Übung zur Vorlesung

Automatisierte Logik und Programmierung

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2009

Blatt 3 — Abgabetermin: 26.06.2009

Aufgabe 3.1 (Anwendungsbezogene Erweiterung formaler Theorien)

Das n-Dame Problem: Gegeben ein Schachbrett mit nxn Feldern und n Dame Figuren. Eine Dame kann alle Figuren schlagen, die sich auf derselben waagerechten oder senkrechten Linie befinden sowie alle Figuren, die sich auf der von ihr ausgehenden Diagonale befinden. Gesucht sind *alle* möglichen Plazierungen der n Damen auf dem Schachbrett, so daß keine Dame eine andere schlagen kann.

Entwickeln Sie eine formale Spezifikation des n-Dame Problems. Stellen Sie eventuell notwendige Definitionen für neue Begriffe auf und beschreiben Sie einige Gesetze dieser neuen Konzepte.

Aufgabe 3.2 (Synthese von Divide & Conquer Algorithmen)

Erzeugen Sie mithilfe der formalen Synthesestrategie für Divide & Conquer Algorithmen den Quicksort-Algorithmus für das Sortieren von Listen ganzer Zahlen. Welche Lemmata benötigt die Strategie, um die Komponenten herzuleiten?

Hinweise: Wie beim Mergesort-Algorithmus der Vorlesung muß man in zwei Phasen vorgehen, da der Quicksort-Algorithmus eine nichttriviale Dekomposition besitzt. Berücksichtigen Sie auch, daß Quicksort in einem gewissem Sinne invers zu Mergesort arbeitet, also die Dekomposition invers zur Komposition von Mergesort operiert, während die Komposition verhältnismäßig einfach ist und als Ausgangspunkt genommen werden sollte.

Aufgabe 3.3 (Formalisierung von Grundbegriffen der Programmsynthese)

In der Vorlesung haben wir die Grundbegriffe der Programmsynthese semi-formal beschrieben. Für eine formale Programmsynthese innerhalb eines Beweissystems müssen diese Konzepte formalisiert werden. Formalisieren Sie die folgenden Begriffe innerhalb der CTT.

3.3–a Die Klasse aller Spezifikationen als ein Datentyp `SPECIFICATIONS`

3.3–b Die Klasse aller Programme als ein Datentyp `PROGRAMS`

3.3–c Programmkorrektheit als ein “Prädikat” p `ist korrekt` *(beachten Sie Terminierung!)*

3.3–d Erfüllbarkeit von Spezifikationen als ein “Prädikat” *spec* `ist erfüllbar`

3.3–e Die Notation für Programme
`FUNCTION` f`(x:D):R WHERE I(x) RETURNS y SUCH THAT O(x,y) =` `body(x)`

Aufgabe 3.4 (Aufwendig: Formalisierung von Graphen und Bäumen)

Graphen und Bäume sind wichtige Datenstrukturen für eine große Menge von Anwendungsproblemen. Formalisieren Sie eine Theorie endlicher Graphen und Bäume derart, daß sich die folgenden Probleme darin beschreiben lassen.

- Das Cliquen-Problem: Gegeben ein Graph $G = (V, E)$ der Größe n und eine Zahl $k \leq |V|$. Gibt es in G eine Clique der Mindestgröße k?
- Das Independent Set: Gegeben ein Graph $G = (V, E)$ der Größe n und eine Zahl $k \leq |V|$. Gibt es in G eine unabhängige Knotenmenge der Mindestgröße k?
- Das Vertex Cover Problem: Gegeben ein Graph $G = (V, E)$ der Größe n und eine Zahl $k \leq |V|$. Gibt es in G eine Knotenüberdeckung der Maximalgöße k?
- Das Travelling Salesman Problem: Gegeben ein vollständiger gewichteter Graph (G, g). Gibt es in G einen Zyklus dessen Gesamtgewicht unter B liegt?
- Das MWST Problem. Gegeben ein gewichteter Graph (G, g). Bestimme einen minimal spannenden Baum von G.

3.4–a Formalisieren Sie zunächst die wichtigsten Begriffe der Theorie endlicher Graphen und Bäume in der Typentheorie (Beispiele sind im Anhang genannt). Identifizieren Sie dabei Konzepte aus der Theorie endlicher Mengen und Listen, die für eine Formalisierung erforderlich wären.

3.4–b Formulieren Sie Lemmata, welche Zusammenhänge zwischen verschiedenen Begriffen beschreiben, z.B. den Zusammenhang zwischen Cliquen und unabhängige Knotenmengen.

