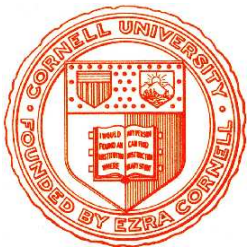
# Automatisierte Logik und Programmierung II

## Teil IV




## Automatisierte Programmierung


**1. Grundkonzepte, Paradigmen & Strategien**

**2. Wissensbasierte Programmentwicklung**

– Divide & Conquer, Globalsuche, sl
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 zurAufgabenstellung selten erkennbar
  – Begründung für Korrektheit des Programms fehlt fast immer

- **Logische Synthese von Programmen hilft**

  – Unterstützung für teil-automatische Konstruktion von Algorithmen
  – Logisches Fundament erhöht Zuverlässigkeit erzeugter Programme
  – 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. §16

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** |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

Costas Array der Ordnung 6 und seine Differenzentafel

## 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** | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

Costas Array der Ordnung 6 und seine Differenzentafel

# 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** | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

Costas Array der Ordnung 6 und seine Differenzentafel

# 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** | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

Costas Array der Ordnung 6 und seine Differenzentafel

# 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 | | |
| | | | | | |
| | | | | | |
| | | | | | |

Costas Array der Ordnung 6 und seine Differenzentafel

# 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**

# Costas-Arrays (1): formale Spezifikation

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

- **Formalisierung vorkommender Begriffe**

# Costas-Arrays (1): formale Spezifikation

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

- **Formalisierung vorkommender Begriffe**
  - `nodups(`$L$`)` $\equiv$ $\forall$`i`$\neq$`j`$\leq$`|p|.` $L$`[i]`$\neq$$L$`[j]`

# Costas-Arrays (1): formale Spezifikation

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

- **Formalisierung vorkommender Begriffe**
  - `nodups(`$L$`)` $\equiv$ $\forall$`i`$\neq$`j`$\leq$`|p|.` $L$`[i]`$\neq$$L$`[j]`
  - `perm(p,`$S$`)` $\equiv$ `nodups(p)` $\wedge$ `range(p)=`$S$

# Costas-Arrays (1): formale Spezifikation

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

- **Formalisierung vorkommender Begriffe**
  - `nodups(`$L$`)` $\equiv$ $\forall$`i`$\neq$`j`$\leq$`|p|.` $L$`[i]`$\neq$$L$`[j]`
  - `perm(p,`$S$`)` $\equiv$ `nodups(p)` $\wedge$ `range(p)=`$S$
  - `dtrow(p,`$j$`)` $\equiv$ `[p[i]-p[i+`$j$`] | i` $\in$ `[1..length(p)-`$j$`]]`

# Costas-Arrays (1): formale Spezifikation

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

- **Formalisierung vorkommender Begriffe**
  - `nodups(`$L$`)` $\equiv$ $\forall$`i`$\neq$`j`$\leq$`|p|.` $L$`[i]`$\neq$$L$`[j]`
  - `perm(p,`$S$`)` $\equiv$ `nodups(p)` $\wedge$ `range(p)=`$S$
  - `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`

# Costas-Arrays (1): formale Spezifikation

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

- **Formalisierung vorkommender Begriffe**
  - `nodups(`$L$`)` $\equiv$ $\forall$`i`$\neq$`j`$\leq$`|p|.` $L$`[i]`$\neq$$L$`[j]`
  - `perm(p,`$S$`)` $\equiv$ `nodups(p)` $\wedge$ `range(p)=`$S$
  - `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**

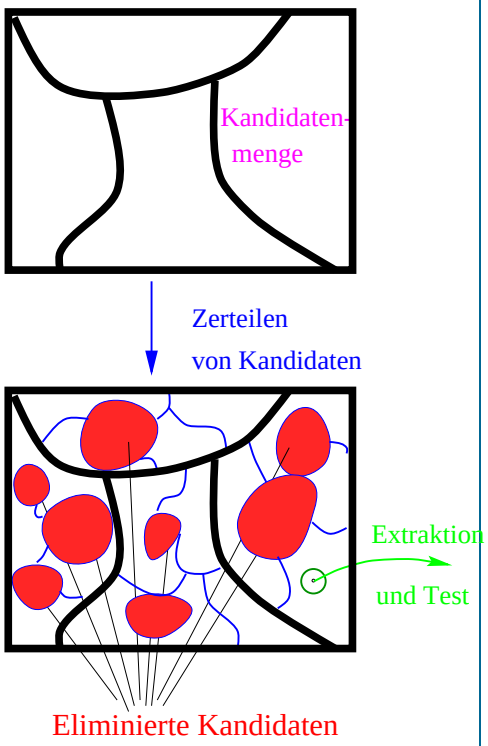
  *Standardstruktur zum Durchsuchen von Lösungsräumen*

  – Codierung von Kandidatenmengen

  – Wiederholtes Aufteilen und Filtern

  auf Basis von Repräsentanten

  – Extraktion konkreter Lösungen aus Repräsentanten




- **Algorithmus wird direkt 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,[])
```

# 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,[])
```

# 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))
   = (if perm(s,{1..n}) ∧∀j<n. nodups(dtrow(s,j)) then {s} else ∅)
      ∪ ⋃{ 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))
   =( if perm(s,{1..n}) ∧ ∀j<n.nodups(dtrow(s,j)) then {s} else ∅)
      ∪ ⋃{ 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))
   =( if perm(s,{1..n}) then {s} else ∅)
      ∪ ⋃{ 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))
   = if perm(s,{1..n}) then {s} else ∅)
      ∪ ⋃{ 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))
   =( if {1..n}\s=∅ then {s} else ∅)
      ∪ ⋃{ 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))
   = (if {1..n}\s=∅ then {s} else ∅)
      ∪ ⋃{ 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))
   = (if {1..n}\s=∅ then {s} else ∅)
      ∪ ⋃{ aux(n,s·i) | i∈{1..n} ∧ nodups(s·i)
                                 ∧ ∀j<|s·i|.nodups(dtrow(s·i,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))
   = (if {1..n}\s=∅ then {s} else ∅)
      ∪ ⋃{ aux(n,s·i) | i∈{1..n} ∧ nodups(s·i)
                        ∧ ∀j<|s·i|.nodups(dtrow(s·i,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))
   = (if {1..n}\s=∅ then {s} else ∅)
      ∪ ⋃{ aux(n,s·i) | i∈{1..n}\s
                        ∧ ∀j<|s·i|.nodups(dtrow(s·i,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))
   = (if {1..n}\s=∅ then {s} else ∅)
      ∪ ⋃{ aux(n,s·i) | i∈{1..n}\s
                        ∧ ∀j<|s·i|. nodups(dtrow(s·i,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))
   = (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,[])
```

# 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,[])
```

---

Ersetze ineffiziente Neuberechnung $\{1..n\}\backslash s$ durch neue Variable

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

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

---

Ergänze neue Variable `pool` zu Definition und Aufruf von aux

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

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

---

Vereinfache Algorithmus mit Vorbedingung `pool={1..n}\s`

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

```
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}, |s·i|) | i∈pool
                          ∧ ∀j<|s|.(s[|s·i|-j]-i)∉dtrow(s,j)}
  in aux(n,[],{1..n}, |[]|)
```

---

Ersetze ineffiziente Neuberechnung `|s|` durch neue Variable `ssize`

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

```
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)
```

---

Vereinfache Algorithmus mit Vorbedingung `ssize=|s|`

# 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)
```

---

Anwendung der Distributivgesetze für `if .. then .. else`

# 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} ∪ ⋃{ aux(n, s·i,pool\{i},ssize+1) | i∈pool
                    ∧ ∀j<ssize. (s[ssize+1-j]-i)∉dtrow(s,j)}
      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)
```

