Research Group
Functional-Logic Development
& Implementation Techniques

FLDIT

fldit-www.cs.tu-dortmund.de

Peter Padawitz
Hubert Wagner
Jens Lechner
Sebastian Venier

Logisch-algebraische Modellierung

Funktionale und regelbasierte Programmierung

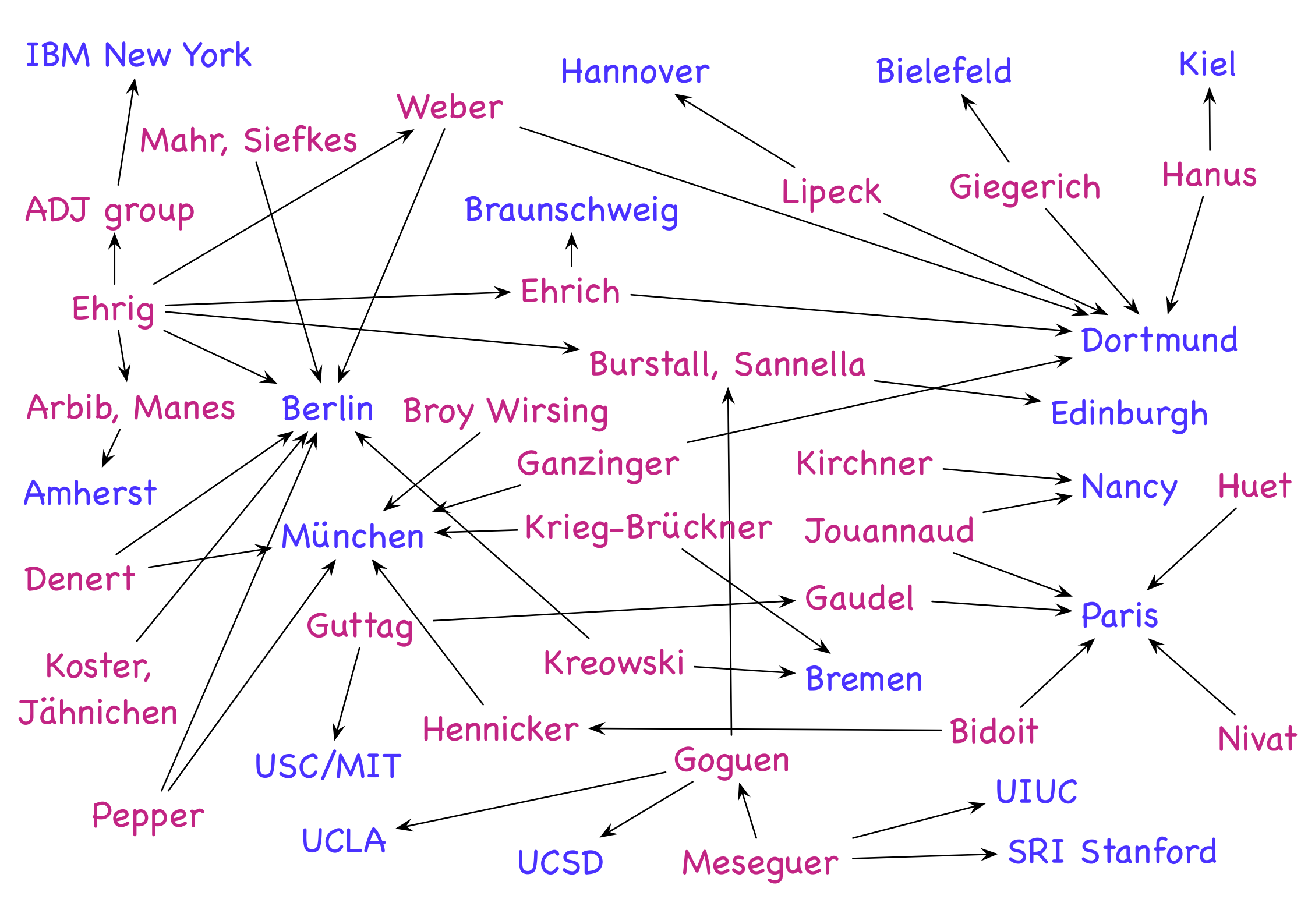
27. Juni 2011

Peter Padawitz

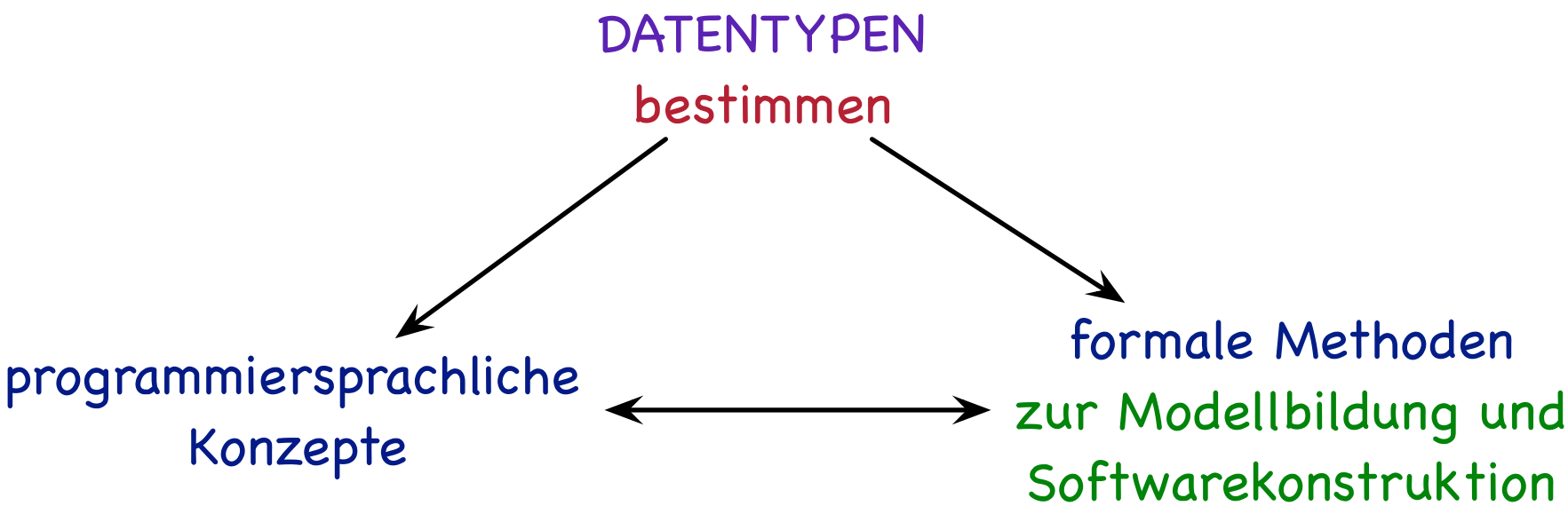
Werdegang

1972-1974	Mathematik-Studium an der TU Hannover
1974-1978	Informatik-Studium an der TU Berlin
1978-1983	WiMi an der TU Berlin
1983	Promotion (mit Auszeichnung)
1983	Forschungsaufenthalt an der Uni Nancy II
1983-1991	Akad. Rat an der Uni Passau
1987	Habilitation
seit 1992	Professor an der TU Dortmund
1998	Forschungsaufenthalt an der Univ. of California at San Diego

Netzwerk der Lehrer und Partner


IBM New York
Hannover
Bielefeld
Kiel
Weber
Mahr, Siefkes
Lipeck
Giegerich
Hanus
ADJ group
Braunschweig
Ehrig
Ehrich
Dortmund