3.4–c Formalisieren Sie die obengenannten Probleme als “Spezifikationstheoreme”, mit denen im Sinne des Prinzips “Beweise als Programme” gezeigt würde, daß für jede Problemstellung eine Lösung konstruierbar ist. Wie müssen dabei Entscheidungsprobleme spezifiziert werden?

Diese Aufgabe ließe sich leicht zu einem Projekt/Studienarbeit erweitern

Beispiele graphentheoretischer Definitionen

- Ein (ungerichteter) *Graph* ist ein Paar $G = (V, E)$, wobei V endliche Menge und $E \subseteq \{ \{v, v'\} \mid v, v' \in V \wedge v \neq v' \}$.
  Ein Graph ist darstellbar als Liste $v_1, ..., v_n, \{v_{i_1}, v'_{i_1}\}, ..., \{v_{i_m}, v'_{i_m}\}$.
- Ein *gerichteter Graph* ist ein Paar $G = (V, E)$, wobei V endliche Menge und $E \subseteq V \times V$.
- Ein *gewichteter Graph* ist ein Graph $G = (V, E)$ mit einer Gewichtungsfunktion $g : E \rightarrow \mathbb{N}$.
- Ein Graph $H = (V_H, E_H)$ ist genau dann *Subgraph* des Graphen $G = (V, E)$ ($H \sqsubseteq G$), wenn alle Ecken und Kanten von H auch Ecken bzw. Kanten in G sind:
  $$(V_H, E_H) \sqsubseteq (V, E) :\Leftrightarrow V_H \subseteq V \wedge E_H \subseteq E$$
- $H = (V_H, E_H)$ ist *isomorph* zu $G = (V, E)$ (kurz: $H \cong G$), wenn die Graphen durch Umbenennung (bijektive Abbildung $h : V_2 \rightarrow V$) ineinander überführt werden können:
  $(V_H, E_H) \cong (V, E) :\Leftrightarrow \exists h : V \rightarrow V_H.(h \text{ bijektiv} \wedge E_H = \{\{h(u), h(v)\} \mid \{u, v\} \in E\})$
- Die *Größe* $|G|$ eines Graphen $G = (V, E)$ ist die Anzahl $|E|$ seiner Kanten.
- Der *Komplementärgraph* des Graphen $G = (V, E)$ ist der Graph $G^c = (V, E^c)$ mit $E^c = \{ \{v, v'\} \mid v, v' \in V \} - E$.
- Eine *Clique* der Größe k im Graphen $G = (V, E)$ ist eine vollständig verbundene Knotenmenge $V' \subseteq V$ mit $|V| = k$.
  Dabei heißt V' *vollständig verbunden*, wenn gilt: $\forall v, v' \in V'. v \neq v' \Rightarrow \{v, v'\} \in E$
- Eine *unabhängige Knotenmenge* der Größe k im Graphen $G = (V, E)$ ist eine Knotenmenge $V' \subseteq V$ mit $|V| = k$ mit der Eigenschaft $\forall v, v' \in V'. v \neq v' \Rightarrow \{v, v'\} \notin E$
- Eine *Knotenüberdeckung* (Vertex cover) des Graphen $G = (V, E)$ ist eine Knotenmenge $V' \subseteq V$ mit der Eigenschaft $\forall \{v, v'\} \in E. v \in V' \vee v' \in V'$
- Ein *Zyklus* (*Kreis*) in einem Graphen $G = (V, E)$ ist eine Menge $V_z = \{v_1, .., v_n\} \subseteq V$ mit der Eigenschaft $\forall i < n. \{v_i, v_{i+1}\} \in E \wedge \{v_n, v_1\} \in E$)
- Ein *Hamiltonscher Kreis* im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
- Ein Graph $G = (V, E)$ heißt zusammenhängend, wenn jeder Knoten in V von jedem anderen Knoten über Kanten aus E erreichbar ist.
- Ein *Baum* ist ein zyklenfreier zusammenhängender Graph.
- Ein *spannender Baum* in einem Graphen $G = (V, E)$ ist ein Subgraph $G_B = (V, E_B)$ von G, der ein Baum ist.
- Ein *minimal spannender Baum* in einem gewichteten Graphen (G, g) ist ein spannender Baum von G mit minimalem Gesamtgewicht.

Detailliertere Formulierungen mancher Konzepte findet man z.B. in

- S. O. Krumke, H. Noltemeier: *Graphentheoretische Konzepte und Algorithmen*, Teubner 2005.
- C. Meinel, M. Mundhenk: *Mathematische Grundlagen der Informatik*, Teubner 2002.
- K. Denecke: *Algebra und Diskrete Mathematik für Informatiker*, Teubner 2003.

