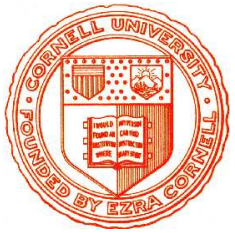
Automatisierte Logik und Programmierung II

Teil IV



Automatisierte Programmierung

1. Grundkonzepte, Paradigmen & Strategien

2. Wissensbasierte Programmentwicklung

– Divide & Conquer, Globalsuche,
Lokalsuche, Problemreduktionsgeneratoren

3. Korrektheitserhaltende Optimierungen

Wozu Automatisierte Programmierung?

- **Softwareproduktion hat viele Probleme**
  - **Zeitaufwendig und teuer**
    - Entwurf und Implementierung fokussiert auf Modellierungs- und Programmiersprachen anstatt auf Eigenschaften des Problembereichs
    - Implementierung meist “von Hand” und ad hoc
    - Einbeziehung der Endanwender zu spät
  - **Zu viele Fehler im Endprodukt**
    - Logischer Zusammenhang zur Aufgabenstellung selten erkennbar
    - Begründung für Korrektheit des Programms fehlt meistens
- **Logische Synthese von Programmen hilft**
  - (Teil-)Automatisierung der Konstruktion von Algorithmen
  - Logisches Fundament erhöht Zuverlässigkeit des erzeugten Programms
  - Automatisierung verringert Entwicklungszeit und -kosten und ermöglicht frühzeitige Validierung durch Endanwender

Programmsynthese: grundsätzliches Vorgehen

Erzeuge korrekte ausführbare Programme aus Spezifikationen

1. **Erstellen einer formalen Spezifikation**
   a) Benötigt Formalisierung des Anwendungsbereichs als **Objekttheorie**
      · Welche Begriffe werden benutzt und was bedeuten sie?
      · Welche mathematischen Gesetze gelten für diese Begriffe?
      } vgl. §15
   b) Präzisierung der Problemstellung in spezifischem Formalismus
2. **Entwurf eines läuffähigen, korrekten Algorithmus**
   – Synthesestrategie generiert Basisversion und Korrektheitsgarantien
   – Synthesesystem muß durch erfahrene Benutzer gesteuert werden
3. **Erzeugung eines effizienten, korrekten Programms**
   a) Benutzergesteuerte Optimierungstechniken verbessern Algorithmus
   b) Übertragung in Zielsprache ermöglicht weitere Optimierungen
   – System garantiert jeweils die Korrekheit der Optimierungen

Programmsynthese am Beispiel: Costas-Arrays

Costas Array der Größe n:

– Permutation von $\{1..n\}$ ohne Duplikate in Zeilen der Differenzentafel

– Hilfreich für Erzeugung leicht decodierbarer Radar- und Sonarsignale

2	4	1	6	5	3
-2	3	-5	1	2	
1	-2	-4	3		
-4	-1	-2			
-3	1				
-1					

Costas Array der Ordnung 6 und seine Differenzentafel

Ziel: Berechnung aller Costas Arrays der Größe n

Costas-Arrays (1): formale Spezifikation

Für $n \geq 1$ berechne alle Permutationen von $\{1..n\}$ ohne Duplikate in Zeilen der Differenzentafel

- **Formalisierung vorkommender Begriffe**
  - `perm(p,`S`)` $\equiv$ `nodups(p)` $\wedge$ `range(p)=`S
  - `nodups(`L`)` $\equiv$ $\forall$`i`$\neq$`j`$\leq$`|p|.` L`[i]`$\neq$$L$`[j]`
  - `dtrow(p,`j`)` $\equiv$ `[p[i]-p[i+`j`]|i`$\in$`[1..length(p)-`j`]]`
- **Aufstellen mathematischer Gesetze**
  - Lemmas zu Eigenschaften von `perm`, `nodups`, `dtrow`
- **Präzisierung der Problemstellung**

```
FUNCTION Costas (n:ℤ) WHERE n≥1
   RETURNS {p:Seq(ℤ) | perm(p,{1..n})
                     ∧ ∀j<|p|.nodups(dtrow(p,j))}
```

Costas-Arrays (2): Erzeugung des Basisalgorithmus

- **Algorithmische Struktur ist Globalsuche**

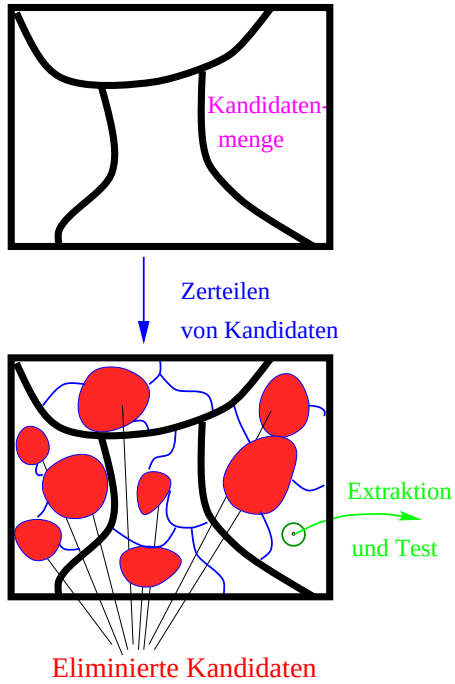
  Standardstruktur zum Durchsuchen von Lösungsräumen

  – Codierung von Kandidatenmengen

  – Wiederholtes Aufteilen und Filtern

  auf Basis von Repräsentanten

  – Extraktion konkreter Lösungen aus Repräsentanten



- **Algorithmus wird aus Schema erzeugt**

```
FUNCTION Costas (n:ℤ) WHERE n≥1
   RETURNS {p:Seq(ℤ) | perm(p,{1..n})
                     ∧ ∀j<|p|.nodups(dtrow(p,j))}
= let rec aux (n:ℤ,s:Seq(ℤ))
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
  =  {p|p∈{s} ∧ perm(p,{1..n}) ∧ ∀j<n.nodups(dtrow(p,j))}
   ∪ ⋃{aux(n,t)|t∈{s·i|i∈{1..n}}
                ∧ nodups(t)∧ ∀j<|t|.nodups(dtrow(t,j))}
  in aux(n,[])
```

Costas-Arrays (3): Optimierung
Algorithmisch-logische Vereinfachung

```
FUNCTION Costas (n:ℤ) ...
= let rec aux (n:ℤ,s:Seq(ℤ))
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
  =  {p|p∈{s} ∧ perm(p,{1..n}) ∧ ∀j<n.nodups(dtrow(p,j))}
    ∪ ⋃{aux(n,t)|t∈{s·i|i∈{1..n}}
                  ∧ nodups(t)∧ ∀j<|t|.nodups(dtrow(t,j))}
  in aux(n,[])
```

```
FUNCTION Costas (n:ℤ) ...
= let rec aux (n:ℤ,s:Seq(ℤ))
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
  = if {1..n}\s=∅ then {s} else ∅
     ∪ ⋃{aux(n,s·i)|i∈{1..n}\s
                  ∧ ∀j<|s|.(s[|s·i|-j]-i)∉dtrow(s,j)}
  in aux(n,[])
```