---

Vereinfachung im Kontext der Bedingung `pool=∅`

# 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)
```

# 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` $\mapsto$ Standardimplementierung positiver ganzer Zahlen

`ssize:ℤ` $\mapsto$ Standardimplementierung positiver ganzer Zahlen

`s:Seq(ℤ)`, Elemente werden hinten angehängt $\mapsto$ umgekehrt verkettete Liste

`pool:Set(ℤ)`: Elemente werden aus fester Menge entnommen $\mapsto$ Bitvektor

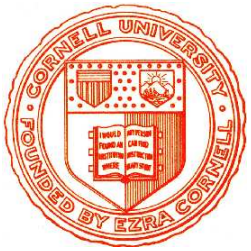
Letzter Schritt vor Übertragung in konkrete Programmiersprache

# 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

# Automatisierte Logik und Programmierung

## Einheit 18




## Syntheseparadigmen & -strategien


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

# Komponenten eines Syntheseverfahrens

**Erzeuge korrekte ausführbare Programme aus Spezifikationen**

- **Vorarbeit: Formale Spezifikation als Ausgangspunkt**
  - Beschreibung von Anwendungsbereich und Problemstellung in (umfangreicher, "natürlicher") formaler Spezifikationssprache
  - Objekttheorie formalisiert Eigenschaften neuer Anwendungskonzepte
- **(Semi-)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**

  Spezifikation benötigt eine Reihe von Informationen:
  - Aus welchem Datentyp stammen die Eingaben? ($D$omain)
  - Zu welchem Datentyp gehören die Ausgaben? ($R$ange)
  - Gibt es Beschränkungen an zulässige Eingaben? ($I$nput-Bedingung)
  - Was ist der Zusammenhang zwischen Ein- und Ausgaben? ($O$utput-Bedingung)

  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 Darstellung abhängig von Aufgabenstellung**

  Erhöhte Lesbarkeit durch Schlüsselwörter und instantiierte Form von $I$, $O$
  - Für Bestimmung einer möglichen Lösung
    `FUNCTION` $f(x{:}D){:}R$ `WHERE` $I[x]$ `RETURNS` $y$ `SUCH THAT` $O[x,y]$
  - Für Bestimmung aller möglichen Lösungen
    `FUNCTION` $f(x{:}D)$ `WHERE` $I[x]$ `RETURNS` $\{y{:}R \mid O[x,y]\}$

# Programme und Korrektheit, formal

- **Programm = Spezifikation + Algorithmus**
  - Algorithmen (Programmkörper) sind berechenbare (partielle) Funktionen auf $D \nrightarrow R$, die auf allen zulässigen Eingaben definiert sind
  - Ein **(formales) Programm** ist ein 5-Tupel $prog = (D, R, I, O, body)$ wobei $(D, R, I, O)$ formale Spezifikation, $body{:}D \nrightarrow R$ 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 $body : D \nrightarrow R$ gibt, so daß $prog{=}(spec, body)$ korrekt ist

# 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 effizienter Algorithmen verlangt Spezialstrategien
  - Gegenseitige Simulation der Methoden prinzipiell möglich
- **Wissensbasierte Syntheseverfahren**
  - Wissen über algorithmische Grundstrukturen wird formalisiert als "**Algorithmentheorien**" (Struktur + Korrektheitsaxiome)
  - Strategien verwenden Wissen zur Erzeugung effizienter Algorithmen
  - Ziel ist Unterstützung des Programmierers, nicht Ersetzung
  - Aufwendigere Vorarbeiten, aber erfolgreich in der "Praxis"

## Extraktion aus konstruktivem Beweis eines Theorems

- **Beweise Erfüllbarkeit einer Spezifikation**

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

  1. Erzeuge **Spezifikationstheorem**: $\forall x : D\,.\,\exists y : R.\ I[x] \Rightarrow O[x,y]$
  2. Suche formalen Beweis in konstruktivem logischen Kalkül
  3. Extrahiere aus Beweis einen Algorithmus zur Berechnung von $y$ aus $x$

     Optimiere durch Reduktion und Elimination überflüssiger Information

- **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)

# Synthese eines Quadratwurzelprogramms

## 1. Spezifikationstheorem: $\forall n{:}\mathbb{N}.\ \exists r{:}\mathbb{N}.\ r^2 \le n < (r+1)^2$

– Keine Input-Bedingung erforderlich

## 2. Formaler Beweis mit Standardinduktion

| | |
|---|---|
| $\vdash \forall n{:}\mathbb{N}.\ \exists r{:}\mathbb{N}.\ r^2 \le n < (r+1)^2$ | allR |
| 1. $n{:}\mathbb{N} \vdash \exists r{:}\mathbb{N}.\ r^2 \le n < (r+1)^2$ | NatInd 1 |
| 1.1. $\vdash \exists r{:}\mathbb{N}.\ r^2 \le 0 < (r+1)^2$ | existsR ⌈0⌉ THEN Auto ✓ |
| 1.2. $i{:}\mathbb{N}^+, r{:}\mathbb{N}, r^2 \le i-1 < (r+1)^2 \vdash \exists r{:}\mathbb{N}.\ r^2 \le i < (r+1)^2$ | Decide ⌈$(r+1)^2 \le i$⌉ THEN Auto |
| 1.2.1. $i{:}\mathbb{N}^+, r{:}\mathbb{N}, r^2 \le i-1 < (r+1)^2, (r+1)^2 \le i \vdash \exists r{:}\mathbb{N}.\ r^2 \le i < (r+1)^2$ | existsR ⌈r+1⌉ THEN Auto ✓ |
| 1.2.2. $i{:}\mathbb{N}^+, r{:}\mathbb{N}, r^2 \le i-1 < (r+1)^2, \neg((r+1)^2 \le i) \vdash \exists r{:}\mathbb{N}.\ r^2 \le i < (r+1)^2$ | existsR ⌈r⌉ THEN Auto ✓ |

## 3. Extrahierter Algorithmus enthält Beweisinformationen

```
function 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>> else <r,<Ax,Ax>>
```

# Verbesserungen für die $\lfloor\sqrt{n}\rfloor$–Synthese

- **Verbesserte Extraktion ohne Verifikationsanteile**

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

- **Effizienzorientierte Beweisführung** (vgl. §10, Folie 19)
  - Nutze Binärdarstellung der Zahlen und bestimme Lösung bitweise
  - Benötigt andersartige ("4-adische") Induktionsstruktur im Beweis

```
function 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 stößt schnell an Grenzen**
  - Beweisschritte sind oft zu atomar (Assemblerniveau)
  - Effizienz des Resultats stark abhängig von Beweismethodik
  - Beweise für komplexe Programme interaktiv kaum durchführbar
  - Automatisierung der Synthese effizienter Programme sehr schwierig

    Nur praktikabel in Kombination mit Definitionen und Spezialtaktiken

# Synthese durch Transformation

## Transformiere Spezifikation in effektiv ausführbare Formel

- **Betrachte Spezifikation als (logisches) Initialprogramm**

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

  1. 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 als berechenbar bekannte Prädikate enthält

  2. Extrahiere Programm aus Formel oder interpretiere als Logik-Programm

- **Mechanismus basiert auf Vorwärtsinferenz**
  - Ziel "bessere algorithmische Struktur" läßt sich nicht präzise beschreiben
  - Viel Interaktion oder starke heuristische Steuerung erforderlich

- **Forschungsschwerpunkte**
  - Leistungsfähige Transformationsregeln
  - Effiziente Rewrite Techniken und Heuristiken für Vorwärtsinferenz

# Synthese durch Transformation: Methodik

- **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, elementaren Tautologien, Abstraktionen, dynamisch erzeugten Definitionen, ...