Lösung 3.1 *Ziel dieser Aufgabe ist es, die Formalisierung von Problemspezifikationen einzuüben und dabei eventuell fehlende Begriffe zu formalisieren, sofern dies für die Problemstellung sinnvoller ist als die Verwendung der entsprechenden komplexeren Ausdrücke.*

Die Bedingung dafür, daß die n Damen einander nicht schlagen können ist, daß in jeder waagerechten, senkrechten, aufwärts- und abwärts-diagonalen Reihe höchstens eine Dame steht. Da es nur n waagerechte und senkrechte Reihen gibt, muß jede Dame in genau einer dieser Reihen stehen. Damit genügt es, anstelle einer Repräsentation des gesamten Schachbretts die waagerechten Positionen der Damen in jeder senkrechten Reihe darzustellen, also eine Folge L von n Zahlen zwischen 1 und n zu verwalten. Da in jeder waagerechten Zeilee nur eine Dame stehen kann, darf diese Folge keine doppelten Vorkommen enthalten, muß also eine Permutation der Zahlen $\{1..n\}$ sein.

Zusätzlich müssen die aufwärts- und abwärts-diagonalen Reihen sicher sein: steht in Reihe `i` an Position L`[i]` eine Dame, so darf die Position L`[j]` der Dame in Reihe `j` nicht genau L`[i]+(j-i)` oder L`[i]-(j-i)` sein. Dies beschreiben wir durch zwei neue Einzelbegriffe, die wir insgesamt mit dem Begriff `safe(`L`)` zusammenfassen.

```
perm(L,S)              ≡ nodups(L) ∧ range(L)=S
free_up_diagonal(L)    ≡ ∀i,j<|L|.i≠j ⇒ L[i]+(j-i)≠L[j]
free_down_diagonal(L)  ≡ ∀i,j<|L|.i≠j ⇒ L[i]-(j-i)≠L[j]
safe(L)                ≡ free_up_diagonal(L) ∧ free_down_diagonal(L)
```

Man kann das Prädikat `safe(`L`)` auch direkter ausdrücken:

```
safe(L)                ≡ ∀i,j<|L|.i≠j ⇒ |L[i]-L[j]| ≠ |i-j|
```

Die Spezifikation lautet nun

```
FUNCTION queens(n:ℕ):Set(Seq(ℤ)) WHERE true
  RETURNS {nq | perm(nq,{1..n}) ∧ safe(nq)}
```

Eine Modellierung mit Arrays wäre ebenfalls möglich, führt aber zu einer aufwendigeren Spezifikation und Lösung.

Lösung 3.2

Die Synthese des Quicksort-Algorithmus verfolgt die Reihenfolge, die wir auch bei der `merge`-Funktion benutzt haben. Zunächst wird eine Listenerzeugungsfunktion als `Compose`-Operator samt Spezifikationsprädikat O_C und Domain `D'` *ausgewählt*. Hierzu passend bestimmen wir `G` (`sort` oder `Id`) und $\succ$, stellen mittels Axiom 2 die Spezifikation für `Decompose` auf, synthetisieren diese separat, und bestimmen schließlich die primitiven Eingaben und deren direkte Lösung.

1. Wähle Listenerzeugungsfunktion `append` (‘$\circ$’) als `Compose`-Operator.

   Deren Spezifikation ist O_C`(`S_1`,`S_2`,S)` $\hat{=}$ `S=`$S_1$$\circ$$S_2$ und der Domain der Hilfsfunktion ist dementsprechend `D'` $\hat{=}$ `Seq(`$\mathbb{Z}$`)`. Das bedeutet, daß die Dekompositionsfunktion die Liste `L` in zwei Listen L_1 und L_2 zerteilen wird.

2. Die Wahl von `D'` $\hat{=}$ `Seq(`$\mathbb{Z}$`)` läßt es zu, als Hilfsfunktion `G` wieder die Funktion `G` $\hat{=}$ `sort` zu wählen, was uns `O'(`L_1`,`S_1`)` $\hat{=}$ `SORT(`L_1`,`S_1`)` $\hat{=}$ `rearranges(`L_1`,`S_1`)`$\wedge$`ordered(`S_1`)` und `I'(`L_1`)` $\hat{=}$ `true` liefert.

3. Als wohlfundierte Verkleinerungsrelation $\succ$ auf `Seq(`$\mathbb{Z}$`)` wählen wir die übliche Längenordnung für Listen: `L`$\succ$$L_2$ $\hat{=}$ `|L|>|`L_2`|`.

