Übung zur Vorlesung

Automatisierte Logik und Programmierung

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2006

Blatt 4 — Abgabetermin: —

Aufgabe 4.1

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.

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

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

4.1–c Programmkorrektheit als ein "Prädikat" p `ist korrekt` *(beachten Sie Terminierung!)*

4.1–d Erfüllbarkeit von Spezifikationen als ein "Prädikat" *spec* `ist erfüllbar`

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

Aufgabe 4.2 (Synthese durch Transformationen)

Gegeben sei die folgende Spezifikation des Sortierproblems

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

wobei das Prädikat `ordered` wie folgt definiert sei

`ordered(`S`)` $\equiv$ $\forall$`i` $\in$ `{1..|`S`|-1}.`S`[i]≤`S`[i+1]`

Synthetisieren Sie ein Sortierprogramm durch Anwendung von Transformationen.
Stellen Sie dazu eine Spezifikationsformel auf und transformieren Sie die Ausgabebedingung durch Äquivalenzumformungen bis ein rekursives Sortierprogramm entsteht. Begründen Sie, warum das erzeugte Programm terminiert.

Hinweise: Je nach Art der Umformungen können sehr unterschiedliche Algorithmen erzeugt werden. Stellen Sie die nötigen Lemmata, die nicht bereits im Appendix B des Skriptes zu finden sind, separat auf, ohne sie formal zu beweisen.

Aufgabe 4.3 (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 im Falle des 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 (invers zur Dekomposition von Mergesort) verhältnismäßig einfach ist und als Ausgangspunkt genommen werden sollte.

Lösung 4.1 *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.*

4.1–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` ≡ `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 `2` gehört).

`SPECIFICATIONS` ≡ `D:TYPES` × `R:TYPES` × `D`→𝔹 × `D`×`R`→𝔹

4.1–b Die Klasse aller Programme ergu]ibt sich aus derjenigen der Spezifikationen durch Hinzunahme eines Programmkörpers `body:D`↛`R`. Um dies beschreiben zu können, geben wir Selektoren `D(spec)` und `R(spec)` für den Domain und Range einer Spezifikation an

`PROGRAMS` ≡ `spec:SPEC` × `let (D,R,I,O)=` *spec* `in {x:D|I(x)}`→`R`

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

p `ist korrekt` ≡ `let ((D,R,I,O),body) =` *p* `in` ∀`x:D. I(x)` ⇒ `O(x,body(x))`

4.1–d Erfüllbarkeit von Spezifikationen folgt ebenfalls unmittelbar

spec `ist erfüllbar` ≡ `let((D,R,I,O),body) =` *p* `in` ∃`body:{x:D|I(x)}`→`R. (spec,body) ist korrekt`

4.1–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` $O_{x,y}$ = $body_{f,x}$
≡ `(`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 4.2 *Diese Aufgabe hat mehrere Ziele. Zum einen soll sie zeigen, daß Lösungen, die durch Nachdenken erzeugt wurden, sich tatsächlich mit Hilfe von Äquivalenztransformationen ableiten (und damit verifizieren) läßt. Zum anderen macht sie relativ schnell deutlich, daß die Anwendung von Äquivalenztransformationen zur Lösung eines Programmierproblems ohne eine Strategie nicht zum Ziele führt.*
In diesem Übungsblatt steht die Korrektheit einer Ableitung im Vordergrund und nicht die Methode, wie diese gefunden wird. Der Algorithmus soll als logische Formel beschrieben werden und es ist zu zeigen, mit welchen Äquivalenztransformationen man zu diesem Algorithmus kommen kann. Im Laufe dieses Prozesses lohnt es, sich Gedanken über Systematik zu machen.

Entsprechend der Denkweise der Synthese durch Transformationen führen wir als erstes ein neues Prädikat `SORT` ein, das durch die folgende Äquivalenz definiert ist:

∀`L,S:Seq(`ℤ`).` `true` ⇒ `SORT(L,S)` ⇔ `rearranges(L,S)` ∧ `ordered(S)`

Da Sortierverfahren notwendigerweise rekursiv sind, muß das Ziel der Transformationen eine rekursive Formel sein, in der auf der rechten Seite außer `SORT` nur Ausdrücke vorkommen, die sich leicht in Algorithmen umwandeln lassen. Wir wollen 4 Algorithmen ableiten.

