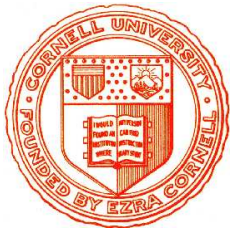
Automatisierte Logik und Programmierung

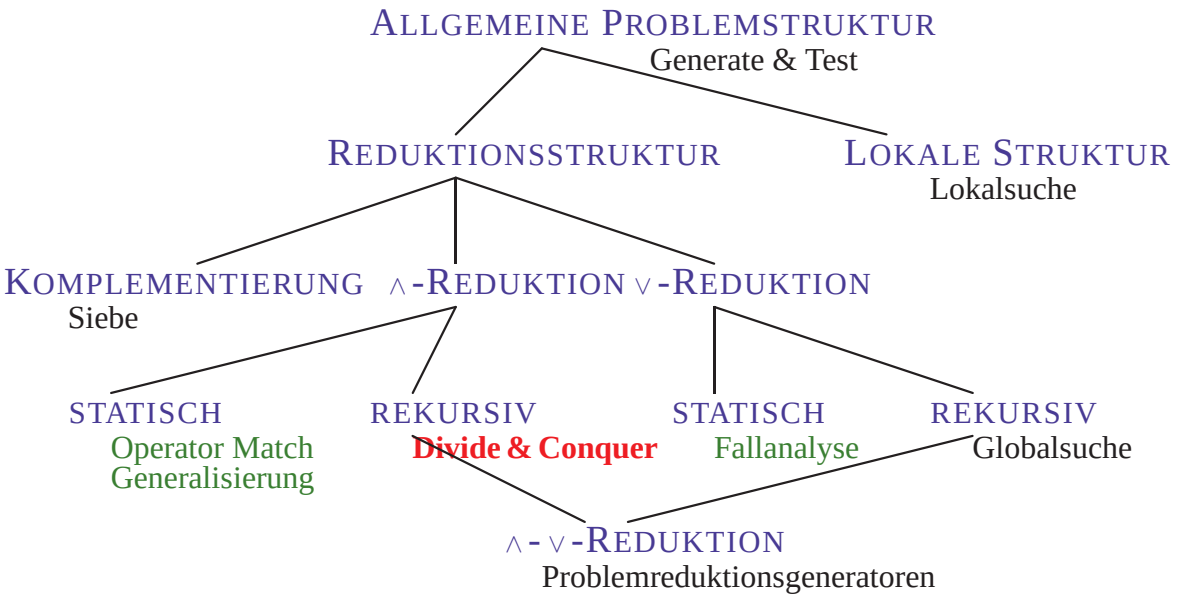
Einheit 18



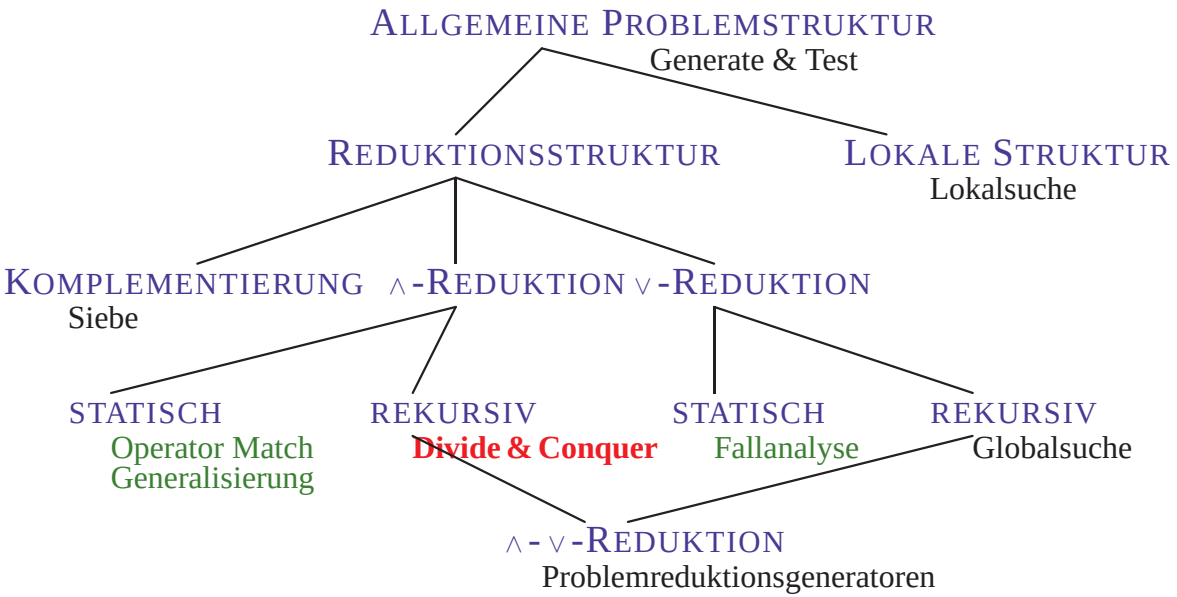
Synthese von
Divide & Conquer Algorithmen

1. Algorithmenschema
2. Korrektheit
3. Synthesestrategie
4. Wissensbasierte Unterstützung

Divide & Conquer Algorithmen

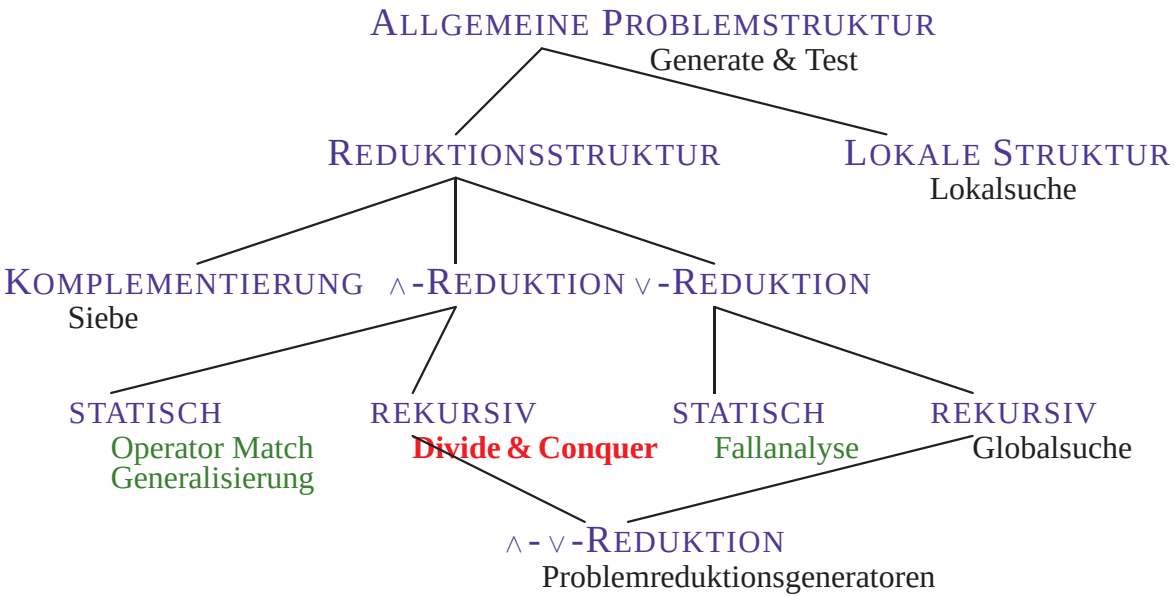


Divide & Conquer Algorithmen



- **Effiziente Verarbeitung strukturierter Daten**
  - Sehr gebräuchliche und einfache Programmiertechnik
  - **Aufteilen**: Zerlege Problem in kleinere Teilprobleme
  - **Erobern**: Löse Teilprobleme separat und setze Lösungen zusammen

Divide & Conquer Algorithmen



- **Effiziente Verarbeitung strukturierter Daten**
  - Sehr gebräuchliche und einfache Programmiertechnik
  - **Aufteilen**: Zerlege Problem in kleinere Teilprobleme
  - **Erobern**: Löse Teilprobleme separat und setze Lösungen zusammen
- **Rekursive $\wedge$-Reduktion des Problems**
  - Lösung benötigt alle Teillösungen gleichzeitig
  - Teillösungen bauen aufeinander auf

Ein typischer Divide & Conquer Algorithmus

- **Maximum einer nichtleeren Liste von Zahlen**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
   SUCH THAT  m∈L ∧ ∀x∈L.x≤m
≡ if |L|=1 then hd(L)
           else let a.L'=L
                in let m'=maxL(L')
                   in if a<m' then m' else a
```

Ein typischer Divide & Conquer Algorithmus

- **Maximum einer nichtleeren Liste von Zahlen**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
   SUCH THAT  m∈L ∧ ∀x∈L.x≤m
≡ if |L|=1 then hd(L)
          else let a.L'=L
               in let m'=maxL(L')
                  in if a<m' then m' else a
```

Einfache (primitive) Eingaben (`|L|=1`) erhalten direkte Lösung `hd(L)`

Ein typischer Divide & Conquer Algorithmus

- **Maximum einer nichtleeren Liste von Zahlen**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
   SUCH THAT  m∈L ∧ ∀x∈L.x≤m
≡ if |L|=1 then hd(L)
          else let a.L'=L
               in let m'=maxL(L')
                  in if a<m' then m' else a
```

Einfache (primitive) Eingaben (`|L|=1`) erhalten direkte Lösung `hd(L)`

Andernfalls Dekomposition der Eingabe mit `HdTl: L ↦ a.L'`

Ein typischer Divide & Conquer Algorithmus

- **Maximum einer nichtleeren Liste von Zahlen**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
   SUCH THAT  m∈L ∧ ∀x∈L.x≤m
≡ if |L|=1 then hd(L)
          else let a.L'=L
                  in let m'=maxL(L')
                     in if a<m' then m' else a
```