   Man beachte, daß diese Relation aufgrund der Wahl von `G` $\hat{=}$ `sort` nicht nur auf L_2 sondern auch auf L_1 anewandt werden muß.

4. Wir konstruieren nun die Spezifikation für `Decompose` mithilfe von Axiom 2. Es muß gelten

   $O_D(L, L_1, L_2) \wedge SORT(L_1, S_1) \wedge SORT(L_2, S_2) \wedge S = S_1 \circ S_2 \Rightarrow SORT(L, S)$

   Hierbei müssen wir nun die Definition von $SORT(L,S) \hat{=} rearranges(L,S) \wedge ordered(S)$ auflösen, jedes Vorkommen von S durch $S_1 \circ S_2$ ersetzen, und dann Gesetze über `rearranges`, `ordered` und `append` anwenden, in denen *hinreichende* Bedingungen für die Gültigkeit von Schlüssen genannt sind, die sich ausschließlich durch L, L_1 und L_2 ausdrücken lassen.

   Dies ist notwendigerweise ein heuristischer Schritt, der nur auf dem bereits vorhandenem Wissen aufsetzen kann, wenn er automatisch ablaufen soll.

   - $rearranges(L_1, S_1) \wedge rearranges(L_2, S_2) \Rightarrow rearranges(L, S_1 \circ S_2)$ gilt unter der Voraussetzung, daß $rearranges(L, L_1 \circ L_2)$ gilt. Lemma B.2.26.7/12
   - $ordered(S_1) \wedge ordered(S_2) \Rightarrow ordered(S_1 \circ S_2)$ hat als Voraussetzung, daß alle Elemente von S_1 kleiner als alle Elemente von S_2 sind. Lemma über `ordered`

     Wir kürzen dies ab durch $S_1 \leq S_2 \hat{=} \forall x_1 \in S_1. \forall x_2 \in S_2. x_1 \leq x_2$
   - Die Voraussetzung $S_1 \leq S_2$ benutzt noch die falschen Variablen und wir müssen sie in eine Aussage über L_1 und L_2 umformulieren. Dies ist jedoch nicht schwer, da sich jede Aussage der Form $\forall x_i \in S_i.\ p(x_i)$ in $\forall x_i \in L_i.\ p(x_i)$ umformulieren läßt, wenn $rearranges(L_i, S_i)$ gilt. $S_1 \leq S_2$ läßt sich daher äquivalent in $L_1 \leq L_2$ umwandeln. Lemma B.2.26.3
   - Zusammen mit der Relation $\succ$ ergibt sich somit als Nachbedingung $O_D(L, L_1, L_2)$ die Formel $|L|>|L_1| \wedge |L|>|L_2| \wedge rearranges(L, L_1 \circ L_2) \wedge L_1 \leq L_2$
   - Die obige Nachbedingung ist nur erfüllbar, wenn L mindestens ein Element enthält. Die nachfolgende Synthese der Decompositionsfunktion wird jedoch zeigen, daß diese Voraussetzung nicht ausreicht, um einen guten Algorithmus zu synthetisieren. Vielmehr muß gefordert werden, daß L mindestens 2 Elemente enthält. Wir drücken dies durch eine Formel über die Struktur von L aus, nämlich $rest(L) \neq []$

   Insgesamt erhalten wir als Spezifikation für `Decompose`

   $f_d(L{:}Seq(\mathbb{Z})){:}Seq(\mathbb{Z}) \times Seq(\mathbb{Z})$ WHERE $rest(L) \neq []$
   RETURNS L_1, L_2 SUCH THAT $|L|>|L_1| \wedge |L|>|L_2| \wedge rearranges(L, L_1 \circ L_2) \wedge L_1 \leq L_2$

   Diese Spezifikation wird durch die Funktion `part` erfüllt, die wir unten synthetisieren werden.

5. Die Vorbedingung für Korrektheit von `Decompose` war $rest(L) \neq []$, was Listen mit $rest(L) = []$ zu primitiven Eingaben werden läßt, für die wir eine direkte Lösung benötigen. Wir stellen hierzu die Spezifikation auf

   $f_p(L{:}Seq(\mathbb{Z})){:}Seq(\mathbb{Z})$ WHERE $rest(L) = []$
   RETURNS S SUCH THAT $rearranges(L,S) \wedge ordered(S)$

   Eine direkte Lösung ist naheliegend, da $rest(L) = []$ äquivalent ist zu $L = [] \vee L = [first(L)]$ und somit $rearranges(L,S)$ nur $S = L$ zuläßt. Da sowohl leere als auch einelementige Listen geordnet sind, ist $Directly\text{-}Solve(L) \hat{=} L$ die gewünschte Lösung.