Selection Sort:
```
sort(L) = if L=[]  then []  else let y=min(L) in  y.sort(L-y)
```

Insertion Sort:
```
sort(L) = if L=[]  then []  else let x=first(L) and L'=rest(L)
                                 in  insert(x,sort(L'))
```

Quicksort:
```
sort(L) = if L=[]  then []  else let x=first(L) and L'=rest(L)
                                 in  sort(L'_{<x}) ∘ x.sort(L'_{≥x})
```

Mergesort:
```
sort(L) = if L=[]  then []  else let x=first(L) and L'=rest(L) and k=|L'|÷2
                                 in  insert(x, merge(sort(L'_[1..k]),sort(L'_[k+1..|L'|])))
```

Diese Algorithmen entsprechen den folgenen logischen Formeln.

$$\forall L,S{:}Seq(\mathbb{Z}).\ \ true \Rightarrow$$
$$\quad SSORT(L,S) \ \Leftrightarrow\ L=[] \ \wedge\ S=[]$$
$$\qquad \vee\ \exists a{:}\mathbb{Z}.\exists S'{:}Seq(\mathbb{Z}).\ \ a\in L \ \wedge\ \forall x\in L.x\geq a \ \wedge\ SSORT(L-a,S') \ \wedge\ S=a.S'$$

$$\forall L,S{:}Seq(\mathbb{Z}).\ \ true \Rightarrow$$
$$\quad ISORT(L,S) \ \Leftrightarrow\ L=[] \ \wedge\ S=[]$$
$$\qquad \vee\ \exists a{:}\mathbb{Z}.\exists L',S'{:}Seq(\mathbb{Z}).\ \ L=a.L' \ \wedge\ ISORT(L',S') \ \wedge\ S=S'_{<a}\circ x.S'_{\geq a}$$

$$\forall L,S{:}Seq(\mathbb{Z}).\ \ true \Rightarrow$$
$$\quad QSORT(L,S) \ \Leftrightarrow\ L=[] \ \wedge\ S=[]$$
$$\qquad \vee\ \exists a{:}\mathbb{Z}.\exists L',S_1,S_2{:}Seq(\mathbb{Z}).\ \ L=a.L'$$
$$\qquad\ \wedge\ QSORT(L'_{<a},S_1) \ \wedge\ QSORT(L'_{\geq a},S_2) \ \wedge\ S=S_1\circ a.S_2$$

$$\forall L,S{:}Seq(\mathbb{Z}).\ \ true \Rightarrow$$
$$\quad MSORT(L,S) \ \Leftrightarrow\ L=[] \ \wedge\ S=[]$$
$$\qquad \vee\ \exists a,k{:}\mathbb{Z}.\exists L',S',S_1,S_2{:}Seq(\mathbb{Z}).\ \ L=a.L' \ \wedge\ k=|L'|\div 2$$
$$\qquad\ \wedge\ MSORT(L'_{[1..k]},S_1) \ \wedge\ MSORT(L'_{[k+1..|L'|]},S_2)$$
$$\qquad\ \wedge\ S'=merge(S_1,S_2) \ \wedge\ S=S'_{<a}\circ a.S'_{\geq a}$$

Dabei sei definiert

$$insert(x,S) \ \equiv\ S_{<x}\circ x.S_{\geq x}$$
```
merge(S1,S2) ≡ if S1=[] then S2 else if  S2=[] then S1
                        else let x.S1'=S1 and y.S2'=S2 in
                          if x<y then x.merge(S1',S2) else y.merge(S1,S2')
```

(Genau besehen wird `merge` *als rekursive Funktion* letrec `merge(S1,S2)...` *definiert)*

Wie man sieht haben alle Formeln eine first/rest-Zerlegung gemeinsam, die entweder auf der Ausgabevariablen `S` oder auf der Eingabe `L` stattfindet. Diese Zerlegung wird in allen Fällen auf die gleiche Art begonnen werden. Die letzten beiden Formeln nehmen zusätzlich einen Split der Eingaben vor, bevor sortiert wird. Es ist daher zu erwarten, daß ihre Ableitung sich zu einem Teil aus der des Insertion-Sort Algorithmus ergibt, zu dem nun Argumente über Splitting und zusammenmischen hinzukommen werden. Alle Ableitungen werden Lemmata über `ordered` benötigen, die wir zum Schluß ergänzen müssen.