- **Gleichwertig zum Prinzip "Beweise als Programme"**

  – Transformationen durch Beweise mit Gleichheitslemmata simulierbar

  – Beweisregeln können als Rewrite-Regeln beschrieben werden

- **Methode skaliert schlecht**

  – Transformationen haben zu niedriges Inferenzniveau

  – Suchraum explodiert bei nichttrivialen Problemen

  Nur sinnvoll bei Einsatz von high-level Transformationen (siehe Folie 10)

# Wissensverarbeitung in der Programmsynthese

## Zielgerichtete Entwicklung guter Algorithmen

- **Synthese im Kleinen ist zu generisch**
  - Fokus auf Logik oder Rewriting statt auf Programmierung
    Erkenntnisse aus der Algorithmik kommen nicht zum Tragen
  - Steuerung durch "normale" Programmier ist kaum möglich
  - Keine echte Unterstützung bei der Entwicklung von Programmen
- **Praktische Programmiermethodik verwendet Wissen**
  - Welche grundsätzlichen Algorithmenstrukturen gibt es?
  - Welche Algorithmenstrukturen sind für welche Probleme 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 $O[x,y]$

  1. Wähle eine algorithmische Grundstruktur für die Lösung
  2. Bestimme Komponenten eines schematischen Grundalgorithmus
  3. Prüfe, ob Komponenten die Korrektheitsaxiome des Schemas erfüllen
  4. Instantiiere Algorithmenschema (und optimiere Resultat)
- **Forschungsschwerpunkte**
  - Analyse der allgemeinen Struktur einer Klasse von Algorithmen
  - Schematisierung durch Komponenten und Korrektheitsaxiome
  - Techniken zur Verfeinerung von Standardstrukturen

# Globalsuch Synthese: Costas-Arrays Problem (§20)

- **Problemspezifikation**

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

# Globalsuch Synthese: Costas-Arrays Problem (§20)

- **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))}

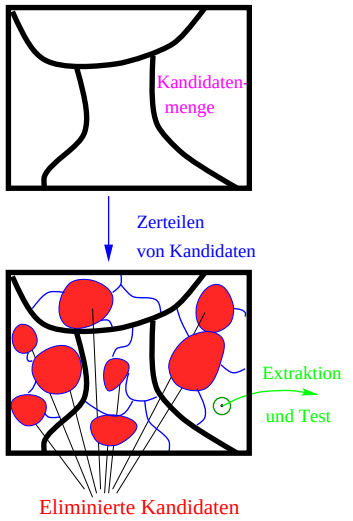
- **Grundstruktur von Globalsuch-Algorithmen** (mathematische Notation)

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

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

  in $f_{gs}(x, s_0(x))$

- **Bedeutung der markierten 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 $x$



# Globalsuch Synthese: Costas-Arrays Problem (§20)

- **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))}

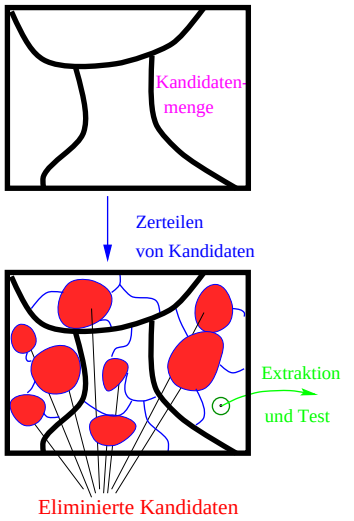
- **Grundstruktur von Globalsuch-Algorithmen** (mathematische Notation)

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

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

  in $f_{gs}(x, s_0(x))$

- **Bedeutung der markierten 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}) $\wedge$ ..}

  $\equiv$ function $f_{gs}$(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 (§19)

- **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 $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Komponenten für einen Sortieralgorithmus**

  *primitive* $\equiv$ $\lambda$L. null?(L)
  *Directly-solve* $\equiv$ $\lambda$L. []
  *Decompose* $\equiv$ $\lambda$L. 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 $\times$ $\lambda$S.S
  *Compose* $\equiv$ $\lambda S_1, S_2, S_3.\ S_1 \circ S_2 \circ S_3$

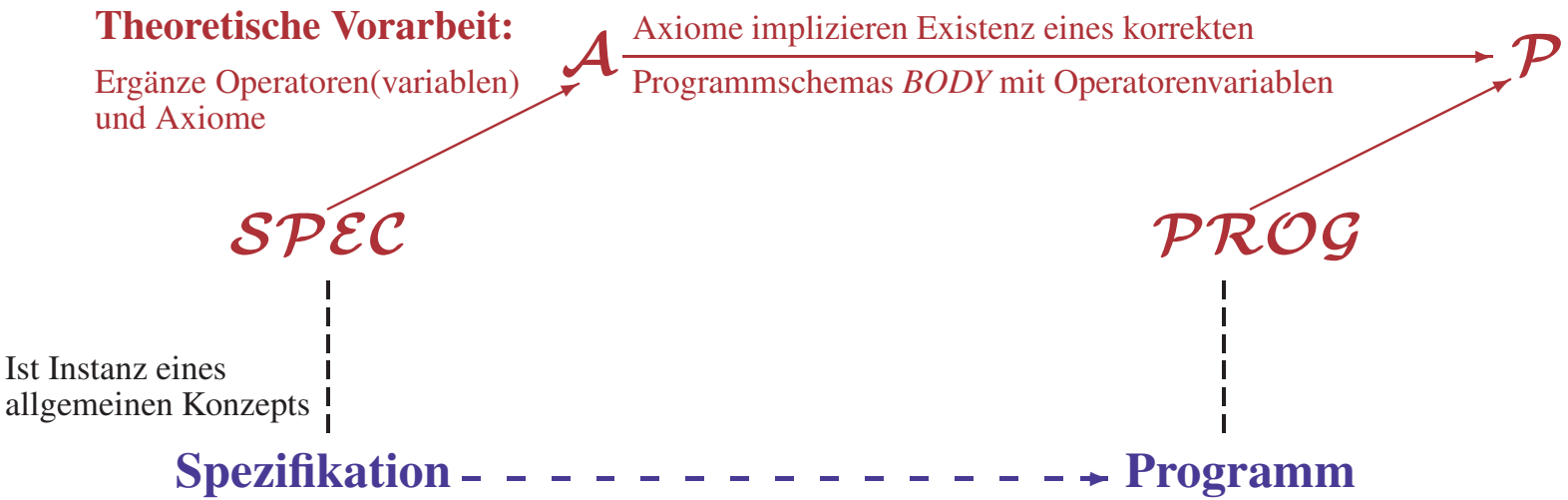
- **Instantiierter Algorithmus**

  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

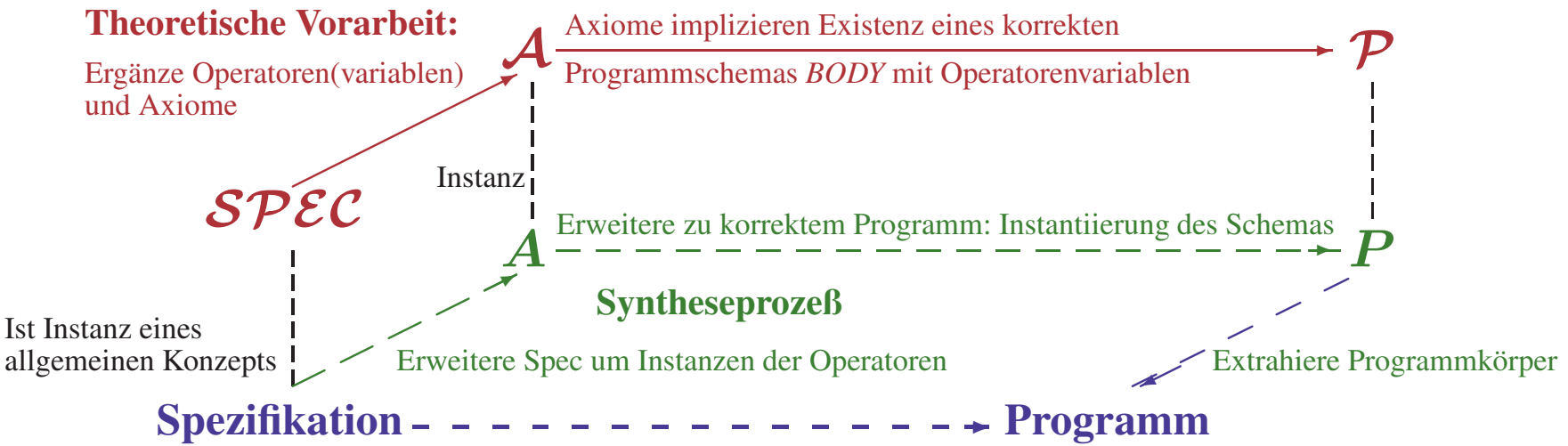
**Spezifikation** - - - - - - - - - - - - - - - - → **Programm**