Costas-Arrays (3): Optimierung
Inkrementelle Berechnung von Teilausdrücken

```
FUNCTION Costas (n:ℤ) ...
= let rec aux (n:ℤ,s:Seq(ℤ))
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
  = if {1..n}\s=∅ then {s} else ∅
     ∪ ⋃{aux(n,s·i)|i∈{1..n}\s
                    ∧ ∀j<|s|.(s[|s·i|-j]-i)∉dtrow(s,j)}
  in aux(n,[])
```

```
FUNCTION Costas (n:ℤ) ...
= let rec aux(n:ℤ,s:Seq(ℤ),pool:Seq(ℤ),ssize:ℤ)
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
      ∧ pool={1..n}\s ∧ ssize=|s|
  = if pool=∅ then {s} else ∅
     ∪ ⋃{aux(n, s·i,pool\{i},ssize+1)|i∈pool
                    ∧ ∀j<ssize.(s[ssize+1-j]-i)∉dtrow(s,j)}
  in aux(n,[],{1..n},0)
```

Costas-Arrays (3): Optimierung
Fallanalyse

```
FUNCTION Costas (n:ℤ) ...
= let rec aux(n:ℤ,s:Seq(ℤ),pool:Seq(ℤ),ssize:ℤ)
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
      ∧ pool={1..n}\s ∧ ssize=|s|
  = if pool=∅ then {s} else ∅
     ∪ ⋃{aux(n, s·i,pool\{i},ssize+1)|i∈pool
                  ∧ ∀j<ssize.(s[ssize+1-j]-i)∉dtrow(s,j)}
  in aux(n,[],{1..n},0)
```

```
FUNCTION Costas (n:ℤ) ...
= let rec aux(n:ℤ,s:Seq(ℤ),pool:Seq(ℤ),ssize:ℤ)
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
      ∧ pool={1..n}\s ∧ ssize=|s|
  = if pool=∅  then {s}
     else ⋃{aux(n, s·i,pool\{i},ssize+1)|i∈pool
                  ∧ ∀j<ssize.(s[ssize+1-j]-i)∉dtrow(s,j)}
  in aux(n,[],{1..n},0)
```

Costas-Arrays (3): Optimierung
Wahl der Implementierung abstrakter Datentypen

```
FUNCTION Costas (n:ℤ) WHERE n≥1 ...
= let rec aux(n:ℤ,s:Seq(ℤ),pool:Seq(ℤ),ssize:ℤ)
  WHERE nodups(s) ∧ ∀j<|s|.nodups(dtrow(s,j))
      ∧ pool={1..n}\s ∧ ssize=|s|
  = if pool=∅  then {s}
     else ⋃{aux(n, s·i,pool\{i},ssize+1) | i∈pool
                  ∧ ∀j<ssize.(s[ssize+1-j]-i)∉dtrow(s,j)}
  in aux(n,[],{1..n},0)
```

`n:ℤ` ↦ Standardimplementierung positiver ganzer Zahlen

`ssize:ℤ` ↦ Standardimplementierung positiver ganzer Zahlen

`s:Seq(ℤ)`, Elemente werden hinten angehängt ↦ umgekehrt verkettete Liste

`pool:Set(ℤ)`: Elemente werden aus fester Menge entnommen ↦ Bitvektor

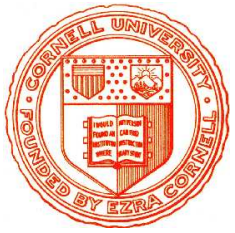
Letzter Schritt vor Übertragung in konkrete Programmiersprache

Ansätze zur Entwicklung von Syntheseverfahren

- **Anwendung generischer Inferenztechniken**
  - **Beweise als Programme**
    - Automatischer Beweiser + Programmextraktion aus Beweisen
  - **Transformation von Formeln**
    - Rewrite-Techniken + Extraktion von Programmen aus Formeln
  - Gut zur Illustration der Prinzipien ("Synthese im Kleinen")
  - Konstruktion aufwendigerer Algorithmen verlangt Spezialstrategien
- **Wissensbasierte Syntheseverfahren**
  - Wissen über algorithmische Grundstrukturen formalisiert als "**Algorithmentheorien**" (Struktur + Korrektheitsaxiome)
  - Strategien verwenden Wissen zur Erzeugung effizienter Algorithmen
  - Ziel ist Unterstützung des Programmierers (statt Ersetzung)
  - Aufwendigere Vorarbeiten aber erfolgreich in der "Praxis"

Automatisierte Logik und Programmierung

Einheit 17



Syntheseparadigmen & -strategien

1. Grundkonzepte
2. Synthese im Kleinen
   – Beweise als Programme
   – Synthese durch Transformationen
3. Wissensbasierte Programmentwicklung

Komponenten einer Programmsynthese

Erzeuge korrekte ausführbare Programme aus Spezifikationen

- **Formale Spezifikation als Ausgangspunkt**
  - Beschreibung von Anwendungsbereich und Problemstellung in (umfangreicher, "natürlicher") formaler Spezifikationssprache
  - Objekttheorie formalisiert Eigenschaften neuer Anwendungskonzepte
- **Automatische Algorithmensynthese**
  - Syntheseparadigmen: grundsätzliche Vorgehensweisen
    Theoretische Begründung der Korrektheit erzeugter Algorithmen
  - Synthesestrategien: Verfahren zur Steuerung der Synthese
    Dokumentation getroffene Entscheidungen durch Trace der Strategie
- **Optimierung und Datentypverfeinerung**
  - Verbesserung des erzeugten Basisalgorithmus
  - Wahl geeigneter Implementierungen vorkommender Datentypen
  - Sprachabhängige Optimierung bei Übertragung in Programmiersprache

Formalismus zur Spezifikation von Problemen

- **Programme berechnen im Endeffekt Funktionen**
  - Aus welchem Datentyp stammen die Eingaben? Domain D
  - Zu welchem Datentyp gehören die Ausgaben? Range R
  - Gibt es Beschränkungen an zulässige Eingaben? Input-Bedingung I
  - Was ist der Zusammenhang zwischen Ein- und Ausgaben? Output-Bedingung O
  - Eine **formale Spezifikation** ist ein Quadrupel $spec = (D, R, I, O)$, wobei D und R Datentypen, I Prädikat über D, O Prädikat über $D \times R$
- **Syntaktische Notation abhängig von Aufgabenstellung**

  Erhöhte Lesbarkeit durch Schlüsselwörter und instantiierte Form von I, O
  - Bestimmung einer möglichen Lösung

    `FUNCTION` $f(x{:}D){:}R$ `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$
  - Bestimmung aller möglichen Lösungen