Wir beginnen mit der Herleitung der first/rest-Zerlegung und der Lösung für den Basisfall. Der Einfachheit unterdrücken wir ab sofort die Zeile $\forall L,S:Seq(\mathbb{Z}).\ true\ \Rightarrow$

$\forall L,S:Seq(\mathbb{Z}).\ \ true \Rightarrow\ \ SORT(L,S) \Leftrightarrow rearranges(L,S) \wedge ordered(S)$

B.2.3.1 mit $\mathbb{Z}$ und L + Zerlegung

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge rearranges(L,S) \wedge ordered(S)$
$\quad \vee \exists a:\mathbb{Z}.\exists L':Seq(\mathbb{Z}).\ L=a.L \wedge rearranges(L,S) \wedge ordered(S)$

Substitution

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge rearranges([],S) \wedge ordered(S)$
$\quad \vee \exists a:\mathbb{Z}.\exists L':Seq(\mathbb{Z}).\ L=a.L' \wedge rearranges(a.L',S) \wedge ordered(S)$

B.2.26.1/2

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[] \wedge ordered(S)$
$\quad \vee \exists a:\mathbb{Z}.\exists L':Seq(\mathbb{Z}).\ L=a.L' \wedge a\in S \wedge rearranges(L',S-a) \wedge ordered(S)$

Substitution

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[] \wedge ordered([])$
$\quad \vee \exists a:\mathbb{Z}.\exists L':Seq(\mathbb{Z}).\ L=a.L' \wedge a\in S \wedge rearranges(L',S-a) \wedge ordered(S)$

ordered.1

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists L':Seq(\mathbb{Z}).\ L=a.L' \wedge a\in S \wedge rearranges(L',S-a) \wedge ordered(S)$ **1**

Selection Sort

Da `rearranges` kommutativ ist, gilt 1 auch für `S` anstelle von `L` und umgekehrt.

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists S':Seq(\mathbb{Z}).\ S=a.S' \wedge a\in L \wedge rearranges(L-a,S') \wedge ordered(S)$

ordered.2

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists S':Seq(\mathbb{Z}).\ S=a.S' \wedge a\in L \wedge rearranges(L-a,S')$
$\quad \wedge ordered(S') \wedge \forall x\in S'.a\leq x$

B.2.26.12/3

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists S':Seq(\mathbb{Z}).\ S=a.S' \wedge a\in L \wedge rearranges(L-a,S')$
$\quad \wedge ordered(S') \wedge \forall x\in L-a.a\leq x$

Definition von SORT

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists S':Seq(\mathbb{Z}).\ S=a.S' \wedge a\in L \wedge SORT(L-a,S') \wedge \forall x\in L-a.a\leq x$

Arithmetik-Ergänzung

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists S':Seq(\mathbb{Z}).\ S=a.S' \wedge a\in L \wedge SORT(L-a,S') \wedge \forall x\in L-a.a\leq x \wedge a\leq a$

B.1.11.6 (analog)

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists S':Seq(\mathbb{Z}).\ S=a.S' \wedge a\in L \wedge SORT(L-a,S') \wedge \forall x\in L.a\leq x$

Umsortieren, der Lesbarkeit wegen

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists S':Seq(\mathbb{Z}).\ a\in L \wedge \forall x\in L.a\leq x \wedge\ \ SORT(L-a,S') \wedge S=a.S'$

Insertion Sort

Wir beginnen wieder bei 1

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists L':Seq(\mathbb{Z}).\ L=a.L' \wedge a\in S \wedge rearranges(L',S-a) \wedge ordered(S)$

ordered 5

$SORT(L,S) \Leftrightarrow\ \ L=[] \wedge S=[]$
$\quad \vee \exists a:\mathbb{Z}.\exists L':Seq(\mathbb{Z}).\ L=a.L' \wedge a\in S \wedge rearranges(L',S-a)$
$\quad \wedge ordered(S-a) \wedge S=(S-a)_{<a}\circ a.(S-a)_{\geq a}$ **2**

