Automatisierte Logik und Programmierung II

Teil V

Automatisierte Programmierung

1. **Grundkonzepte & Vorgehensweise**
2. **Synthese im Kleinen**
   – Paradigmen & Strategien
3. **Wissensbasierte Programmentwicklung**
4. **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 zwischen Aufgabe und Lösung selten erkennbar
    - Programmierer geben keine Begründung für Korrektheit ihres Programms
- **Logische Synthese von Programmen hilft**
  - Werkzeuge zur (Teil-)Automatisierung der Konstruktion von Algorithmen
  - Logisches Fundament erhöht Zuverlässigkeit des erzeugten Programms
  - Automatisierung verringert Entwicklungszeit und -kosten und ermöglicht frühzeitige Validierung durch Endanwender

Kernaspekte einer Programmsynthese

Erzeuge korrekte ausführbare Programme aus Spezifikationen

- **Formale Spezifikation als Ausgangspunkt**
  - Formale Beschreibung von Anwendungsbereich und Problemstellung
  - Verlangt Fixierung einer formalen Sprache
- **Methoden für automatische Algorithmensynthese**
  - Benötigen theoretische Resultate über Korrektheit erzeugter Algorithmen
  - Syntheseparadigma: zulässige Manipulationen garantieren Korrektheit
  - Synthesestrategie automatisiert Anwendung zulässiger Operationen
  - Trace der Strategie dokumentiert getroffene Entscheidungen
- **Optimierung und Datentypverfeinerung**
  - Verbesserung des erzeugten Basisalgorithmus
  - Auswahl geeigneter Implementierungen der vorkommenden Datentypen
  - Sprachabhängige Optimierung bei Übertragung in Programmiersprache

Historische Entwicklung

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

Automatisierte Logik und Programmierung

Lektion 15

Grundkonzepte der Programmsynthese

1. Formale Grundbegriffe
2. Formalisierung von Anwendungsbereichen
3. Programmsynthese am Beispiel

Programmsynthese: grundsätzliches Vorgehen

1. **Erstellen einer formalen Spezifikation**
   – Benötigt Formalisierung des Anwendungsbereichs als "**Objekttheorie**"
   – Welche Begriffe werden benutzt und was bedeuten sie?
   – Welche mathematischen Gesetze gelten für diese Begriffe?
2. **Entwurf eines läuffähigen, korrekten Algorithmus**
   – Synthesestrategie generiert Basisversion und Korrektheitsgarantien
3. **Erzeugung eines effizienten, korrekten Programms**
   – Benutzergesteuerte Optimierungstechniken verbessern Algorithmus
   – Übertragung in Zielsprache ermöglicht weitere Optimierungen
   – System garantiert Korrekheit der Optimierungen

Spezifikation von Programmierproblemen

- **Programme berechnen im Endeffekt Funktionen**
  - Aus welchem Datentyp stammen die Eingaben? Domain D
  - Zu welchem Datentyp gehören die Ausgaben? Range R
  - Gibt es Beschränkungen an zulässige Eingaben? Input-Bedingung I
  - Was ist der Zusammenhang zwischen Ein- und Ausgaben? Output-Bedingung O
- **Spezifikationen als formale Objekte**
  - Eine **formale Spezifikation** ist ein Quadrupel $spec = (D, R, I, O)$ wobei D und R Datentypen, I Prädikat über D, O Prädikat über $D \times R$
  - D, R, I, O sind in einer Spezifikationssprache zu beschreiben
- **Zwei mögliche Aufgabenstellungen**
  - Programm soll eine mögliche Lösung bestimmen,
    FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$
  - Programm soll alle möglichen Lösungen bestimmen
    FUNCTION $f(x{:}D)$ WHERE $I[x]$ RETURNS $\{y{:}R \mid O[x,y]\}$

Programme und Korrektheit

- **Programm = Spezifikation + Algorithmus**
  - Algorithmen (Programmkörper) sind berechenbare (partielle) Funktionen auf $D \not\to R$, die auf allen zulässigen Eingaben definiert sind