    `FUNCTION` $f(x{:}D)$ `WHERE` $I[x]$ `RETURNS` $\{y{:}R \mid O[x,y]\}$

Programme und Korrektheit

- **Programm = Spezifikation + Algorithmus**
  - Algorithmen (Programmkörper) sind berechenbare (partielle) Funktionen auf $D \not\to R$, die auf allen zulässigen Eingaben definiert sind
  - Eine **formales Programm** ist ein 5-Tupel $prog = (D, R, I, O, body)$ wobei (D, R, I, O) formale Spezifikation, $body{:}D \not\to R$ berechenbar
  - Syntaktische Notation erlaubt Bezug auf Funktionsnamen in $body$

    $\texttt{FUNCTION}\ f(x{:}D){:}R\ \texttt{WHERE}\ I[x]\ \texttt{RETURNS}\ y\ \texttt{SUCH THAT}\ O[x,y] \equiv body[f,x]$

    $\texttt{FUNCTION}\ f(x{:}D)\ \texttt{WHERE}\ I[x]\ \texttt{RETURNS}\ \{y{:}R \mid O[x,y]\} \equiv body[f,x]$
- **Korrektheit von Programmen**
  - $prog$ **ist korrekt**, falls $\forall x : D\,.\ I[x] \Rightarrow O[x, body(x)]$
- **Syntheseziel: Erfüllbarkeit von Spezifikationen**
  - $spec$ **ist erfüllbar** (synthetisierbar), falls es eine Funktion $body : D \not\to R$ gibt, so daß $prog{=}(spec, body)$ korrekt ist

Ansätze zur Programmsynthese

- **KI-Vision: Automatisches Programmieren**
  - Intelligenter Agent ersetzt menschlichen Programmierer
  - Ziel ist vollautomatische Erzeugung von Programmen aus Spezifikationen
  - Strategien konzentrieren sich auf logisch-deduktive Verfahren
    Forschungen lieferten wichtige theoretische Grundlagen
  - Verfahren erzeugen oft nur einfache Algorithmen $\mapsto$ Synthese im Kleinen
- **Software-Engineering: Programmierunterstützung**
  - Benutzergesteuerte Erzeugung von Programmen mit Korrektheitsgarantie
    Synthesewerkzeug unterstützt den menschlichen Programmierer
  - Strategien verwenden symbolisch-algebraische Techniken und
    theoretische Grundlagen aus der Deduktion als Fundament
    Forschung liefert Formalisierung von Programmierwissen und -methoden
  - Verfahren liefern praktisch wertvolle Algorithmen
    $\mapsto$ Wissensbasierte Programmentwicklung

Synthese im Kleinen

Anwendung generischer Inferenzmechanismen

- **Beweise als Programme**
  - Extraktion aus konstruktivem Beweis eines Theorems
  - Fokus liegt auf automatischen Beweisverfahren
- **Synthese durch Transformationen**
  - Äquivalenzumformungen in ausführbare, meist rekursive Form
  - Fokus liegt auf algebraisch motivierten Rewrite-Techniken
- **Unterschiedliche theoretische Fundamente**
  - Problemrepräsentation, Lösungsmethodik, Extraktion des Algorithmus
  - Gegenseitige Simulation prinzipiell möglich
- **Nur gut zur Illustration von Prinzipien**
  - Konstruktion aufwendiger Algorithmen verlangt Spezialstrategien

Beweise als Programme

- **Beweise Erfüllbarkeit einer Spezifikation**

  `FUNCTION` f`(`x`:`D`):`R `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$

  – Erzeuge **Spezifikationstheorem**: $\forall x : D\,.\,\exists y : R\,.\; I[x] \Rightarrow O[x,y]$

  – Suche formalen Beweis in konstruktivem logischen Kalkül

  – Extrahiere aus Beweis einen Algorithmus zur Berechnung von y aus x

- **Rechtfertigung durch Curry Howard Isomorphismus**

  – Konstruktiver Beweis zeigt, wie Ausgabe aus Eingabe bestimmt wird

  – Implizit enthaltenes funktionales Programm ist garantiert korrekt

- **Forschungsschwerpunkte**

  – Ausdrucksstarke Kalküle

  – Effiziente Beweisstrategien und Beweisplaner (Induktion)

  – Effiziente Extraktionsmechanismen (Algorithmen ohne Beweisballast)

Formaler Beweis: Integerquadratwurzel

```
⊢ ∀n:ℕ. ∃r:ℕ. r²≤n<(r+1)²
BY allR
  n:ℕ
  ⊢ ∃r:ℕ. r²≤n<(r+1)²
  BY NatInd 1
  .....basecase.....
      ⊢ ∃r:ℕ. r²≤0<(r+1)²
  ✓   BY existsR ⌈0⌉ THEN Auto
  .....upcase.....
      i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²
      ⊢ ∃r:ℕ. r²≤i<(r+1)²
      BY Decide ⌈(r+1)²≤i⌉ THEN Auto
      .....Case 1.....
          i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²,  (r+1)²≤i
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      ✓   BY existsR ⌈r+1⌉ THEN Auto'
      .....Case 2.....
          i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²,  ¬((r+1)²≤i)
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      ✓   BY existsR ⌈r⌉ THEN Auto
```

Beweisterm: Integerquadratwurzel

- **Mit Korrektheitsbeweisinformation**

```
let rec sqrt n
= if n=0 then <0,<Ax,Ax>>
  else let <r,%1>  = sqrt (n-1)
       in  if (r+1)²≤n  then <r+1,<Ax,Ax>>ᵢ
           else <r,<Ax,Ax>>
```

- **Nach Projektion und Optimierung**

```
let rec sqrt n
= if n=0 then 0
  else let r = sqrt (n-1)
       in  if (r+1)²≤n  then r+1
           else r
```

$\lfloor\sqrt{n}\rfloor$ – Synthese eines effizienteren Algorithmus durch effizienzorientierte Beweisführung

```
⊢ ∀n:ℕ. ∃r:ℕ. r²≤n<(r+1)²
BY allR
  n:ℕ
  ⊢ ∃r:ℕ. r²≤n<(r+1)²
  BY NatInd4 1
  .....basecase.....
      ⊢ ∃r:ℕ. r²≤0<(r+1)²
  ✓   BY existsR ⌈0⌉ THEN Auto
  .....upcase.....
      i:ℕ,  r:ℕ,  r²≤i÷4<(r+1)²
      ⊢ ∃r:ℕ. r²≤i<(r+1)²
      BY Decide ⌈((2*r)+1)²≤i⌉ THEN Auto
      .....Case 1.....
          i:ℕ,  r:ℕ,  r²≤i÷4<(r+1)²,  ((2*r)+1)²≤i
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      ✓   BY existsR ⌈(2*r)+1⌉ THEN Auto'
      .....Case 2.....
          i:ℕ,  r:ℕ,  r²≤i÷4<(r+1)²,  ¬(((2*r)+1)²≤i)
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      ✓   BY existsR ⌈2*r⌉ THEN Auto
```

```
let rec sqrt n
= if n=0 then 0
  else let r = sqrt (n÷4)
       in  if (2*r+1)²≤n  then 2*r+1
           else 2*r
```