Burstall, Sannella
Arbib, Manes
Berlin
Broy Wirsing
Edinburgh
Ganzinger
Kirchner
Nancy
Huet
Amherst
München
Krieg-Brückner
Jouannaud
Denert
Gaudel
Paris
Guttag
Koster,
Jähnichen
Kreowski
Bremen
Hennicker
Bidoit
Nivat
USC/MIT
Goguen
Pepper
UIUC
UCLA
UCSD
Meseguer
SRI Stanford

Modellierung und Programmierung
bestehen i.w. in der Auswahl adäquater
Datenstrukturen und darauf zugeschnittener
Programmstrukturen, kurz:
DATENTYPEN.



Der Zoo der Modellierungsansätze und Spezifikationsprachen

Boolesche Algebra

Horn-Logik

Fixpunkttheorie

Modal- und Temporallogik

Gleichungslogik

Prädikatenlogik

Automaten

reguläre Ausdrücke

Relationenalgebra

λ-Kalkül

Semiringtheorie

pre/postcondition-Kalküle

XML

Z

.....

Entity-Relationship-Modelle

UML

OCL

Lasst euch nicht verwirren
von der Vielfalt der Ansätze und auch nicht
von der von LV zu LV und Artikel zu Artikel
variierenden Benennung oder Darstellung!