- **Programme als formale Objekte**
  - Eine **formales Programm** ist ein 5-Tupel $prog = (D, R, I, O, body)$ wobei (D, R, I, O) formale Spezifikation, $body{:}D \not\to R$ berechenbar
  - *body* darf f rekursiv aufrufen und ist in Programmiersprache zu beschreiben
    - $\mathtt{FUNCTION}\ f(x{:}D){:}R\ \mathtt{WHERE}\ I[x]\ \mathtt{RETURNS}\ y\ \mathtt{SUCH\ THAT}\ O[x,y]\ \equiv\ body[f,x]$
    - $\mathtt{FUNCTION}\ f(x{:}D)\ \mathtt{WHERE}\ I[x]\ \mathtt{RETURNS}\ \{y{:}R \mid O[x,y]\}\ \equiv\ body[f,x]$

- **Korrektheit von Programmen**
  - ***prog* ist korrekt**, falls $\forall x{:}D.\ I[x] \Rightarrow O[x, body(x)]$

- **Syntheseziel: Erfüllbarkeit von Spezifikationen**
  - ***spec* ist erfüllbar** (synthetisierbar), falls es eine Funktion $body{:}D \not\to R$ gibt, so daß $prog{=}(spec, body)$ korrekt ist

Spezifikations- und Programmiersprache

- **Einheitlicher Formalismus für Synthese**
  - Mathematische Sprache mit Programmiernotation
  - Beinhaltet formale Notation für die wichtigsten Datentypen
  - Aufgesetzt auf logischen Basiskalkül (z.B. Typentheorie)
- **Formales Wissen über Standard-Datentypen**
  - Definition der Begriffe im Basiskalkül
  - Verifizierte Lemmata über Eigenschaften der Begriffe
  - Quelle: Inhalt von Lehrmaterial, -büchern und Forschungsergebnissen
- **Objekttheorien: zusätzliches Domänenwissen**
  - Definition neuer Konzepte in einer Spezifikation
  - Lemmata über Grundeigenschaften dieser Konzepte

Aufbau formaler mathematischer (Objekt-)Theorien

Notwendig für Formalisierung von Programmierproblemen

- **Formalisiere Grundkonzepte der Theorie**
  - Systematischer Entwurf analog zum Aufbau der Typentheorie
    - Formale Notation für Datentyp, kanonische & nichtkanonische Elemente
    - Inferenzregeln für Elemente und Datentyp
- **Implementiere Grundkonzepte der Theorie**
  - Formale Definitionen erklären neue Begriffe durch bestehende Terme
  - Taktiken beschreiben neue Inferenzregeln durch existierende Regeln
- **Erstelle erweiterte Objekttheorie**
  - Formalisiere wichtige Begriffe durch Grundkonzepte der Theorie
  - Beweise mathematische Gesetze zu Eigenschaften "abgeleiteter" Konzepte

    Insbesondere Rewrite-Lemmata zu Kombinationen von Operationen

Beispiel: Theorie endlicher Mengen

- **Grundkonzepte**

  Datentyp: $\texttt{Set}(\alpha)$

  Operationen: $\emptyset\texttt{:}\ \texttt{Set}(\alpha)$

  $\texttt{+:}\ \texttt{Set}(\alpha)\times\alpha \rightarrow \texttt{Set}(\alpha)$

  $\in\texttt{:}\ \alpha\times\texttt{Set}(\alpha) \rightarrow \texttt{Bool}$

  Gesetze: $\texttt{a}\notin\emptyset$

  $\texttt{x}\in\texttt{(S+a)} \;\Leftrightarrow\; \texttt{(x=a} \vee \texttt{x}\in\texttt{S)}$

