Übung zur Vorlesung

Automatisierte Logik und Programmierung II

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2014

Blatt 2 — Abgabetermin: 18.06.2014

Aufgabe 2.1 (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 2.2 (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 2.3 (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.

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

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

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

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

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

Lösung 2.1 *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`)` $\equiv$ `nodups(`L`)` $\wedge$ `range(`L`)=`S

`free_up_diagonal(`L`)` $\equiv$ $\forall$`i,j<|`L`|.i≠j` $\Rightarrow$ L`[i]+(j-i)≠`L`[j]`

`free_down_diagonal(`L`)` $\equiv$ $\forall$`i,j<|`L`|.i≠j` $\Rightarrow$ L`[i]-(j-i)≠`L`[j]`

`safe(`L`)` $\equiv$ `free_up_diagonal(`L`)` $\wedge$ `free_down_diagonal(`L`)`

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

`safe(`L`)` $\equiv$ $\forall$`i,j<|`L`|.i≠j` $\Rightarrow$ `|`L`[i]-`L`[j]|` $\neq$ `|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 2.2

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 $\mathtt{O}_C$ 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 $\mathtt{O}_C(\mathtt{S}_1,\mathtt{S}_2,\mathtt{S}) \mathrel{\hat{=}} \mathtt{S}=\mathtt{S}_1\circ\mathtt{S}_2$ und der Domain der Hilfsfunktion ist dementsprechend `D'` $\mathrel{\hat{=}}$ `Seq(`$\mathbb{Z}$`)`. Das bedeutet, daß die Dekompositionsfunktion die Liste `L` in zwei Listen $\mathtt{L}_1$ und $\mathtt{L}_2$ zerteilen wird.

2. Die Wahl von `D'` $\mathrel{\hat{=}}$ `Seq(`$\mathbb{Z}$`)` läßt es zu, als Hilfsfunktion `G` wieder die Funktion `G` $\mathrel{\hat{=}}$ `sort` zu wählen, was uns $\mathtt{O'}(\mathtt{L}_1,\mathtt{S}_1) \mathrel{\hat{=}} \mathtt{SORT}(\mathtt{L}_1,\mathtt{S}_1) \mathrel{\hat{=}} \mathtt{rearranges}(\mathtt{L}_1,\mathtt{S}_1) \wedge \mathtt{ordered}(\mathtt{S}_1)$ und $\mathtt{I'}(\mathtt{L}_1) \mathrel{\hat{=}} \mathtt{true}$ liefert.

3. Als wohlfundierte Verkleinerungsrelation $\succ$ auf `Seq(`$\mathbb{Z}$`)` wählen wir die übliche Längenordnung für Listen: $\mathtt{L}\succ\mathtt{L}_2 \mathrel{\hat{=}} |\mathtt{L}|>|\mathtt{L}_2|$.

   Man beachte, daß diese Relation aufgrund der Wahl von `G` $\mathrel{\hat{=}}$ `sort` nicht nur auf $\mathtt{L}_2$ sondern auch auf $\mathtt{L}_1$ 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 $SORT(L,S) \mathrel{\hat{=}} rearranges(L,S) \wedge ordered(S)$ 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 $S_1 \leq S_2 \mathrel{\hat{=}} \forall x_1 \in S_1. \forall x_2 \in S_2. x_1 \leq x_2$
   - 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`

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

   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

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

   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 $Directly\text{-}Solve(L) \mathrel{\hat{=}} L$ die gewünschte Lösung.

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

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

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 Fd(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))≠[] 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' ∧ a=a' ∧ PART(L',L1',L2') ∧ O_C(a,L1',L2',L1,L2) ⇒ PART(L,L1,L2)

   Dabei ist PART(L,L1,L2) eine Abkürzung für |L|>|L1| ∧ |L|>|L2| ∧ rearranges(L,L1∘L2) ∧ 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') ⇒ rearranges(a.L',L1∘L2) gilt unter der Voraussetzung rearranges(a.L1'∘L2',L1∘L2).
   - |L|>|L1| wandeln wir um in |L1'∘L2'|≥|L1| und |L|>|L2| in |L1'∘L2'|≥|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' ⇒ L1≤L2.

