\leadsto$ *Mengen* von Gleichungen, Fallunterscheidungen
2. Ersetze rechte Seite durch **Programmaufruf** (rekursiv oder Hintergrundwissen).
3. Ersetze Unterterm mit ungebundener Variable durch **Unterfunktion**.

Grundidee



- In jeder Iteration *entwickle* die *beste* Hypothese mit den wenigsten *Fällen*.
- Bearbeiten einer Hypothese führt zu *mehreren Nachfolgern*.

Partitionierung der Beispiele, Fallunterscheidung

- Anti-unifizierte Terme unterscheiden sich an min. einer Position bez. des Konstruktors.
- Partitionierung der Beispiele bez. des Konstruktors an dieser Position

Beispiele

```
reverse   []      =   []
reverse (a:[])    = (a:[])
reverse (a:b:[]) = (b:a:[])
```

anti-unifizierter Term

```
reverse x = y
```

An der Wurzelposition stehen die Konstruktoren `[]` und `(:)`

{1}

```
reverse []  = []
```

{2, 3}

```
reverse (x:xs)  = (y:ys)
```

Partitionierung der Beispiele, Fallunterscheidung

- Anti-unifizierte Terme unterscheiden sich an min. einer Position bez. des Konstruktors.
- Partitionierung der Beispiele bez. des Konstruktors an dieser Position

Beispiele

```
reverse   []      =   []
reverse (a:[])    = (a:[])
reverse (a:b:[])  = (b:a:[])
```

anti-unifizierter Term

```
reverse x = y
```

An der Wurzelposition stehen die Konstruktoren `[]` und `(:)`

$\{1\}$

```
reverse []  = []
```

$\{2, 3\}$

```
reverse (x:xs)  = (y:ys)
```

Partitionierung der Beispiele, Fallunterscheidung

- Anti-unifizierte Terme unterscheiden sich an min. einer Position bez. des Konstruktors.
- Partitionierung der Beispiele bez. des Konstruktors an dieser Position

Beispiele

```
reverse   []      =   []
reverse  (a:[])   = (a:[])
reverse  (a:b:[]) = (b:a:[])
```

anti-unifizierter Term

```
reverse x = y
```

An der Wurzelposition stehen die Konstruktoren `[]` und `(:)`

$\{1\}$

```
reverse []  = []
```

$\{2,3\}$

```
reverse (x:xs)  = (y:ys)
```

Programmaufruf

```
reverse [a,b] = [b,a]        snoc [x] y = [x,y]
```

$$\Downarrow \{x \leftarrow b, y \leftarrow a\} \Downarrow$$

```
reverse [a,b] = snoc ? ?
```

- Passt eine Ausgabe auf jene einer anderen Funktion *f*, so kann diese durch einen Aufruf von *f* ersetzt werden.
- Konstruktion der Argumente des Aufrufs ist eine neues Induktionsproblem.
- E/A Beispiele werden abduziert:
  - Eingaben bleiben die gleichen.
  - Ausgaben sind die substituierten Eingaben der passenden Ausgabe.

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 berechnet 1. Argument

fun_2 berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 berechnet 1. Argument

fun_2 berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 berechnet 1. Argument

fun_2 berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 berechnet 1. Argument

fun_2 berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 berechnet 1. Argument fun_2 berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

$\texttt{fun}_1$ berechnet 1. Argument $\texttt{fun}_2$ berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 berechnet 1. Argument

fun_2 berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Programmaufruf – Beispiel

Beispielgleichung:

```
reverse [a,b] = b:[a]
```

Hintergrundwissen:

```
snoc [x] y = x:[y]
```

`(b:[a])` passt auf `(x:[y])` mit Substitution

$$\{x \leftarrow b, y \leftarrow a\}$$

ersetze rechte Seite von `reverse`

```
reverse [a,b] = snoc (fun1 [a,b]) (fun2 [a,b])
```

fun_1 berechnet 1. Argument

fun_2 berechnet 2. Argument

abduzierte Beispiele

Rechte Seite von `reverse` und subst. 1./2. Argument von `snoc`

```
fun1 [a,b] = [b]
```

```
fun2 [a,b] = a
```

Unterfunktionen

Beispielgleichungen:

```
reverse [a]   = [a]
reverse [a,b] = [b,a]
```

Initiale Hypothese:

```
reverse (x:xs) = (y:ys)
```

- Jeder Subterm der rechten Seite mit ungebundenen Variablen wird durch einen Aufruf einer Unterfunktion ersetzt.
- Konstruktion der Unterfunktion ist ein neues Induktionsproblem.
- E/As der Unterfunktion werden abduziert:
  - Eingaben bleiben bestehen.
  - Ausgaben werden die korrespondierenden Unterterme.

Unterfunktionen – Beispiele

Beispielgleichungen:

```
reverse [a]   = (a: [])
reverse [a,b] = (b:[a])
```

Initiale Hypothese:

```
reverse (x:xs) = (y : ys)
```

erhalte Konstruktoren und ersetze Variablen der rechten Seite

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

abduzierten E/As der Unterfunktionen

```
fun1 [a]   = a          fun2 [a]   = []
fun1 [a,b] = b          fun2 [a,b] = [a]
```

Unterfunktionen – Beispiele

Beispielgleichungen:

```
reverse [a]   = (a: [])
reverse [a,b] = (b:[a])
```

Initiale Hypothese:

```
reverse (x:xs) = (y : ys)
```

erhalte Konstruktoren und ersetze Variablen der rechten Seite

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

abduzierten E/As der Unterfunktionen

```
fun1 [a]   = a          fun2 [a]   = []
fun1 [a,b] = b          fun2 [a,b] = [a]
```

Unterfunktionen – Beispiele

Beispielgleichungen:

```
reverse [a]   = (a: [])
reverse [a,b] = (b:[a])
```

Initiale Hypothese:

```
reverse (x:xs) = (y : ys)
```

erhalte Konstruktoren und ersetze Variablen der rechten Seite

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

abduzierten E/As der Unterfunktionen

```
fun1 [a]   = a
fun1 [a,b] = b
```

```
fun2 [a]   = []
fun2 [a,b] = [a]
```

Unterfunktionen – Beispiele

Beispielgleichungen:

```
reverse [a]   = (a: [])
reverse [a,b] = (b:[a])
```

Initiale Hypothese:

```
reverse (x:xs) = (y) : (ys)
```

erhalte Konstruktoren und ersetze Variablen der rechten Seite

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

abduzierten E/As der Unterfunktionen

```
fun1 [a]   = a
fun1 [a,b] = b
```

```
fun2 [a]   = []
fun2 [a,b] = [a]
```

Unterfunktionen – Beispiele

Beispielgleichungen:

```
reverse [a]   = (a: [])
reverse [a,b] = (b:[a])
```

Initiale Hypothese:

```
reverse (x:xs) = (y : ys)
```

erhalte Konstruktoren und ersetze Variablen der rechten Seite

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

abduzierten E/As der Unterfunktionen

```
fun1 [a]   = a
fun1 [a,b] = b
```

```
fun2 [a]   = []
fun2 [a,b] = [a]
```