  $\texttt{(S+a)+x = (S+x)+a}$

  $\texttt{(S+a)+a = S+a}$

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

- **Implementierung**

  $\emptyset \equiv \texttt{nil}$

  $\texttt{+} \equiv \lambda\texttt{a,S. a.S}$

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

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

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

Theorie endlicher Mengen – Abgeleitete Konzepte

`empty?`	$\equiv$	λS. if S=$\emptyset$ then tt else ff
$\subseteq$	$\equiv$	λS,S'. $\forall$x$\in$S.x$\in$S'
$\{$*list-exp*$\}$	$\equiv$	*list-exp*.nil
$\{i..j\}$	$\equiv$	ind(j-i; _,_.$\emptyset$; $\{j\}$; diff,j-set. j-set+(j-diff))
$\{f_x \mid x \in S \wedge p_x\}$	$\equiv$	list_ind(S; $\emptyset$; a,_,GSF. if p_x[a/x] then GSF+f_x[a/x] else GSF
$\lvert S \rvert$	$\equiv$	list_ind(S; 0; a,S',card. if a$\in$S' then card else card+1)
-	$\equiv$	λS,a. $\{$x$\mid$x$\in$S$\wedge$x$\neq$a$\}$
$\cup$	$\equiv$	λS,S'. list_ind(S'; S; a,_,union.union+a)
$\cap$	$\equiv$	λS,S'. $\{$x$\mid$x$\in$S$\wedge$x$\in$S'$\}$
$\setminus$	$\equiv$	λS,S'. $\{$x$\mid$x$\in$S$\wedge$x$\notin$S'$\}$
$\bigcup$	$\equiv$	λFAMILY. list_ind(FAMILY; $\emptyset$; S,FAM,Union.Union$\cup$S)
$\bigcap$	$\equiv$	λFAMILY. list_ind(FAMILY; fail; S,FAM,inter. if empty?(FAM) then S else inter$\cap$S)
`map`	$\equiv$	λf,S. $\{$f(x)$\mid$x$\in$S$\}$
`reduce`	$\equiv$	λop,S. list_ind(S; fail; a,S',redS'. if empty?(S') then a else if a$\in$S' then redS' else op(redS',a))
$T =_{Set} S \uplus S'$	$\equiv$	$T =_{Set} S \cup S'$ $\wedge$ empty?($S \cap S'$)

Wichtigste Bestandteile der Formalisierungssprache

$\mathbb{B}$, `true`, `false`	Data type of boolean expressions, explicit truth values
$\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$	Boolean connectives
$\forall$x$\in$S.p, $\exists$x$\in$S.p	Limited boolean quantifiers (on finite sets and sequences)
`if p then a else b`	Conditional
`Seq`(α)	Data type of finite sequences over members of α
`null?`, $\in$, $\sqsubseteq$	Decision procedures: emptiness, membership, prefix
[], [a], [i..j], [a_1...a_n]	Empty/ singleton sequence, subrange, literal sequence former
a.L, L·a	prepend `a`, append `a` to `L`
[f(x)|x$\in$L$\wedge$p(x)], |L|, L[i]	General sequence former, length of `L`, `i`-th element,
`domain(L)`, `range(L)`	The sets {1..|L|} and {L[i]|i$\in$domain(L)}
`nodups(L)`	Decision procedure: all the `L[i]` are distinct (no duplicates)
`Set`(α)	Data type of *finite* sets over members of α
`empty?`, $\in$, $\subseteq$	Decision procedures: emptiness, membership, subset
$\emptyset$, {a}, {i..j}, {a_1...a_n}	Empty set, singleton set, integer subset, literal set former
S+a, S-a	element addition, element deletion
{f(x)|x$\in$S$\wedge$p(x)}, |S|	General set former, cardinality
S$\cup$T, S$\cap$T, S\T	Union, intersection, set difference
$\bigcup$FAMILY, $\bigcap$FAMILY	Union, intersection of a family of sets

Costas-Arrays Problem

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					

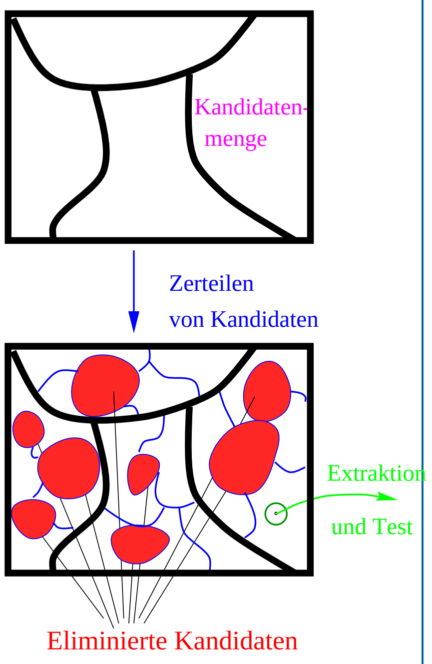
Costas Array der Ordnung 6 und seine Differenzentafel

Ziel: Berechnung aller Costas Arrays der Größe n

Berechnung aller Costas Arrays

Bis 1988 keine effiziente Lösungsalgorithmen bekannt

- **Aufzählung und Testen ist exponentiell**
  - Wie analysiert man Lösungskandidaten ohne sie aufzuzählen?
- **Lösung benutzt Globalsuche**
  - Codierung von Kandidatenmengen
  - Wiederholtes Aufteilen und und Filtern
    auf Basis von Repräsentanten
  - Extraktion konkreter Lösungen aus Repräsentanten