Einfache (primitive) Eingaben (`|L|=1`) erhalten direkte Lösung `hd(L)`

Andernfalls Dekomposition der Eingabe mit `HdTl: L ↦ a.L'`

Einfache Teillösung für `a` `a=id(a)`

Rekursive Lösung für `L'` `m'=maxL(L')`

Ein typischer Divide & Conquer Algorithmus

- **Maximum einer nichtleeren Liste von Zahlen**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
   SUCH THAT  m∈L ∧ ∀x∈L.x≤m
≡ if |L|=1 then hd(L)
          else let a.L'=L
               in let m'=maxL(L')
                  in if a<m' then m' else a
```

Einfache (primitive) Eingaben (`|L|=1`) erhalten direkte Lösung `hd(L)`

Andernfalls Dekomposition der Eingabe mit `HdTl: L ↦ a.L'`

Einfache Teillösung für `a` `a=id(a)`

Rekursive Lösung für `L'` `m'=maxL(L')`

Komposition der Teillösungen `a` und `m'` mit der `max`:$\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$

Ein typischer Divide & Conquer Algorithmus

- **Maximum einer nichtleeren Liste von Zahlen**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
   SUCH THAT  m∈L ∧ ∀x∈L.x≤m
≡ if |L|=1 then hd(L)
          else let a.L'=L
                  in let m'=maxL(L')
                     in if a<m' then m' else a
```

Einfache (primitive) Eingaben (`|L|=1`) erhalten direkte Lösung `hd(L)`

Andernfalls Dekomposition der Eingabe mit `HdTl: L ↦ a.L'`

Einfache Teillösung für `a` `a=id(a)`

Rekursive Lösung für `L'` `m'=maxL(L')`

Komposition der Teillösungen `a` und `m'` mit der `max`:$\mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$