Es gibt nämlich nur zwei Klassen von Modellen:

konstruktorbasierte und destruktorbasierte (verhaltens-, zustandsbasierte)

Listen, Bäume, Graphen	Automaten, Transitionssysteme
endliche (Multi-)Mengen	Flussgraphen
Relationen	Klassenhierarchien (UML)
Felder, Matrizen	attributierte Strukturen (XML)
endliche Datenstrukturen	unendliche Datenstrukturen
Grammatiken	Kripke-Strukturen
rekursive Datentypen	corekursive Datentypen

Die grundlegenden Modellkonstruktionen
kennt man aus der
Mengenlehre:

Produkt

$A1 \times \ldots \times An$

Summe
(disjunkte Vereinigung)

$A1 + \ldots + An$

Funktionsräume

$A \to B$

Bildung von
Quotienten

$A/\sim$

dient der Abstraktion

Bildung von
Unterstrukturen

$B \subset A$

dient der Restriktion

Eine Programmiersprache, Spezifikationssprache, Grammatik, ...
bildet eine konkrete Syntax.

Diese wird durch Übersetzung in die Sprache der mathematischen Logik,
also in Terme bzw. Formeln
zur abstrakten Syntax.

Die abstrakte Syntax beschreibt entweder
die Elemente des Modells selbst
(konstruktorbasierter Ansatz)

oder diejenigen Funktionen,
deren Anwendung auf ein Element des Modells
dessen "Verhalten" sichtbar macht ("Methoden", "Attribute")
(destruktorbasierter Ansatz)

Initiale und finale Modelle

Ein Term t mit Variablen x1,...,xn besteht aus Konstruktoren
und bezeichnet eine n-stellige Funktion
f : A1 x ... x An -> A.

Terme bilden ein initiales Modell.
Weitere konstruktorbasierte Modelle erhält man durch
Interpretation der Terme.

Ein Coterm t mit Covariablen x1,...,xn besteht aus Destruktoren,
bezeichnet eine Funktion
f : A -> A1 + ... + An
und entspricht einem Flussdiagramm mit n Ausgängen.

Coterme bezeichnen "Anfragen" an Objekte
destruktorbasierter Modelle.
Die Folgen möglicher Antworten auf alle Anfragen bilden
ein finales Modell.

Rekursive Datentypen benutzen
Produkt, Summe, Abstraktion und Restriktion,
um aus bekannten Datentypen neue zu bilden.

Sie werden definiert als Lösung von Gleichungen
zwischen Typausdrücken mit Mengenvariablen.

Fixpunktsätze liefern die Konstruktion der Lösungen
und damit die Definition rekursiver Datentypen.

Die Semantik einzelner Objekte,
rekursiver Funktionsbeschreibungen
und logischer Programme für Relationen
erhält man mit Hilfe ähnlicher Fixpunktsätze als
Lösung von Gleichungen.

```
blink = 0:1:blink
```
kleinste Lösung in blink: 0:1:0:1:0:1:...

```
nats(n) = n:nats(n+1)
```
kleinste Lösung in nats(5): 5:6:7:8:...