Beweise als Programme: Eigenschaften

- **Mathematisch elegante Konstruktionsmethode**
  - Extrahiere Beweisterm t aus vollständigem Beweis für
    $$\forall x : D\,.\,\exists y : R\,.\; I[x] \Rightarrow O[x, y]$$
  - Konstruiere den Programmkörper λ`x.fst(`t`(x))`
  - Optimiere durch Reduktion und Elimination überflüssiger Information
- **Korrektheitsgarantien aber geringe Effizienz**
  - Effizienz stark abhängig von Beweismethodik
  - Gewisse Effizienzsteigerungen durch verbesserte Extraktionsverfahren
- **Beweisschritte zu atomar (Assemblerniveau)**
  - Beweise für komplexe Programme interaktiv kaum durchführbar
  - Automatisierung sehr schwierig:

    Analyse der Formel, Unifikation, Suche nach geeigneten Induktionen

Nur praktikabel in Kombination mit Definitionen und Spezialtaktiken

Synthese durch Transformation

- **Transformiere in effektiv ausführbare Formel**
  - Spezifikation ist ineffektive Formel
  - Transformationen verbessern algorithmisches Verhalten der Formel
  - Vorwärtsschließen ohne konkret vorgegebenes Ziel
- **Verwende Spezifikation als (logisches) Initialprogramm**

  FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y] \equiv \boldsymbol{O[x,y]}$
  - Transformiere Programmkörper in äquivalente Formel der Gestalt:

    FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y] \equiv \boldsymbol{O'[x,y,f]}$

    die außer f nur noch berechenbare Prädikate enthält
  - Extrahiere Programm aus Formel oder interpretiere als Logik-Programm
- **Forschungsschwerpunkte**
  - Leistungsfähige Transformationsregeln
  - Effiziente Rewrite Techniken und Heuristiken für Vorwärtsinferenz

Einfache transformationsbasierte Synthese

- **Anwendung bedingter Ersetzungsregeln der Form**

  $$\forall z{:}T\,.\;\; B[z] \;\Rightarrow\; (\;\; Q[z] \;\Leftrightarrow\; Q'[z] \;\;)$$

  "*Ersetze Vorkommen von $Q[z]$ durch $Q'[z]$, falls $B[z]$ erfüllt ist*"

  – Regeln sind Äquivalenzen oder Verfeinerungen (Implikationen)
- **Regeln ergeben sich aus**
  – Lemmata der Wissensbasis, elementare Tautologien und Abstraktionen
  – Dynamisch erzeugte Definitionen und Kombinationsformen
- **Mechanismus basiert auf Vorwärtsinferenz**
  – Ziel ist bestimmte Struktur der Formel zu erreichen
  – Starke heuristische Steuerung notwendig
- **Gleichwertig zum Prinzip Beweise als Programme**
  – Transformationen durch Beweise mit Gleichheitslemmata simulierbar
  – Beweisregeln können als Rewrite-Regeln beschrieben werden
  – Nicht skalierend: Suchraum explodiert bei nichttrivialen Problemen
  – **Inferenzniveau zu niedrig** – ignoriert bekanntes Programmierwissen

Wissensverarbeitung in der Programmsynthese

Zielgerichtete Entwicklung guter Algorithmen

- **Synthese im Kleinen ist zu allgemein**
  - – Fokus auf Logik statt auf Programmierung
  - – Steuerung durch “normale” Programmier kaum möglich
  - – Keine echte Unterstützung bei der Entwicklung von Programmen
- **Programmiermethodik verwendet Wissen**
  - – Welche grundsätzlichen Algorithmenstrukturen gibt es?
  - – Welche Algorithmenstrukturen sind für ein Problem geeignet?
- **Synthese sollte formales Programmierwissen verarbeiten**
  - – Umsetzung von Programmiermethodik in Entwurfsstrategien
  - – Schematisierung von algorithmischer Grundstrukturen
  - – Axiome für Korrektheit des schematischen Algorithmus

Aufwendige Theoretische Grundlagenarbeit entlastet Syntheseprozeß

Wissensbasierte Programmentwicklung: Grundidee

- **Erzeuge Algorithmen in einem Schritt**
  - – Anpassung eines schematischen Algorithmus an eine Problemstellung
  - – Aus historischer Sicht: Anwendung einer High-Level Transformation
- **Vorgehen formalisiert systematische Programmierung**

  Gegeben sei die Spezifikation

  `FUNCTION` f`(`x`:`D`):`R `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$
  - – Wähle eine algorithmische Grundstruktur für die Lösung
  - – Bestimme Komponenten eines schematischen Grundalgorithmus
  - – Prüfe, ob Komponenten die Korrektheitsaxiome des Schemas erfüllen
  - – Instantiiere Algorithmenschema
- **Forschungsschwerpunkte**
  - – Analyse der allgemeinen Struktur einer Klasse von Algorithmen
  - – Schematisierung durch Komponenten und Korrektheitsaxiome
  - – Techniken zur Verfeinerung von Standardstrukturen

Globalsuch Synthese: Costas-Arrays Problem

- **Problemspezifikation**

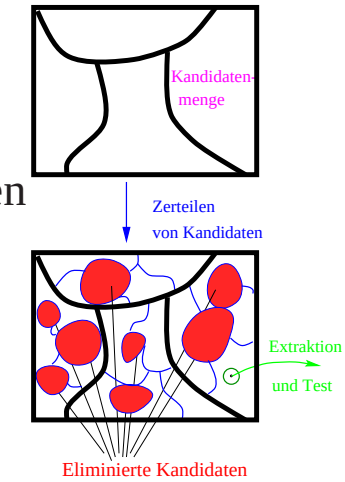
  FUNCTION Costas (n:$\mathbb{Z}$) WHERE n$\geq$1
  RETURNS {p:Seq($\mathbb{Z}$) | perm(p,{1..n}) $\wedge$ $\forall$j<|p|.nodups(dtrow(p,j))}

- **Grundstruktur von Globalsuch-Algorithmen**

  FUNCTION $f(x{:}D)$ WHERE $I[x]$ RETURNS $\{y{:}R \mid O[x,y]\}$

  $\equiv$ letrec $f_{gs}(x,s) = \{z \mid z \in ext[s] \wedge O[x,z]\} \cup \bigcup\{f_{gs}(x,t) \mid t \in split[x,s] \wedge \Phi[x,t]\}$
  in $f_{gs}(x, s_0(x))$

- **Bedeutung der Komponenten**
  - s: Deskriptor für Mengen von Lösungskandidaten
  - $ext(s)$: Direkte Extraktion von Lösungskandidaten z aus Deskriptoren
  - $O(x,z)$: Ausgabebedingung, verwendet zur endgültigen Selektion
  - $split(x,s)$: Rekursive Aufteilung von Kandidatenmengen
  - $\Phi(x,s)$: Filter zur Elimination unnötiger Deskriptoren
  - $s_0(x)$: Initialdeskriptor für Eingabe x