Quantifizierung von `S-a`

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'},\texttt{S'}{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{a}\in\texttt{S} \wedge \texttt{rearranges(L',S')}\\ & \wedge \texttt{ordered(S')} \wedge \texttt{S=S'}_{<a} \circ \texttt{a.(S')}_{\geq a} \wedge \texttt{S'=S-a}\end{aligned}$$

$\texttt{S'=S-a}$ ist überflüssig nach 6., $\texttt{a}\in\texttt{S}$ ist überflüssig nach B.2.9.11/2

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'},\texttt{S'}{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{rearranges(L',S')}\\ & \wedge \texttt{ordered(S')} \wedge \texttt{S=S'}_{<a} \circ \texttt{a.S'}_{\geq a}\end{aligned}$$

3

Definition von `SORT`

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'},\texttt{S'}{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{SORT(L',S')} \wedge \texttt{S=S'}_{<a} \circ \texttt{a.S'}_{\geq a}\end{aligned}$$

Quicksort

Die Ableitung ist bis zum Punkt 2 identisch mit Insertion Sort, muß dann allerdings noch ein zusätzliches Splitting der Ausgabe durchführen, die genau am Punkt `a` gespalten wird, der bereits für das Einfügen vorgesehen ist.

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'}{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{a}\in\texttt{S} \wedge \texttt{rearranges(L',S-a)}\\ & \wedge \texttt{ordered(S-a)} \wedge \texttt{S=(S-a)}_{<a} \circ \texttt{a.(S-a)}_{\geq a}\end{aligned}$$

B.2.26.12, B.2.26.16 mit $\texttt{p(x)}\equiv\texttt{x<a}$ $\texttt{p(x)}\equiv\texttt{x}\geq\texttt{a}$

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'}{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{a}\in\texttt{S} \wedge \texttt{rearranges(L',S-a)}\\ & \wedge \texttt{rearranges(L'}_{<a}\texttt{,(S-a)}_{<a}\texttt{)} \wedge \texttt{rearranges(L'}_{\geq a}\texttt{,(S-a)}_{\geq a}\texttt{)}\\ & \wedge \texttt{ordered(S-a)} \wedge \texttt{S=(S-a)}_{<a} \circ \texttt{a.(S-a)}_{\geq a}\end{aligned}$$

ordered 8

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'}{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{a}\in\texttt{S} \wedge \texttt{rearranges(L',(S-a)}_{<a}\circ\texttt{(S-a)}_{\geq a}\texttt{)}\\ & \wedge \texttt{rearranges(L'}_{<a}\texttt{,(S-a)}_{<a}\texttt{)} \wedge \texttt{rearranges(L'}_{\geq a}\texttt{,(S-a)}_{\geq a}\texttt{)}\\ & \wedge \texttt{ordered(S-a)} \wedge \texttt{S=(S-a)}_{<a} \circ \texttt{a.(S-a)}_{\geq a}\end{aligned}$$

ordered 7

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'}{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{a}\in\texttt{S} \wedge \texttt{rearranges(L',(S-a)}_{<a}\circ\texttt{(S-a)}_{\geq a}\texttt{)}\\ & \wedge \texttt{rearranges(L'}_{<a}\texttt{,(S-a)}_{<a}\texttt{)} \wedge \texttt{rearranges(L'}_{\geq a}\texttt{,(S-a)}_{\geq a}\texttt{)}\\ & \wedge \texttt{ordered((S-a)}_{<a}\texttt{)} \wedge \texttt{ordered((S-a)}_{\geq a}\texttt{)} \wedge \texttt{S=(S-a)}_{<a} \circ \texttt{a.(S-a)}_{>a}\end{aligned}$$

Quantifizierung von $\texttt{(S-a)}_{<a}$, $\texttt{(S-a)}_{\geq a}$

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'},,\texttt{S}_1,\texttt{S}_2{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'} \wedge \texttt{a}\in\texttt{S} \wedge \texttt{rearranges(L',S}_1\circ\texttt{S}_2\texttt{)}\\ & \wedge \texttt{rearranges(L'}_{<a}\texttt{,S}_1\texttt{)} \wedge \texttt{rearranges(L'}_{\geq a}\texttt{,S}_2\texttt{)}\\ & \wedge \texttt{ordered(S}_1\texttt{)} \wedge \texttt{ordered(S}_2\texttt{)} \wedge \texttt{S=S}_1\circ\texttt{a.S}_2 \wedge \texttt{S}_1\texttt{=(S-a)}_{<a} \wedge \texttt{S}_2\texttt{=(S-a)}_{\geq a}\end{aligned}$$