# Algorithmenschemata: Formaler Hintergrund




- **Entwurfsmethodik basiert auf algebraischer Denkweise**
  - Komponenten und Axiome eines Algorithmenschemas bilden eine Algorithmentheorie $\mathcal{A}$ als Erweiterung der Theorie der Spezifikationen
  - Algorithmentheorie läßt sich kanonisch zu Programmtheorie $\mathcal{P}$ erweitern welche die abstrakte Theorie korrekter Programme erweitert

# Algorithmenschemata: Formaler Hintergrund




- **Entwurfsmethodik basiert auf algebraischer Denkweise**
  - Komponenten und Axiome eines Algorithmenschemas bilden eine Algorithmentheorie $\mathcal{A}$ als Erweiterung der Theorie der Spezifikationen
  - Algorithmentheorie läßt sich kanonisch zu Programmtheorie $\mathcal{P}$ erweitern welche die abstrakte Theorie korrekter Programme erweitert
- **Syntheseprozeß basiert auf Instantiierung**
  - Problem besteht aus konkreter Spezifikation + Domänenbeschreibung
  - Synthese bestimmt Instanz von $\mathcal{A}$, und prüft ob Axiome erfüllt sind
  - Korrekte Lösung ergibt sich unmittelbar durch Instanz des Schemas in $\mathcal{P}$

## WORTWAHL ENTSTAMMT THEORIE ALGEBRAISCHER SPEZIFIKATIONEN

- **Eine formale Theorie $\mathcal{T}$ besteht aus**
  - $S$: Menge von Sortennamen (Namen für Datentypen)
  - $\Omega$: Familie von Operationsnamen (zusammen mit Typisierung)
  - $Ax$: Menge von Axiomen für Datentypen und Operationen
  - · Theorie der endlichen Mengen (§16)
  - · $\mathcal{SPEC}$: Theorie der Spezifikationen
    Sorten: $D$, $R$; Operationen: $I$:$D\to\mathbb{B}$, $O$:$D\times R\to\mathbb{B}$; keine Axiome
  - · $\mathcal{PROG}$: 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: $\forall x{:}D.\, I(x) \Rightarrow O(x,\text{body}(x))$
- **Modell $T$ für Theorie $\mathcal{T}$**
  - Menge von Datentypen und Operationen, welche die Typbedingungen einhalten und alle Axiome aus $\mathcal{T}$ erfüllen (logisch "Instanz" der Theorie)
- **$\mathcal{T}_1$ erweitert $\mathcal{T}_2$**
  - Alle Sortennamen, Operationsnamen, Axiome von $\mathcal{T}_2$ existieren in $\mathcal{T}_1$
  - Definition der Erweiterung von Modellen analog

# Algorithmenschemata als algebraisches Konzept

- **Formalisierung bekannter algorithmischer Strukturen**
  - Suchstrukturen: Divide & Conquer, Lokalsuche, Globalsuche, Dynamische Programmierung, Siebe, ...
  - Glue-Code Schemata: Operator Match, Generalisierung, Fallanalyse
  - Kriterien für Korrekheit sind im Prinzip auch bekannt
- **Algorithmentheorie $\mathcal{A}$ für eine Algorithmenstruktur**
  - Erweiterung $\mathcal{A}$ der Spezifikationstheorie $\mathcal{SPEC}$
  - Eine kanonische Erweiterung zu einer Programmtheorie $\mathcal{P}$ (Erweiterung von $\mathcal{PROG}$) muß existieren

    D.h. es gibt ein abstraktes Programmschema *BODY*, so daß für jedes Modell $A$ von $\mathcal{A}$ die Instanz $BODY_A$ eine korrekte Lösung für die in $A$ enthaltene Spezifikation ist
- **Satz:** Eine Spezifikation *spec* ist erfüllbar, wenn es eine Algorithmentheorie $\mathcal{A}$ und ein Modell $A$ für $\mathcal{A}$ gibt, das *spec* erweitert

# Divide & Conquer Schema als Algorithmentheorie

- **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 $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Algorithmentheorie enthält Zusatzkomponenten neben $(D, R, I, O)$**
  - Sorten für Typisierung der Komponenten: $D$', $R$'
  - Operationen für Berechnung der Lösung: *primitive*:$D \to \mathbb{B}$, *Directly-solve*:$D \to R$

    *Compose*: $R' \times R \to R$, $g$: $D' \to R'$, *Decompose*: $D \to D' \times D$
  - Operationen für Spezifikation dieser Komponenten:

    $O_D$: $D \times D' \times D \to \mathbb{B}$, $I'$: $D' \to \mathbb{B}$, $O'$: $D' \times R' \to \mathbb{B}$, $O_C$: $R' \times R \times R \to \mathbb{B}$
  - Operation für Terminierungsaussagen: $\succ$: $D \times D \to \mathbb{B}$
  - Axiome für relative Korrektheit von *Directly-solve*, *Compose*, $g$ und *Decompose*
  - Axiom: $\succ$ *ist wohlfundierte Ordnung auf* $D$

  Nicht alle Komponenten werden im endgültigen Algorithmus gebraucht

- **Kanonische Erweiterbarkeit der Theorie ist beweisbar**
  - Für jedes Modell $A$ der Divide & Conquer Theorie ist

    *BODY(A)* $\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$

    else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

    ein korrektes Programm

    (Details in Einheit 19)

# Synthese auf Basis von Algorithmentheorien

- **Erweitere Spezifikation zu passend zur Algorithmentheorie**
  - Gegeben: FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS $y$ SUCH THAT $O[x,y]$
  - Wähle eine geeignete Algorithmentheorie $\mathcal{A}$
  - Erweitere die Spezifikation zu einem Modell $A$ von $\mathcal{A}$
    Hierzu bestimme Komponenten und beweise, daß Axiome erfüllt sind
  - Instantiiere Programmschema *BODY* / entferne irrelevante Komponenten

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

- **Beispiel: Sortieralgorithmen** (Informale Beschreibung)
  - Gegeben: FUNCTION sort(L:Seq($\mathbb{Z}$)):Seq($\mathbb{Z}$) RETURNS $S$