- **Algorithmus nach Bestimmung der Komponenten**

  FUNCTION Costas (n:$\mathbb{Z}$) WHERE n$\geq$1 RETURNS {p:Seq($\mathbb{Z}$) | perm(p,{1..n}) $\wedge$ ..}

  $\equiv$ letrec f_{gs}(n,s)
  = {p | p$\in${s} $\wedge$ perm(p,{1..n}) $\wedge$ $\forall$j<n. nodups(dtrow(p,j))}
  $\cup \bigcup\{f_{gs}(x,t) \mid t \in \{s{\cdot}i \mid i \in \{1..n\}\} \wedge$ nodups(t) $\wedge \forall$j<|t|.nodups(dtrow(t,j))$\}$
  in f_{gs}(n,[])

Divide & Conquer Synthese: Sortieralgorithmen

- **Problemspezifikation**

  FUNCTION sort(L:Seq($\mathbb{Z}$)):Seq($\mathbb{Z}$) RETURNS S
  SUCH THAT rearranges(L,S) $\wedge$ ordered(S)

- **Grundstruktur von Divide & Conquer Algorithmen**

  FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$
  $\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else (*Compose* $\circ\, g \times f \circ$ *Decompose*) (x)

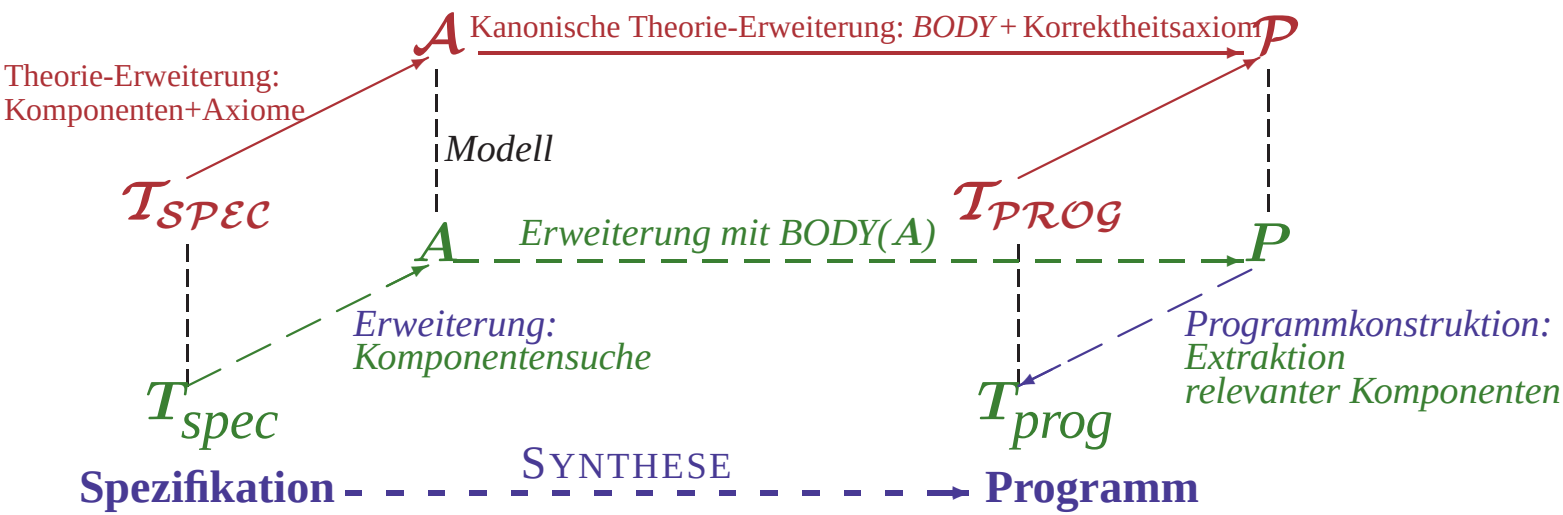
- **Komponenten für einen Sortieralgorithmus**

  primitive $\equiv$ λL. null?(L)
  Directly-solve $\equiv$ λL. []
  Decompose $\equiv$ λL. let a=L[|L|/2] in ($L_{<a}$, $L_{=a}$, $L_{>a}$) ($L_{<a} \equiv$ [x|x$\in$L$\wedge$x<a])
  g $\equiv$ sort $\times$ λS.S
  Compose $\equiv$ $\lambda S_1, S_2, S_3.\ S_1 \circ S_2 \circ S_3$

- **Instantiierter Algorithmus**

  FUNCTION sort(L:Seq($\mathbb{Z}$)):Seq($\mathbb{Z}$) RETURNS S
  SUCH THAT rearranges(L,S) $\wedge$ ordered(S)
  $\equiv$ if null?(L) then []
  else let a=L[|L|/2]
  in sort([x|x$\in$L$\wedge$x<a]) $\circ$ [x|x$\in$L$\wedge$x=a] $\circ$ sort([x|x$\in$L$\wedge$x>a])

Algorithmenschemata: Formaler Hintergrund



- **Algorithmische Theorien beschreiben Entwurfsmethode**
  - Algorithmentheorie = Komponenten+Axiome eines Algorithmenschemas
  - Formale Brücke zwischen allgemeiner Theorie der Spezifikationen und abstrakter Theorie korrekter Programme
- **Syntheseprozeß instantiiert allgemeine Theorien**
  - Problem = konkrete Spezifikation + Domänenbeschreibung
  - Synthese: bestimme Komponenten, welche die Axiome erfüllen
  - Korrekte Lösung ergibt sich kanonisch aus Algorithmentheorie

Technischer Grundbegriff: Algebraische Theorien

- **Eine formale Theorie $\mathcal{T}$ hat drei Komponenten**
  - S: Menge von Sortennamen (Namen für Datentypen)
  - Ω: Familie von Operationsnamen (zusammen mit Typisierung)
  - Ax: Menge von Axiomen für Datentypen und Operationen
- **Modell T für Theorie $\mathcal{T}$**
  - T ist Struktur für $\mathcal{T}$: Menge von Datentypen passend zu S und Operationen typisiert gemäß Ω
  - Struktur T erfüllt alle Axiome aus $\mathcal{T}$
- **Theorie $\mathcal{T}_1$ erweitert $\mathcal{T}_2$**
  - Alle Sortennamen, Operationsnamen, Axiome von $\mathcal{T}_2$ existieren in $\mathcal{T}_1$

  Struktur T_1 erweitert T_2
  - Alle Datentypen und Operationen von T_2 existieren auch in T_1 und haben die gleiche Typisierung bezüglich $\mathcal{T}_2$