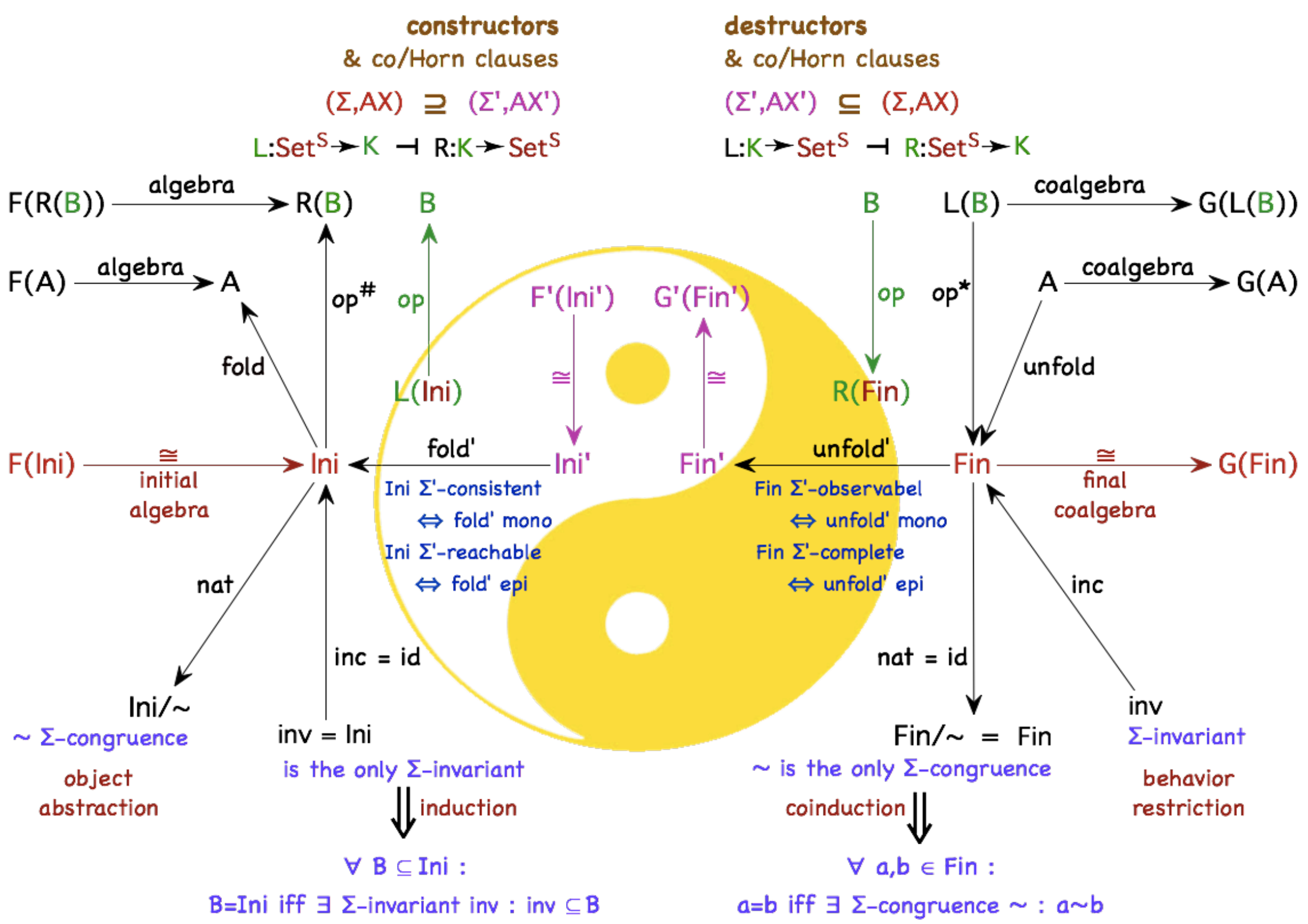
```
factorial(n) = if n == 1 then 1 else n*factorial(n-1)
```
kleinste Lösung in factorial: Fakultätsfunktion

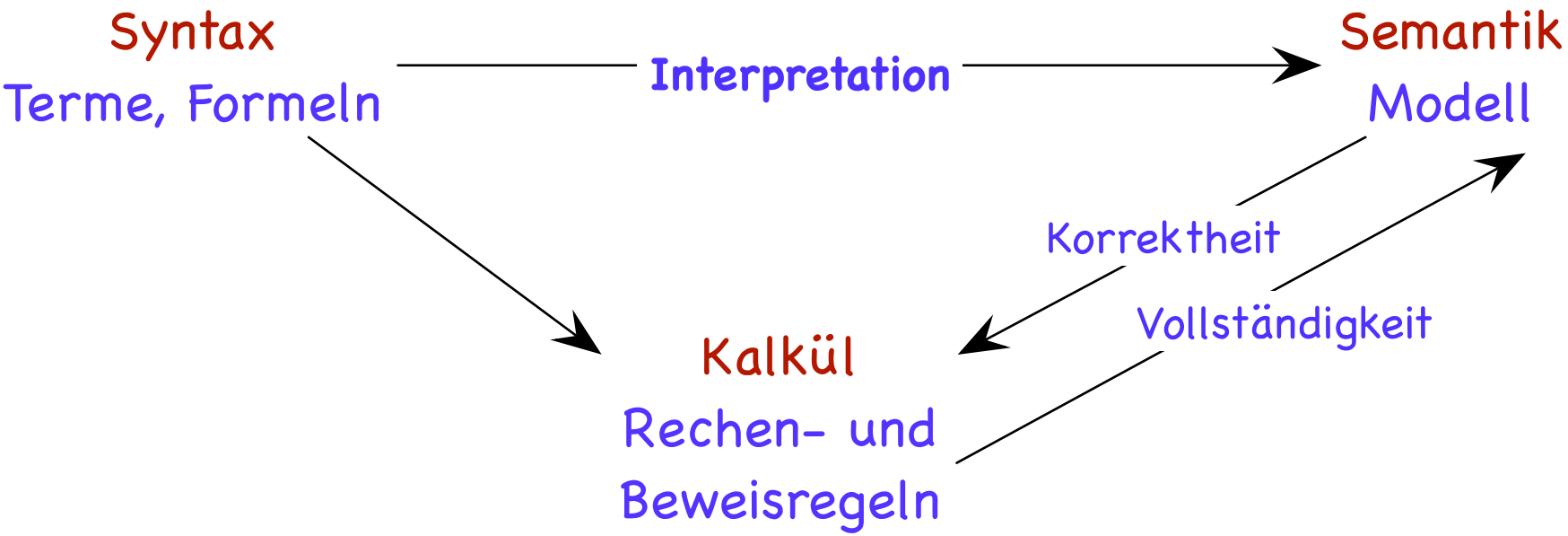
```
bintree(T) = empty + bintree(T) x T x bintree(T)
```
kleinste Lösung in bintree(int): Menge aller endlichen binären Bäume mit ganzzahligen Knoteneinträgen