6. Damit sind alle Komponenten bestimmt. Wir instantiieren den Divide & Conquer Algorithmus und erhalten

   FUNCTION sort(L:Seq($\mathbb{Z}$)):Seq($\mathbb{Z}$)
   RETURNS S SUCH THAT $rearranges(L,S) \wedge ordered(S)$
   = if $rest(L) = []$ then L
   else let $L_1, L_2 = part(L)$ in $sort(L_1) \circ sort(L_2)$

Offen bleibt noch die Synthese der Funktion `part`, die wir wie folgt spezifiziert hatten.

```
FUNCTION part(L:Seq(ℤ)):Seq(ℤ)×Seq(ℤ)  WHERE rest(L)≠[]
  RETURNS L₁,L₂  SUCH THAT |L|>|L₁| ∧ |L|>|L₂| ∧ rearranges(L,L₁∘L₂) ∧ L₁≤L₂
```

Die Synthese geht in der gleichen Reihenfolge vor wie die des Mergesort-Algorithmus. Zunächst wird eine Listenzerlegungsfunktion als `Decompose`-Operator samt Verkleinerungsrelation $\succ$, Spezifikationsprädikat O_D und Domain `D'` *ausgewählt*. Hierzu passend bestimmen wir `G` (`part` oder `Id`) und die passende Vorbedingung, verifizieren `Decompose` mit Axiom 3 und erhalten darüber ein Prädikat für primitiven Eingaben. Mittels Axiom 2 stellen wir eine Spezifikation für `Decompose` auf, synthetisieren diese separat (, was einfach ist), und bestimmen schließlich die direkte Lösung für primitive Eingaben.

1. Wir wählen die Listenzerlegungsfunktion `FirstRest` als `Decompose`-Operator.

   Deren Spezifikation ist O_D`(L,a',L')` $\hat{=}$ `L=a'.L'` und der Domain der Hilfsfunktion ist dementsprechend `D'` $\hat{=}$ $\mathbb{Z}$.

2. Als wohlfundierte Verkleinerungsrelation $\succ$ auf `Seq(`$\mathbb{Z}$`)` wählen wir wieder `L`$\succ$`L'` $\hat{=}$ `|L|>|L'|`.

3. Da aufgrund des Definitionsbereichs `D'` $\hat{=}$ $\mathbb{Z}$ die Hilfsfunktion `G` nicht `part` sein kann, wählen wir `G` $\hat{=}$ `Id` mit `O'(a',a)` $\hat{=}$ `a=a'` und `I'(a')` $\hat{=}$ `true`.

4. Die Verifikation von `Decompose` gemäß Axiom 3 liefert

   ```
   FUNCTION F_d(L:Seq(ℤ)):ℤ×Seq(ℤ)  WHERE rest(L)≠[] ∧ ¬primitive(L)
     RETURNS a',L'  SUCH THAT |L|>|L'| ∧ L=a'.L' ∧ rest(L')≠[]
   =  FirstRest(L)
   ```

   Da `L'` genau `rest(L)` ist und – aufgrund der Vorbedingung des rekursiven Aufrufs von `part` – keinen leeren Rest haben darf, muß als zusätzliche Vorbedingung `rest(rest(L))`$\neq$`[]` gefordert werden, was zu `primitive(L)` $\hat{=}$ `rest(rest(L))=[]` führt.

5. Axiom 2 liefert uns nun die Nachbedingung der Kompositionsfunktion für `part`:

   `L=a'.L'` $\wedge$ `a=a'` $\wedge$ `PART(L',L₁',L₂')` $\wedge$ O_C`(a,L₁',L₂',L₁,L₂)` $\Rightarrow$ `PART(L,L₁,L₂)`

   Dabei ist `PART(L,L₁,L₂)` eine Abkürzung für `|L|>|L₁|` $\wedge$ `|L|>|L₂|` $\wedge$ `rearranges(L,L₁∘L₂)` $\wedge$ `L₁≤L₂`, die wir natürlich gleich wieder auflösen müssen, wobei wir für `L` immer `a.L'` einsetzen und in O_C jedes Vorkommen von `L'` durch die anderen 5 Parameter ersetzen müssen.