Endliche Mengen als algebraische Theorie

- **Abstrakte Theorie = Struktur + Grundgesetze**

  Sorten: $\mathtt{Set}(\alpha)$

  Operationsnamen: $\emptyset\colon \mathtt{Set}(\alpha)$

  $+\colon \mathtt{Set}(\alpha)\times\alpha \to \mathtt{Set}(\alpha)$

  $\in\colon \alpha\times\mathtt{Set}(\alpha) \to \mathtt{Bool}$

  Axiome: $a \notin \emptyset$

  $x \in (S+a) \Leftrightarrow (x=a \vee x \in S)$

  $(S+a)+x = (S+x)+a$

  $(S+a)+a = S+a$

  $P(\emptyset) \wedge (\forall S{:}\mathtt{Set}(\alpha).P(S) \Rightarrow \forall a{:}\alpha.P(S+a)) \Rightarrow \forall S{:}\mathtt{Set}(\alpha).P(S)$

- **Modell = Implementierung durch existierende Konzepte**

  $\emptyset \equiv \mathtt{nil}$

  $+ \equiv \lambda a,S.\ a.S$

  $\in \equiv \lambda a,S.\ \exists x \in S.x =_b a$

  $=_{Set} \equiv \lambda S,T.\ (\forall a \in S.\ a \in T) \wedge (\forall a' \in T.\ a' \in S)$

  $\mathtt{Set}(\alpha) \equiv (S,T){:}\alpha\ \mathtt{list}\ //\ S =_{Set} T$

  Alternative Modelle sind möglich (Bitvektoren, sortierte Listen,...)

Algorithmenschemata als algebraische Theorien

- $\mathcal{T}_{\mathcal{SPEC}}$: **Theorie der Spezifikationen**
  - Sorten: D, R; Operationen: $I{:}D{\to}\mathbb{B}$, $O{:}D{\times}R{\to}\mathbb{B}$; keine Axiome
  - **Problemtheorie**: Erweiterung von $\mathcal{T}_{\mathcal{SPEC}}$
    Spezifikation eines Programms durch Bedingungen an I und O
- $\mathcal{T}_{\mathcal{PROG}}$: **Theorie korrekter Programme**
  - Sorten: D, R; Operationen: $I{:}D{\to}\mathbb{B}$, $O{:}D{\times}R{\to}\mathbb{B}$, body: $D{\not\to}R$
  - Axiome: $\forall x{:}D.\, I(x) \Rightarrow O(x,\mathrm{body}(x))$
  - **Programmtheorie**: Erweiterung von $\mathcal{T}_{\mathcal{PROG}}$
    Beschreibung einer Klasse korrekter Programme
- **Algorithmentheorie**: **Theorie einer Algorithmenstruktur**
  - Problemtheorie $\mathcal{A}$ mit kanonischer Erweiterung zu Programmtheorie:
  - Es gibt ein abstraktes Programmschema *BODY*, so daß für jedes Modell A von $\mathcal{A}$ die Spezifikation $spec_A$ zusammen mit *BODY*(A) korrekt ist
    (Für jedes Modell A von $\mathcal{A}$ hat $spec_A$ eine Standardlösung)

Divide & Conquer Schema als Algorithmentheorie

Sorten : D, R, D', R'

Operationen : $I: D \to \mathbb{B}$, $O: D \times R \to \mathbb{B}$, $\succ: D \times D \to \mathbb{B}$,
Decompose: $D \to D' \times D$, $O_D: D \times D' \times D \to \mathbb{B}$, I': $D' \to \mathbb{B}$,
Compose: $R' \times R \to R$, O': $D' \times R' \to \mathbb{B}$, $O_C: R' \times R \times R \to \mathbb{B}$,
$g: D' \to R'$, *Directly-solve*: $D \to R$, *primitive*: $D \to \mathbb{B}$

Axiome : `FUNCTION` f_p`(`x`:`D`):`R `WHERE` $I[x] \wedge$ *primitive*$[x]$
`RETURNS` y `SUCH THAT` $O[x, y]$
$\equiv$ *Directly-solve*$[x]$ ist korrekt
⋮
$\succ$ *ist wohlfundierte Ordnung auf* D

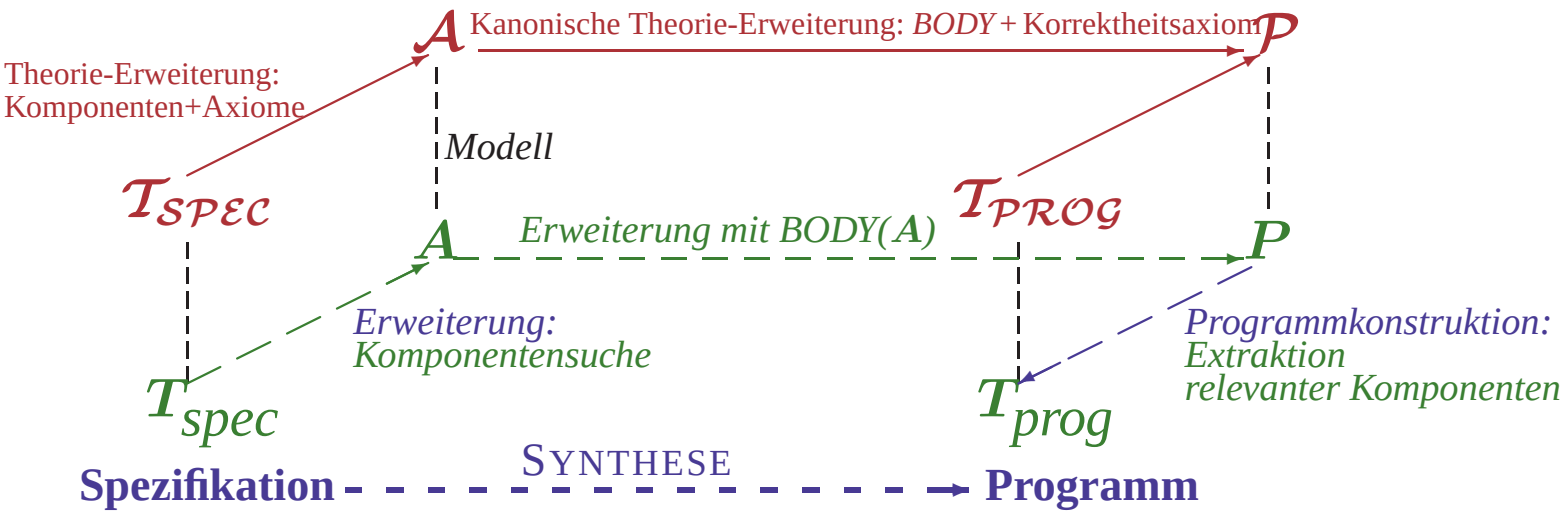
Für jedes Modell A der Divide & Conquer Theorie ist

BODY(A) $\equiv$ `if` *primitive*$[x]$ `then` *Directly-solve*$[x]$
`else` (*Compose* $\circ$ $g \times f$ $\circ$ *Decompose*) (x)

ein korrektes Programm

(Details in Einheit 18)