- **Vereinheitlichung durch separierte Beschreibung**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
  RETURNS m SUCH THAT m∈L ∧ ∀x∈L.x≤m
≡ if |L|=1 then hd(L) else (max ∘ (id×maxL) ∘ HdTl)(L)
```

Algorithmus zur Verarbeitung der Liste folgt festem Schema

Allgemeines Divide & Conquer Schema

Grundstruktur aller 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)

Allgemeines Divide & Conquer Schema

Grundstruktur aller 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)

- **5 zentrale Komponenten der Algorithmentheorie**
  - *Decompose*: $D \to D' \times D$ Aufspalten der Eingabe in Teilprobleme
  - Rekursiver Aufruf von f zusammen mit Hilfsfunktion g: $D' \to R'$
    - Funktionsprodukt $g \times f(x,y) = (g(x),\ f(y))$
  - *Compose*: $R' \times R \to R$ Zusammensetzen der Teillösungen
  - *Directly-solve*: $D \to R$: Direkte Lösung für einfache Teilprobleme
  - *primitive*: $D \to \mathbb{B}$: Test, ob Eingabe "einfach" ist

Allgemeines Divide & Conquer Schema

Grundstruktur aller 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)

- **5 zentrale Komponenten der Algorithmentheorie**
  - *Decompose*: $D \to D' \times D$ Aufspalten der Eingabe in Teilprobleme
  - Rekursiver Aufruf von f zusammen mit Hilfsfunktion g: $D' \to R'$
    - Funktionsprodukt $g \times f(x,y) = (g(x),\ f(y))$
  - *Compose*: $R' \times R \to R$ Zusammensetzen der Teillösungen
  - *Directly-solve*: $D \to R$: Direkte Lösung für einfache Teilprobleme
  - *primitive*: $D \to \mathbb{B}$: Test, ob Eingabe "einfach" ist

Korrektheit folgt aus wenigen Voraussetzungen

Korrektheit des Divide & Conquer Schemas

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$

ist korrekt, wenn 6 Axiome erfüllt sind

Korrektheit des Divide & Conquer Schemas

FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$
$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$
ist korrekt, wenn 6 Axiome erfüllt sind

---

1. Direkte Lösung korrekt für primitive Eingaben

FUNCTION $f_p(x{:}D){:}R$ WHERE $I[x] \wedge \mathit{primitive}[x]$ RETURNS y SUCH THAT $O[x,y]$

Korrektheit des Divide & Conquer Schemas

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$

ist korrekt, wenn 6 Axiome erfüllt sind

---

1. **Direkte Lösung korrekt für primitive Eingaben**

   FUNCTION $f_p(x{:}D){:}R$ WHERE $I[x] \wedge \mathit{primitive}[x]$ RETURNS y SUCH THAT $O[x,y]$

2. **Ausgabebedingung O rekursiv zerlegbar in O_D, O' und O_C**

   $O_D[x,y',y] \wedge O'[y',z'] \wedge O[y,z] \wedge O_C[z',z,t] \Rightarrow O[x,t]$

   (Strong Problem Reduction Principle)

Korrektheit des Divide & Conquer Schemas

FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$
$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$
ist korrekt, wenn 6 Axiome erfüllt sind

---

1. Direkte Lösung korrekt für primitive Eingaben

FUNCTION $f_p(x{:}D){:}R$ WHERE $I[x] \wedge \mathit{primitive}[x]$ RETURNS y SUCH THAT $O[x,y]$

2. Ausgabebedingung O rekursiv zerlegbar in O_D, O' und O_C

$$O_D[x,y',y] \wedge O'[y',z'] \wedge O[y,z] \wedge O_C[z',z,t] \Rightarrow O[x,t]$$

(Strong Problem Reduction Principle)

3. Dekomposition erfüllt O_D und 'verkleinert' Problem

FUNCTION $f_d(x{:}D){:}D' \times D$ WHERE $I[x] \wedge \neg \mathit{primitive}[x]$
RETURNS y', y SUCH THAT $I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x,y',y]$

Korrektheit des Divide & Conquer Schemas

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$

ist korrekt, wenn 6 Axiome erfüllt sind

---

1. **Direkte Lösung korrekt für primitive Eingaben**

   FUNCTION $f_p(x{:}D){:}R$ WHERE $I[x] \wedge \mathit{primitive}[x]$ RETURNS y SUCH THAT $O[x,y]$

2. **Ausgabebedingung O rekursiv zerlegbar in O_D, O' und O_C**

   $O_D[x,y',y] \wedge O'[y',z'] \wedge O[y,z] \wedge O_C[z',z,t] \Rightarrow O[x,t]$

   (Strong Problem Reduction Principle)

3. **Dekomposition erfüllt O_D und 'verkleinert' Problem**

   FUNCTION $f_d(x{:}D){:}D' \times D$ WHERE $I[x] \wedge \neg \mathit{primitive}[x]$
   RETURNS y', y SUCH THAT $I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x,y',y]$

4. **Hilfsfunktion g erfüllt O'**

   FUNCTION $g(y'{:}D'){:}R'$ WHERE $I'[y']$ RETURNS z' SUCH THAT $O'[y',z']$

Korrektheit des Divide & Conquer Schemas

FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$
$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$
ist korrekt, wenn 6 Axiome erfüllt sind

---

1. **Direkte Lösung korrekt für primitive Eingaben**
   FUNCTION $f_p(x{:}D){:}R$ WHERE $I[x] \wedge \mathit{primitive}[x]$ RETURNS y SUCH THAT $O[x,y]$
2. **Ausgabebedingung O rekursiv zerlegbar in O_D, O' und O_C**
   $O_D[x,y',y] \wedge O'[y',z'] \wedge O[y,z] \wedge O_C[z',z,t] \Rightarrow O[x,t]$
   (Strong Problem Reduction Principle)
3. **Dekomposition erfüllt O_D und 'verkleinert' Problem**
   FUNCTION $f_d(x{:}D){:}D' \times D$ WHERE $I[x] \wedge \neg \mathit{primitive}[x]$
   RETURNS y', y SUCH THAT $I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x,y',y]$
4. **Hilfsfunktion g erfüllt O'**
   FUNCTION $g(y'{:}D'){:}R'$ WHERE $I'[y']$ RETURNS z' SUCH THAT $O'[y',z']$
5. **Komposition erfüllt O_C**
   FUNCTION $f_c(z',z{:}R' \times R){:}R$ WHERE true RETURNS y SUCH THAT $O_C[z',z,t]$

Korrektheit des Divide & Conquer Schemas

FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$
$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$
ist korrekt, wenn 6 Axiome erfüllt sind

---

1. **Direkte Lösung korrekt für primitive Eingaben**
   FUNCTION $f_p(x{:}D){:}R$ WHERE $I[x] \wedge \mathit{primitive}[x]$ RETURNS y SUCH THAT $O[x,y]$
2. **Ausgabebedingung O rekursiv zerlegbar in O_D, O' und O_C**
   $O_D[x,y',y] \wedge O'[y',z'] \wedge O[y,z] \wedge O_C[z',z,t] \Rightarrow O[x,t]$
   (Strong Problem Reduction Principle)
3. **Dekomposition erfüllt O_D und 'verkleinert' Problem**
   FUNCTION $f_d(x{:}D){:}D' \times D$ WHERE $I[x] \wedge \neg \mathit{primitive}[x]$
   RETURNS y', y SUCH THAT $I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x,y',y]$
4. **Hilfsfunktion g erfüllt O'**
   FUNCTION $g(y'{:}D'){:}R'$ WHERE $I'[y']$ RETURNS z' SUCH THAT $O'[y',z']$
5. **Komposition erfüllt O_C**
   FUNCTION $f_c(z',z{:}R' \times R){:}R$ WHERE true RETURNS y SUCH THAT $O_C[z',z,t]$
6. **Verkleinerungsrelation $\succ$ ist wohlfundierte Ordnung auf D**
   Sechste Komponente der Theorie, nötig nur für Terminierungsbeweis

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if $\mathit{primitive}[x]$ then $\mathit{Directly\text{-}solve}[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

FUNCTION $f_p(x{:}D){:}R$ WHERE $I[x] \wedge \mathit{primitive}[x]$ RETURNS y SUCH THAT $O[x,y]$

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if $\mathit{primitive}[x]$ then $\mathit{Directly\text{-}solve}[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

  Nichtprimitive Eingaben: $I[x] \wedge \neg\mathit{primitive}[x]$

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

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

  Nichtprimitive Eingaben: $I[x] \wedge \neg \mathit{primitive}[x]$

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

  – *Decompose*$[x]$ liefert y', y mit $O_D[x, y', y]$ und $x \succ y$ Axiom 3

FUNCTION $f_d(x{:}D){:}D' \times D$ WHERE $I[x] \wedge \neg \mathit{primitive}[x]$
RETURNS y', y SUCH THAT $I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x, y', y]$

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

  Nichtprimitive Eingaben: $I[x] \wedge \neg \mathit{primitive}[x]$

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

  – *Decompose*$[x]$ liefert y', y mit $O_D[x, y', y]$ und $x \succ y$ Axiom 3

  – $g(y')$ liefert z' mit $O'[y', z']$ Axiom 4

  FUNCTION $g(y'{:}D'){:}R'$ WHERE $I'[y']$ RETURNS z' SUCH THAT $O'[y', z']$

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if $\mathit{primitive}[x]$ then $\mathit{Directly\text{-}solve}[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

  Nichtprimitive Eingaben: $I[x] \wedge \neg\mathit{primitive}[x]$

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

  – $\mathit{Decompose}[x]$ liefert y', y mit $O_D[x, y', y]$ und $x \succ y$ Axiom 3

  – $g(y')$ liefert z' mit $O'[y', z']$ Axiom 4

  – $f(y)$ liefert z mit $O[y, z]$ Induktionsannahme

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if $\mathit{primitive}[x]$ then $\mathit{Directly\text{-}solve}[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

  Nichtprimitive Eingaben: $I[x] \wedge \neg\mathit{primitive}[x]$

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

  – $\mathit{Decompose}[x]$ liefert y', y mit $O_D[x, y', y]$ und $x \succ y$ Axiom 3

  – $g(y')$ liefert z' mit $O'[y', z']$ Axiom 4

  – $f(y)$ liefert z mit $O[y, z]$ Induktionsannahme

  – $\mathit{Compose}[z', z]$ liefert t mit $O_C[z', z, t]$ Axiom 5

FUNCTION $f_c(z', z{:}R' \times R){:}R$ WHERE true RETURNS y SUCH THAT $O_C[z', z, t]$

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if $\mathit{primitive}[x]$ then $\mathit{Directly\text{-}solve}[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über $(D, \succ)$**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

  Nichtprimitive Eingaben: $I[x] \wedge \neg\mathit{primitive}[x]$