$\texttt{rearranges(L',S}_1\circ\texttt{S}_2\texttt{)}$ ist überflüssig nach B.2.26.7/14/13

$\texttt{S}_1\texttt{=(S-a)}_{<a} \wedge \texttt{S}_2\texttt{=(S-a)}_{\geq a}$ ist überflüssig nach B.2.26.3/B.2.12.

$\texttt{a}\in\texttt{S}$ ist überflüssig nach B.2.9.11/2

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'},\texttt{S}_1,\texttt{S}_2{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'}\\ & \wedge \texttt{rearranges(L'}_{<a}\texttt{,S}_1\texttt{)} \wedge \texttt{rearranges(L'}_{\geq a}\texttt{,S}_2\texttt{)}\\ & \wedge \texttt{ordered(S}_1\texttt{)} \wedge \texttt{ordered(S}_2\texttt{)} \wedge \texttt{S=S}_1\circ\texttt{a.S}_2\end{aligned}$$

Definition von `SORT`

$$\begin{aligned}\texttt{SORT(L,S)} \Leftrightarrow\ & \texttt{L=[]} \wedge \texttt{S=[]}\\ \vee\ & \exists \texttt{a}{:}\mathbb{Z}.\exists \texttt{L'},\texttt{S}_1,\texttt{S}_2{:}\texttt{Seq}(\mathbb{Z}).\ \texttt{L=a.L'}\\ & \wedge \texttt{SORT(L'}_{<a}\texttt{,S}_1\texttt{)} \wedge \texttt{SORT(L'}_{\geq a}\texttt{,S}_2\texttt{)} \wedge \texttt{S=S}_1\circ\texttt{a.S}_2\end{aligned}$$

Mergesort

Die Ableitung ist bis zum Punkt 3 identisch mit Insertion Sort, muß dann allerdings noch ein zusätzliches Splitting der (restlichen) Eingabe durchführen.

$$\texttt{SORT(L,S)} \Leftrightarrow \texttt{L=[]} \wedge \texttt{S=[]} \vee \exists \texttt{a:}\mathbb{Z}.\exists \texttt{L',S':Seq}(\mathbb{Z})\texttt{.L=a.L'} \wedge \texttt{rearranges(L',S')} \wedge \texttt{ordered(S')} \wedge \texttt{S=S'}_{<a} \circ \texttt{a.(S')}_{\geq a}$$

Splitting.1 mit $\texttt{k=|L'|}\div 2$

$$\texttt{SORT(L,S)} \Leftrightarrow \texttt{L=[]} \wedge \texttt{S=[]} \vee \exists \texttt{a,k:}\mathbb{Z}.\exists \texttt{L',S':Seq}(\mathbb{Z})\texttt{.L=a.L'} \wedge \texttt{k=|L'|}\div 2 \wedge \texttt{rearranges(L'}_{\texttt{[1..k]}} \circ \texttt{L'}_{\texttt{[k+1..|L'|]}}\texttt{,S')} \wedge \texttt{ordered(S')} \wedge \texttt{S=S'}_{<a} \circ \texttt{a.(S')}_{\geq a}$$

merge.3

$$\texttt{SORT(L,S)} \Leftrightarrow \texttt{L=[]} \wedge \texttt{S=[]} \vee \exists \texttt{a,k:}\mathbb{Z}.\exists \texttt{L',S',}S_1\texttt{,}S_2\texttt{:Seq}(\mathbb{Z})\texttt{.L=a.L'} \wedge \texttt{k=|L'|}\div 2 \wedge \texttt{rearranges(L'}_{\texttt{[1..k]}}\texttt{,}S_1\texttt{)} \wedge \texttt{rearranges(L'}_{\texttt{[k+1..|L'|]}}\texttt{,}S_2\texttt{)} \wedge \texttt{ordered(}S_1\texttt{)} \wedge \texttt{ordered(}S_2\texttt{)} \wedge \texttt{S'=merge(}S_1\texttt{,}S_2\texttt{)} \wedge \texttt{S=S'}_{<a} \circ \texttt{a.(S')}_{\geq a}$$