Synthese mit Algorithmenschemata präzisiert



- **Satz:** Eine Spezifikation *spec* ist erfüllbar, wenn es eine Algorithmentheorie $\mathcal{A}$ und ein Modell A für $\mathcal{A}$ gibt, das die Struktur T_{spec} erweitert
- **Synthese erweitert Problemtheorien zu Algorithmentheorien**
  - Wähle eine Algorithmentheorie $\mathcal{A}$
  - Erweitere die Spezifikation zu einem Modell A von $\mathcal{A}$
    Bestimme Komponenten und beweise, daß Axiome erfüllt sind
  - Instantiierung von *BODY(A)* liefert Modell einer Programmtheorie
  - Extrahiere Programm als Lösung der Spezifikation

Einfaches Algorithmenschema: Operator Match

Problemreduktion auf bekannte Algorithmen

- ***spec* reduzierbar auf *spec′*** **(*spec* $\trianglelefteq$ *spec′*)**

  $\forall x{:}D\,.\,I[x] \Rightarrow \exists x'{:}D'\,.\,(\,I'[x'] \wedge \forall y'{:}R'\,.\,O'[x',y'] \Rightarrow \exists y{:}R\,.\,O[x,y]\,)$

  – Transformation von Ein- und Ausgaben einer gegebene Spezifikation

  – *spec* hat stärkere Eingabebedingung, schwächere Ausgabebedingung

- **Reduzierbarkeit liefert statische Problemtransformation**

  – *spec* $= (D, R, I, O)$ *ist erfüllbar, wenn es eine erfüllbare Spezifikation* $spec' = (D', R', I', O')$ *gibt mit* $spec \trianglelefteq spec'$

  – Beweis für $spec \trianglelefteq spec'$ liefert Substitutionen $\vartheta{:}D \to D'$, $\sigma{:}D \times R' \to R$

  – Ist $body'$ korrekter Algorithmus für $spec'$, dann definiert

  body(*x*) $\equiv$ σ(*x*,*body′*(ϑ(*x*))) einen korrekten Algorithmus für *spec*

- **Einfaches Syntheseverfahren**

  – Suche erfüllbare Spezifikation $spec'$ und beweise $spec \trianglelefteq spec'$

  – Extrahiere σ und ϑ und spezialisiere *body'* zu $\lambda x.\, \sigma$(*x*,*body'*(ϑ(*x*)))

  Auch geeignet zur Verfeinerung anderer Algorithmentheorien ↦ später

Einfaches Algorithmenschema: Generalisierung

Problemreduktion für mengenwertige Spezifikationen

- ***spec* spezialisiert *spec′*** **(*spec* ≪ *spec′*)**

  $R \subseteq R' \;\wedge\; \forall x{:}D\,.\,I[x] \Rightarrow \exists x'{:}D'\,.\,(\,I'[x'] \wedge \forall y{:}R\,.\,O[x,y] \Rightarrow O'[x',y]\,)$

  – Ein- und Ausgabebedingungen von *spec* stärker als *spec′*

  – Andere Bezeichungsart: *spec′* generalisiert *spec*

- **Spezialisierung liefert Problemtransformation**

  – *spec* = `FUNCTION` f`(`x`:`D`)` `WHERE` $I[x]$ `RETURNS` $\{y{:}R \mid O[x,y]\}$ *ist erfüllbar, wenn* $spec \ll spec'$ *für eine erfüllbare Spezifikation* $spec'$ *gilt*

  – Beweis für $spec \ll spec_A$ liefert Substitution $\vartheta{:}D \to D$'

  – *body*(*x*) ≡ $\{\, y \mid y \in body'(\vartheta(x)) \wedge O(x,y) \,\}$ ist korrekt für *spec*

- **Syntheseverfahren**

  – Suche erfüllbare Spezifikation *spec′* und beweise $spec \ll spec'$

  – Extrahiere ϑ, spezialisiere *body'* zu $\lambda x.\, \{y \mid y \in body'(\vartheta(x)) \wedge O(x,y)\}$

  Geeignet zur Verfeinerung mengenwertiger Algorithmentheorien ↦ später

Einfaches Algorithmenschema: Fallunterscheidung

Statische $\vee$-Reduktion durch Zerlegung des Problems

- **$spec$ zerlegbar in $spec_1..spec_n$** ($spec = \cup_i spec_i$)

  $\forall x{:}D\,.\,I[x] \Rightarrow I_1[x] \vee .. \vee I_n[x] \;\wedge\; \forall i{\leq}n\,.\,\forall y{:}R.\; O_i[x,y] \Rightarrow O[x,y]$)

  – Eingabebedingung zerlegbar in Eingabebedingungen der $spec_i$

  – Ausgabebedingung der $spec_i$ stärker

- **Zerlegbarkeit liefert Fallunterscheidung**

  – *$spec$ ist erfüllbar, wenn es erfüllbare Spezifikationen $spec_i$ gibt mit $spec = \cup_i spec_i$*

  – Sind $body_i$ korrekte Algorithmen für $spec_i$, dann ist der Algorithmus *body*(x) $\equiv$ if $I_1[x]$ then *body*$_{A_1}[x]$.. else *body*$_{A_n}[x]$ korrekt für $spec$

- **Syntheseverfahren "Derived Antecedants"**

  – Suche erfüllbare Spezifikation $spec_1$ mit stärkerer Ausgabebedingung

  – Wiederhole mit $spec'{=}(D, R, I \wedge \neg I_1, O)$ bis Zerlegung komplett

  – Setze einzelne Lösungen durch Fallunterscheidung zusammen

  Auch geeignet zur Verfeinerung anderer Algorithmentheorien ↦ später

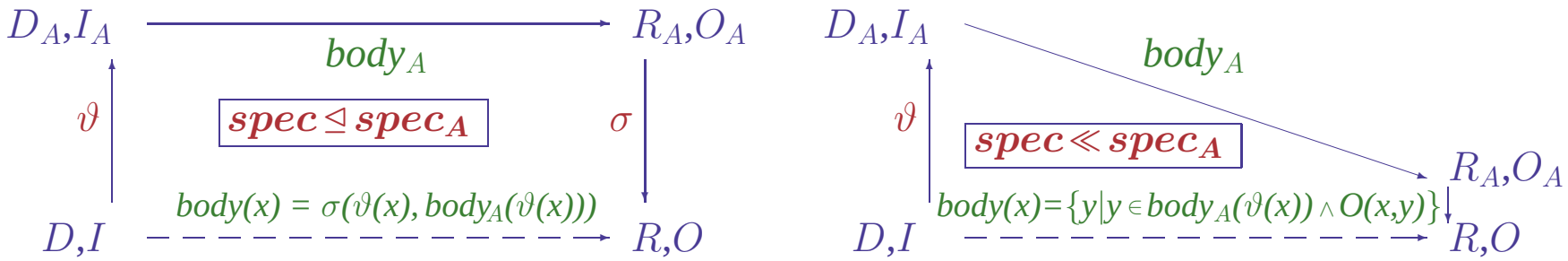
Unterstützungstechnik: Derived Antecedants