   - `rearranges(L',L₁'∘L₂')` $\Rightarrow$ `rearranges(a.L',L₁∘L₂)` gilt unter der Voraussetzung `rearranges(a.L₁'∘L₂',L₁∘L₂)`.
   - `|L|>|L₁|` wandeln wir um in `|L₁'∘L₂'|`$\geq$`|L₁|` und `|L|>|L₂|` in `|L₁'∘L₂'|`$\geq$`|L₂|`.
   - `L₁≤L₂` müssen wir unverändert stehen lassen, da die Voraussetzung `L₁'≤L₂'` sich kaum einbringen läßt, ohne gleich eine Lösung zu generieren. Es wäre jedoch ein Verlust von Informationen, die Voraussetzung `L₁'≤L₂'` völlig zu unterschlagen. Wir bringen sie daher in eine Implikation ein und erhalten `L₁'≤L₂'` $\Rightarrow$ `L₁≤L₂`.

   Insgesamt erhalten wir folgende Spezifikation für `Compose`, wobei wir der Übersichtlichkeit halber die Bedingung `L₁'≤L₂'` $\Rightarrow$ `L₁≤L₂` aufgebrochen haben.

   ```
   FUNCTION F_c(a,L₁',L₂':ℤ×Seq(ℤ)×Seq(ℤ)):Seq(ℤ)×Seq(ℤ)  WHERE  L₁'≤L₂'
     RETURNS L₁,L₂
     SUCH THAT L₁≤L₂ ∧ rearranges(a.L₁'∘L₂',L₁∘L₂) ∧ |L₁'∘L₂'|≥|L₁| ∧ |L₁'∘L₂'|≥|L₂|
   ```

   Die Lösung für dieses Problem ist denkbar einfach, da wir nur den Wert `a` in eine der beiden Listen `L₁'` oder `L₂'` einfügen müssen, also `L₁:=a.L₁'` / `L₂:=L₂'` oder `L₁:=L₁'` / `L₂:=a.L₂'`

setzen. Beide Lösungen erfüllen alle Bedingungen mit Ausnahme von $L_1 \leq L_2 \;\hat{=}\; \forall x_1 \in L_1 . \forall x_2 \in L_2 .\; x_1 \leq x_2$. Die richtige Lösung ist nun durch eine Fallunterscheidung $a.L_1' \leq L_2' \;\vee\; L_1' \leq a.L_2'$ zu bestimmen, wobei wir nun die Voraussetzung $L_1' \leq L_2'$ einbringen können, um die Bedingung zu vereinfachen. Insgesamt erhalten wir als Kompositionsfunktion

$\texttt{Compose(a,L}_1\texttt{',L}_2\texttt{')} \;\hat{=}\; \texttt{if a} \leq \texttt{L}_2\texttt{' then (a.L}_1\texttt{', L}_2\texttt{') else (L}_1\texttt{', a.L}_2\texttt{')}$

6. Als letzten Schritt generieren wir die direkte Lösung für primitive Eingaben

   $\texttt{FUNCTION f}_p\texttt{(L:Seq(}\mathbb{Z}\texttt{)):Seq(}\mathbb{Z}\texttt{)}\times\texttt{Seq(}\mathbb{Z}\texttt{)}\;\; \texttt{WHERE rest(L)} \neq \texttt{[]} \wedge \texttt{rest(rest(L))=[]}$
   $\texttt{RETURNS L}_1\texttt{,L}_2\;\; \texttt{SUCH THAT |L|>|L}_1\texttt{|} \wedge \texttt{|L|>|L}_2\texttt{|} \wedge \texttt{rearranges(L,L}_1 \circ \texttt{L}_2\texttt{)} \wedge \texttt{L}_1 \leq \texttt{L}_2$

   Die Vorbedingung $\texttt{rest(L)} \neq \texttt{[]} \wedge \texttt{rest(rest(L))=[]}$ besagt, daß L genau zwei Elemente besitzt, nämlich $x_1 \;\hat{=}\; \texttt{first(L)}$ und $x_2 \;\hat{=}\; \texttt{first(rest(L))}$. Diese müssen auf die beiden Ziellisten verteilt werden, wobei das größere in L_2 landen muß. Wieder führt eine Strategie zur Fallanalyse zum Ziel und liefert

   $\texttt{Directly-solve(L)} \;\hat{=}\; \texttt{let [x}_1\texttt{,x}_2\texttt{] = L in if x}_1 \leq \texttt{x}_2 \texttt{ then (x}_1\texttt{,x}_2\texttt{) else (x}_2\texttt{,x}_1\texttt{)}$

   Die Spezifikation wäre nicht erfüllbar, wenn L *nicht mindestens 2 Elemente hätte: eine Verteilung der Elemente von* L *auf zwei echt kleinere Listen gelingt nicht bei leeren und einelementigen Listen. Bei einer automatischen Synthese von Quicksort würde dies erst relativ spät entdeckt und würde zu einer Revision des Verfahrens führen, bei der allerdings immer nur neue Bedingungen für* primitive *generiert werden während die restlichen Schritte unverändert bleiben und jeweils nur neu auf Korrektheit überprüft werden.*

