Übung zur Vorlesung

Automatisierte Logik und Programmierung

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2004

Blatt 6 — Abgabetermin: —

Aufgabe 6.1 (Optimierung)

In Übung 5 haben wir einen Globalsuch-Algorithmus für das n-Damen Problem synthetisiert.

```
FUNCTION queens(n:ℤ):Set(Seq(ℤ)) WHERE n≥1
   RETURNS {nq | perm(nq,{1..n}) ∧ safe(nq)}
≡ if nodups([]) ∧ safe([]) then queens_gs(n,[]) else ∅

FUNCTION queens_gs(n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ safe(V)
  RETURNS {nq | perm(nq,{1..n}) ∧ V⊑nq ∧ safe(nq)}
≡     {nq | nq∈{V} ∧ perm(nq,{1..n}) ∧ safe(nq)}
  ∪ ⋃{ queens_gs(n,W) | W∈{V·i|i∈{1..n}} ∧ nodups(W) ∧ safe(W)}
```

Optimieren Sie diesen Algorithmus mit den in der Vorlesung vorgestellten Techniken.

Aufgabe 6.2 (Rückblick, 1. Teil)

Die folgenden Aufgaben sind vorgesehen als Kontrollfragen zur Überprüfung des eigenen Kenntnisstandes. Sie entsprechen in ihrer Thematik dem Spektrum einer mündlichen Prüfung. Die Antworten sind größtenteils im Skript bzw. auf den Folien enthalten, allerdings nur selten an auffälliger Stelle. Versuchen sie, diese zunächst ohne Ihre Unterlagen zu beantworten.

6.2–a Welche 5 prinzipiellen Möglichkeiten einer automatischen Unterstützung für formales Beweisen gibt es?

6.2–b Welche Doppelrolle spielt die Sprache ML für interaktiver Beweissysteme?

6.2–c Schreiben Sie ein ML-Programm `prim`, welches einen Primzahltest durchführt.

6.2–d Durch welche Datenstruktur werden Sequenzenbeweise in ML repräsentiert? Nennen sie auch die wichtigsten Zugriffs- und Manipulationsoperatoren.

6.2–e Welche Besonderheit besitzt der Termeditor von NuPRL und welche Vorteile ergeben sich daraus?

6.2–f Welcher Mechanismus ist erforderlich, um konservative Erweiterungen der Typentheorie durch Abstraktionen fehlerfrei zu ermöglichen?

6.2–g Erklären Sie die besondere Rolle von Taktiken bei der Beweisführung.

6.2–h Wodurch unterscheiden sich Verfeinerungs- und Transformationstaktiken?

6.2–i Warum sind Beweise, die mit Taktiken geführt werden, immer korrekt?

6.2–j Welche Voraussetzungen müssen erfüllt sein, um eine Entscheidungsprozedur in einem Beweissystem einzusetzen?

6.2–k Warum gibt es keine Entscheidungsprozedur für die gesamte Arithmetik?

6.2–l Welche Art von Problemen entscheidet die Prozedur `arith`?

6.2–m Erklären Sie die grundsätzliche Arbeitsweise der Prozedur `arith`.

6.2–n Erklären Sie die grundsätzliche Arbeitsweise der Prozedur `supinf`.

6.2–o Erklären Sie die grundsätzliche Arbeitsweise der Prozedur `equality`.

6.2–p Welche Probleme besitzen Beweissysteme, die sich ausschließlich auf Entscheidungsprozeduren und Benutzerinteraktion stützen?

6.2–q Welche Arten von Beweisführung gibt es für die Logik erster Stufe und wie funktionieren sie?

6.2–r Beschreiben Sie die Funktionsweise von JProver.

Aufgabe 6.3 (Rückblick, 2.Teil)

6.3–a Warum kann es ein vollständiges Programmsynthesesystem niemals geben?

6.3–b Beschreiben Sie die Phasen einer formalen Entwicklung von Software.

6.3–c Welche Komponenten von Programmen und Spezifikationen sind für Programmsynthese von Bedeutung?

6.3–d Geben Sie eine präzise Definition der Begriffe "Programmkorrektheit" und "Erfüllbarkeit von Spezifikationen" auf der Basis der oben genannten Komponenten.