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

  – $\mathit{Decompose}[x]$ liefert y', y mit $O_D[x, y', y]$ und $x \succ y$ Axiom 3

  – $g(y')$ liefert z' mit $O'[y', z']$ Axiom 4

  – $f(y)$ liefert z mit $O[y, z]$ Induktionsannahme

  – $\mathit{Compose}[z', z]$ liefert t mit $O_C[z', z, t]$ Axiom 5

  – t ist das Ergebnis ($f(x) = t$) und es gilt $O[x, t]$ Axiom 2

$$\boldsymbol{O_D[x, y', y] \wedge O'[y', z'] \wedge O[y, z] \wedge O_C[z', z, t] \Rightarrow O[x, t]}$$

Divide & Conquer Schema: Korrektheitsbeweis

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

$\equiv$ if $\mathit{primitive}[x]$ then $\mathit{Directly\text{-}solve}[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über (D,$\succ$)**

  Primitive Eingaben: $I[x] \wedge \mathit{primitive}[x]$

  – $f(x) = \mathit{Directly\text{-}solve}[x]$ Korrektheit folgt direkt aus Axiom 1

  Nichtprimitive Eingaben: $I[x] \wedge \neg\mathit{primitive}[x]$

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

  – $\mathit{Decompose}[x]$ liefert y', y mit $O_D[x, y', y]$ und $x \succ y$ Axiom 3

  – $g(y')$ liefert z' mit $O'[y', z']$ Axiom 4

  – $f(y)$ liefert z mit $O[y, z]$ Induktionsannahme

  – $\mathit{Compose}[z', z]$ liefert t mit $O_C[z', z, t]$ Axiom 5

  – t ist das Ergebnis ($f(x) = t$) und es gilt $O[x, t]$ Axiom 2

- **Terminierung: Wohlfundiertheit von $\succ$** Axiom 6

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

$O_D(x,y',y) \wedge O(y,z) \wedge O'(y',z') \wedge O_C(z,z',t) \Rightarrow O(x,t)$,

`FUNCTION` $F_d(x:D):D' \times D$ `WHERE` $I[x] \wedge \neg$*primitive*$[x]$ `RETURNS` y', y
`SUCH THAT` $I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x,y',y] \equiv$ *Decompose*$[x]$ ist korrekt

`FUNCTION` $F_c(z,z':R \times R'):R$
`RETURNS` t `SUCH THAT` $O_C[z,z',t] \equiv$ *Compose*$[z,z']$ ist korrekt

`FUNCTION` $F_G(y':D'):R'$ `WHERE` $I'[y']$
`RETURNS` z' `SUCH THAT` $O'[y',z'] \equiv g[y']$ ist korrekt

$\succ$ *ist wohlfundierte Ordnung auf* D

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

$O_D(x,y',y) \wedge O(y,z) \wedge O'(y',z') \wedge O_C(z,z',t) \Rightarrow O(x,t)$,

`FUNCTION` $F_d(x:D):D' \times D$ `WHERE` $I[x] \wedge \neg$*primitive*$[x]$ `RETURNS` y', y
`SUCH THAT` $I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x,y',y] \equiv$ *Decompose*$[x]$ ist korrekt

`FUNCTION` $F_c(z,z':R \times R'):R$
`RETURNS` t `SUCH THAT` $O_C[z,z',t] \equiv$ *Compose*$[z,z']$ ist korrekt

`FUNCTION` $F_G(y':D'):R'$ `WHERE` $I'[y']$
`RETURNS` z' `SUCH THAT` $O'[y',z'] \equiv g[y']$ ist korrekt

$\succ$ *ist wohlfundierte Ordnung auf* D

Für jedes Modell A der Divide & Conquer Theorie ist folgender Algorithmus eine korrekte Lösung

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

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen

Spezifikation: `FUNCTION minL(L:Seq(ℤ)):ℤ WHERE L≠[]`
`SUCH THAT m∈L ∧ ∀x∈L.m≤x`

Ziel: Bestimme Komponenten des D&C Algorithmenschemas für `minL`

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen

Spezifikation: `FUNCTION minL(L:Seq(ℤ)):ℤ WHERE L≠[]`
`SUCH THAT m∈L ∧ ∀x∈L.m≤x`

Ziel: Bestimme Komponenten des D&C Algorithmenschemas für `minL`

1. Es gibt nur wenige sinnvolle Arten, die Eingabe zu zerlegen

– Wissensbank sollte mögliche Zerlegungen von Listen speichern

– Wähle Zerlegung in Anfang (`hd`) und Rest (`tl`)

– Liefert Komponente *Decompose* $\equiv$ `HdTl: Seq(`α`)→`$\alpha\times$`Seq(`α`)`,
mit Extra-Domäne $D' \equiv \mathbb{Z}$, Ausgabebedingung $O_D[L, a, L'] \equiv L{=}a.L'$
und wohlfundierte Ordnung $L \succ L' \equiv |L| > |L'|$ $\mapsto$ **Axiom 6**

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen

Spezifikation: `FUNCTION minL(L:Seq(ℤ)):ℤ WHERE L≠[]`
`SUCH THAT m∈L ∧ ∀x∈L. m≤x`

Ziel: Bestimme Komponenten des D&C Algorithmenschemas für `minL`

1. Es gibt nur wenige sinnvolle Arten, die Eingabe zu zerlegen

- Wissensbank sollte mögliche Zerlegungen von Listen speichern
- Wähle Zerlegung in Anfang (`hd`) und Rest (`tl`)
- Liefert Komponente *Decompose* $\equiv$ `HdTl: Seq(`α`)→`$\alpha\times$`Seq(`α`)`, mit Extra-Domäne $D' \equiv \mathbb{Z}$, Ausgabebedingung $O_D[L, a, L'] \equiv L{=}a.L'$ und wohlfundierte Ordnung $L \succ L' \equiv |L| > |L'|$ $\mapsto$ **Axiom 6**

2. Hilfsfunktion g muß konstruiert werden

- Auf Ausgabe von *Decompose* muß $g\times$`minL` angewandt werden
- Auf Resultat von $g\times$`minL` muß *Compose* angewandt werden
- Hilfsfunktion g muß auf $D' \equiv \mathbb{Z}$ operieren, also auf Elementen der Liste
- Da *Compose* ohnehin synthetisiert wird, ist "nichts tun" die beste Wahl
- Liefert Komponente $\boldsymbol{g} \equiv id{:}\mathbb{Z}\to\mathbb{Z}$ mit Eingabebedingung $I'[a] \equiv$ `true`, Bildbereich $R' \equiv \mathbb{Z}$ und Ausgabebedingung $O'[a, a'] \equiv a'{=}a$ $\mapsto$ **Axiom 4**

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen (2)

3. Mit g kann Korrektheit von *Decompose* geprüft werden $\mapsto$ **Axiom 3**

– Für nichtprimitive Listen muß $I'[a] \wedge I[L'] \wedge\ L \succ L'\ \wedge O_D[L, a, L']$ gelten, also $\texttt{true} \wedge L' \neq \texttt{[]} \wedge \texttt{|}L\texttt{|>|}L'\texttt{|} \wedge L = a.L'$, wobei $\texttt{HdTl(}L\texttt{)=(}a, L'\texttt{)}$

– Formel muß transformiert werden in (hinreichende) Bedingung an L

– $L' \neq \texttt{[]} \wedge \texttt{|}L\texttt{|>|}L'$ ist äquivalent zu $\texttt{|}L\texttt{|>1}$

– Vorbedingung $L \neq \texttt{[]}$ liefert Komponente *primitive(L)* $\equiv$ $\texttt{|}L\texttt{|=1}$

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen (2)

3. Mit g kann Korrektheit von *Decompose* geprüft werden $\mapsto$ **Axiom 3**

– Für nichtprimitive Listen muß $I'[a] \wedge I[L'] \wedge\ L \succ L'\ \wedge O_D[L, a, L']$ gelten, also $\texttt{true} \wedge L' \neq \texttt{[]} \wedge \texttt{|}L\texttt{|>|}L'\texttt{|} \wedge L = a.L'$, wobei $\texttt{HdTl(}L\texttt{)=(}a, L'\texttt{)}$

– Formel muß transformiert werden in (hinreichende) Bedingung an L

– $L' \neq \texttt{[]} \wedge \texttt{|}L\texttt{|>|}L'$ ist äquivalent zu $\texttt{|}L\texttt{|>1}$

– Vorbedingung $L \neq \texttt{[]}$ liefert Komponente *primitive(L)* $\equiv\ \texttt{|}L\texttt{|=1}$

4. Axiom 2 liefert Ausgabebedingung von *Compose* $\mapsto$ **Axiom 2**

– Vereinfache $L = a.L'\ \wedge\ a' = a\ \wedge\ (m' \in L' \wedge \forall x \in L'.\, m' \leq x)\ \wedge\ O_C[a', m', m]$
$\Rightarrow\ m \in L\ \wedge\ \forall x \in L.\, m \leq x$ zu Bedingung an m

– Möglich sind $m = m'$ und $m = a$: vereinfache $\forall x \in L.\, m \leq x$ für diese Fälle

– Liefert $O_C[a, m', m]\ \equiv\ (m = m' \wedge m' \leq a) \vee (m = a \wedge a \leq m')$

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen (2)

3. Mit g kann Korrektheit von *Decompose* geprüft werden ↦ **Axiom 3**

– Für nichtprimitive Listen muß $I'[a] \wedge I[L'] \wedge L \succ L' \wedge O_D[L, a, L']$ gelten, also $\texttt{true} \wedge L' \neq \texttt{[]} \wedge \texttt{|}L\texttt{|>|}L'\texttt{|} \wedge L = a.L'$, wobei $\texttt{HdTl(}L\texttt{)=(}a, L'\texttt{)}$

– Formel muß transformiert werden in (hinreichende) Bedingung an L

– $L' \neq \texttt{[]} \wedge \texttt{|}L\texttt{|>|}L'$ ist äquivalent zu $\texttt{|}L\texttt{|>1}$

– Vorbedingung $L \neq \texttt{[]}$ liefert Komponente *primitive(L)* $\equiv$ $\texttt{|}L\texttt{|=1}$

4. Axiom 2 liefert Ausgabebedingung von *Compose* ↦ **Axiom 2**

– Vereinfache $L = a.L' \wedge a' = a \wedge (m' \in L' \wedge \forall x \in L'.\, m' \leq x) \wedge O_C[a', m', m]$
$\Rightarrow m \in L \wedge \forall x \in L.\, m \leq x$ zu Bedingung an m

– Möglich sind $m = m'$ und $m = a$: vereinfache $\forall x \in L.\, m \leq x$ für diese Fälle

– Liefert $O_C[a, m', m] \equiv (m = m' \wedge m' \leq a) \vee (m = a \wedge a \leq m')$

5. Erneute Synthese liefert Algorithmus für *Compose* ↦ **Axiom 5**

– Disjunktion legt Algorithmenschema Fallunterscheidung nahe

– Lösung ist trivial in beiden Fällen $m' \leq a$ bzw. $a \leq m'$

– Liefert *Compose(a,m')* $\equiv$ $\texttt{if}\ m' \leq a\ \texttt{then}\ m'\ \texttt{else}\ a$

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen (3)

6. Synthese liefert Algorithmus für *Directly-solve* ↦ **Axiom 1**

- Vereinfache $m \in L \wedge \forall x \in L.\, m \leq x$ im Kontext $\texttt{|}L\texttt{|=1}$
- Aus $\texttt{|}L\texttt{|=1}$ folgt $L\texttt{=[hd(}L\texttt{)]}$
- $\forall x \in \texttt{[hd(}L\texttt{)].}\ m \leq x$ ist äquivalent zu $m \leq \texttt{hd(}L\texttt{)}$
- Aus $m \in \texttt{[hd(}L\texttt{)]}$ folgt $m\texttt{=hd(}L\texttt{)}$
- Liefert Komponente *Directly-solve*$(L) \equiv \texttt{hd(}L\texttt{)}$

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen (3)

6. Synthese liefert Algorithmus für *Directly-solve* $\mapsto$ **Axiom 1**

- Vereinfache $m \in L \wedge \forall x \in L.\, m \leq x$ im Kontext $|L|=1$
- Aus $|L|=1$ folgt $L=[hd(L)]$
- $\forall x \in [hd(L)].\, m \leq x$ ist äquivalent zu $m \leq hd(L)$
- Aus $m \in [hd(L)]$ folgt $m=hd(L)$
- Liefert Komponente *Directly-solve*$(L) \equiv hd(L)$

7. Alle Komponenten sind bestimmt, alle Axiome geprüft

- Instantiiere das Divide & Conquer Schema für `minL`