Globalsuchalgorithmen sind systematisch erzeugbar

Phasen einer formale Algorithmenentwicklung

1. **Erstellen der nötigen Objekttheorie**
   - Formalisierung vorkommender neuer Begriffe
   - Aufstellen mathematischer Gesetze für diese Begriffe

2. **Erstellen der formalen Spezifikation**

3. **Entwurf eines korrekten Basisalgorithmus**

4. **Verifizierte algorithmische Optimierung**

5. **Implementierung**
   - Auswahl geeigneter Implementierungen für abstrakte Datentypen
   - Ggf. Compilierung und sprachabhängige Optimierung

Unterstützung durch Synthesesystem

Steuerung durch erfahrenen Benutzer

Costas-Arrays (1): Objekttheorie

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

- **Formalisierung vorkommender Begriffe:**

  $\texttt{dtrow}(L,j) \;\equiv\; [L[\texttt{i}]-L[\texttt{i}+j] \mid \texttt{i} \in [1..|L|-j]]$

  $\texttt{perm}(L,S) \;\equiv\; \texttt{nodups}(L) \wedge \texttt{range}(L)=S$

- **Aufstellen mathematischer Gesetze:**

  $\forall$L,L':Seq($\mathbb{Z}$).$\forall$i:$\mathbb{Z}$.$\forall$j:$\mathbb{N}$.

  1. dtrow([],j) = []
  2. j≤|L| ⇒ dtrow(i.L,j) = (i-L[j]).dtrow(L,j)
  3. j≠0 ⇒ dtrow([i],j) = []
  4. L⊑L' ⇒ dtrow(L,j) ⊑ dtrow(L',j)
  5. j≥|L| ⇒ dtrow(L,j) = []
  6. j≤|L| ⇒ dtrow(L·i,j) = dtrow(L,j)·(L[|L|+1-j]-i)

  ⋮

Costas-Arrays (2): formale Spezifikation

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

$D \mapsto \mathbb{Z}$

$R \mapsto \texttt{Seq}(\mathbb{Z})$

$I \mapsto \lambda\texttt{n. n}\geq\texttt{1}$

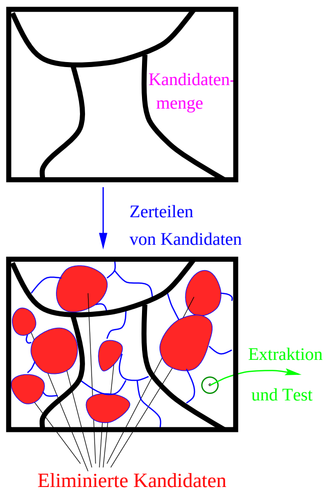
$O \mapsto \lambda\texttt{n,p. perm(p,}\{\texttt{1..n}\}\texttt{)} \wedge \forall\texttt{j}\in\texttt{domain(p).nodups(dtrow(p,j))}$

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

Costas-Arrays (3): Erzeugung des Basisalgorithmus

- **Grundstruktur eines Globalsuchalgorithmus**

$$\begin{array}{l}\texttt{let rec } \texttt{f}_{gs}\texttt{(x,s) = } \{\, \texttt{z} \mid \texttt{z} \in \texttt{ext(s)} \wedge \texttt{O(x,z)} \,\} \\ \qquad\qquad\qquad\qquad \cup \bigcup \{\, \texttt{f}_{gs}\texttt{(x,t)} \mid \texttt{t} \in \texttt{split(x,s)} \wedge \Phi\texttt{(x,t)} \,\} \\ \texttt{in } \; \texttt{f}_{gs}\texttt{(x,}\texttt{s}_0\texttt{(x))}\end{array}$$



  - s: Deskriptor für Mengen von Lösungskandidaten
  - $\texttt{s}_0\texttt{(x)}$: Initialdeskriptor für Eingabe x
  - split(x,s): Rekursive Aufteilung von Kandidatenmengen
  - Φ(x,s): Filter zur Elimination unnötiger Deskriptoren
  - ext(s): direkte Extraktion von Lösungskandidaten z aus Deskriptoren
  - O(x,z): Ausgabebedingung, verwendet zur endgültigen Selektion