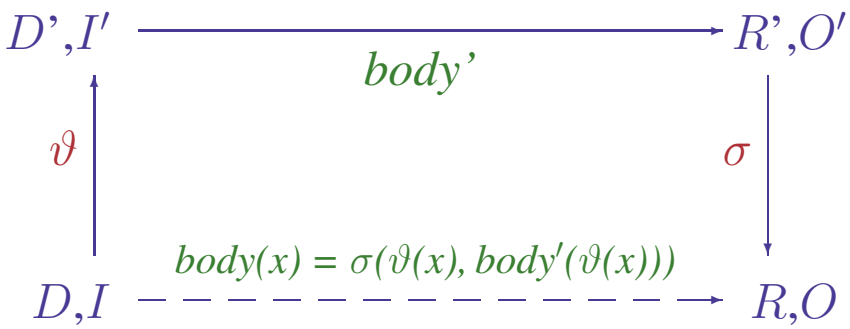
    SUCH THAT rearranges($L$,$S$) $\wedge$ ordered($S$)
  - Wähle *Decompose* als Zerlegung von L in Anfang und Rest
  - Nur möglich wenn Liste nicht leer (*primitive*(L) $\equiv$ null?(L))
  - *Directly-solve*(L) muss [] und $g$ sollte id sein
  - Bestimmung von *Compose* benötigt weitere Synthese, liefert "Einfügen"

# Einfaches Algorithmenschema: Operator Match

## Problemreduktion auf bekannte Algorithmen

- **Es gibt ein verwandtes Problem auf anderem Domain**
  - Vereinigen endlicher Mengen $S_1 \cup S_2$ entspricht Verkettung von Listen
    Konvertiere Mengen $S_1, S_2$ in Listen $L_1, L_2$ der Elemente
    Konvertiere Liste $L_1 \circ L_2$ zurück in Mengendarstellung
  - Komplementieren von $S \subseteq \{x_1, .., x_n\}$ entspricht Bitvektor-Invertierung
    Konvertiere Menge $S$ in Bitvektor $v{:}\{1, .., n\} \to \{0, 1\}$
    Konvertiere $\bar{v} = \lambda i.1 - v(i)$ zurück in Mengendarstellung
- **Formales Konzept: Reduzierbarkeit von Spezifikationen**
  - ***spec* reduzierbar auf *spec′*** (***spec* ⊴ *spec′***)
    $\equiv \forall x{:}D\,.\,I[x] \Rightarrow \exists x'{:}D'.\,(I'[x'] \wedge \forall y'{:}R'.\,O'[x', y'] \Rightarrow \exists y{:}R.\,O[x, y])$
  - Transformation von Ein- und Ausgaben einer gegebenen Spezifikation
    *spec* hat stärkere Eingabebedingung, schwächere Ausgabebedingung
  - Transformationen sind aus Beweis generierbar (Beweise-als-Programme)

# Synthese durch Reduzierbarkeit $\trianglelefteq$



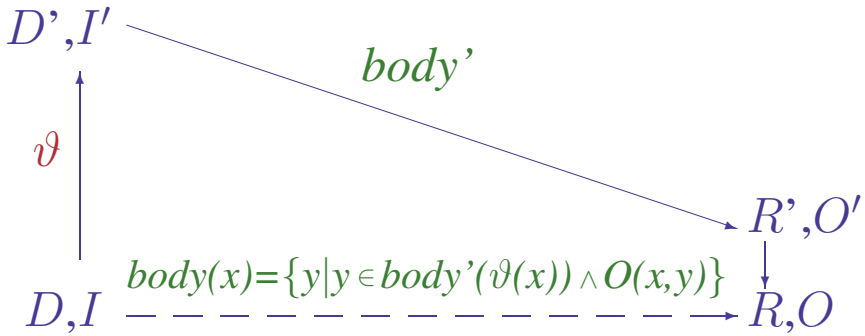

- **Reduzierbarkeit liefert statische Problemtransformation**
  - Eine Spezifikation $spec = (D, R, I, O)$ ist erfüllbar, wenn es eine erfüllbare Spezifikation $spec' = (D', R', I', O')$ gibt mit $spec \trianglelefteq spec'$
  - Beweis für $spec \trianglelefteq spec'$ liefert Substitutionen $\vartheta: D \to D'$, $\sigma: D' \times R' \to R$
  - Ist $body'$ korrekter Algorithmus für $spec'$, dann definiert $body(x) \equiv \sigma(\vartheta(x), body'(\vartheta(x)))$ einen korrekten Algorithmus für $spec$
- **Einfaches Syntheseverfahren "Operator Match"**
  - Suche erfüllbare Spezifikation $spec'$ und beweise $spec \trianglelefteq spec'$
  - Extrahiere $\sigma$ und $\vartheta$ und spezialisiere $body'$ zu $\lambda x.\, \sigma(\vartheta(x), body'(\vartheta(x)))$

  Auch geeignet zur Verfeinerung anderer Algorithmentheorien $\mapsto$ später

## Problemreduktion für mengenwertige Spezifikationen

- **Es gibt ein allgemeineres Problem auf ähnlichem Domain**
  - Suche nach Costas Arrays der Größe $n$ (spezielle Permutationen der Ordnung $n$) ist ist Spezialfall einer Suche auf Listen über $\{1, .., n\}$
    Konvertiere Eingabe $n$ in die Menge $\{1, .., n\}$
    Verwende allgemeine Suchstruktur auf Listen über Mengen
    Entferne Listen, die keine Costas Arrays darstellen
- **Formales Konzept: Spezialisierung von Spezifikationen**
  - ***spec*** spezialisiert ***spec′*** (***spec*** $\ll$ ***spec′***, $spec'$ generalisiert $spec$ )
    $\equiv R \subseteq R' \ \wedge \ \forall x{:}D\,.\,I[x] \Rightarrow \exists x'{:}D'\,.\,(I'[x'] \wedge \forall y{:}R\,.\,O[x,y] \Rightarrow O'[x',y])$
  - Ein- und Ausgabebedingungen von $spec$ stärker als $spec'$
  - Liefert Transformation der Eingaben einer gegebenen Spezifikation
    Ausgaben können übernommen und weiter verfeinert werden

# Synthese durch Spezialisierung ≪




- **Spezialisierung liefert statische Problemtransformation**
  - Spezifikation $spec =$ `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 $spec \ll spec'$ liefert Substitution $\vartheta: D \to D'$
  - $body(x) \equiv \{\, y \mid y \in body'(\vartheta(x)) \wedge O(x,y) \,\}$ ist ein korrekter Algorithmus für $spec$, wenn $body'$ korrekt für $spec'$ ist
- **Syntheseverfahren**
  - Suche erfüllbare Spezifikation $spec'$ und beweise $spec \ll spec'$
  - Extrahiere $\vartheta$, spezialisiere *body'* zu $\lambda x.\, \{y \mid y \in body'(\vartheta(x)) \wedge O(x,y)\}$

Geeignet zur Verfeinerung mengenwertiger Algorithmentheorien ↦ später

# Unterstützungstechnik: Vorwärtsinferenz

- **Hauptproblem ist automatisches Witness-Finding**
  - Reduktion/Spezialisierung beweist $\forall x{:}D.I[x] \Rightarrow \exists x'{:}D'.(I'[x'] \wedge ...)$
  - Bestimmung von $x'$ benötigt Anwendung von Gleichheitslemmata
  - Unifikation mit hunderten von Gleichheitsregeln ist zu schwer
- **Methodik: heuristische Anwendung von Reduktionsregeln**
  - z.B. $\forall \mathtt{n}{:}\mathbb{Z}.\ \mathtt{n} \geq 1 \Rightarrow \exists \mathtt{M}{:}\mathtt{Set}(\mathbb{Z}).\forall \mathtt{p}{:}\mathtt{Seq}(\mathbb{Z}).\ O(\mathtt{n},\mathtt{p}) \Rightarrow \mathtt{range}(\mathtt{p}) \subseteq \mathtt{M}$
    mit $O(\mathtt{n},\mathtt{p}) \equiv \mathtt{perm}(\mathtt{p},\{1..\mathtt{n}\}) \wedge \forall \mathtt{j} < |\mathtt{p}|.\mathtt{nodups}(\mathtt{dtrow}(\mathtt{p},\mathtt{j}))$
  - Heuristik: Suche Folgerungen von $O(\mathtt{n},\mathtt{p})$, die $\mathtt{range}(\mathtt{p})$ und $\subseteq$ enthalten
    · Auffalten von perm: $\mathtt{perm}(\mathtt{p},\{1..\mathtt{n}\}) \Rightarrow \mathtt{range}(\mathtt{p}) = \{1..\mathtt{n}\}$
    · Mengengleichheit: $S = S' \Rightarrow S \subseteq S'$
    Matching von $O(\mathtt{n},\mathtt{p}) \Rightarrow \mathtt{range}(\mathtt{p}) \subseteq \mathtt{M}$ mit $O(\mathtt{n},\mathtt{p}) \Rightarrow \mathtt{range}(\mathtt{p}) \subseteq \{1..\mathtt{n}\}$
    ist erfolgreicht und liefert $\mathtt{M} \equiv \{1..\mathtt{n}\}$
- **Heuristik steuert Vorwärtsinferenz syntaktisch**
  - Auswahl von Rewrite Regeln aus Lemmas (Äquivalenzen / Implikationen)
  - Ziel ist Erreichen einer bestimmten Formelstruktur
    z.B. Vorkommen bestimmter Formeln, die ein Matching ermöglichen
    Beschränkung der vorkommenden Variablen, Größe der Formel etc.