Definition von SORT

$$\texttt{SORT(L,S)} \Leftrightarrow \texttt{L=[]} \wedge \texttt{S=[]} \vee \exists \texttt{a,k:}\mathbb{Z}.\exists \texttt{L',S',}S_1\texttt{,}S_2\texttt{:Seq}(\mathbb{Z})\texttt{.L=a.L'} \wedge \texttt{k=|L'|}\div 2 \wedge \texttt{SORT(L'}_{\texttt{[1..k]}}\texttt{,}S_1\texttt{)} \wedge \texttt{SORT(L'}_{\texttt{[k+1..|L'|]}}\texttt{,}S_2\texttt{)} \wedge \texttt{S'=merge(}S_1\texttt{,}S_2\texttt{)} \wedge \texttt{S=S'}_{<a} \circ \texttt{a.(S')}_{\geq a}$$

Lemmata

Die folgenden Lemmata sind im Verlaufe der Ableitungen erforderlich geworden.

ordered

1. $\texttt{ordered([])}$
2. $\texttt{ordered(a.S')} \Leftrightarrow \texttt{ordered(S')} \wedge \forall \texttt{x} \in \texttt{S'.a} \leq \texttt{x}$
3. $\texttt{ordered(S)} \Rightarrow \forall \texttt{i,k} \in \texttt{domain(S).i} \leq \texttt{k} \Rightarrow \texttt{ordered(S}_{\texttt{[i..k]}}\texttt{)}$
4. $\texttt{a} \in \texttt{S} \Rightarrow \texttt{ordered(S)} \Rightarrow \exists \texttt{k} \in \texttt{domain(S).S}_{<a}\texttt{=S}_{\texttt{[1..k]}} \wedge \texttt{S}_{\geq a}\texttt{=S}_{\texttt{[k+1..|S|]}}$
5. $\texttt{a} \in \texttt{S} \Rightarrow \texttt{ordered(S)} \Leftrightarrow \texttt{ordered(S-a)} \wedge \texttt{S=(S-a)}_{<a} \circ \texttt{a.(S-a)}_{\geq a}$ *abgeleitet aus 4 und B.2.25.4*
6. $\texttt{a} \in \texttt{S} \wedge \texttt{ordered(S)} \Rightarrow \texttt{S'=S-a} \Leftrightarrow \texttt{S=S'}_{<a} \circ \texttt{a.S'}_{\geq a}$ *abgeleitet aus 5 und B.2.25.4*
7. $\texttt{a} \in \texttt{S} \Rightarrow \texttt{ordered(S)} \Leftrightarrow \texttt{ordered(S}_{<a}\texttt{)} \wedge \texttt{ordered(S}_{\geq a}\texttt{)}$
8. $\texttt{a} \in \texttt{S} \Rightarrow \texttt{ordered(S)} \Rightarrow \texttt{S=S}_{<a} \circ \texttt{S}_{\geq a}$

splitting

1. $\forall \texttt{k} \in \texttt{domain(L).\ L = L}_{\texttt{[1..k]}} \circ \texttt{L}_{\texttt{[k+1..|L|]}}$

merge

1. $\texttt{rearranges(merge(}S_1\texttt{,}S_2\texttt{),}S_1 \circ S_2\texttt{)}$
2. $\texttt{ordered(}S_1\texttt{)} \wedge \texttt{ordered(}S_2\texttt{)} \Rightarrow \texttt{ordered(merge(}S_1\texttt{,}S_2\texttt{))}$
3. $\texttt{ordered(S)} \wedge \texttt{rearranges(}L_1 \circ L_2\texttt{,S)} \Leftrightarrow \exists S_1,S_2\texttt{:Seq}(\mathbb{Z})\texttt{.\ ordered(}S_1\texttt{)} \wedge \texttt{ordered(}S_2\texttt{)} \wedge \texttt{rearranges(}L_1\texttt{,}S_1\texttt{)} \wedge \texttt{rearranges(}L_2\texttt{,}S_2\texttt{)} \wedge \texttt{S=merge(}S_1\texttt{,}S_2\texttt{)}$

S_1/S_2 *entsteht durch Filterung. Aufwendiger Beweis*

B.2.25 (neu!!)