größte Lösung in bintree(int): ... oder unendlichen ...

```
sorted(L) <==> length(L) <= 1 \/
                exists x,y,L': (L == x:y:L' /\ x <= y /\ sorted(y:L'))
```
kleinste Lösung in sorted: Menge aller sortierten Listen

```
unsorted(L) <==> length(L) > 1 /\
                  forall x,y,L': (L =/= x:y:L' \/ x > y \/ unsorted(y:L'))
```
größte Lösung in unsorted: Menge aller unsortierten Listen


constructors
& co/Horn clauses
(Σ,AX) ⊇ (Σ',AX')
L:Set^S → K ⊣ R:K → Set^S
destructors
& co/Horn clauses
(Σ',AX') ⊆ (Σ,AX)
L:K → Set^S ⊣ R:Set^S → K
F(R(B)) algebra R(B)
F(A) algebra A
op#
op
B
F'(Ini') ≅ L(Ini)
G'(Fin') ≅ R(Fin)
op
op*
B
L(B) coalgebra G(L(B))
A coalgebra G(A)
fold
unfold
F(Ini) ≅ Ini
initial algebra
fold'
Ini'
Fin'
unfold'
Fin ≅ G(Fin)
final coalgebra
Ini Σ'-consistent
⇔ fold' mono
Ini Σ'-reachable
⇔ fold' epi
Fin Σ'-observabel
⇔ unfold' mono
Fin Σ'-complete
⇔ unfold' epi
nat
inc = id
nat = id
inc
Ini/~
~ Σ-congruence
object abstraction
inv = Ini
is the only Σ-invariant
induction
Fin/~ = Fin
~ is the only Σ-congruence
coinduction
inv
Σ-invariant
behavior restriction
∀ B ⊆ Ini :
B=Ini iff ∃ Σ-invariant inv : inv ⊆ B
∀ a,b ∈ Fin :
a=b iff ∃ Σ-congruence ~ : a~b



p(t1,...,tn) <== φ	Hornaxiom (definiert kleinste Relation) Anwendung auf co-Horngoals
p(t1,...,tn) ==> φ	co-Hornaxiom (definiert größte Relation) Anwendung auf Horngoals
p(t1,...,tn) ==> φ	co-Horngoal (beweisbar mit Induktion) Anwendung auf Hornaxiome
p(t1,...,tn) <== φ	Horngoal (beweisbar mit Coinduktion) Anwendung auf co-Hornaxiome

Was bringt logisch-algebraische Modellierung?

Eine Sprache, die Datentypen präzise beschreibt und
Rechen-, Lösungs- und Beweismethoden zur Verfügung stellt,
die es erlauben, in dem Modell zu rechnen.

Eine Schnittstelle zwischen
benutzerfreundlichen/vorgegebenen/maschinenorientierten
Formaten/Sprachen und damit die Möglichkeit,
die Korrektheit von Übersetzungen
zwischen verschiedenen Sprachen zu beweisen,
so dass die Ergebnisse der Rechnungen im Modell
auch in den aus ihm heraus oder dort hinein kompilierten Sprachen
gelten.

Funktionales Programmieren

Mächtiges Typkonzept bewirkt die Erkennung der meisten semantischen Fehler während der Übersetzung

Polymorphe Typen, generische Funktionen und Funktionen höherer Ordnung ermöglichen ein Höchstmaß an Wiederverwendbarkeit und schneller Adaptierbarkeit an spezielle Anforderungen sowie benutzerfreundliche Test- und Verifikationsmethoden

Konstruktorbasierte Datentypen erlauben sehr komplexe Fallunterscheidungen mit Hilfe von Datenmustern

Lazy evaluation ermöglicht die direkte Berechnung von Gleichungslösungen wie in logischer/relationaler Programmierung (Prolog, Datalog, SQL)