```
FUNCTION minL(L:Seq(ℤ)):ℤ WHERE L≠[]
      SUCH THAT  m∈L  ∧  ∀x∈L.m≤x
   ≡ if |L|=1 then hd(L)
               else let a.L'=L
                    in let m'= minL(L')
                       in if m'≤a then m' else a
```

Synthese eines Divide & Conquer Algorithmus
Minimum einer nichtleeren Liste von Zahlen (3)

6. Synthese liefert Algorithmus für *Directly-solve* ↦ **Axiom 1**

- Vereinfache $m \in L \wedge \forall x \in L.\, m \leq x$ im Kontext $|L|=1$
- Aus $|L|=1$ folgt $L=[hd(L)]$
- $\forall x \in [hd(L)].\, m \leq x$ ist äquivalent zu $m \leq hd(L)$
- Aus $m \in [hd(L)]$ folgt $m=hd(L)$
- Liefert Komponente *Directly-solve*$(L) \equiv hd(L)$

7. Alle Komponenten sind bestimmt, alle Axiome geprüft

- Instantiiere das Divide & Conquer Schema für `minL`

```
FUNCTION minL(L:Seq(ℤ)):ℤ WHERE L≠[]
      SUCH THAT  m∈L  ∧  ∀x∈L.m≤x
   ≡ if |L|=1 then hd(L)
               else let a.L'=L
                    in let m'= minL(L')
                       in if m'≤a then m' else a
```

Alle durchgeführten Schritte sind automatisierbar

Synthesestrategie für Divide & Conquer

Zerlege Problem in Spezifikationen für Teilprobleme

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

Synthesestrategie für Divide & Conquer

Zerlege Problem in Spezifikationen für Teilprobleme

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

1. Wähle $\succ$ und *Decompose* aus Wissensbank $\mapsto$ O_D,D', **Axiom 6**

Synthesestrategie für Divide & Conquer

Zerlege Problem in Spezifikationen für Teilprobleme

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

1. Wähle $\succ$ und *Decompose* aus Wissensbank $\mapsto$ **O_D,D', Axiom 6**

2. Konstruiere Hilfsfunktion g: $\mapsto$ **O', I', Axiom 4**

– Heuristik: $g{:=}f$, falls $D'{=}D$, sonst $g{:=}id$

Synthesestrategie für Divide & Conquer

Zerlege Problem in Spezifikationen für Teilprobleme

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

1. Wähle $\succ$ und *Decompose* aus Wissensbank $\mapsto$ O_D,D', **Axiom 6**

2. Konstruiere Hilfsfunktion g: $\mapsto$ O', I', **Axiom 4**

– Heuristik: $g{:=}f$, falls $D'{=}D$, sonst $g{:=}id$

3. Verifiziere *Decompose*, generiere Vorbedingung $\mapsto$ *primitive*, **Axiom 3**

– Heuristik: abgeleitete zusätzliche Vorbedingung für O_D ist $\neg primitive[x]$

Synthesestrategie für Divide & Conquer

Zerlege Problem in Spezifikationen für Teilprobleme

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

1. Wähle $\succ$ und *Decompose* aus Wissensbank $\mapsto$ **O_D,D', Axiom 6**

2. Konstruiere Hilfsfunktion g: $\mapsto$ **O', I', Axiom 4**

– Heuristik: $g{:=}f$, falls $D'{=}D$, sonst $g{:=}id$

3. Verifiziere *Decompose*, generiere Vorbedingung $\mapsto$ ***primitive*, Axiom 3**

– Heuristik: abgeleitete zusätzliche Vorbedingung für O_D ist $\neg primitive[x]$

4. Konstruiere *Compose* $\mapsto$ **O_C, Axiome 5 & 2**

– Heuristik: Erzeuge O_C mit Axiom 2;
Synthetisiere *Compose* gemäß Axiom 5

Synthesestrategie für Divide & Conquer

Zerlege Problem in Spezifikationen für Teilprobleme

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

1. Wähle $\succ$ und *Decompose* aus Wissensbank $\mapsto$ **O_D,D', Axiom 6**

2. Konstruiere Hilfsfunktion g: $\mapsto$ **O', I', Axiom 4**

– Heuristik: $g{:=}f$, falls $D'{=}D$, sonst $g{:=}id$

3. Verifiziere *Decompose*, generiere Vorbedingung $\mapsto$ ***primitive***, **Axiom 3**

– Heuristik: abgeleitete zusätzliche Vorbedingung für O_D ist $\neg primitive[x]$

4. Konstruiere *Compose* $\mapsto$ **O_C, Axiome 5 & 2**

– Heuristik: Erzeuge O_C mit Axiom 2;
Synthetisiere *Compose* gemäß Axiom 5

5. Konstruiere *Directly-solve* $\mapsto$ **Axiom 1**

– Heuristik: Suche nach vorgefertigten Lösungen, sonst neue Synthese

– Falls dies nicht möglich ist, konstruiere eingeschränkte Vorbedingung $\hat{I}$

Synthesestrategie für Divide & Conquer

Zerlege Problem in Spezifikationen für Teilprobleme

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

1. Wähle $\succ$ und *Decompose* aus Wissensbank $\mapsto$ **O_D,D', Axiom 6**

2. Konstruiere Hilfsfunktion g: $\mapsto$ **O', I', Axiom 4**

– Heuristik: $g{:=}f$, falls $D'{=}D$, sonst $g{:=}id$

3. Verifiziere *Decompose*, generiere Vorbedingung $\mapsto$ ***primitive***, **Axiom 3**

– Heuristik: abgeleitete zusätzliche Vorbedingung für O_D ist $\neg primitive[x]$

4. Konstruiere *Compose* $\mapsto$ **O_C, Axiome 5 & 2**

– Heuristik: Erzeuge O_C mit Axiom 2;
Synthetisiere *Compose* gemäß Axiom 5

5. Konstruiere *Directly-solve* $\mapsto$ **Axiom 1**

– Heuristik: Suche nach vorgefertigten Lösungen, sonst neue Synthese

– Falls dies nicht möglich ist, konstruiere eingeschränkte Vorbedingung $\hat{I}$

6. Instantiiere das Divide & Conquer Schema

Man kann auch in anderer Reihenfolge vorgehen

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

$\equiv$ if *primitive*$[x]$ then *Directly-solve*$[x]$ else $(\mathit{Compose} \circ g \times f \circ \mathit{Decompose})(x)$

Man kann auch in anderer Reihenfolge vorgehen

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

- **Grundstrategie**
  - Wähle $\succ$ und *Decompose* aus Wissensbank
  - Konstruiere g heuristisch
  - Bestimme *primitive* durch Verifikation von *Decompose*
  - Konstruiere Spezifikation und Lösung für *Compose*
  - Konstruiere *Directly-solve*

Man kann auch in anderer Reihenfolge vorgehen

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

- **Grundstrategie**
  - Wähle $\succ$ und *Decompose* aus Wissensbank
  - Konstruiere g heuristisch
  - Bestimme *primitive* durch Verifikation von *Decompose*
  - Konstruiere Spezifikation und Lösung für *Compose*
  - Konstruiere *Directly-solve*
- **Variante**
  - Wähle $\succ$ und *Decompose* aus Wissensbank
  - Konstruiere *Compose* heuristisch
  - Konstruiere Spezifikation und Lösung für g
  - Bestimme *primitive* und konstruiere *Directly-solve*

Man kann auch in anderer Reihenfolge vorgehen

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

- **Grundstrategie**
  - Wähle $\succ$ und *Decompose* aus Wissensbank
  - Konstruiere g heuristisch
  - Bestimme *primitive* durch Verifikation von *Decompose*
  - Konstruiere Spezifikation und Lösung für *Compose*
  - Konstruiere *Directly-solve*
- **Variante**
  - Wähle $\succ$ und *Decompose* aus Wissensbank
  - Konstruiere *Compose* heuristisch
  - Konstruiere Spezifikation und Lösung für g
  - Bestimme *primitive* und konstruiere *Directly-solve*
- **Umgekehrte Strategie**
  - Wähle *Compose* aus Wissensbank
  - Konstruiere g heuristisch
  - Konstruiere Spezifikation und Lösung für *Decompose*; bestimme $\succ$
  - Bestimme *primitive* und konstruiere *Directly-solve*

Synthese eines Sortieralgorithmus
Verwendung der umgekehrten Divide & Conquer Synthese

Spezifikation:
```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true
    RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)
```

Synthese eines Sortieralgorithmus
Verwendung der umgekehrten Divide & Conquer Synthese

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. Wähle *Compose*$[a', S'] \equiv a'.S'$ ($O_C[a', S', S] \equiv S{=}a'.S'$, $R' \equiv \mathbb{Z}$)

Synthese eines Sortieralgorithmus
Verwendung der umgekehrten Divide & Conquer Synthese

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. Wähle *Compose*$[a', S'] \equiv a'.S'$ ($O_C[a', S', S] \equiv S{=}a'.S'$, $R' \equiv \mathbb{Z}$)

2. Konstruiere Hilfsfunktion $g \equiv$ `id` ($O'[a, a'] \equiv a{=}a'$, $I'[L'] \equiv$ `true`)

Synthese eines Sortieralgorithmus
Verwendung der umgekehrten Divide & Conquer Synthese

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. Wähle *Compose*$[a', S'] \equiv a'.S'$ ($O_C[a', S', S] \equiv S{=}a'.S'$, $R' \equiv \mathbb{Z}$)

2. Konstruiere Hilfsfunktion $g \equiv$ `id` ($O'[a, a'] \equiv a{=}a'$, $I'[L'] \equiv$ `true`)

3. Konstruiere *Decompose* **und** $\succ$ **mit Axiomen 2 und 3**

– $O_D[L, a, L'] \wedge a{=}a' \wedge L \succ L' \wedge$ `rearranges(`L', S'`)` $\wedge$ `ordered(`S'`)` $\wedge\ S{=}a'.S'$
$\Rightarrow$ `rearranges(`L, S`)` $\wedge$ `ordered(`S`)`

– Liefert als Spezifikation $O_D[L, a, L'] \equiv a \in L \wedge \forall x \in L.\, a' \leq x \wedge L' = L - a$,
als Lösung *Decompose(L)* $\equiv$ `let a=minL(`L`) in (a,`L`-a)`
als wohlfundierte Ordnung $L \succ L' \equiv$ `|`L`|>|`L'`|`

Synthese eines Sortieralgorithmus
Verwendung der umgekehrten Divide & Conquer Synthese

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. Wähle *Compose*$[a', S'] \equiv a'.S'$ ($O_C[a', S', S] \equiv S{=}a'.S'$, $R' \equiv \mathbb{Z}$)

2. Konstruiere Hilfsfunktion $g \equiv$ `id` ($O'[a, a'] \equiv a{=}a'$, $I'[L'] \equiv$ `true`)

3. Konstruiere *Decompose* **und** $\succ$ **mit Axiomen 2 und 3**

– $O_D[L, a, L'] \wedge a{=}a' \wedge L \succ L' \wedge$ `rearranges(`L', S'`)` $\wedge$ `ordered(`S'`)` $\wedge S{=}a'.S'$
$\Rightarrow$ `rearranges(`L, S`)` $\wedge$ `ordered(`S`)`

– Liefert als Spezifikation $O_D[L, a, L'] \equiv a \in L \wedge \forall x \in L.\, a' \leq x \wedge L' = L - a$,
als Lösung *Decompose(L)* $\equiv$ `let a=minL(`L`) in (a,`L`-a)`
als wohlfundierte Ordnung $L \succ L' \equiv$ `|`L`|>|`L'`|`

4. Vorbedingung für Korrektheit von *Decompose* **ist** $L \neq$ `[]`

– Liefert *primitive*$[L] \equiv L{=}$`[]` und *Directly-solve*$[L] \equiv$ `[]`

Synthese eines Sortieralgorithmus
Verwendung der umgekehrten Divide & Conquer Synthese

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. Wähle *Compose*$[a', S'] \equiv a'.S'$ ($O_C[a', S', S] \equiv S{=}a'.S'$, $R' \equiv \mathbb{Z}$)

2. Konstruiere Hilfsfunktion $g \equiv$ `id` ($O'[a, a'] \equiv a{=}a'$, $I'[L'] \equiv$ `true`)

3. Konstruiere *Decompose* **und** $\succ$ **mit Axiomen 2 und 3**

– $O_D[L, a, L'] \wedge a{=}a' \wedge L \succ L' \wedge$ `rearranges(`L', S'`)` $\wedge$ `ordered(`S'`)` $\wedge\ S{=}a'.S'$
$\Rightarrow$ `rearranges(`L, S`)` $\wedge$ `ordered(`S`)`

– Liefert als Spezifikation $O_D[L, a, L'] \equiv a \in L \wedge \forall x \in L.\, a' \leq x \wedge L' = L - a$,
als Lösung *Decompose*$(L) \equiv$ `let a=minL(`L`) in (a,`L`-a)`
als wohlfundierte Ordnung $L \succ L' \equiv$ `|`L`|>|`L'`|`

4. Vorbedingung für Korrektheit von *Decompose* **ist** $L \neq$ `[]`

– Liefert *primitive*$[L] \equiv L{=}$`[]` und *Directly-solve*$[L] \equiv$ `[]`

6. Instantiiere das Divide & Conquer Schema

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) ....
≡ if L=[]then[] else let a=minL(L), L'=L-a in a.sort(L')
```