Unterfunktionen – Beispiele

Beispielgleichungen:

```
reverse [a]   = (a: [])
reverse [a,b] = (b:[a])
```

Initiale Hypothese:

```
reverse (x:xs) = (y : ys)
```

erhalte Konstruktoren und ersetze Variablen der rechten Seite

```
reverse (x:xs) = fun1 (x:xs) : fun2 (x:xs)
```

abduzierten E/As der Unterfunktionen

```
fun1 [a]   = a
fun1 [a,b] = b
```

```
fun2 [a]   = []
fun2 [a,b] = [a]
```

Gliederung

Problem

Suchproblem mit exponentiellem Aufwand

- Operatoranwendung läßt Auswahl:
  - viele Hintergrundfunktionen
  - viele “*matchings*”
  - viele Partitionsmöglichkeiten
- mehrere Operatoren

$\Longrightarrow$ Hoher Verzweigungsfaktor
$\Longrightarrow$ Plateaus im Suchraum

Traditionelle Lösung – Zusätzliches Wissen

Templates oder *Programmschemata*

Dies ist:

- *Expertenwissen*
- *problemspezifisch*

Es sollte aber sein:

- *benutzerunabhängig*
- *problemunabhängig*

Problem

Suchproblem mit exponentiellem Aufwand

$\Longrightarrow$ Hoher Verzweigungsfaktor

$\Longrightarrow$ Plateaus im Suchraum

Traditionelle Lösung – Zusätzliches Wissen

Templates oder *Programmschemata*

Dies ist:

- *Expertenwissen*
- *problemspezifisch*
- *schlecht diskriminierend*

Es sollte aber sein:

- *benutzerunabhängig*
- *problemunabhängig*
- *stark diskriminierend*

Problem

Suchproblem mit exponentiellem Aufwand

$\Longrightarrow$ Hoher Verzweigungsfaktor

$\Longrightarrow$ Plateaus im Suchraum

Traditionelle Lösung – Zusätzliches Wissen

Templates oder *Programmschemata*

Dies ist:

- *Expertenwissen*
- *problemspezifisch*
- *schlecht diskriminierend*

Es sollte aber sein:

- *benutzerunabhängig*
- *problemunabhängig*
- *stark diskriminierend*

Problem

Suchproblem mit exponentiellem Aufwand

$\Longrightarrow$ Hoher Verzweigungsfaktor

$\Longrightarrow$ Plateaus im Suchraum

Traditionelle Lösung – Zusätzliches Wissen

Templates oder *Programmschemata*

Dies ist:

- *Expertenwissen*
- *problemspezifisch*
- *schlecht diskriminierend*

Es sollte aber sein:

- *benutzerunabhängig*
- *problemunabhängig*
- *stark diskriminierend*

Problem

Suchproblem mit exponentiellem Aufwand

$\Longrightarrow$ Hoher Verzweigungsfaktor

$\Longrightarrow$ Plateaus im Suchraum

Traditionelle Lösung – Zusätzliches Wissen

Templates oder *Programmschemata*

Dies ist:

- *Expertenwissen*
- *problemspezifisch*
- *schlecht diskriminierend*

Es sollte aber sein:

- *benutzerunabhängig*
- *problemunabhängig*
- *stark diskriminierend*

Anpassen des Schemas an die Daten und nicht umgekehrt!

Universale Eigenschaften von Typmorphismen nutzen!

Catamorphismen auf Listen

Fold - ein Schema für strukturelle Rekursion

```
fold :: (a → b → b) → b → [a] → b
fold f v []     = v
fold f v (x:xs) = x `f` (fold f v xs)
```

```
fold f v (a:b:c:d:[]) ⇝ a `f` (b `f` (c `f` (d `f` [])))
```

Fold - Universale Eigenschaft

```
g []     = v
g (x:xs) = f x (g xs)
```

$\Longleftrightarrow$

```
g = fold f v
```

Ermöglicht `map` / `filter` / `reduce` Schema

Universale Eigenschaften von Typmorphismen nutzen!

Catamorphismen auf Listen

Fold - ein Schema für strukturelle Rekursion

```
fold :: (a → b → b) → b → [a] → b
fold f v []     = v
fold f v (x:xs) = x `f` (fold f v xs)
```

```
fold f v (a:b:c:d:[]) ⇝ a `f` (b `f` (c `f` (d `f` [])))
```

Fold - Universale Eigenschaft

```
g []     = v
g (x:xs) = f x (g xs)
```

$\Longleftrightarrow$

```
g = fold f v
```

Ermöglicht `map` / `filter` / `reduce` Schema

Neuer Operator – Fold Einführung

Ein analytischer “Vorverarbeitungsschritt” exklusiv anzuwenden

1. Programmschema als **Funktion höherer Ordnung** einführen

Versuche universelle Eigenschaft mit E/A zu erfüllen:

Für Funktion `g`:

1. Ist `g` definiert für `[]`?
2. Finde ein `f`, sodass für alle Beispiele subsumiert von `g (x:xs)` gilt:

```
f x (g xs) = g (x:xs)
```

3. Konstruktion von `f` ist neues Induktionsproblem.

Fold – Beispiel

Beispielgleichungen:

```
reverse        []   =       []
reverse (a:    [] ) =      [a]
reverse (a:   [b]) =    [b,a]
reverse (a:[b,c]) = [c,b,a]
```

Überprüfe universale Eigenschaft:

```
reverse []       = v
reverse (x:xs) = f x (reverse xs)
```

$\Longrightarrow$

```
v = []
f = fun3
```

Reformuliere Problem:

```
reverse x = fold fun3 [] x
```

Abduziere E/As:

```
fun3 a     []  =      [a]
fun3 a    [b] =    [b,a]
fun3 a [c,b] = [c,b,a]
```

Fold – Beispiel

Beispielgleichungen:

```
reverse        []   =       []
reverse (a:    [] ) =      [a]
reverse (a:   [b]) =    [b,a]
reverse (a:[b,c]) = [c,b,a]
```

Überprüfe universale Eigenschaft:

```
reverse []       = v                    ⟹    v = []
reverse (x:xs) = f x (reverse xs)             f = fun3
```

Reformuliere Problem:

```
reverse x = fold fun3 [] x
```

Abduziere E/As:

```
fun3 a     []  =      [a]
fun3 a    [b]  =    [b,a]
fun3 a [c,b]  = [c,b,a]
```

Fold – Beispiel

Beispielgleichungen:

```
reverse        []   =       []
reverse (a:    [] ) =      [a]
reverse (a:   [b]) =    [b,a]
reverse (a:[b,c]) = [c,b,a]
```

Überprüfe universale Eigenschaft:

```
reverse []       = v                          v = []
reverse (x:xs) = f x (reverse xs)      ⟹      f = fun3
```

Reformuliere Problem:

```
reverse x = fold fun3 [] x
```

Abduziere E/As:

```
fun3 a     []  =      [a]
fun3 a    [b]  =    [b,a]
fun3 a [c,b]  = [c,b,a]
```

Fold – Beispiel

Beispielgleichungen:

```
reverse        []   =       []
reverse (a:    [] ) =      [a]
reverse (a:   [b]) =    [b,a]
reverse (a:[b,c]) = [c,b,a]
```