7. Wir instantiieren den Divide & Conquer Algorithmus zu

   $\texttt{FUNCTION part(L:Seq(}\mathbb{Z}\texttt{)):Seq(}\mathbb{Z}\texttt{)}\times\texttt{Seq(}\mathbb{Z}\texttt{)}\;\; \texttt{WHERE rest(L)} \neq \texttt{[]} \wedge \texttt{rest(rest(L))=[]}$
   $\texttt{RETURNS L}_1\texttt{,L}_2\;\; \texttt{SUCH THAT |L|>|L}_1\texttt{|} \wedge \texttt{|L|>|L}_2\texttt{|} \wedge \texttt{rearranges(L,L}_1 \circ \texttt{L}_2\texttt{)} \wedge \texttt{L}_1 \leq \texttt{L}_2$
   $= \texttt{if rest(rest(L))=[]}$
   $\texttt{then let [x}_1\texttt{,x}_2\texttt{] = L in if x}_1 \leq \texttt{x}_2 \texttt{ then (x}_1\texttt{,x}_2\texttt{) else (x}_2\texttt{,x}_1\texttt{)}$
   $\texttt{else let a.L' = L}$
   $\texttt{in let L}_1\texttt{',L}_2\texttt{' = part(L')}$
   $\texttt{in if } \forall \texttt{x} \in \texttt{L}_2\texttt{'.a} \leq \texttt{x then (a.L}_1\texttt{', L}_2\texttt{') else (L}_1\texttt{', a.L}_2\texttt{')}$

Gesamtlösung

$\texttt{FUNCTION sort(L:Seq(}\mathbb{Z}\texttt{)):Seq(}\mathbb{Z}\texttt{)}\;\; \texttt{RETURNS S}\;\; \texttt{SUCH THAT rearranges(L,S)} \wedge \texttt{ordered(S)}$
$= \texttt{if rest(L)=[] then L}$
$\texttt{else let L}_1\texttt{,L}_2\texttt{=part(L) in sort(L}_1\texttt{)}\circ\texttt{sort(L}_2\texttt{)}$

$\texttt{FUNCTION part(L:Seq(}\mathbb{Z}\texttt{)):Seq(}\mathbb{Z}\texttt{)}\times\texttt{Seq(}\mathbb{Z}\texttt{)}\;\; \texttt{WHERE rest(L)} \neq \texttt{[]} \wedge \texttt{rest(rest(L))=[]}$
$\texttt{RETURNS L}_1\texttt{,L}_2\;\; \texttt{SUCH THAT |L|>|L}_1\texttt{|} \wedge \texttt{|L|>|L}_2\texttt{|} \wedge \texttt{rearranges(L,L}_1 \circ \texttt{L}_2\texttt{)} \wedge \texttt{L}_1 \leq \texttt{L}_2$
$= \texttt{if rest(rest(L))=[]}$
$\texttt{then let [x}_1\texttt{,x}_2\texttt{] = L in if x}_1 \leq \texttt{x}_2 \texttt{ then (x}_1\texttt{,x}_2\texttt{) else (x}_2\texttt{,x}_1\texttt{)}$
$\texttt{else let a.L' = L}$
$\texttt{in let L}_1\texttt{',L}_2\texttt{' = part(L')}$
$\texttt{in if } \forall \texttt{x} \in \texttt{L}_2\texttt{'.a} \leq \texttt{x then (a.L}_1\texttt{', L}_2\texttt{') else (L}_1\texttt{', a.L}_2\texttt{')}$

Lösung 3.3 *Ziel dieser Aufgabe ist es, die Grundbegriffe der Programmsynthese so zu formalisieren, daß man später formale Beweise über die Synthetisierbarkeit von Programmen führen und die entsprechenden Theoreme innerhalb eines Syntheseprozesses verwenden kann.*

3.3–a Die Klasse aller Spezifikationen ist die Klasse aller 4-Tupel $spec$ = (D,R,I,O), wobei D und R Datentypen (also Elemente von TYPES $\equiv \mathbb{U}_1$, I ein Prädikat über D und O ein Prädikat über D×R ist.

Diese Klasse läßt sich als ein Datentyp SPECIFICATIONS beschreiben, der allerdings von einer höheren Ordnung ist (d.h. zu $\mathbb{U}_2$ gehört).