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

   ```
   FUNCTION Fc(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

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

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

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

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

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

   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

   $\texttt{FUNCTION part}(L{:}\texttt{Seq}(\mathbb{Z})){:}\texttt{Seq}(\mathbb{Z}) \times \texttt{Seq}(\mathbb{Z}) \;\; \texttt{WHERE rest}(L) \neq [] \;\wedge\; \texttt{rest}(\texttt{rest}(L)) = []$
   $\texttt{RETURNS } L_1, L_2 \;\; \texttt{SUCH THAT } |L| > |L_1| \;\wedge\; |L| > |L_2| \;\wedge\; \texttt{rearranges}(L, L_1 \circ L_2) \;\wedge\; L_1 \leq L_2$
   $= \texttt{if rest}(\texttt{rest}(L)) = []$
   $\quad \texttt{then let } [x_1,x_2] = L \texttt{ in if } x_1 \leq x_2 \texttt{ then } (x_1,x_2) \texttt{ else } (x_2,x_1)$
   $\quad \texttt{else let } a.L' = L$
   $\quad\quad \texttt{in let } L_1', L_2' = \texttt{part}(L')$
   $\quad\quad\quad \texttt{in if } \forall x \in L_2' . a \leq x \texttt{ then } (a.L_1', L_2') \texttt{ else } (L_1', a.L_2')$

Gesamtlösung

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

$\texttt{FUNCTION part}(L{:}\texttt{Seq}(\mathbb{Z})){:}\texttt{Seq}(\mathbb{Z}) \times \texttt{Seq}(\mathbb{Z}) \;\; \texttt{WHERE rest}(L) \neq [] \;\wedge\; \texttt{rest}(\texttt{rest}(L)) = []$
$\texttt{RETURNS } L_1, L_2 \;\; \texttt{SUCH THAT } |L| > |L_1| \;\wedge\; |L| > |L_2| \;\wedge\; \texttt{rearranges}(L, L_1 \circ L_2) \;\wedge\; L_1 \leq L_2$
$= \texttt{if rest}(\texttt{rest}(L)) = []$
$\quad \texttt{then let } [x_1,x_2] = L \texttt{ in if } x_1 \leq x_2 \texttt{ then } (x_1,x_2) \texttt{ else } (x_2,x_1)$
$\quad \texttt{else let } a.L' = L$
$\quad\quad \texttt{in let } L_1', L_2' = \texttt{part}(L')$
$\quad\quad\quad \texttt{in if } \forall x \in L_2' . a \leq x \texttt{ then } (a.L_1', L_2') \texttt{ else } (L_1', a.L_2')$

Was hier herauskommt ist NICHT Quicksort, da in jeder Iteration der Partitionierung das Pivotelement gewechselt wird. Deswegen muß der aufwendigere Test $\forall x \in L_2' . a \leq x$ aufgerufen werden, was die Komplexitaet (garantiert) bei $\mathcal{O}(n \log^2 n)$ landen läßt. Der Grund ist die Reduktionsordnung, die verlangt, daß die Teillisten beide echt kleiner werden. Eine (pseudo-)lexikographische Ordnung wäre da besser. Außerdem muß die Partitionierung mit einem konstanten Pivot-Element durchgeführt werden.

Dieser Algorithmus $\texttt{Part}_a(L)$ sollte vorher mit einer genaueren, eigenen Aufgabenstellung synthetisiert werden, wenn er sich nicht auf natütliche Art aus der Synthesebedingung für Decompose herleiten läßt. Dann können wir operator match verwenden, um $\texttt{Part}(L)$ auf $\texttt{Part}_a(L)$ anzupassen - die Auswahl des a wäre dann eine leichte Vorbedingung.

Lösung 2.3 *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.*

2.3–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$\to\mathbb{B}$ $\times$ D×R$\to\mathbb{B}$

2.3–b Die Klasse aller Programme ergu]ibt sich aus derjenigen der Spezifikationen durch Hinzunahme eines Programmkörpers body:D$\not\to$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)}$\to$R

2.3–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))

2.3–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)}$\to$R. (spec,body) ist korrekt

2.3–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.