Überprüfe universale Eigenschaft:

```
reverse []       = v
reverse (x:xs) = f x (reverse xs)
```

$\Longrightarrow$

```
v = []
f = fun3
```

Reformuliere Problem:

```
reverse x = fold fun3 [] x
```

Abduziere E/As:

```
fun3 a     []  =      [a]
fun3 a    [b] =    [b,a]
fun3 a [c,b] = [c,b,a]
```

Fold – Beispiel

Beispielgleichungen:

```
reverse []     = []
reverse (a:[]) = [a]
reverse (a:[b]) = [b,a]
reverse (a:[b,c]) = [c,b,a]
```

Überprüfe universale Eigenschaft:

```
reverse []     = v
reverse (x:xs) = f x (reverse xs)
```

$\Longrightarrow$

```
v = []
f = fun3
```

Reformuliere Problem:

```
reverse x = fold fun3 [] x
```

Abduziere E/As:

```
fun3 a     []  = [a]
fun3 a    [b]  = [b,a]
fun3 a [c,b]  = [c,b,a]
```

Fold – Beispiel

Beispielgleichungen:

```
reverse []     = []
reverse (a:[]) = [a]
reverse (a:[b]) = [b,a]
reverse (a:[b,c]) = [c,b,a]
```

Überprüfe universale Eigenschaft:

```
reverse []     = v
reverse (x:xs) = f x (reverse xs)
```

$\Longrightarrow$

```
v = []
f = fun3
```

Reformuliere Problem:

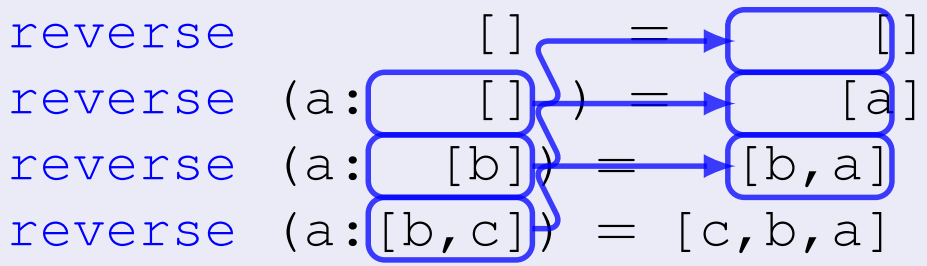
```
reverse x = fold fun3 [] x
```

Abduziere E/As:

```
fun3 a []     = [a]
fun3 a [b]    = [b,a]
fun3 a [c,b]  = [c,b,a]
```

Fold – Beispiel

Beispielgleichungen:



Überprüfe universale Eigenschaft:

```
reverse []     = v
reverse (x:xs) = f x (reverse xs)
```

$\Longrightarrow$

```
v = []
f = fun3
```

Reformuliere Problem:

```
reverse x = fold fun3 [] x
```

Abduziere E/As:

```
fun3 a []    =    [a]
fun3 a [b]   =  [b,a]
fun3 a [c,b] = [c,b,a]
```

Optimierungen

Sei *IO* eine Beispielmenge der Form `fun` a_i b_i $= o_i$ mit $i = 1 \ldots n$.

`map` verwenden

Wenn lgg(*IO*) gleich `fun x xs = y:xs`

- ignoriere 2. Argument in E/As von `fun`
- `g x = map fun x`

`filter` verwenden

Ist *IO* in zwei disjunkte Mengen IO_1 und IO_2 teilbar, sodass

lgg(IO_1): `fun x xs = x:xs` lgg(IO_2): `fun x xs = xs`

- dann sei `fun'` eine neue Funktion mit

```
fun' ai = True  -- (*@ f r ai aus IO1 @*)
fun' ai = False -- (*@ f r ai aus IO2 @*)
```

- `g x = filter fun' x`

Funktionale Programmierung und Kategorientheorie

- In der Kategorie der Mengen **Set**
  - seien *Objekte (Mengen)* Typen
  - und *Morphismen* totale Funktionen zwischen Typen
- Morphismen aus der Einermenge ($\{()\}$), d.h. ein terminales Objekt,

  $$a : \mathbf{1} \to A$$

  weisen einem Typ A einen Wert a zu.
- Funktionskomposition

  $$f \circ a : \mathbf{1} \to B$$

  beschreibt die Anwendung der Funktion $f : A \to B$ auf den Wert $a : A$

Terminologie

Sei $\mathcal{C}$ eine *distributive* Kategorie, d.h. mit *finiten Produkten* $(\times, \mathbf{1})$, *finiten Koprodukten* $(+, \mathbf{0})$, und der Distribution von *Produkten über Koprodukten*.

$A \times B$ ist das Produkt von A und B

mit Projektionen

$$\begin{aligned} fst_{A,B} &: A \times B \to A \\ snd_{A,B} &: A \times B \to B \end{aligned}$$

$A + B$ ist das Koprodukt von A und B

mit Injektionen

$$\begin{aligned} inl_{A,B} &: A \to A + B \\ inr_{A,B} &: B \to A + B \end{aligned}$$

Terminologie fortges.

Das Paar $f \otimes g : C \to A \times B$

der Morphismen $f : C \to A$ und $g : C \to B$ ist der eindeutige Morphismus, sodass

$$\begin{aligned} fst_{A,B} \circ f \otimes g &= f \\ snd_{A,B} \circ f \otimes g &= g \end{aligned}$$

Die Fallunterscheidung $f \oplus g : A + B \to C$

der Morphismen $f : A \to C$ und $g : B \to C$ ist der eindeutige Morphismus, sodass

$$\begin{aligned} f \oplus g \circ inl_{A,B} &= f \\ f \oplus g \circ inr_{A,B} &= g \end{aligned}$$

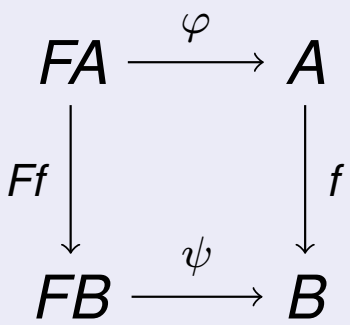
F-Algebren

F-Algebra — funktorinduzierte Algebra

Sei $F : \mathcal{C} \to \mathcal{C}$ ein Endofunktor. Ein F-Algebra (Algebra mit Signatur F) ist ein Tupel $\mathbf{A} = (A, \varphi)$, bestehend aus einem Objekt A (Träger) und einem Morphismus $\varphi : FA \to A$ (Struktur) aus $\mathcal{C}$.

F-Algebra Morphismus

Seien $\mathbf{A} = (A, \varphi)$ und $\mathbf{B} = (B, \psi)$ F-Algebren, ein Homomorphismus zwischen $\mathbf{A}$ und $\mathbf{B}$ ist ein Morphismus $f : A \to B$ aus $\mathcal{C}$, sodass



$$f \circ \varphi = \psi \circ Ff$$

Kategorie der F-Algebren und initiale Objekte

Kategorie der F -Algebras – $\mathbf{Alg}_F$

F-Algebren und F-Algebra Morphismen mit Identität und Komposition.