Wichtiges Programmierwissen für D&C-Synthesen

- **Standard-Zerlegungen für Typen (D)** $\mapsto$ *Decompose*,O_D,*D'*
  - Endliche Listen / Folgen (`Seq(`α`)`): `HdTl, ListSplit`
    - `ListSplit(`L`)` $\equiv$ `([L`$_i$`|i`$\in$`[1..|L|÷2]],[L`$_i$`|i`$\in$`[1+|L|÷2..|L|]])`
      mit $O_D[L, L_1, L_2] \equiv L{=}L_1 \circ L_2$, $D' \equiv \text{Seq}(\alpha)$
  - Endliche Mengen (`Set(`α`)`): `ArbRest`
  - Produkträume (+ Mengen, Folgen, ...): Zerlegung in Einzelkomponenten
  - Natürlichen Zahlen ($\mathbb{N}$): Vorgängerfunktion

Wichtiges Programmierwissen für D&C-Synthesen

- **Standard-Zerlegungen für Typen (D)** $\mapsto$ *Decompose*,O_D,*D'*
  - Endliche Listen / Folgen (`Seq(`α`)`): `HdTl, ListSplit`
    - `ListSplit(`L`)` $\equiv$ `([L`$_i$`|i`$\in$`[1..|L|÷2]],[L`$_i$`|i`$\in$`[1+|L|÷2..|L|]])`
      mit $O_D[L, L_1, L_2] \equiv L{=}L_1{\circ}L_2$, $D' \equiv \mathrm{Seq}(\alpha)$
  - Endliche Mengen (`Set(`α`)`): `ArbRest`
  - Produkträume (+ Mengen, Folgen, ...): Zerlegung in Einzelkomponenten
  - Natürlichen Zahlen ($\mathbb{N}$): Vorgängerfunktion
- **Standard-Wohlordnungen auf Typen (D)** $\mapsto \succ$
  - Folgen und Mengen: Längen/Größenordnung $L \succ L' \equiv |L| > |L'|$
  - Produkträume (+ Mengen, Folgen, ...): Lexikographische Ordnung
    - $(a_1, b_1) \succ (a_2, b_2) \equiv a_1 > a_2 \vee (a_1 = a_2 \wedge b_1 > b_2)$
  - Zahlen ($\mathbb{N}$/$\mathbb{Z}$): Zahlenordnung bzw. Absolutordnung

Wichtiges Programmierwissen für D&C-Synthesen

- **Standard-Zerlegungen für Typen (D)** $\mapsto$ *Decompose*,O_D,*D'*
  - Endliche Listen / Folgen (`Seq(`α`)`): `HdTl, ListSplit`
    - `ListSplit(`L`)` $\equiv$ `([L`$_i$`|i`$\in$`[1..|L|÷2]],[L`$_i$`|i`$\in$`[1+|L|÷2..|L|]])`
      mit $O_D[L, L_1, L_2] \equiv L{=}L_1{\circ}L_2$, $D' \equiv \mathrm{Seq}(\alpha)$
  - Endliche Mengen (`Set(`α`)`): `ArbRest`
  - Produkträume (+ Mengen, Folgen, ...): Zerlegung in Einzelkomponenten
  - Natürlichen Zahlen ($\mathbb{N}$): Vorgängerfunktion
- **Standard-Wohlordnungen auf Typen (D)** $\mapsto \succ$
  - Folgen und Mengen: Längen/Größenordnung $L \succ L' \equiv |L|{>}|L'|$
  - Produkträume (+ Mengen, Folgen, ...): Lexikographische Ordnung
  - Zahlen ($\mathbb{N}$/$\mathbb{Z}$): Zahlenordnung bzw. Absolutordnung
- **Standard-Kompositionen für Typen (R)** $\mapsto$ *Compose*,O_C,*R'*
  - Endliche Folgen: `cons` $(a.l)$, `append` $(L_1{\circ}L_2)$
  - Endliche Mengen: `insert` $(S{+}a)$, `union` $(S_1{\cup}S_2)$
  - Produkträume (+ Mengen, Folgen, ...): Komponentenweise Komposition
  - Natürlichen Zahlen ($\mathbb{N}$): Nachfolgerfunktion

Unterstützende Techniken der D&C-Strategie

- **Heuristische Fixierung von g**
  - Falls $D'{=}D$, wähle $g{:=}f$, $R'{:=}R$, $O'{:=}O$ und $I'{:=}I$
  - Falls $D'{\neq}D$, wähle $g{:=}id$, $R'{:=}D'$, $O'[y',z']'{:=}z'{=}y'$ und $I'[y']{:=}\texttt{true}$

  Analoge Entscheidungen, falls $R'{=}R$ durch Wahl von *Compose*

Unterstützende Techniken der D&C-Strategie

- **Heuristische Fixierung von g**
  - Falls $D'{=}D$, wähle $g{:=}f$, $R'{:=}R$, $O'{:=}O$ und $I'{:=}I$
  - Falls $D'{\neq}D$, wähle $g{:=}id$, $R'{:=}D'$, $O'[y',z']'{:=}z'{=}y'$ und $I'[y']{:=}\texttt{true}$

  Analoge Entscheidungen, falls $R'{=}R$ durch Wahl von *Compose*
- **Inferenzmechanismus: Abgeleitete Vorbedingungen**
  - Bestimme Voraussetzungen für Gültigkeit einer Formel durch zielgerichteten Einsatz von Lemmas entsprechend vorkommender Begriffe
  - Voraussetzungen sind verbleibende Vorbedingungen beim Beweisversuch

Unterstützende Techniken der D&C-Strategie

- **Heuristische Fixierung von g**
  - Falls $D'{=}D$, wähle $g{:=}f$, $R'{:=}R$, $O'{:=}O$ und $I'{:=}I$
  - Falls $D'{\neq}D$, wähle $g{:=}id$, $R'{:=}D'$, $O'[y',z']'{:=}z'{=}y'$ und $I'[y']{:=}\texttt{true}$

  Analoge Entscheidungen, falls $R'{=}R$ durch Wahl von *Compose*
- **Inferenzmechanismus: Abgeleitete Vorbedingungen**
  - Bestimme Voraussetzungen für Gültigkeit einer Formel durch zielgerichteten Einsatz von Lemmas entsprechend vorkommender Begriffe
  - Voraussetzungen sind verbleibende Vorbedingungen beim Beweisversuch
- **Inferenzmechanismus: Fallanalyse**
  - Erzeugung von Alternativen über existierende Prädikate
  - Partielle Auswertung der Einzelfälle

  Liefert Programmstücke mit Fallunterscheidungen

Unterstützende Techniken der D&C-Strategie

- **Heuristische Fixierung von g**
  - Falls $D'{=}D$, wähle $g{:=}f$, $R'{:=}R$, $O'{:=}O$ und $I'{:=}I$
  - Falls $D'{\neq}D$, wähle $g{:=}id$, $R'{:=}D'$, $O'[y', z']'{:=}z'{=}y'$ und $I'[y']{:=}\texttt{true}$

  Analoge Entscheidungen, falls $R'{=}R$ durch Wahl von *Compose*
- **Inferenzmechanismus: Abgeleitete Vorbedingungen**
  - Bestimme Voraussetzungen für Gültigkeit einer Formel durch zielgerichteten Einsatz von Lemmas entsprechend vorkommender Begriffe
  - Voraussetzungen sind verbleibende Vorbedingungen beim Beweisversuch
- **Inferenzmechanismus: Fallanalyse**
  - Erzeugung von Alternativen über existierende Prädikate
  - Partielle Auswertung der Einzelfälle

  Liefert Programmstücke mit Fallunterscheidungen
- **Inferenzmechanismus: Operator match**
  - Synthese von Programmstücken durch Anpassung an bekannte Lösungen

Divide & Conquer Synthese von Sortieralgorithmen

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

Divide & Conquer Synthese von Sortieralgorithmen

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

1. Wähle $L \succ L' \equiv$ `|`L`|>|`L'`|` **und** *Decompose* $\equiv$ `ListSplit`

Liefert $O_D[L, L_1, L_2] \equiv L{=}L_1 \circ L_2$, $D' \equiv$ `Seq(`α`)`

Divide & Conquer Synthese von Sortieralgorithmen

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. **Wähle** $L \succ L' \equiv$ `|L|>|L'|` **und** *Decompose* $\equiv$ `ListSplit`

   Liefert $O_D[L, L_1, L_2] \equiv L{=}L_1 \circ L_2$, $D' \equiv$ `Seq(`α`)`

2. **Konstruiere Hilfsfunktion** $g \equiv$ `sort` $\mapsto O' \equiv O$, $I'[L'] \equiv$ `true`

Divide & Conquer Synthese von Sortieralgorithmen

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. **Wähle** $L \succ L' \equiv |L|>|L'|$ **und** *Decompose* $\equiv$ `ListSplit`

   Liefert $O_D[L, L_1, L_2] \equiv L{=}L_1 \circ L_2$, $D' \equiv \mathtt{Seq}(\alpha)$