Funktionales Programmieren

Quicksort-Algorithmus in Haskell

```
quicksort :: (a -> a -> Bool) -> [a] -> [a]

quicksort rel (a:s) = quicksort rel [b | b <- s, rel b a] ++ a:
                      quicksort rel [b | b <- s, rel a b]
quicksort _ s       = s

quicksort (<) [6,5,7,2,3,9,0]     ===>     [0,2,3,5,6,7,9]
```

Typ

Programm ist Gleichungssystem

Funktionales Programmieren

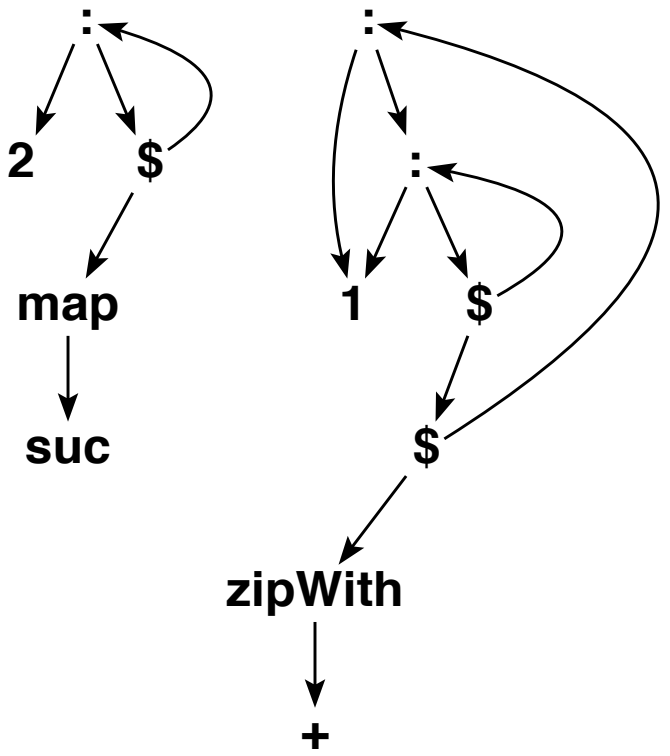
Neben der Gleichungslösung erlaubt lazy evaluation
das Rechnen mit unendlichen Strukturen (z.B. Prozessmodellen)

Primzahlgenerator

```
prims, nats :: [Int]
sieve       :: [Int] -> [Int]

prims = sieve nats
        where nats = 2:map (+1) nats
              sieve (p:s) = p:sieve [n | n <- s, n`mod`p /= 0]

take 11 prims      ===>      [2,3,5,7,11,13,17,19,23,29,31]
```



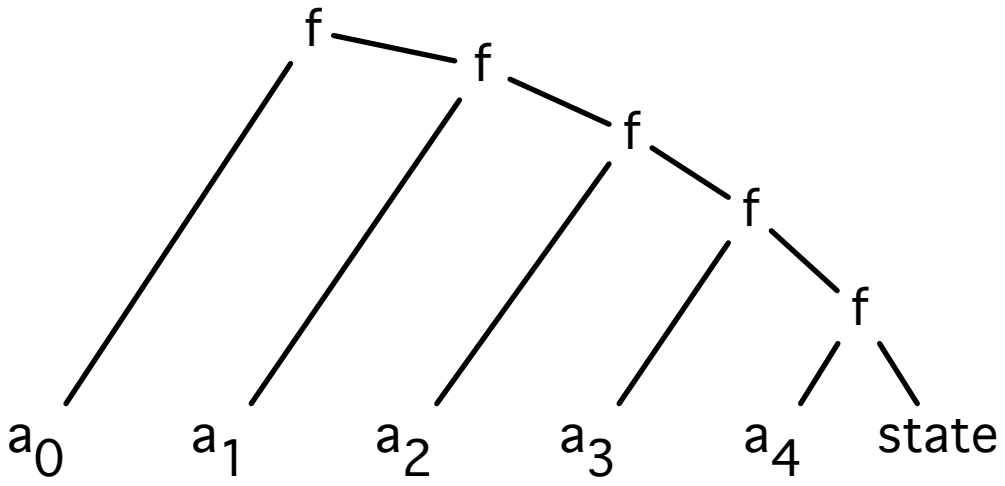
Fibonacci-Folge

```
fib :: [Int]

fib = 1:tfib where tfib = 1:zipWith (+) fib tfib

take 11 fib   ===>   [1,1,2,3,5,8,13,21,34,55,89]
```

Funktionales Programmieren



Listenfaltung (Iteration)

```
foldr :: (a -> state -> state) -> state -> [a] -> state
foldr f state []     = state
foldr f state (a:as) = f a (foldr f state as)
```