Initiale F-Algebra – $\mathbf{Alg}_F$

Ein initiales Objekt $\mathbf{Alg}_F$ ist eine initiale F-Algebra $\mu F = (\mu F, in_F)$

F-Catamorphismus von φ – $(\!|\, \varphi \,|\!)_F$

Für Endofunktor F mit initialer F-Algebra existiert für jede F-Algebra $\mathbf{A} = (A, \varphi)$ ein eindeutiger Morphismus $(\!|\, \varphi \,|\!)_F : \mu F \to A$, sodass

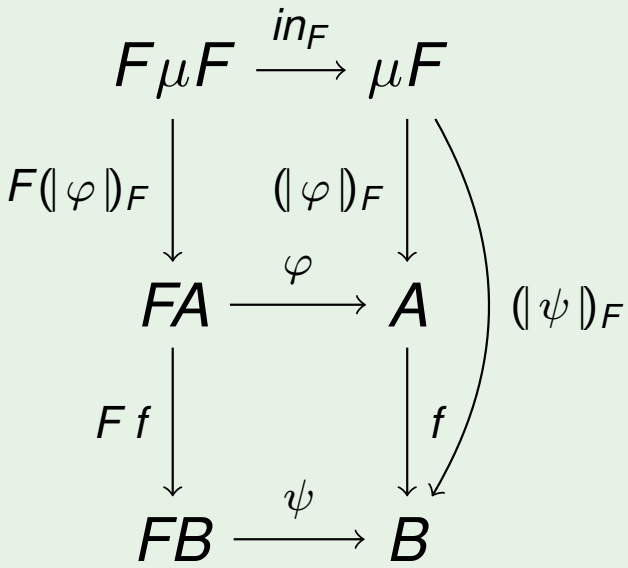
$$\begin{array}{ccc} F\mu F & \xrightarrow{in_F} & \mu F \\ {\scriptstyle F(\!|\varphi|\!)_F}\downarrow & & \downarrow{\scriptstyle (\!|\varphi|\!)_F} \\ FA & \xrightarrow{\varphi} & A \end{array} \qquad f \circ in_F = \varphi \circ F(\!|\, \varphi \,|\!)_F$$

Initiale F-Algebren als Fixpunkte von F

Gegeben:

- initiale F-Algebra $(\mu F, in)$
- beliebige F-Algebren $\varphi : FA \to A$ und $\psi : FB \to B$
- Morphismus $f : A \to B$

Dann:

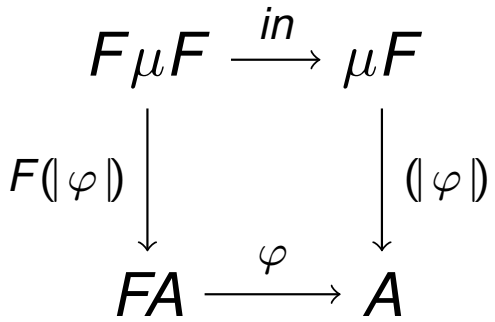


1. $(\!|\varphi|\!)_F \circ in_F = \varphi \circ F(\!|\varphi|\!)_F$
2. $id = (\!|in_F|\!)_F$
3. $f \circ \varphi = \psi \circ F f \Rightarrow f \circ (\!|\varphi|\!)_F = (\!|\psi|\!)_F$
4. $in_F^{-1} = (\!|F\, in_F|\!)_F$

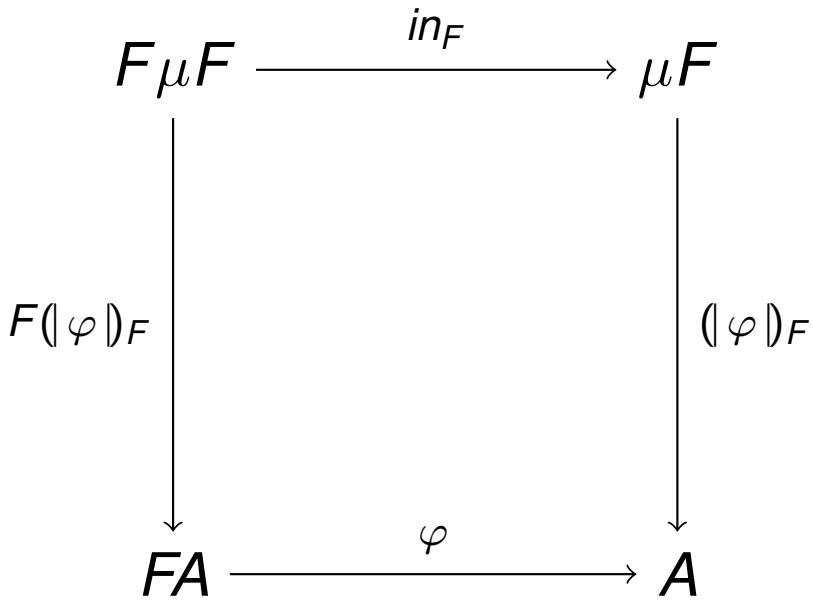
$in_F : F\mu F \to \mu F$ ist ein Isomorphismus und μF ist der Fixpunkt von F.

Initiale Algebren und induktive Datentypen

- Der Endofunktor F beschreibt die Signatur
- Die initiale Algebra $\mathbf{A} = (\mu F, in)$ ist der induktive Type
  - sowohl der reine Container (μF)
  - als auch die Konstruktoren (in)
- Catamorphismen, Zeugen der Initialität, korrespondieren mit induktiv definierten Funktionen:
  - Der Zieltyp und die Funktionen des induktive Schritts sind Algebren.
  - Der Catamorphismus beschreibt die induktive Funktion als Ganzes.