2. **Konstruiere Hilfsfunktion** $g \equiv$ `sort` $\mapsto O' \equiv O$, $I'[L'] \equiv$ `true`

3. **Verifiziere** *Decompose*

   `FUNCTION` f_d`(L:Seq(ℤ)):Seq(ℤ)×Seq(ℤ) WHERE` $\neg$*primitive*`[L]`
   `RETURNS` L_1, L_2 `SUCH THAT` $L \succ L_1 \wedge L \succ L_2 \wedge L_1 \circ L_2 = L$

   Vorbedingung für Korrektheit: *primitive*`[L]` $\equiv |L| \leq 1$

Divide & Conquer Synthese von Sortieralgorithmen

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. **Wähle** $L \succ L' \equiv |L|>|L'|$ **und** *Decompose* $\equiv$ `ListSplit`

   Liefert $O_D[L, L_1, L_2] \equiv L{=}L_1 \circ L_2$, $D' \equiv \mathtt{Seq}(\alpha)$

2. **Konstruiere Hilfsfunktion** $g \equiv$ `sort` $\mapsto O' \equiv O$, $I'[L'] \equiv$ `true`

3. **Verifiziere** *Decompose*

   `FUNCTION` f_d`(L:Seq(ℤ)):Seq(ℤ)×Seq(ℤ) WHERE` $\neg$*primitive*`[L]`
   `RETURNS` L_1, L_2 `SUCH THAT` $L \succ L_1 \wedge L \succ L_2 \wedge L_1 \circ L_2 = L$

   Vorbedingung für Korrektheit: *primitive*`[L]` $\equiv |L| \leq 1$

4. **Konstruiere** *Compose*

   $L \succ L_1 \wedge L \succ L_2 \wedge L_1 \circ L_2 = L \wedge$ `SORT(`L_1, S_1`)` $\wedge$ `SORT(`L_2, S_2`)` $\wedge O_C[S_1, S_2, S] \Rightarrow$ `SORT(L,S)`

   Liefert als Spezifikation für *Compose* $\mapsto$ *Synthese folgt später*

   `FUNCTION` f_c`(`S_1, S_2`:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)`
   `WHERE ordered(`S_1`)∧ordered(`S_2`)`
   `RETURNS S SUCH THAT ordered(S) ∧ rearranges(S,`$S_1 \circ S_2$`)`

Divide & Conquer Synthese von Sortieralgorithmen

Spezifikation: `FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

1. **Wähle** $L \succ L' \equiv |L|>|L'|$ **und** *Decompose* $\equiv$ `ListSplit`

   Liefert $O_D[L, L_1, L_2] \equiv L{=}L_1 \circ L_2$, $D' \equiv \mathtt{Seq}(\alpha)$

2. **Konstruiere Hilfsfunktion** $g \equiv$ `sort` $\mapsto O' \equiv O$, $I'[L'] \equiv$ `true`

3. **Verifiziere** *Decompose*

   `FUNCTION` f_d`(L:Seq(ℤ)):Seq(ℤ)×Seq(ℤ) WHERE` $\neg$*primitive*`[L]`
   `RETURNS` L_1, L_2 `SUCH THAT` $L \succ L_1 \wedge L \succ L_2 \wedge L_1 \circ L_2 = L$

   Vorbedingung für Korrektheit: *primitive*`[L]` $\equiv |L| \leq 1$

4. **Konstruiere** *Compose*

   $L \succ L_1 \wedge L \succ L_2 \wedge L_1 \circ L_2 = L \wedge$ `SORT(`L_1, S_1`)` $\wedge$ `SORT(`L_2, S_2`)` $\wedge O_C[S_1, S_2, S] \Rightarrow$ `SORT(L,S)`

   Liefert als Spezifikation für *Compose* $\mapsto$ *Synthese folgt später*

   `FUNCTION` f_c`(`S_1, S_2`:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)`
   `WHERE ordered(`S_1`)`$\wedge$`ordered(`S_2`)`
   `RETURNS S SUCH THAT ordered(S)`$\wedge$`rearranges(S,`$S_1 \circ S_2$`)`

5. **Konstruiere** *Directly-solve*

   `FUNCTION` f_p`(L:Seq(ℤ)):Seq(ℤ) WHERE |L|≤1`
   `RETURNS S SUCH THAT rearranges(L,S)`$\wedge$`ordered(S)`

   Liefert *Directly-solve*`[L]` $\equiv$ `L`

Ermittelte Komponenten

$$
\begin{aligned}
\mathit{Decompose}[L] &\equiv \texttt{ListSplit}[L] \\
&\equiv \texttt{([}L_i\texttt{|i}\in\texttt{[1..|}L\texttt{|}\div\texttt{2]],[}L_i\texttt{|i}\in\texttt{[1+|}L\texttt{|}\div\texttt{2..|}L\texttt{|]])} \\
g &\equiv \texttt{sort} \\
\mathit{Compose}[S_1, S_2] &\equiv \texttt{merge(}S_1\texttt{,}S_2\texttt{)} \qquad \mapsto \textit{nächste Folie} \\
\mathit{Directly\text{-}solve}[L] &\equiv L \\
\mathit{primitive}[L] &\equiv |L|\leq\texttt{1}
\end{aligned}
$$

Ermittelte Komponenten

$$
\begin{array}{lll}
\mathit{Decompose}[L] & \equiv & \texttt{ListSplit}[L] \\
 & \equiv & (\texttt{[}L_i\texttt{|i}\in\texttt{[1..|}L\texttt{|}\div\texttt{2]],[}L_i\texttt{|i}\in\texttt{[1+|}L\texttt{|}\div\texttt{2..|}L\texttt{|]])} \\
g & \equiv & \texttt{sort} \\
\mathit{Compose}[S_1, S_2] & \equiv & \texttt{merge}(S_1, S_2) \\
\mathit{Directly\text{-}solve}[L] & \equiv & L \\
\mathit{primitive}[L] & \equiv & |L|\leq\texttt{1}
\end{array}
$$

↦ *nächste Folie*

6. Instantiiere das Divide & Conquer Schema

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true
   RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)
≡  if |L|≤1 then L
          else let L1.L2 = ( [ Li|i∈[1..|L|÷2] ],
                             [ Li|i∈[1+|L|÷2..|L|] ] )
               in  merge(sort(L1),sort(L2))
```

Ermittelte Komponenten

$$
\begin{array}{lll}
\mathit{Decompose}[L] & \equiv & \texttt{ListSplit}[L] \\
 & \equiv & (\texttt{[}L_i\texttt{|i}\in\texttt{[1..|}L\texttt{|}\div\texttt{2]],[}L_i\texttt{|i}\in\texttt{[1+|}L\texttt{|}\div\texttt{2..|}L\texttt{|]])} \\
g & \equiv & \texttt{sort} \\
\mathit{Compose}[S_1, S_2] & \equiv & \texttt{merge}(S_1, S_2) \\
\mathit{Directly\text{-}solve}[L] & \equiv & L \\
\mathit{primitive}[L] & \equiv & |L|\leq\texttt{1}
\end{array}
$$

↦ *nächste Folie*

6. Instantiiere das Divide & Conquer Schema

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true
   RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)
≡  if |L|≤1 then L
          else let L1.L2 = ( [ Li|i∈[1..|L|÷2] ],
                             [ Li|i∈[1+|L|÷2..|L|] ] )
               in  merge(sort(L1),sort(L2))
```

Algorithmus ist korrekt per Konstruktion

Divide & Conquer Synthese der `merge`-Funktion

FUNCTION merge(S_1,S_2:Seq($\mathbb{Z}$)$\times$Seq($\mathbb{Z}$)):Seq($\mathbb{Z}$)
  WHERE ordered(S_1)$\wedge$ordered(S_2)
  RETURNS S SUCH THAT ordered(S) $\wedge$ rearranges(S, $S_1 \circ S_2$)

Divide & Conquer Synthese der `merge`-Funktion

```
FUNCTION merge(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
   WHERE ordered(S1)∧ordered(S2)
   RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
```

1. Wähle *Compose* ≡ `cons` $(O_C[a, S', S] \equiv S{=}a\,.\,S',\ R' \equiv \mathbb{Z})$

Divide & Conquer Synthese der `merge`-Funktion

```
FUNCTION merge(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
   WHERE ordered(S1)∧ordered(S2)
   RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
```

1. **Wähle** *Compose* $\equiv$ `cons` $(O_C[a, S', S] \equiv S{=}a\,.\,S',\ R' \equiv \mathbb{Z})$
2. **g kann nicht `merge` sein. Wähle $g \equiv$ `id`** $(O'[a, a'] \equiv a{=}a')$

Divide & Conquer Synthese der `merge`-Funktion

```
FUNCTION merge(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
   WHERE ordered(S1)∧ordered(S2)
   RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
```

1. **Wähle** *Compose* $\equiv$ `cons` $(O_C[a, S', S] \equiv S{=}a\,.\,S',\ R' \equiv \mathbb{Z})$
2. **g kann nicht `merge` sein. Wähle $g \equiv$ `id`** $(O'[a, a'] \equiv a{=}a')$
3. **Konstruiere $\succ$ auf `Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)`:**
   - $(S_1, S_2) \succ (S_1', S_2') \equiv$ `|`S_1`|>|`S_1'`| ∨ (|`S_1`|=|`S_1'`| ∧ |`S_2`|>|`S_2'`|)`
   - Kombination Längenordnung für Listen + lexikographische Ordnung für Produkte