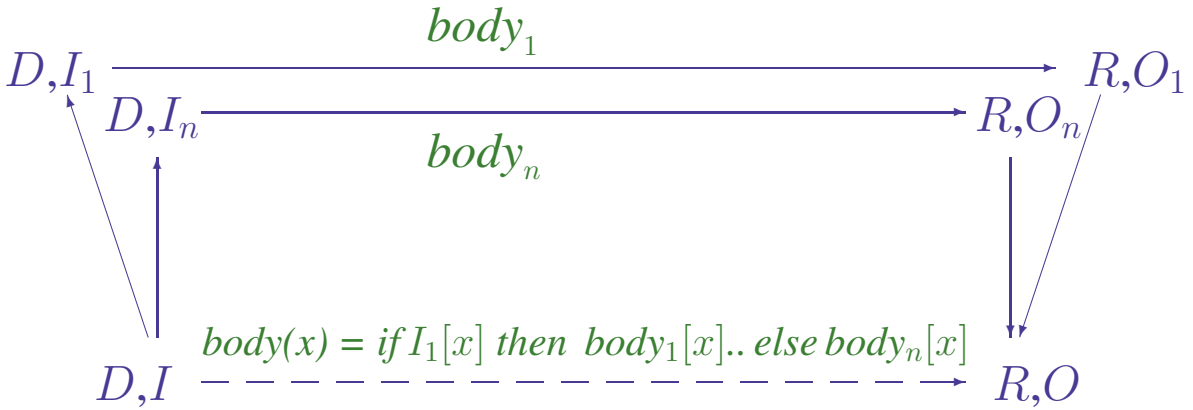
# Einfaches Algorithmenschema: Fallanalyse

## Statische Reduktion durch Zerlegung des Problems

- **Es gibt Lösungen für Teile der Eingabe**
  - Sortieren einer Liste $L$ kann gelöst werden durch Zerteilen von $L$ in $L_1$ und $L_2$, getrenntes sortieren von $L_1$ und $L_2$ und Mischen der Resultate
  - Verfahren terminiert nur wenn $|L_1|<L$ und $|L_2|<L$, also $|L|\geq 2$
  - Lösung für $|L|<2$ ist einfach, da die Liste bereits sortiert ist
- **Formales Konzept: Zerlegen von Spezifikationen**
  - ***spec* zerlegbar in $spec_1$..$spec_n$** $(spec = \cup_i spec_i)$

    $\equiv \forall x{:}D\,.\,I[x] \Rightarrow I_1[x] \vee .. \vee I_n[x] \;\wedge\; \forall i{\leq}n\,.\,\forall y{:}R.\; O_i[x,y] \Rightarrow O[x,y])$
  - Eingabebedingung zerlegbar in Eingabebedingungen der $spec_i$
  - Ausgabebedingung der $spec_i$ muß jeweils stärker sein

# Synthese durch Fallanalyse




- **Zerlegbarkeit liefert Fallunterscheidung**
  - Eine Spezifikation $spec = (D, R, I, O)$ 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 $body_1[x]..$ else $body_n[x]$ korrekt für $spec$
- **Syntheseverfahren suzt sich auf "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

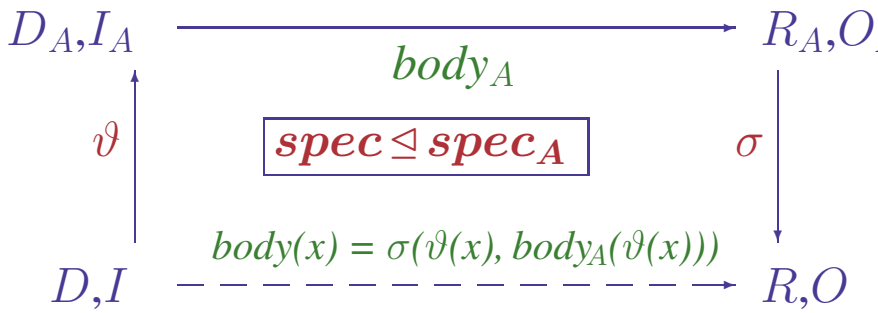
- **Bestimme Vorbedingung $P$ für Gültigkeit einer Formel $F$**
  - Die Formel $F$ enthält möglicherweise Existenzquantoren
    z.B. $F_{append} \equiv \exists L_1, L_2.\ |L|>|L_1| \wedge |L|>|L_2| \wedge L_1 \circ L_2 = L$
  - Vorbedingung $P$ darf quantifizierte Variablen von $F$ nicht enthalten
    z.B. Vorbedingung für $F_{append}$ darf nur die Variable $L$ verwenden
- **Herleitung durch Anwendung von Reduktionsregeln**
  - Rewriting mit Äquivalenzumformungen und bekannten Implikationen
    - Vorbedingung für $\exists L_1, L_2.\ L = L_1 \circ L_2$ ist `True`
    - $L = L_1 \circ L_2$ impliziert $|L| = |L_1| + |L_2|$
    - Vorbedingung für $|L_1| + |L_2| > |L_1|$ ist $|L_2| > 0$ (Arithmetik, analog $|L_1| > 0$)
    - Vorbedingung für $F_{append}$ ist $P[L] \equiv |L| > 1$
- **Steuerung ist Heuristische Vor-/Rückwärtsinferenz**
  - Vorwärts: Äquivalentes Umschreiben von Formeln mit Gleichheiten
  - Rückwärts: Aufsammeln von Bedingungen der Äquivalenzen

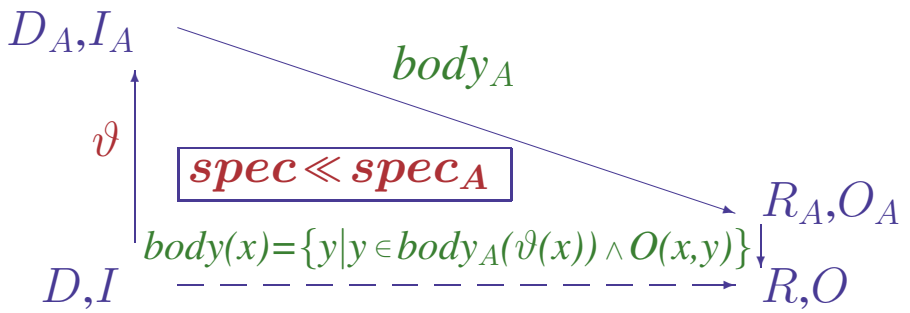
## Modellbildung durch statische Reduktion auf Teilmodelle