Cons-Listen als initiale F-Algebra — Beispiel

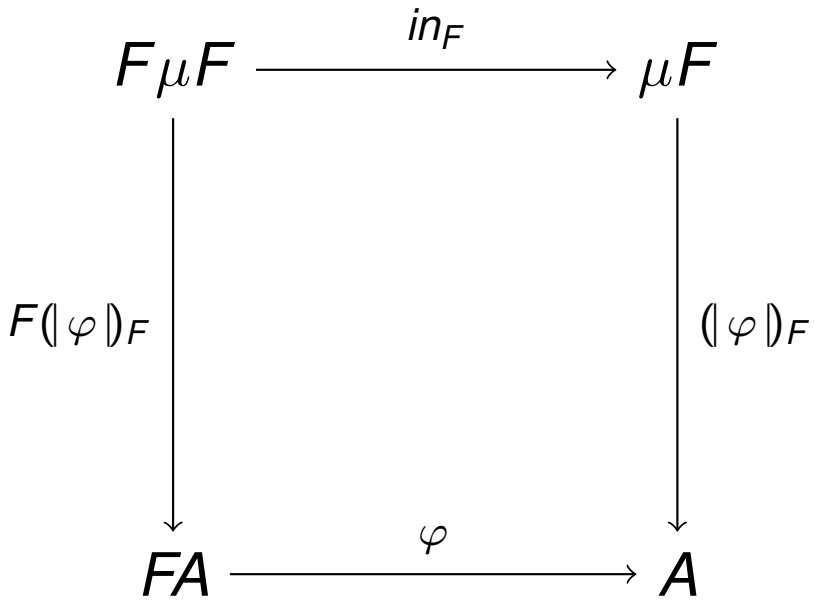


Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,

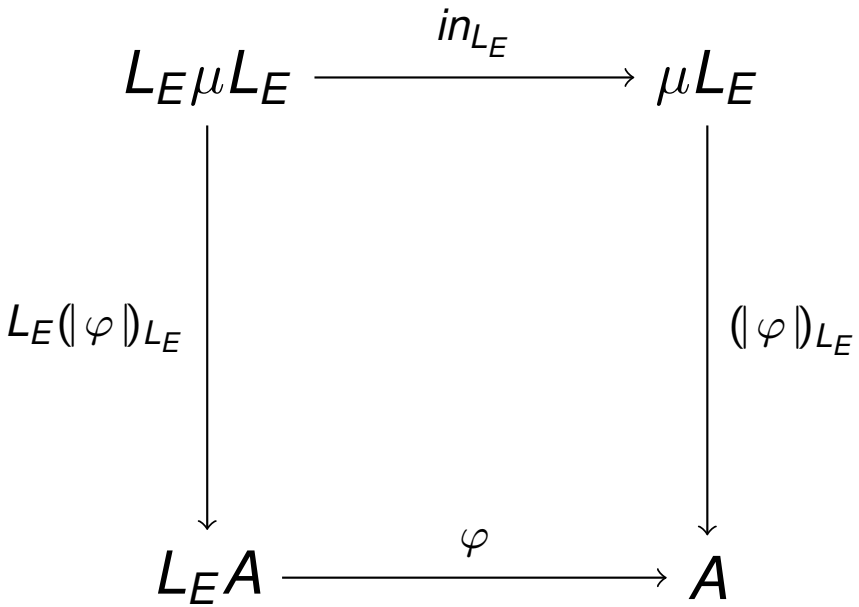


Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

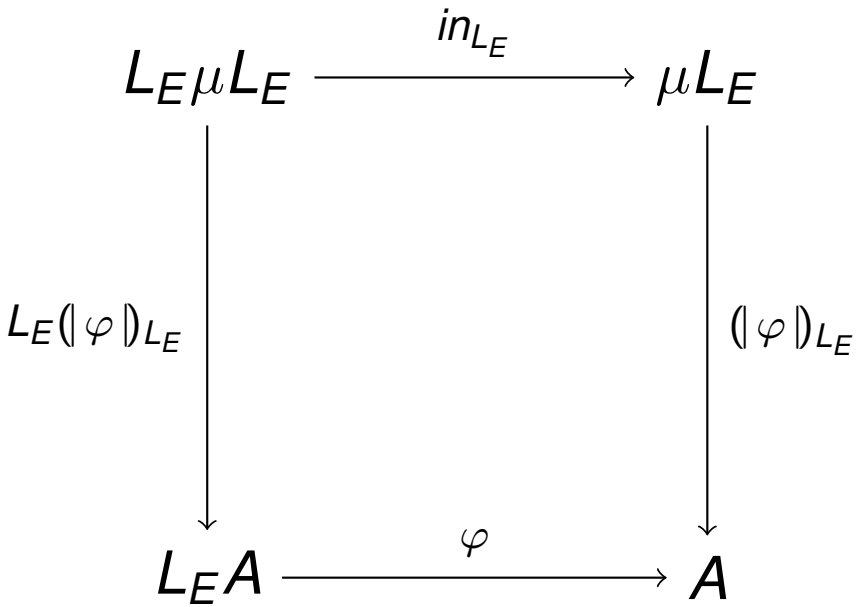
- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_1 \oplus id_E \otimes f$,



Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

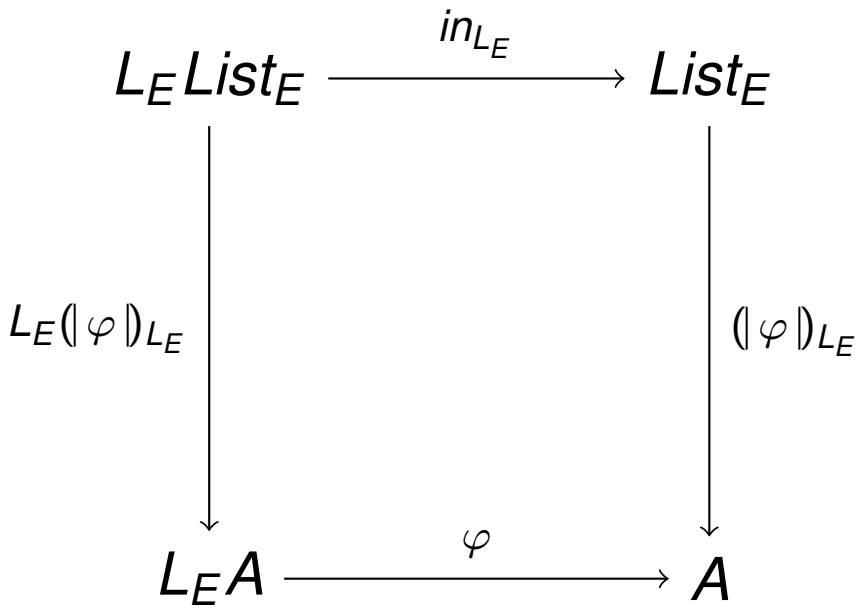
- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,
- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_1 \oplus id_E \otimes f$,



Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

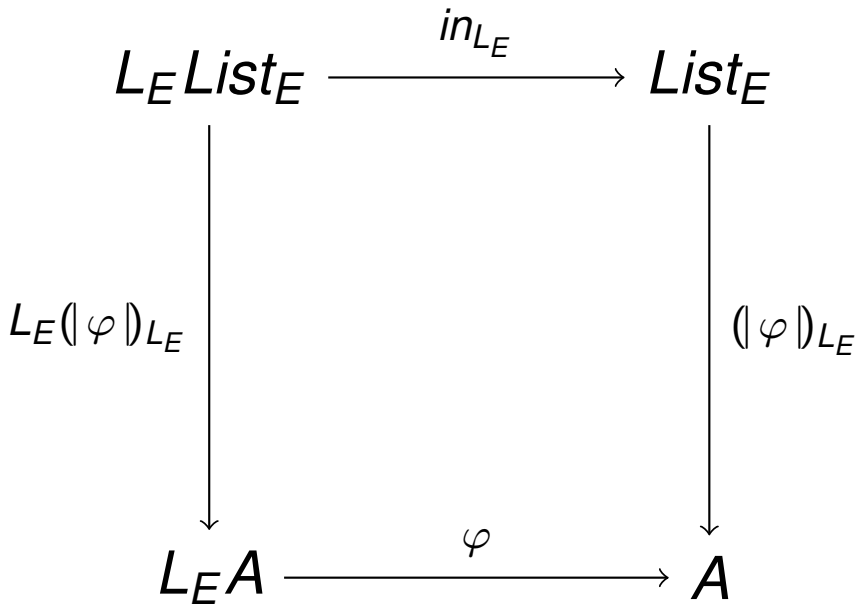
- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,
- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_1 \oplus id_E \otimes f$,



Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,
- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_1 \oplus id_E \otimes f$,