Horner-Schema

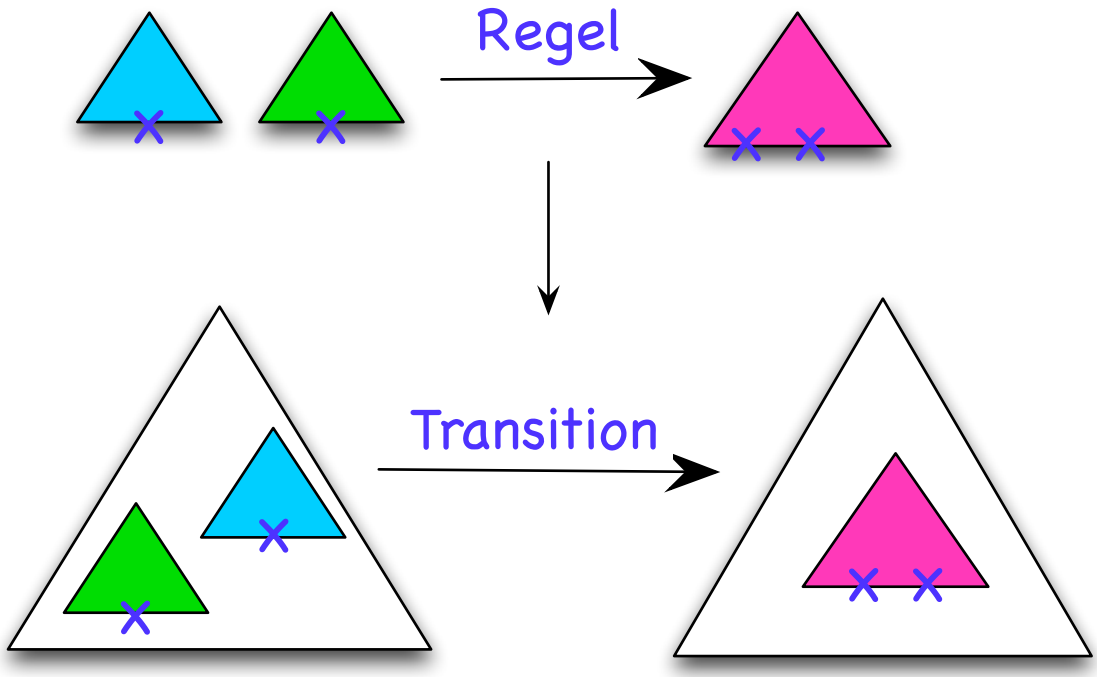
```
horner as x = foldr f (last as) (init as)
              where f a state = a+state*x
```

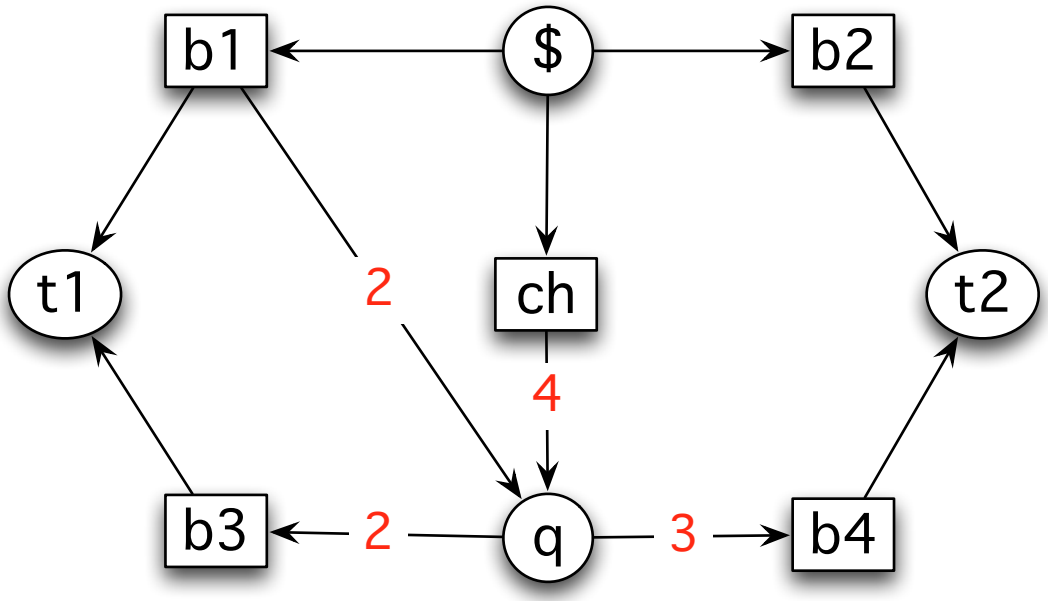
monadisch (imperativer Stil)

```
foldr :: (a -> state -> state) -> [a] -> IO state
foldr f []     = read
foldr f (a:as) = do state <- foldr f as
                    write (f a state)
```