- **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 geeignete Algorithmentheorie $\mathcal{A}$ aus der Wissensbank
  - 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 $\sigma$ und $\vartheta$

**Syntheseverfahren erfordert keine weiteren Inferenzen!** ↦ Beispiele folgen



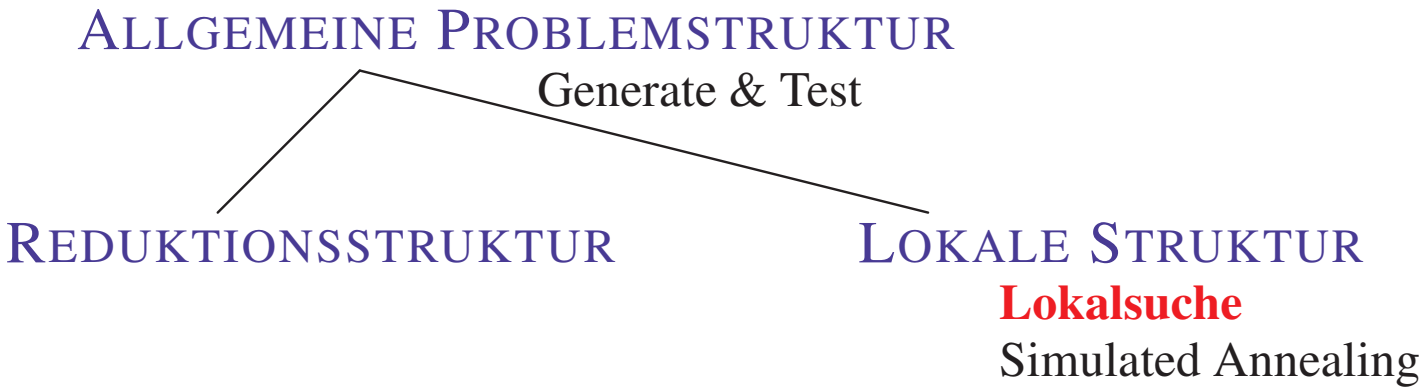



# Hierarchie algorithmischer Strukturen

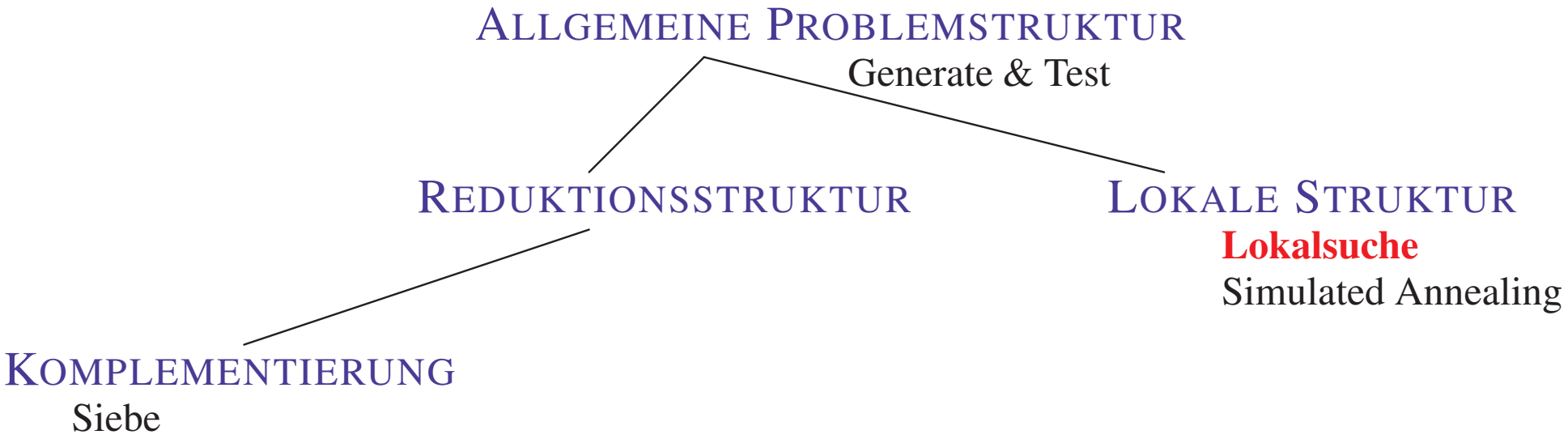