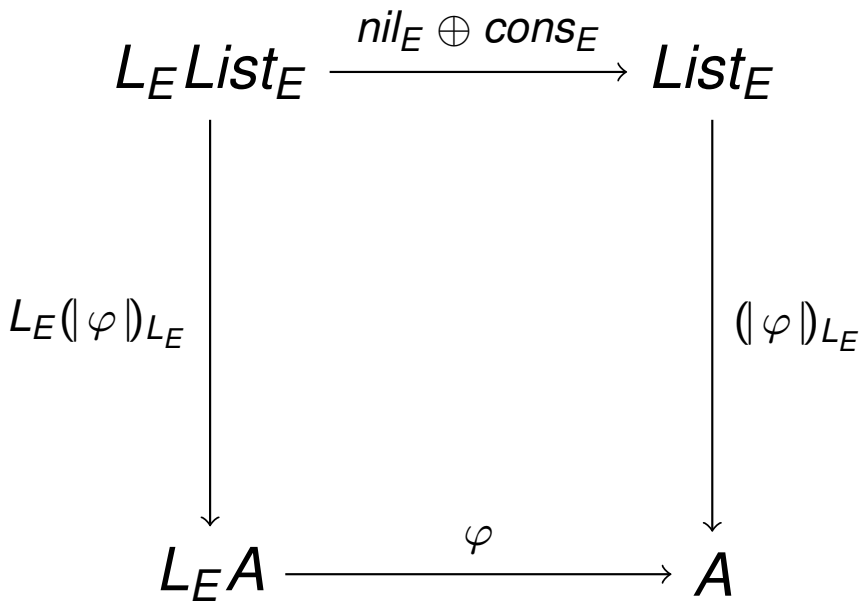


Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,



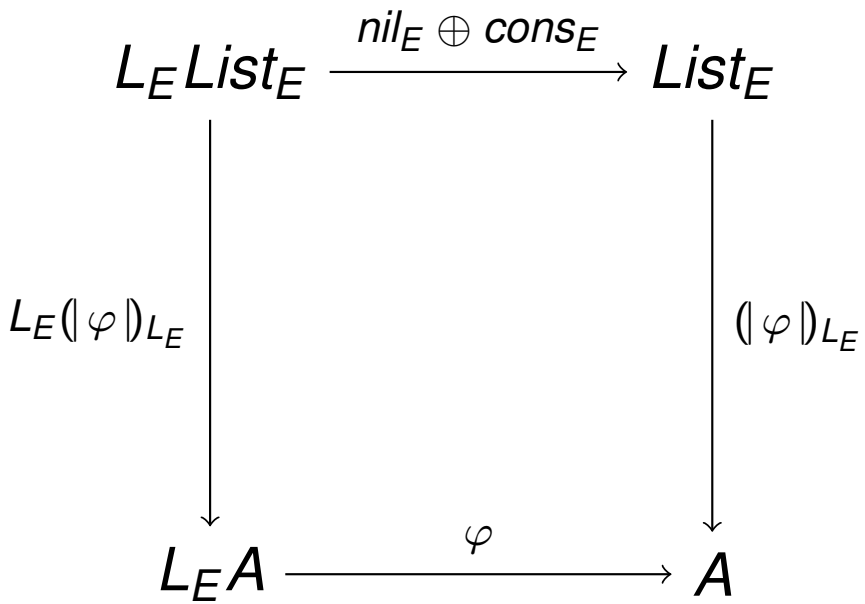
Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

sodass

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,



Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

sodass

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,

$$\begin{array}{ccc} \mathbf{1} + (E \times List_E) & \xrightarrow{nil_E \oplus cons_E} & List_E \\ \Big\downarrow{\scriptstyle L_E(\!|\varphi|\!)_{L_E}} & & \Big\downarrow{\scriptstyle (\!|\varphi|\!)_{L_E}} \\ \mathbf{1} + (E \times A) & \xrightarrow{\varphi} & A \end{array}$$

Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

sodass

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,

$$\begin{array}{ccc} \mathbf{1} + (E \times List_E) & \xrightarrow{nil_E \oplus cons_E} & List_E \\ \Big\downarrow{\scriptstyle L_E(\!|\varphi|\!)_{L_E}} & & \Big\downarrow{\scriptstyle (\!|\varphi|\!)_{L_E}} \\ \mathbf{1} + (E \times A) & \xrightarrow{\varphi} & A \end{array}$$

Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

sodass

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,

$$\begin{array}{ccc} \mathbf{1} + (E \times List_E) & \xrightarrow{nil_E \oplus cons_E} & List_E \\ \downarrow{\scriptstyle id_{\mathbf{1}} \oplus (id_E \otimes (\!| \varphi |\!)_{L_E})} & & \downarrow{\scriptstyle (\!| \varphi |\!)_{L_E}} \\ \mathbf{1} + (E \times A) & \xrightarrow{\varphi} & A \end{array}$$

Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

sodass

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,

wobei $\varphi = g \oplus h$.

$$\begin{array}{ccc} \mathbf{1} + (E \times List_E) & \xrightarrow{nil_E \oplus cons_E} & List_E \\ \Big\downarrow{\scriptstyle id_{\mathbf{1}} \oplus (id_E \otimes (\!|\varphi|\!)_{L_E})} & & \Big\downarrow{\scriptstyle (\!|\varphi|\!)_{L_E}} \\ \mathbf{1} + (E \times A) & \xrightarrow{\varphi} & A \end{array}$$

Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

sodass

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,

wobei $\varphi = g \oplus h$.

$$\begin{array}{ccc} \mathbf{1} + (E \times List_E) & \xrightarrow{nil_E \oplus cons_E} & List_E \\ \downarrow{\scriptstyle id_{\mathbf{1}} \oplus (id_E \times (\!| g \oplus h |\!)_{L_E})} & & \downarrow{\scriptstyle (\!| g \oplus h |\!)_{L_E}} \\ \mathbf{1} + (E \times A) & \xrightarrow{g \oplus h} & A \end{array}$$

Cons-Listen als initiale F-Algebra — Beispiel

Für einen beliebigen Typ E sei L_E ein Endofunktor mit

- Fixpunkt $\mu L_E = List_E$ und
- $in_{L_E} = nil_E \oplus cons_E$,

sodass

- für einen beliebigen Typ A, $L_E A = \mathbf{1} + E \times A$, und
- für ein beliege Funktion f, $L_E f = id_{\mathbf{1}} \oplus id_E \otimes f$,