Regelbasiertes Programmieren

Regeln beschreiben
nichtdeterministische Übergänge
zwischen strukturierten Zuständen





buy1 : $ ⟶ ticket1 q q

buy2 : $ ⟶ ticket2 q

buy3 : q q ⟶ ticket1

buy4 : q q q ⟶ ticket2

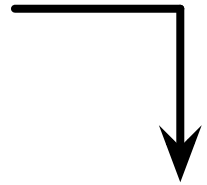
change : $ ⟶ q q q q

Regelbasierte Sprachen

Haskell, Python, Scala, F#

Maude, Curry, Elan,

O'Haskell, **Expander2**

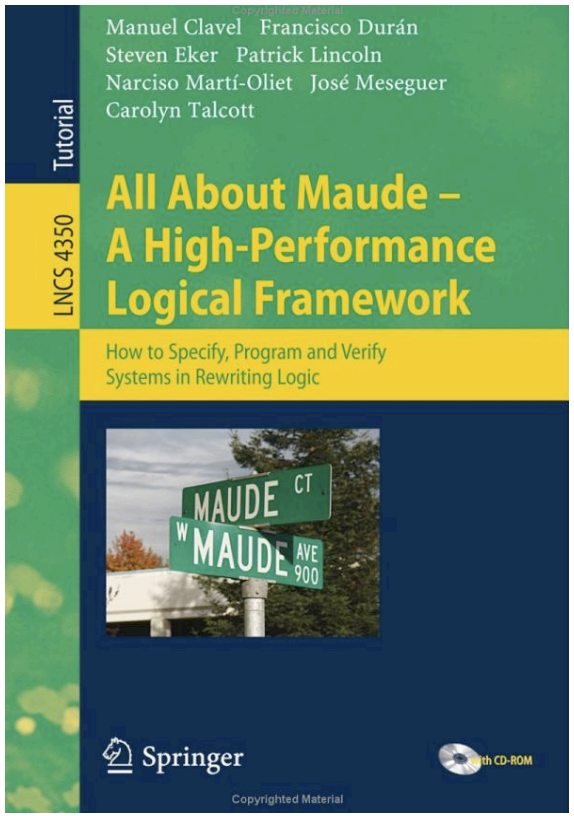


Seminar

Executable Specification Languages

Fachprojekte/Projektgruppen

Bachelor/Master/Diplomarbeiten



Lehrveranstaltungen

fldit-www.cs.tu-dortmund.de

Funktionale Programmierung - 2V+1Ü — WS 10/11

BA 3. Sem. Pflicht-LV

Diplom HS SPGs 1: Software-Konstruktion

+ 4: Algorithmen, Komplexität und formale Modelle

Übersetzerbau - 2V+1Ü — WS 10/11

BA 5. Sem. Wahlpflicht Software (alternativ: Softwarekonstruktion)

Diplom HS Pflicht

Einführung in den logisch-algebraischen Systementwurf - 3 SWS

BA 5. od. 6. Sem. Wahl-LV — WS 10/11

Diplom HS SPGs 1 + 4 + 5: Sicherheit und Verifikation

Proseminar Kategorientheoretische Grundlagen - 2 SWS

BA 5. od. 6. Sem. — SS 11

Lehrveranstaltungen

fldit-www.cs.tu-dortmund.de

Logisch-algebraischer Systementwurf - 4 SWS — SS 11

MA 2. od. 3. Sem. im Schwerpunkt Software

Diplom HS SPGs 1 + 4 + 5

Funktionales und regelbasiertes Programmieren - 4 SWS — SS 11

MA 2. od. 3. Sem. im Schwerpunkt Software

Diplom HS SPGs 1 + 4 + 5

Seminar Executable Specification Languages - 2 SWS

MA ab 2. Sem. — jedes Semester

Diplom HS SPGs 1 + 4 + 5

Fachprojekt Rapid Prototyping mit Expander2 - 4 SWS — ~ SS 12

BA 6. Sem.

Baum- und graphbasierte Übersetzungs- und Analysetechniken - 4 SWS

MA 2. od. 3. Sem. im Schwerpunkt Software — ~ SS 12

Diplom HS SPGs 1 + 4 + 5