- **Bestimme Vorbedingungen für Gültigkeit einer Formel**
  - Formel F darf Quantoren enthalten
  - Vorbedingung P darf evtl. manche Variablen von F nicht enthalten
  - z.B. Vorbedingung für $F \equiv \exists L_1, L_2.\ |L|>|L_1| \wedge |L|>|L_2| \wedge L_1 \circ L_2 = L$
    darf nur die Variable L verwenden (Lösung $P[L] \equiv |L|>1$)
- **Herleitung durch Anwendung von Reduktionsregeln**
  - Rewriting mit Äquivalenzumformungen und Implikationen

    z.B. True $\Rightarrow \exists L_1, L_2.\ L_1 \circ L_2$ und $L_1 \circ L_2 = L \Rightarrow |L_1|+|L_2| = |L|$ und $i+j>i \Leftrightarrow j>0$
  - Inferenzsystem muß hunderte von Rewrite Regeln verwalten
- **Steuerung ist heuristisch**
  - Anwendung von Regeln auf quantorenfreien Teil der Formel
  - Vorwärtsinferenz: Ziel ist syntaktischer (intensionaler) Natur

    z.B. Beschränkung der erlaubten Variablen, Größe der Formel etc.

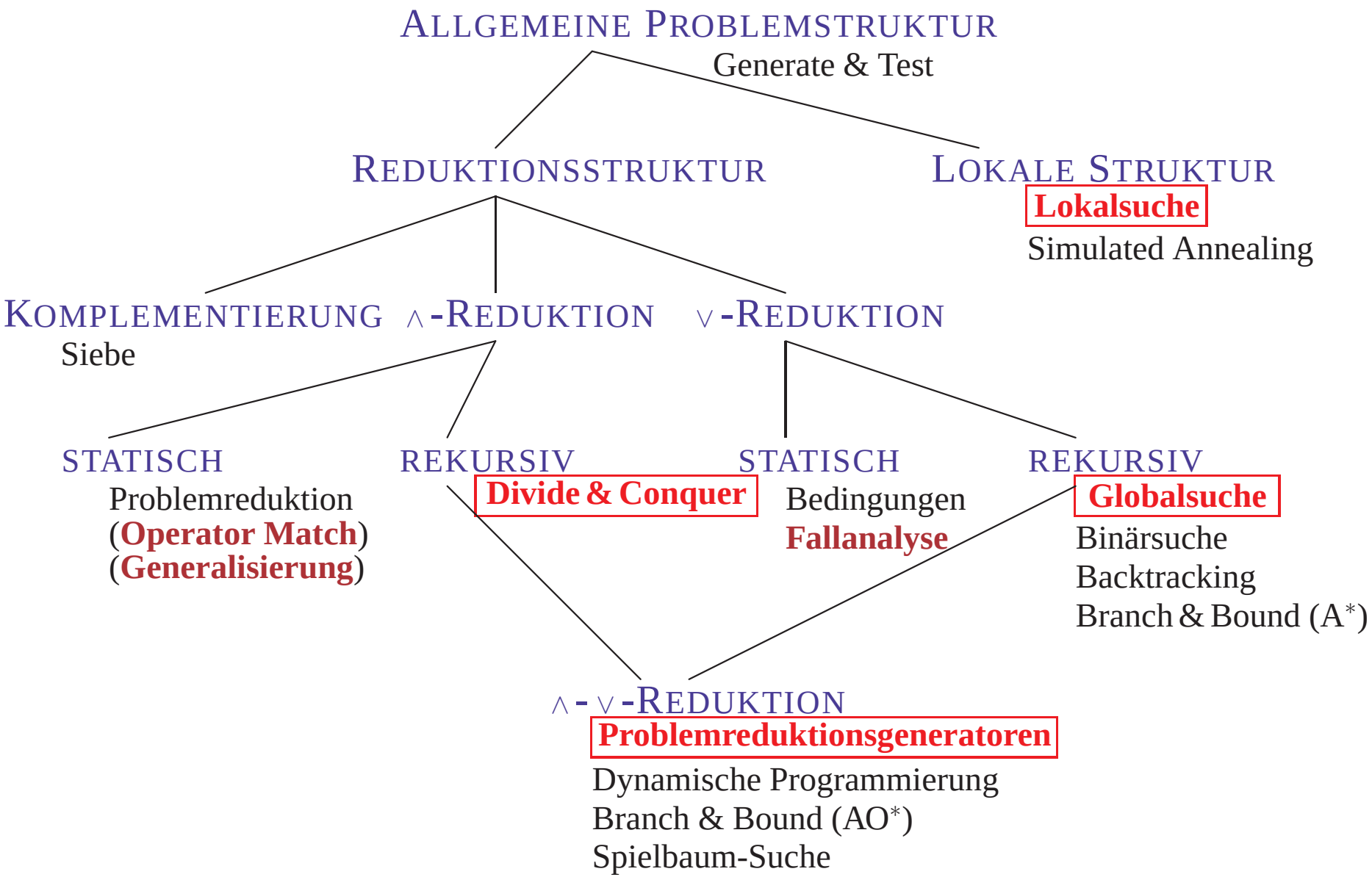
Modellbildung durch statische Reduktion auf Teilmodelle



- **Speichere generische Modelle von Algorithmentheorien**
  - Standardkomponenten der Algorithmentheorie für wichtige Domänen
  - Axiome von $\mathcal{A}$ sind für diese Modelle a-priori bewiesen
- **Synthese reduziert Spezifikation auf gespeichertes Modell**
  - Wähle eine Algorithmentheorie $\mathcal{A}$
  - Wähle Modell A passend zur Grunddomäne von $spec$ (Liste, Menge, Baum,...)
  - Versuche $spec \trianglelefteq spec_A$ automatisch zu beweisen (proofs-as-programs)
    Mengenwertige Synthese analog mit Generalisierung $spec \ll spec_A$
  - Spezialisierung von $body_A$ korrekt für extrahierte Substitutionen σ und ϑ

Syntheseverfahren erfordert keine weiteren Inferenzen! ↦ Beispiele später

Hierarchie algorithmischer Strukturen



Algorithmenschemata: Vorteile für Synthese

- **Effizientes Syntheseverfahren**
  - **Voruntersuchungen entlasten Syntheseprozeß** zur Laufzeit
    - Beweislast verlagert in Entwurf und Beweis der Algorithmentheorien
  - **Verfeinerung vorgefertigter Teillösungen** (Modelle) möglich
    - zielgerichtetes Vorgehen, Verifikation der Axiome entfällt
  - Echte **Kooperation zwischen Mensch und Computer**
    - Mensch: Entwurfsentscheidungen — Computer: formale Details
- **Erzeugung effizienter Algorithmen**
  - Vorgabe einer effizienten Grundstruktur durch Theoreme
  - Individuelle Optimierung nachträglich
- **Wissensbasiertes Vorgehen**
  - Erkenntnisse über Algorithmen als Theoreme verwendbar
- **Formales theoretisches Fundament**
  - Leicht in das Konzept beweisbasierter Systeme integrierbar