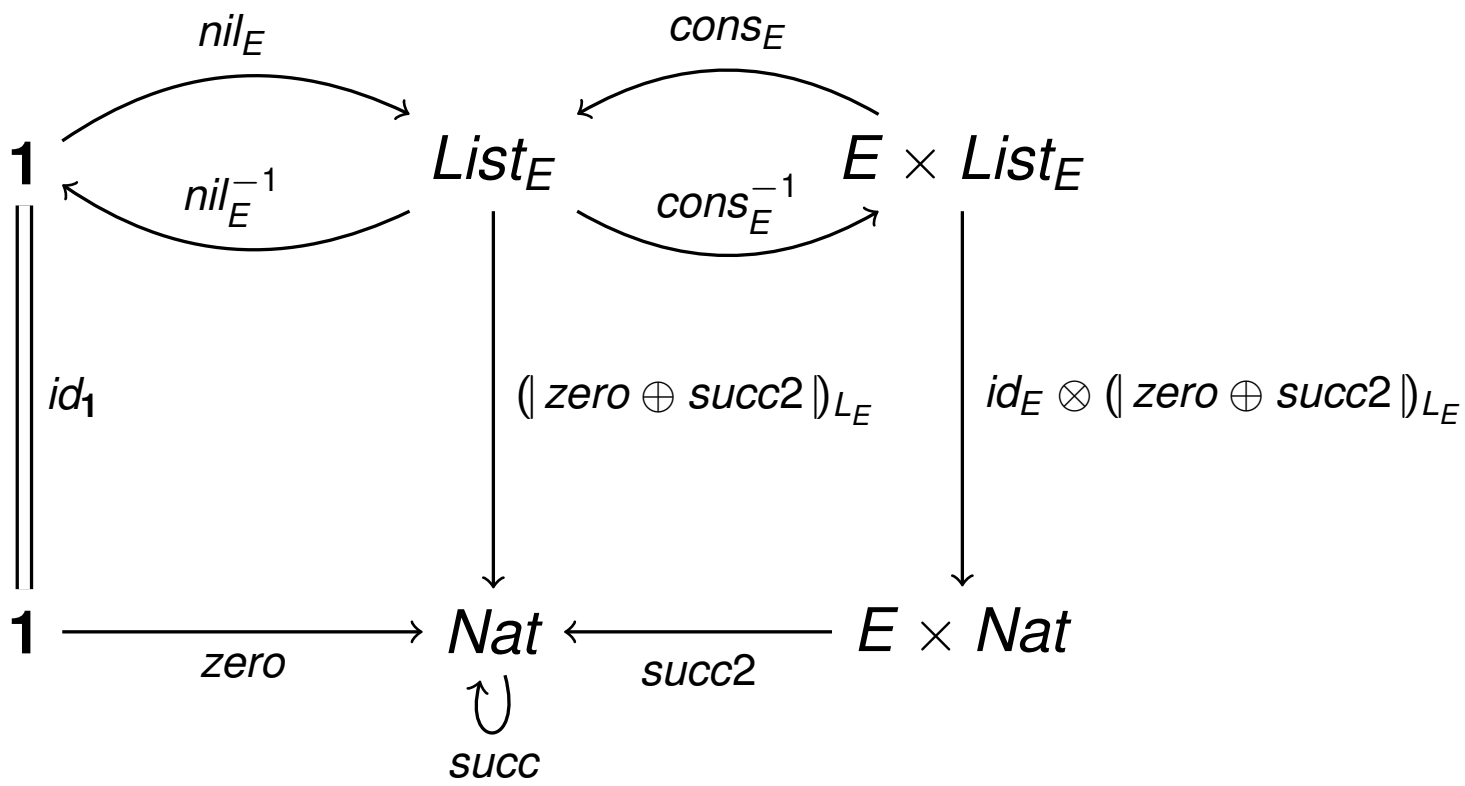
wobei $\varphi = g \oplus h$.

$$\begin{array}{ccccc}
\mathbf{1} & \xrightarrow{nil_E} & List_E & \xleftarrow{cons_E} & E \times List_E \\
\| & & \downarrow{\scriptstyle (\!| g \oplus h |\!)_{L_E}} & & \downarrow{\scriptstyle id_E \times (\!| g \oplus h |\!)_{L_E}} \\
\mathbf{1} & \xrightarrow{g} & A & \xleftarrow{h} & (E \times A)
\end{array}$$

$id_{\mathbf{1}}$

$List_E$-Iteration ist der eindeutige Morphismus $(\!| g \oplus h |\!)_{L_E} : List_E \to A$.

`length` als L_E-Catamorphismus



$$succ2 = succ \circ snd_{E,Nat}$$

```
length, length' :: [e] → Nat
length     []    = 0
length (x:xs) = 1 + (length xs)

length'       = cata (const 0 λ/ (+1).snd )
```

Implementierung in HASKELL

```
{-# OPTIONS_GHC -XTypeOperators -XTypeFamilies   #-}
module MorphEx where

import Generics.Pointless.Combinators
import Generics.Pointless.Functors hiding (Nat, Cons)
import Generics.Pointless.RecursionPatterns

-- Datentypdefinition

data List a = NilL | Cons a (List a) deriving (Show)

-- Typinstanz von Pattern Functor
type instance PF (List a) = Const One :+: (Const a :*: Id)
```

Implementierung in HASKELL fortges.

```
instance Mu (List a) where
    inn (Left _)        = NilL
    inn (Right (a,l))   = Cons a l
    out NilL            = Left _L
    out (Cons a l)      = Right (a,l)

-- Catamorphismus Beispiel

length :: List a → Nat
length = cata ( _L :: (List a) ) ( g λ/ h )
  where
  g _ = Z
  h (_,n) = S n
```

Abduktion der E/A Beispiele

- Koprodukte induzieren Partitionen
- Terminale Objekte sind konstante Funktionen
- Produkte induzieren Funktionen mit Tupelparametern
- Fixpunkte werden durch entsprechende “kleinere” Ausgaben ersetzt
- Alles andere bleibt unverändert

```
length = cata ( _L :: (List a) ) ( g λ/ h )


length []          = Z          g []           = Z
length (a:[])      = (S Z)
length (a:b:[])    = (S(S Z))   h (a, Z)       = (S Z)
length (a:b:c:[]) = (S(S(S Z))) h (a, S Z)     = (S(S Z))
   )                            h (a, S(S Z)) = (S(S(S Z)))
```

Gliederung

Empirischer Vergleich I

Funktion	BK	ohne $(\|\cdot\|)$	mit $(\|\cdot\|)$	Δ
`allodd`	—	↻	6	6
`and`	—	↻	2	2
`evens`	—	23	4	19
`fib`	`add`	187	187	0
`hanoi`	—	5	5	0
`lengths`	—	36	2	34
`powset`	`append`	76	5	71
`odd/even`	—	2/2	2/2	0/0
`shiftr`	—	10	3	7
`sum`	—	6	2	1
`switch`	—	10	↻	—
`zeros`	—	6	2	4

Anzahl der Iterationen

↻ Zeitlimit überschritten, BK Hintergrundwissen

Empirischer Vergleich II

	ohne $(\|\cdot\|)$	mit $(\|\cdot\|)$	Δ
	CPU in Sekunden		
Σ	16.4859	5.2871	11.1988
max	8.7245	3.0962	5.6283
min	0.0001	0.0001	0.0000
$\tilde{X}$	0.0080	0.0001	0.0079
$\varnothing$	0.3111	0.0999	0.2112
σ_X	1.2694	0.4512	0.8182
	Iterationen		
Σ	2107	555	1552
max	1208	206	1002
min	1	1	0
$\tilde{X}$	5	2	3
$\varnothing$	39.7500	10.4700	29.2800
σ_X	167.2719	35.8448	131.4271

$\varnothing$ Mittel, $\tilde{X}$ Median, σ_X Std.Abw.