3. $\texttt{a} \in \texttt{L} \Leftrightarrow \exists \texttt{k} \in \texttt{domain(L).L=L}_{\texttt{[1..k-1]}} \circ \texttt{a.L}_{\texttt{[k+1..|L|]}}$
4. $\texttt{a} \in \texttt{L} \Rightarrow \texttt{L'=L-a} \Leftrightarrow \exists \texttt{k} \in \texttt{domain(L').L=L'}_{\texttt{[1..k]}} \circ \texttt{a.L'}_{\texttt{[k+1..|L'|]}}$
5. $\texttt{a} \in \texttt{S} \Rightarrow \texttt{S}_{<a}\texttt{=(S-a)}_{<a} \wedge \texttt{S}_{\geq a}\texttt{=a.(S-a)}_{\geq a}$

Im Appendix B war ein TeX-Fehler, was $\texttt{L}_{\texttt{[k..j]}}$ betrifft. Die Indizes sind nicht zu sehen.

Lösung 4.3

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 $O'(L_1,S_1) \;\hat{=}\; \texttt{SORT}(L_1,S_1) \;\hat{=}\; \texttt{rearranges}(L_1,S_1) \wedge \texttt{ordered}(S_1)$ und $I'(L_1) \;\hat{=}\; \texttt{true}$ 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

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

   Hierbei müssen wir nun die Definition von $\texttt{SORT}(L,S) \;\hat{=}\; \texttt{rearranges}(L,S) \wedge \texttt{ordered}(S)$ 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.

   - $\texttt{rearranges}(L_1,S_1) \wedge \texttt{rearranges}(L_2,S_2) \Rightarrow \texttt{rearranges}(L, S_1{\circ}S_2)$ gilt unter der Voraussetzung, daß $\texttt{rearranges}(L, L_1{\circ}L_2)$ gilt. Lemma B.2.26.7/12
   - $\texttt{ordered}(S_1) \wedge \texttt{ordered}(S_2) \Rightarrow \texttt{ordered}(S_1{\circ}S_2)$ 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 $\texttt{rearranges}(L_i,S_i)$ 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

     $$|L|>|L_1| \;\wedge\; |L|>|L_2| \;\wedge\; \texttt{rearranges}(L,L_1{\circ}L_2) \;\wedge\; L_1{\leq}L_2$$

- 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`

`FUNCTION` 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

   `FUNCTION` 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 hierfür 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-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(`$\mathbb{Z}$`)):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,`$L_1 \circ L_2$`)` $\wedge$ $L_1 \leq L_2$

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(`$\mathbb{Z}$`)):`$\mathbb{Z}\times$`Seq(`$\mathbb{Z}$`)  WHERE rest(L)`$\neq$`[] $\wedge$ $\neg$`primitive(L)`
   `  RETURNS a',L'  SUCH THAT |L|>|L'| ∧ L=a'.L' ∧ rest(L')`$\neq$`[]`
   `=  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', `L_1`', `L_2`')` $\wedge$ O_C`(a, `L_1`', `L_2`', `L_1`, `L_2`)` $\Rightarrow$ `PART(L, `L_1`, `L_2`)`

   Dabei ist `PART(L, `L_1`, `L_2`)` eine Abkürzung für `|L|>|`L_1`|` $\wedge$ `|L|>|`L_2`|` $\wedge$ `rearranges(L,`$L_1\circ L_2$`)` $\wedge$ $L_1\leq L_2$, 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',`L_1`'`$\circ$``$L_2$`')` $\Rightarrow$ `rearranges(a.L',`$L_1\circ L_2$`)` gilt unter der Voraussetzung `rearranges(a.`$L_1$`'`$\circ$``L_2`', `$L_1\circ L_2$`)`.
   - `|L|>|`L_1`|` wandeln wir um in `|`L_1`'`$\circ$``$L_2$`'|`$\geq$`|`$L_1$`|` und `|L|>|`$L_2$`|` in `|`$L_1$`'`$\circ$``L_2`'|`$\geq$`|`L_2`|`.
   - $L_1\leq L_2$ müssen wir unverändert stehen lassen, da die Voraussetzung $L_1'\leq L_2'$ sich kaum einbringen läßt, ohne gleich eine Lösung zu generieren. Es wäre jedoch ein Verlust von Informationen, die Voraussetzung $L_1'\leq L_2'$ völlig zu unterschlagen. Wir bringen sie daher in eine Implikation ein und erhalten $L_1'\leq L_2' \Rightarrow L_1\leq L_2$.