- **Globalsuchalgorithmus für Costas-Arrays Problem**

$$\begin{array}{l}\texttt{let costas(n) =} \\ \quad \texttt{let rec aux(n,s)} \\ \quad \texttt{= } \{\, \texttt{p} \mid \texttt{p} \in \{\texttt{s}\} \wedge \texttt{perm(p,}\{\texttt{1..n}\}\texttt{)} \wedge \forall \texttt{j<n.nodups(dt-row(p,j))} \,\} \\ \qquad \cup \bigcup \{\, \texttt{aux(x,t)} \mid \texttt{t} \in \{\, \texttt{s}\cdot\texttt{i} \mid \texttt{i} \in \{\texttt{1..n}\} \,\} \\ \qquad\qquad\qquad\qquad \wedge \; \texttt{nodups(t)} \wedge \; \forall \texttt{j<|t|.nodups(dt-row(t,j))} \} \\ \quad \texttt{in aux(n,[])}\end{array}$$

COSTAS-ARRAYS (4a): Simplifikationen

```
let costas(n) =
   let rec aux(n,s)
   = { p | p ∈ {s} ∧ perm(p,{1..n}) ∧ ∀j<n. nodups(dt-row(p,j)) }
      ∪ ⋃{ aux(x,t) | t ∈ { s·i | i ∈ {1..n}}  ∧  nodups(t)
                                    ∧  ∀j<|t|. nodups(dt-row(t,j)) }
   in aux(n,[])
```

Domänenwissen: $\{\texttt{p} \mid \texttt{p} \in \{\texttt{s}\} \wedge P(\texttt{p})\} \quad \equiv \quad \texttt{if}\ P(\texttt{s})\ \texttt{then}\ \{\texttt{s}\}\ \texttt{else}\ \emptyset$

COSTAS-ARRAYS (4A): SIMPLIFIKATIONEN

```
let costas(n) =
   let rec aux(n,s)
   = if perm(s,{1..n}) ∧ ∀j<n. nodups(dt-row(s,j)) then {s} else ∅
     ∪ ⋃{ aux(x,t) | t ∈ { s·i | i ∈ {1..n} } ∧ nodups(t)
                               ∧ ∀j<|t|. nodups(dt-row(t,j)) }
   in aux(n,[])
```

Domänenwissen: perm(s,{1..n}) ⇒ |s|=n

Kontext der Formel: perm(s,{1..n})

Kontext der Loopinvariante: ∀j<|s|. nodups(dt-row(s,j))

COSTAS-ARRAYS (4A): SIMPLIFIKATIONEN

```
let costas(n) =
   let rec aux(n,s)
   = if perm(s,{1..n}) then {s} else ∅
      ∪ ⋃{ aux(x,t) | t ∈ { s·i | i ∈ {1..n} }  ∧ nodups(t)
                                ∧ ∀j<|t|.nodups(dt-row(t,j)) }
   in aux(n,[])
```

Domänenwissen:

`perm(s,{1..n})` ≡ `s⊆{1..n}` ∧ `{1..n}⊆s` ∧ `nodups(s)`

`{1..n}⊆s` ≡ `{1..n}\s=∅`

Kontext der Loopinvariante: `nodups(s)`

Rahmenbedingung für Deskriptoren `J({1..n},s)`: `s⊆{1..n}`

COSTAS-ARRAYS (4A): SIMPLIFIKATIONEN

```
let costas(n) =
  let rec aux(n,s)
  = if {1..n}\s=∅ then {s} else ∅
     ∪ ⋃{ aux(x,t) | t ∈ { s·i | i ∈ {1..n} } ∧ nodups(t)
                                ∧ ∀j<|t|. nodups(dt-row(t,j)) }
  in aux(n,[])
```

Domänenwissen:

$\{\, f(\mathtt{x},\mathtt{t}) \mid \mathtt{t} \in \{g(\mathtt{s},\mathtt{i}) \mid \mathtt{i} \in S\} \wedge h(\mathtt{t})\} = \{\, f(\mathtt{x},g(\mathtt{s},\mathtt{i})) \mid \mathtt{i} \in S \wedge h(g(\mathtt{s},\mathtt{i}))\}$

Costas-Arrays (4a): Simplifikationen

```
let costas(n) =
  let rec aux(n,s)
  = if {1..n}\s=∅ then {s} else ∅
      ∪ ⋃{ aux(x,s·i) | i ∈{1..n} ∧ nodups(s·i)
                        ∧ ∀j<|s·i|. nodups(dt-row(s·i,j)) }
  in aux(n,[])
```