Auf Grundlage von 53 Beispielproblemen.

Fazit

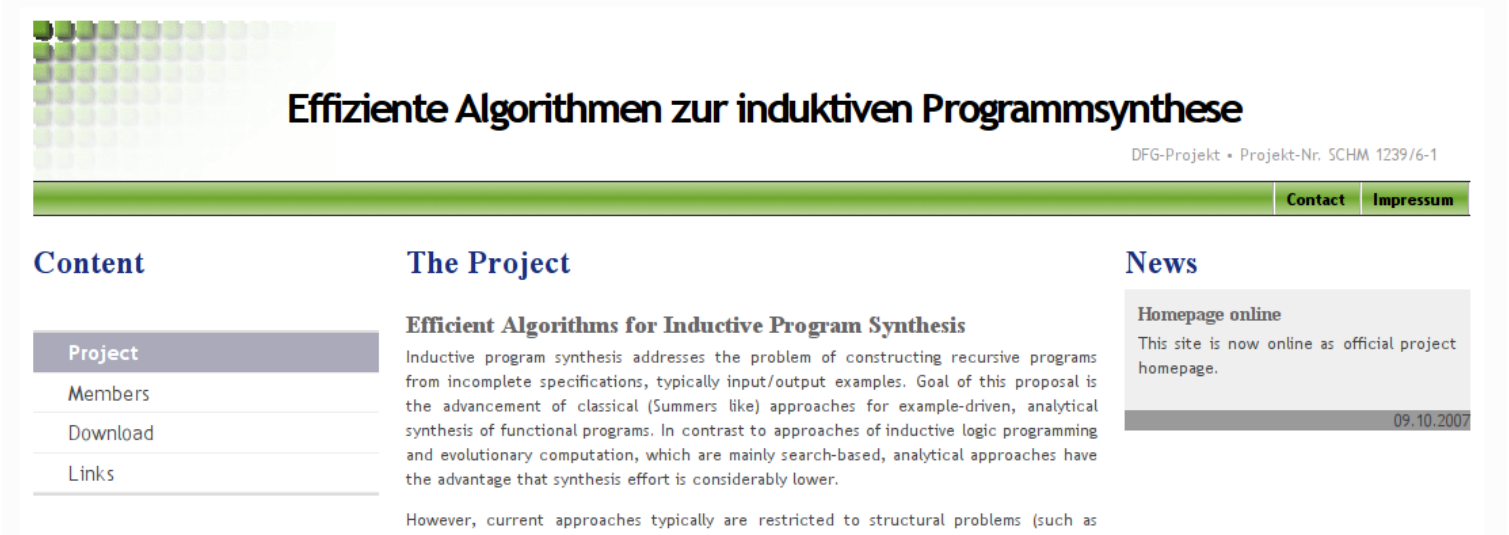
- Catamorphismen lassen sich als Rekursionsschema für IPS datengetrieben verwenden
- Die Anwendbarkeit lässt sich durch Überprüfen der universellen Eigenschaften in den E/As zeigen.
- Die Verwendung von Schemata
  - reduziert die Anzahl der Schleifendurchläufe
  - verbessert die Mächtigkeit des Algorithmus insgesamt

Analytische Verwendung anderer Schemata könnte die Fähigkeit von IP Systemen weiter verbessern.

Ausblick

- Sensitivität $\stackrel{?}{=} 1$, Spezifität < 1
- Programmschemata bez. universeller Eigenschaften in Entscheidungsbaum von *speziell zu generell* anordnen.
  - Andere Morphismen für strukturelle Rekursion ausprobieren.
    - *Hylomorphismus* (primitive Rekursion)
    - *Paramorphismus* (primitive Rekursion via Tupel)
    - *Anamorphismus* (strukturelle Koinduktion)
    - *Apomorphismus* (primitive Korekursion via Tupel)
    - ...
  - Haben derartige Funktionen “aussagekräftige” E/As?
  - Können ihre universellen Eigenschaften ähnlich leicht implementiert werden?
  - Können ihre universellen Eigenschaften diskriminierend?

Unser Projekt



http://www.cogsys.wiai.uni-bamberg.de/effalip/

- Publikationen
- Downloads
- Links

inductive-programming.org

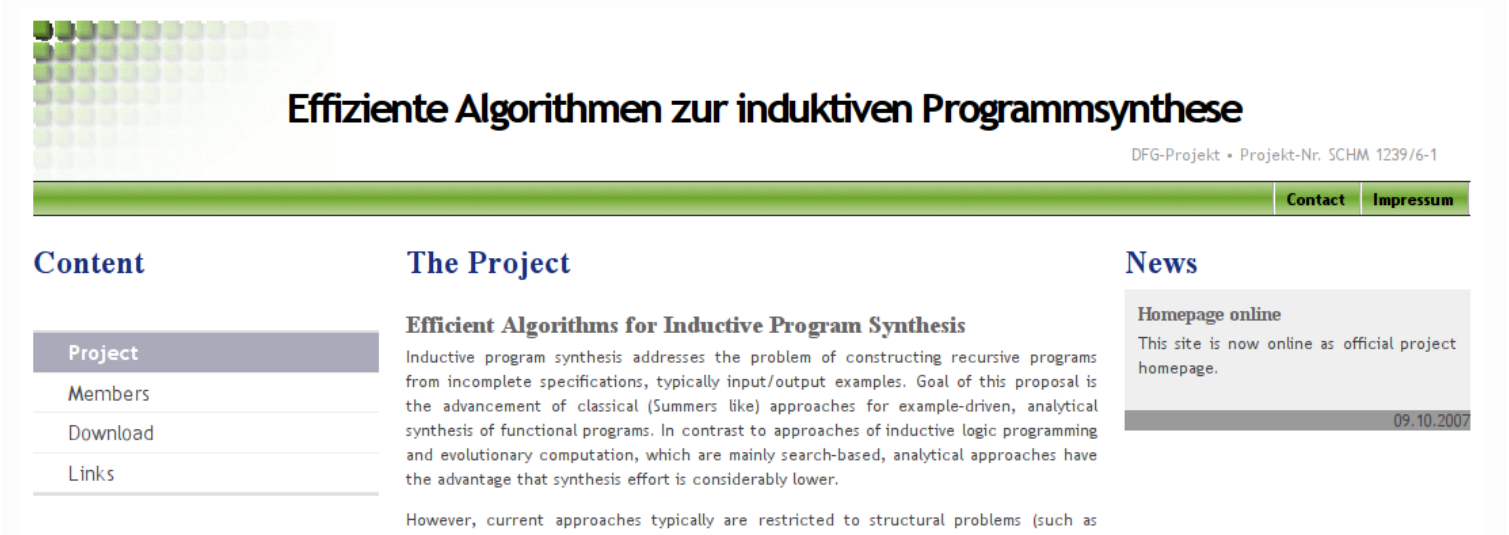
http://www.inductive-programming.org

- Einführung in IPS
- Systemüberblick
- Repositorium mit Testproblemen
- IPS nahe Publikationen
- EMailverteiler
- . . .

Herzlichen Dank!

Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen!

Unser Projekt



http://www.cogsys.wiai.uni-bamberg.de/effalip/

- Publikationen
- Downloads
- Links