SPECIFICATIONS $\equiv$ D:TYPES $\times$ R:TYPES $\times$ D$\rightarrow\mathbb{B}$ $\times$ D×R$\rightarrow\mathbb{B}$

3.3–b Die Klasse aller Programme ergu]bt sich aus derjenigen der Spezifikationen durch Hinzunahme eines Programmkörpers body:D$\not\rightarrow$R. Um dies beschreiben zu können, geben wir Selektoren D(spec) und R(spec) für den Domain und Range einer Spezifikation an

PROGRAMS $\equiv$ spec:SPEC $\times$ let (D,R,I,O)= *spec* in {x:D|I(x)}$\rightarrow$R

3.3–c Die Formalisierung von Programmkorrektheit ergibt sich unmittelbar, da Terminierung durch das dom-Prädikat der rekursiven Funktionstypen beschrieben werden kann.

p ist korrekt $\equiv$ let ((D,R,I,O),body) = p in
$\forall$x:D. I(x) $\Rightarrow$ O(x,body(x))

3.3–d Erfüllbarkeit von Spezifikationen folgt ebenfalls unmittelbar

spec ist erfüllbar $\equiv$ let((D,R,I,O),body) = p in
$\exists$body:{x:D|I(x)}$\rightarrow$R. (spec,body) ist korrekt

3.3–e Die syntaktisch aufbereitete Notation für Programme läßt sich relativ leicht auf die Tupelschreibweise abbilden

FUNCTION f(x:D):R WHERE I_x RETURNS y SUCH THAT $O_{x,y}$ = *body*$_{f,x}$
$\equiv$ (D, R, $\lambda x.I_x$, $\lambda x,y.O_{x,y}$, letrec $f(x)$= *body*$_{f,x}$)

Dabei soll I_x einen beliebiger Ausdruck kennzeichnen, in dem x frei vorkommen darf.

Lösung 3.4 *Hierzu gibt es vorerst keine Musterlösung. Das folgende sind ein paar Gedanken zur Formalisierung*

Es macht wenig Sinne, Graphen axiomatisch zu beschreiben, da die übliche mathematische Definition eine direkte Beschreibung der Struktur von Graphen gibt. Diese läßt sich unmittelbar umsetzen.

Eine erste Idee wäre, Graphen als abhängige Datentypen zu formalisieren. Dies würde zu folgendem Ansatz führen

```
Graph ≡ V:FINSET × SET(V×V)
```

Hierbei ist jedoch zu bedenken, daß das `V` in `SET(V×V)` eigentlich als endliche Aufzählung von Elementen gedacht ist, von der Verwendung her aber als Datentyp eingesetzt wird. Für diese Problematik gibt es zwei Lösungen.

1. `FINSET` wird als die Teilmenge der endlichen Datentypen in $\mathbb{U}_1$ modelliert. In diesem Fall können wir rechts in der Tat die normalen Typkonstruktoren einsetzen.

   Die Modellierung von `FINSET` ist aber nicht ganz einfach (Menge der Typen in $\mathbb{U}_1$, von denen es eine injektive Abbildung in einen der Typen $\{1..n\}$ gibt) und außerdem ist es nicht leicht, auf die Elemente von `V` zuzugreifen.

2. `FINSET` wird als `Set(`$\mathbb{N}$`)` modelliert. In diesem Fall haben wir Zugriff auf die Elemente von `V`. Wir müssen aber nun die "Liste" V zu einem Typ anheben ($\bar{V}$:= `{X:Set(`$\mathbb{N}$`)|X=V}`, um `E` definieren zu können.

Beide Lösungen stellen sich im Endeffekt als zu kompliziert heraus.

Einfacher ist es, zunächst die Struktur von Graphen (Eine Menge von Zahlen und eine von Zahlenpaaren) zu definieren und dann den Zusammenhang zwischen Knoten und Kanten als Prädikat zu formulieren

```
GraphStrukt ≡ Set(ℕ) × Set(ℕ×ℕ)
isgraph(G) = let (V,E)=G in ∀e∈E. e.1∈V ∧ e.2∈V ∧ e.1≠e.2
Graph ≡ {G:GraphStrukt | isgraph(G)}
```

Analog für ungerichtete Graphen

```
binset(T) ≡ e,e:T×T// (e.1=e'.2 ∧ e.2=e'.1)
GraphStrukt-undirected ≡ Set(ℕ) × Set(binset(ℕ))
Graph ≡ {G:GraphStrukt | isgraph(G)}
```

Man beachte, daß durch die Verwendung des Quotiententyps in `binset` nach wie vor Paar-Operationen verwendet werden können.

Auf dieser Basis können nun die weiteren Konzepte formalisiert werden.