Domänenwissen: `i ∈{1..n} ∧ nodups(s·i)  ≡  i ∈{1..n}\s`

COSTAS-ARRAYS (4A): SIMPLIFIKATIONEN

```
let costas(n) =
   let rec aux(n,s)
   = if {1..n}\s=∅ then {s} else ∅
      ∪ ⋃{ aux(x,s·i) | i∈{1..n}\s
                         ∧ ∀j<|s·i|. nodups(dt-row(s·i,j)) }
   in aux(n,[])
```

Domänenwissen:

dt-row(s·i,j) = dt-row(s,j)·(s[|s·i|-j]-i)

nodups($t{\cdot}k$) $\equiv$ nodups(t) $\wedge$ $k \notin t$

∀j<|s·i|.P(j) $\equiv$ ∀j<|s|.P(j) $\wedge$ P(|s|)

dt-row(s·i,|s|) = [s[|s·i|-|s|]-i]

nodups([s[|s·i|-|s|]-i]) $\equiv$ true

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

Kontext der Loopinvariante: ∀j<|s|. nodups(dt-row(s,j))

COSTAS-ARRAYS (4B): ENDLICHE DIFFERENZIERUNG

```
let costas(n) =
   let rec aux(n,s)
   = if {1..n}\s=∅ then {s} else ∅
      ∪ ⋃{ aux(x,s·i) | i ∈ {1..n}\s
                      ∧ ∀j<|s|. (s[|s·i|-j]-i) ∉ dt-row(s,j)}
   in aux(n,[])
```

Ersetze ineffiziente Neuberechnung durch neue Variablen:

`{1..n}\s` ↦ `pool`

`|s·i|` ↦ `ssize`

Integriere Variablen in Definition von `aux`

COSTAS-ARRAYS (4B): ENDLICHE DIFFERENZIERUNG

```
let costas(n) =
   let rec aux(n,s,pool,ssize)
   = if pool=∅ then {s} else ∅
      ∪ ⋃{ aux(x,s·i,pool\{i},ssize+1) | i∈pool
                                       ∧ ∀j<|s|. (s[ssize-j]-i) ∉ dt-row(s,j)}
   in aux(n,[],{1..n},1)
```

Costas-Arrays (4c): Fallanalyse

```
let costas(n) =
   let rec aux(n,s,pool,ssize)
   = if pool=∅ then {s} else ∅
      ∪ ⋃{ aux(x,s·i,pool\{i},ssize+1) | i ∈ pool
                                 ∧ ∀j<|s|. (s[ssize-j]-i) ∉ dt-row(s,j)}
   in aux(n,[],{1..n},1)
```

Domänenwissen:

(if pool=∅ then {s} else ∅) ∪ S = if pool=∅ then {s} ∪ S else S

Costas-Arrays (4c): Fallanalyse

```
let costas(n) =
   let rec aux(n,s,pool,ssize)
   = if pool=∅
      then {s} ∪ ⋃{ aux(x,s·i,pool\{i},ssize+1) | i ∈ pool
                                   ∧ ∀j<|s|. (s[ssize-j]-i) ∉ dt-row(s,j)}
      else ⋃{ aux(x,s·i,pool\{i},ssize+1) | i ∈ pool
                                   ∧ ∀j<|s|. (s[ssize-j]-i) ∉ dt-row(s,j)}
   in aux(n,[],{1..n},1)
```

Domänenwissen: $\bigcup\{ f(\mathtt{i}) \mid \mathtt{i} \in \emptyset\} = \emptyset$

Kontext der Formel: pool=∅

Costas-Arrays (5): Datentypverfeinerung

Ersetze abstrakte Definitionen von Datentypen durch effiziente konkrete Implementierungen

```
let costas(n) =
   let rec aux(n,s,pool,ssize)
   = if pool=∅
      then {s}
      else ⋃{ aux(x,s·i,pool\{i},ssize+1) | i∈pool
                                   ∧ ∀j<|s|. (s[ssize-j]-i)∉dt-row(s,j)}
   in aux(n,[],{1..n},1)
```

---

n: $\mathbb{Z}$ $\mapsto$ Standardimplementierung positiver ganzen Zahlen

s: `Seq(`$\mathbb{Z}$`)`, Elemente werden hinten angehängt $\mapsto$ umgekehrt verkettere Liste

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

ssize: $\mathbb{Z}$ $\mapsto$ Standardimplementierung positiver ganzen Zahlen