## Allgemeine Problemstruktur

Generate & Test

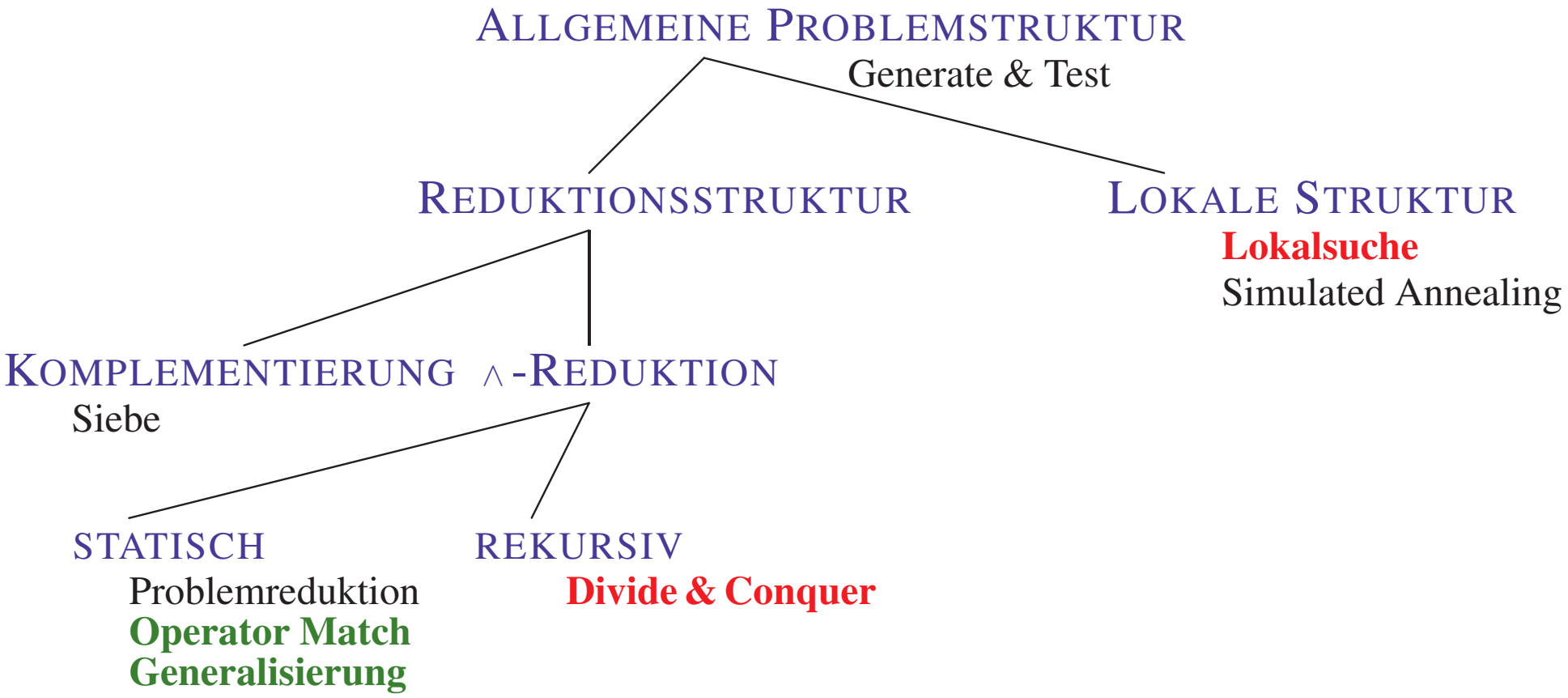
# Hierarchie algorithmischer Strukturen



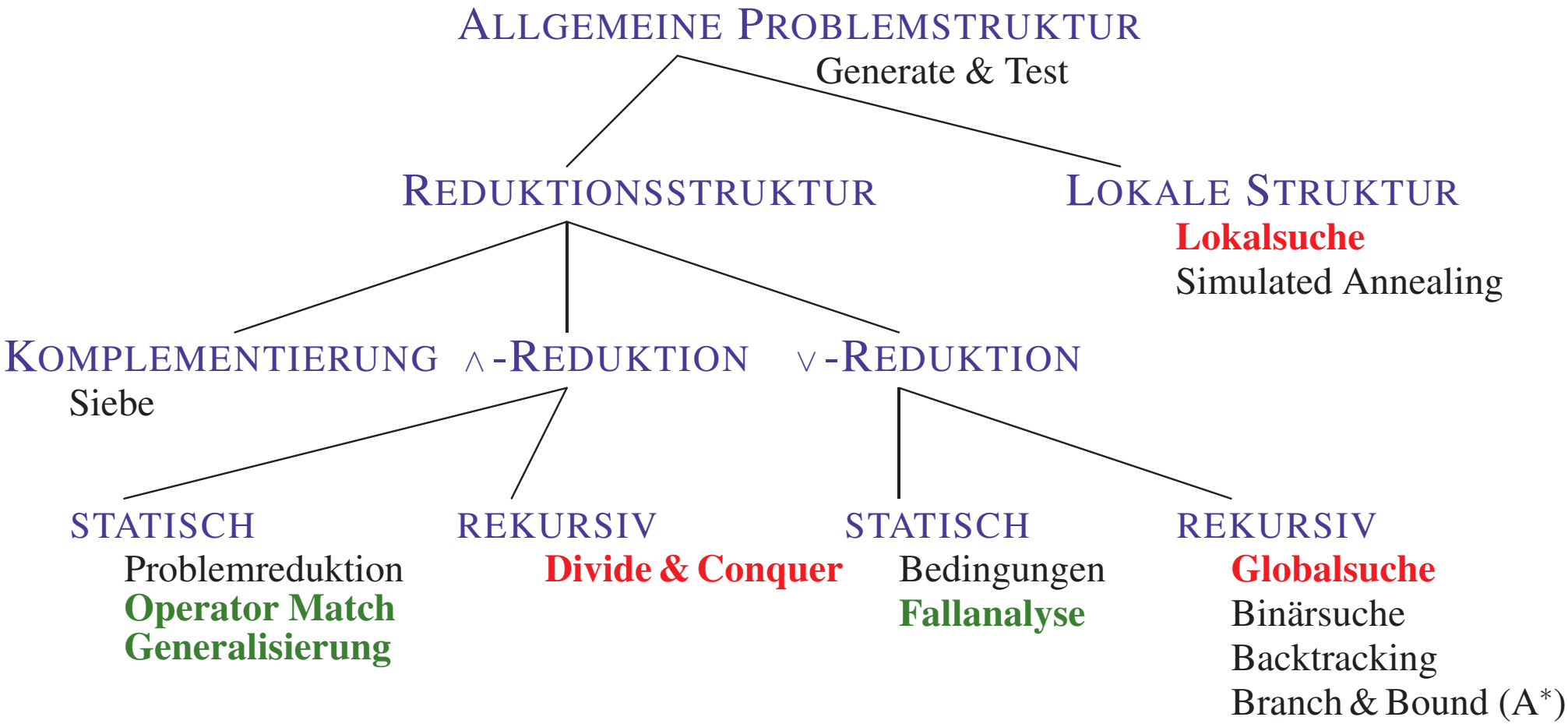
# Hierarchie algorithmischer Strukturen



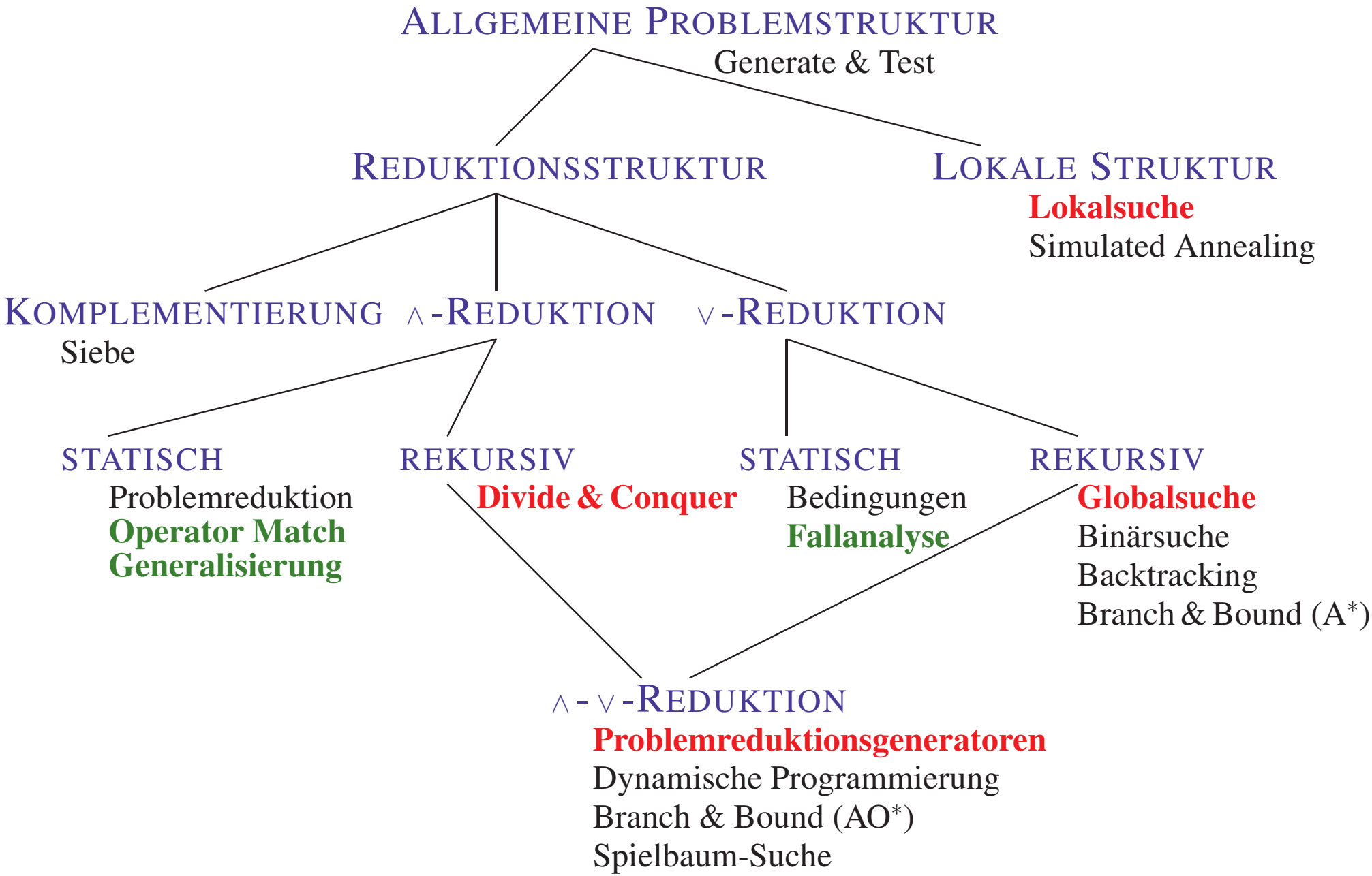
# Hierarchie algorithmischer Strukturen



# Hierarchie algorithmischer Strukturen



# 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