   Insgesamt erhalten wir folgende Spezifikation für `Compose`, wobei wir der Übersichtlichkeit halber die Bedingung $L_1'\leq L_2' \Rightarrow L_1\leq L_2$ aufgebrochen haben.

   `FUNCTION` F_c`(a,`L_1`',`L_2`':`$\mathbb{Z}\times$`Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)):Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)  WHERE  `$L_1'\leq L_2'$
   `  RETURNS `L_1`,`L_2
   `  SUCH THAT `$L_1\leq L_2$` ∧ rearranges(a.`L_1`'`$\circ$``$L_2$`', `$L_1\circ L_2$`) ∧ |`$L_1$`'`$\circ$``L_2`'|`$\geq$`|`L_1`| ∧ |`L_1`'`$\circ$``L_2`'|`$\geq$`|`L_2`|`

   Die Lösung für dieses Problem ist denkbar einfach, da wir nur den Wert `a` in eine der beiden Listen L_1' oder L_2' einfügen müssen, also L_1`:=a.`L_1`'` / L_2`:=`L_2`'` oder L_1`:=`L_1`'` / L_2`:=a.`L_2`'` 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

   `Compose(a,`L_1`',`L_2`')` $\hat{=}$ `if` $a\leq L_2'$ `then (a.`L_1`', `L_2`') else (`L_1`', a.`L_2`')`

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

   `FUNCTION` f_p`(L:Seq(`$\mathbb{Z}$`)):Seq(`$\mathbb{Z}$`)×Seq(`$\mathbb{Z}$`)  WHERE rest(L)`$\neq$`[] ∧ rest(rest(L))=[]`
   `  RETURNS `L_1`,`L_2`  SUCH THAT |L|>|`L_1`| ∧ |L|>|`L_2`| ∧ rearranges(L,`$L_1\circ L_2$`) ∧ `$L_1\leq L_2$

   Die Vorbedingung `rest(L)`$\neq$`[]` $\wedge$ `rest(rest(L))=[]` besagt, daß `L` genau zwei Elemente besitzt, nämlich x_1 $\hat{=}$ `first(L)` und x_2 $\hat{=}$ `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

   `Directly-solve(L)` $\hat{=}$ `let [`x_1`,`x_2`] = L in if `$x_1\leq x_2$` then (`x_1`,`x_2`) else (`x_2`,`x_1`)`

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

```
FUNCTION part(L:Seq(ℤ)):Seq(ℤ)×Seq(ℤ)  WHERE rest(L)≠[] ∧ rest(rest(L))=[]
  RETURNS L1,L2  SUCH THAT |L|>|L1| ∧ |L|>|L2| ∧ rearranges(L,L1∘L2) ∧ L1≤L2
= if rest(rest(L))=[]
     then let [x1,x2] = L in if x1≤x2 then (x1,x2) else (x2,x1)
     else let a.L' = L
          in  let L1',L2' = part(L')
              in  if ∀x∈L2'. a≤x  then (a.L1', L2')  else (L1', a.L2')
```

Gesamtlösung

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ)  RETURNS S  SUCH THAT rearranges(L,S) ∧ ordered(S)
= if rest(L)=[]  then L
                 else  let L1,L2=part(L) in  sort(L1)∘sort(L2)

FUNCTION part(L:Seq(ℤ)):Seq(ℤ)×Seq(ℤ)  WHERE rest(L)≠[] ∧ rest(rest(L))=[]
  RETURNS L1,L2  SUCH THAT |L|>|L1| ∧ |L|>|L2| ∧ rearranges(L,L1∘L2) ∧ L1≤L2
= if rest(rest(L))=[]
     then let [x1,x2] = L in if x1≤x2 then (x1,x2) else (x2,x1)
     else let a.L' = L
          in  let L1',L2' = part(L')
              in  if ∀x∈L2'. a≤x  then (a.L1', L2')  else (L1', a.L2')
```