Divide & Conquer Synthese der `merge`-Funktion

`FUNCTION merge(`S_1`,`S_2`:Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)):Seq(`$\mathbb{Z}$`)`
`WHERE ordered(`S_1`)`$\wedge$`ordered(`S_2`)`
`RETURNS S SUCH THAT ordered(S)`$\wedge$`rearranges(S,`$S_1 \circ S_2$`)`

---

1. **Wähle** *Compose* $\equiv$ `cons` $(O_C[a, S', S] \equiv S{=}a\,.\,S',\ R' \equiv \mathbb{Z})$
2. **g kann nicht `merge` sein. Wähle $g \equiv$ `id`** $(O'[a, a'] \equiv a{=}a')$
3. **Konstruiere $\succ$ auf `Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)`:**
   - $(S_1, S_2) \succ (S_1', S_2') \equiv |S_1|{>}|S_1'| \vee (|S_1|{=}|S_1'| \wedge |S_2|{>}|S_2'|)$
   - Kombination Längenordnung für Listen + lexikographische Ordnung für Produkte
4. **Konstruiere *Decompose* mit Axiom 2 und 3**

   $O_D[S_1, S_2, a', S_1', S_2'] \wedge a{=}a' \wedge \texttt{MERGE}(S_1', S_2', S') \wedge S{=}a.S' \Rightarrow \texttt{MERGE}(S_1, S_2, S)$

   liefert als Spezifikation und Lösung für *Decompose*

   `FUNCTION` f_d`(`S_1`,`S_2`:Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)):`$\mathbb{Z}$`×Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)`
   `WHERE ordered(`S_1`)`$\wedge$`ordered(`S_2`)`$\wedge$`S`$_1{\neq}$`[]`$\wedge$`S`$_2{\neq}$`[]`
   `RETURNS a',`S_1`',`S_2`'`
   `SUCH THAT rearranges(a.(`S_1`'`$\circ$`S`$_2$`'),`$S_1 \circ S_2$`)` (1)
   $\wedge\ \forall x \in S_1' \circ S_2'.\ a \leq x \wedge (S_1, S_2) \succ (S_1', S_2')$ (2)

---

(1) folgt aus `rearranges(`S_1`'`$\circ$`S`$_2$`',S'),S=a.S'` und `rearranges(`$S_1 \circ S_2$`,S)`
(2) folgt aus (1) und aus `ordered(`S_1`)`$\wedge$`ordered(`S_2`)`

Divide & Conquer Synthese der `merge`-Funktion

```
FUNCTION merge(S₁,S₂:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
   WHERE ordered(S₁)∧ordered(S₂)
   RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S₁∘S₂)
```

---

1. Wähle *Compose* $\equiv$ `cons` $(O_C[a,S',S] \equiv S{=}a\,.\,S',\ R' \equiv \mathbb{Z})$

2. g **kann nicht** `merge` **sein. Wähle** $g \equiv$ `id` $(O'[a,a'] \equiv a{=}a')$

3. Konstruiere $\succ$ **auf** $\mathtt{Seq}(\mathbb{Z})\times\mathtt{Seq}(\mathbb{Z})$**:**

– $(S_1,S_2)\succ(S_1',S_2') \equiv |S_1|>|S_1'| \vee (|S_1|=|S_1'| \wedge |S_2|>|S_2'|)$

– Kombination Längenordnung für Listen + lexikographische Ordnung für Produkte

4. Konstruiere *Decompose* **mit Axiom 2 und 3**

$O_D[S_1,S_2,a',S_1',S_2'] \wedge a{=}a' \wedge \mathtt{MERGE}(S_1',S_2',S') \wedge S{=}a.S' \Rightarrow \mathtt{MERGE}(S_1,S_2,S)$

liefert als Spezifikation und Lösung für *Decompose*

```
FUNCTION f_d(S₁,S₂:Seq(ℤ)×Seq(ℤ)):ℤ×Seq(ℤ)×Seq(ℤ)
   WHERE ordered(S₁) ∧ ordered(S₂) ∧ S₁≠[] ∧ S₂≠[]
   RETURNS a',S₁',S₂'
   SUCH THAT rearranges(a.(S₁'∘S₂'), S₁∘S₂)
             ∧ ∀x∈S₁'∘S₂'.a≤x ∧ (S₁,S₂)≻(S₁',S₂')
≡ let x₁.S₁'=S₁, x₂.S₂'=S₂ in if x₁≤x₂ then(x₁,S₁',S₂) else(x₂,S₁,S₂')
```

(Lösung durch Fallanalyse: `a'` muß Minimum von `hd(S₁)` und `hd(S₂)` sein)

Divide & Conquer Synthese der `merge`-Funktion

FUNCTION merge(S_1,S_2:Seq($\mathbb{Z}$)$\times$Seq($\mathbb{Z}$)):Seq($\mathbb{Z}$)
  WHERE ordered(S_1)$\wedge$ordered(S_2)
  RETURNS S SUCH THAT ordered(S) $\wedge$ rearranges(S, $S_1 \circ S_2$)

Divide & Conquer Synthese der `merge`-Funktion

```
FUNCTION merge(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
   WHERE ordered(S1)∧ordered(S2)
   RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
```

5. Vorbedingung für Korrektheit von *Decompose* ist $S_1 \neq [] \wedge S_2 \neq []$

– Liefert *primitive* und Spezifikation für *Directly-solve*

```
FUNCTION fp(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
  WHERE ordered(S1)∧ordered(S2)∧ (S1=[] ∨ S2=[])
  RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
= if S1=[] then S2 else S1
```

Divide & Conquer Synthese der `merge`-Funktion

```
FUNCTION merge(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
   WHERE ordered(S1)∧ordered(S2)
   RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
```

5. Vorbedingung für Korrektheit von *Decompose* ist $S_1 \neq [] \wedge S_2 \neq []$

– Liefert *primitive* und Spezifikation für *Directly-solve*

```
FUNCTION fp(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
  WHERE ordered(S1)∧ordered(S2)∧ (S1=[] ∨ S2=[])
  RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
= if S1=[] then S2 else S1
```

6. Instantiierter Divide & Conquer Algorithmus

```
FUNCTION merge(S1,S2:Seq(ℤ)×Seq(ℤ)):Seq(ℤ)
  WHERE ordered(S1)∧ordered(S2)
  RETURNS S SUCH THAT ordered(S) ∧ rearranges(S, S1∘S2)
= if S1=[] then S2
    elseif S2=[] then S1
      else let x1.S1'=S1 and x2.S2'=S2
           in if x1≤x2 then x1.merge(S1',S2) else x2.merge(S1,S2')
```

Erzeugung Alternativer Sortieralgorithmen

- **Wähle einfachere Dekomposition** (Sortieren durch Einfügen)
  - *Decompose* $\equiv$ `HdTl`, $g \equiv$ `id`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Compose*[`a`, `S`] $\equiv$ `ordered_insert(a,S)`

Erzeugung Alternativer Sortieralgorithmen

- **Wähle einfachere Dekomposition** (Sortieren durch Einfügen)
  - *Decompose* $\equiv$ `HdTl`, $g \equiv$ `id`
  - *primitive*`[L]` $\equiv$ `L=[]`, *Directly-solve*`[L]` $\equiv$ `[]`
  - *Compose*`[a, S]` $\equiv$ `ordered_insert(a,S)`
- **Wähle einfache Komposition** (Sortieren durch Auswahl)
  - *Compose*`[a, S]` $\equiv$ `a.S`, $g \equiv$ `id`
  - *primitive*`[L]` $\equiv$ `L=[]`, *Directly-solve*`[L]` $\equiv$ `[]`
  - *Decompose*`[L]` $\equiv$ `let m=min(L) in (m, L-m)`

Erzeugung Alternativer Sortieralgorithmen

- **Wähle einfachere Dekomposition** (Sortieren durch Einfügen)
  - *Decompose* $\equiv$ `HdTl`, $g \equiv$ `id`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Compose*[`a, S`] $\equiv$ `ordered_insert(a,S)`
- **Wähle einfache Komposition** (Sortieren durch Auswahl)
  - *Compose*[`a, S`] $\equiv$ `a.S`, $g \equiv$ `id`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Decompose*[`L`] $\equiv$ `let m=min(L) in (m, L-m)`
- **Wähle binäre Komposition** (Naives Quicksort)
  - *Compose*[S_1, S_2] $\equiv S_1 \circ S_2$, $g \equiv$ `sort`
  - *primitive*[`L`] $\equiv$ `|L|`$\leq$`1`, *Directly-solve*[`L`] $\equiv$ `L`
  - *Decompose*[`L`] $\equiv$ `let let a=`$L_{[|L|/2]}$ `in (L<a, L`$_{\geq}$`a])`

Erzeugung Alternativer Sortieralgorithmen

- **Wähle einfachere Dekomposition** (Sortieren durch Einfügen)
  - *Decompose* $\equiv$ `HdTl`, $g \equiv$ `id`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Compose*[`a, S`] $\equiv$ `ordered_insert(a,S)`
- **Wähle einfache Komposition** (Sortieren durch Auswahl)
  - *Compose*[`a, S`] $\equiv$ `a.S`, $g \equiv$ `id`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Decompose*[`L`] $\equiv$ `let m=min(L) in (m, L-m)`
- **Wähle binäre Komposition** (Naives Quicksort)
  - *Compose*[S_1, S_2] $\equiv S_1 \circ S_2$, $g \equiv$ `sort`
  - *primitive*[`L`] $\equiv$ `|L|`$\leq$`1`, *Directly-solve*[`L`] $\equiv$ `L`
  - *Decompose*[`L`] $\equiv$ `let let a=`$L_{[|L|/2]}$ `in (L<a, L`$_{\geq}$`a])`

Flexible Strategie mit vielfältigen Anwendungen