6.3–e Welche grundsätzlichen Paradigmen für Programmsynthese kennen Sie? Beschreiben Sie die grundsätzlichen Unterschiede bei der Darstellung des Problems, der Art der logischen Inferenzen bei der Problemlösung und der Konstruktionsmethode für den endgültigen Algorithmus.

6.3–f Welches Problem führt dazu, daß das Prinzip "Beweise-als-Programme" in der Praxis bisher nur von geringer Bedeutung ist?

6.3–g Welcher Programmierstil wird durch eine Synthese mit Formeltransformationen besonders unterstützt?

6.3–h Was ist die charakterisierende Eigenschaft einer Algorithmentheorie?

6.3–i Beschreiben Sie das generelle Verfahren zur Synthese von Algorithmen mit Hilfe von Algorithmenschemata. Wodurch wird dieses Vorgehen gerechtfertigt?

6.3–j Welche Vorteile hat die Synthese mit Algorithmenschemata gegenüber anderen Verfahren?

6.3–k Beschreiben Sie die allgemeine Struktur von Divide & Conquer Algorithmen und die Voraussetzungen für ihre Korrektheit.

6.3–l Beschreiben Sie eine Strategie zur Erzeugung von Divide & Conquer Algorithmen

6.3–m Beschreiben sie die allgemeine Struktur von Globalsuchalgorithmen und die Voraussetzungen für ihre Korrektheit.

6.3–n Beschreiben Sie die allgemeine Strategie zur Erzeugung von Globalsuchalgorithmen

6.3–o Auf welche Art könnte man Synthesestrategien in das formale Konzept typentheoretischer Schlußfolgerungen integrieren?

6.3–p Beschreiben Sie die allgemeine Struktur von Lokalsuchalgorithmen und die Voraussetzungen für ihre Korrektheit.

6.3–q Beschreiben Sie die allgemeine Struktur von Problemreduktionsgeneratoren und die Voraussetzungen für ihre Korrektheit.

6.3–r Beschreiben Sie die Techniken zur Optimierung von synthetisierten Algorithmen.

Lösung 6.1

Die Lösung folgt der Vorgehensweise des Costas-Arrays Problems (animierte Version der Folien). Die Lemmata sind ähnlich - wir geben hier nur wesentlichen die Schritte an.

```
FUNCTION queens(n:ℤ):Set(Seq(ℤ)) WHERE n≥1
   RETURNS {nq | perm(nq,{1..n}) ∧ safe(nq)}
≡ if nodups([]) ∧ safe([]) then queens_gs(n,[]) else ∅
FUNCTION queens_gs(n:ℤ,V:Seq(ℤ)):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ safe(V)
  RETURNS {nq | perm(nq,{1..n}) ∧ V⊑nq ∧ safe(nq)}
≡     {nq | nq∈{V} ∧ perm(nq,{1..n}) ∧ safe(nq)}
  ∪ ⋃{ queens_gs(n,W) | W∈{V·i|i∈{1..n}} ∧ nodups(W) ∧ safe(W)}
```

- Wir beginnen mit der Optimierung der Hauptfunktion

  `nodups([])` und `safe([])` werden mit Lemma B.2.2.4.1 und B.1.11.1. zu `true` vereinfacht. Damit kann die `if`-Anweisung zu einem einfachen Aufruf der Hilfsfunktion im Hauptteil vereinfacht werden und wir erhalten

  ```
  FUNCTION queens(n:ℤ):Set(Seq(ℤ)) WHERE n≥1
     RETURNS {nq | perm(nq,{1..n}) ∧ safe(nq)}
  ≡ queens_gs(n,[])
  ```

- Die erste Menge der Hilfsfunktion wählt `nq` aus einer einelementigen Menge aus und testet dann Bedingungen an `nq`. Die resultierende Menge ist {`nq`}, wenn die bedingungen erfüllt sind und ansonsten leer. Diese Erkenntnis wird ausgedrückt durch das Lemma

  $\{z \mid z \in \{x\} \wedge P[z]\} \quad \equiv \quad \texttt{if } P[x] \texttt{ then } \{x\} \texttt{ else } \emptyset$

  und damit vereinfacht sich der erste Teil der Hilfsfunktion zu dem Ausdruck

  ```
  if perm(V),{1..n}) ∧ safe(V)  then {V}  else ∅
  ```

  Dabei wurde der Kontext noch nicht berücksichtigt. Da `range(V)⊆{1..n} ∧ nodups(V)` und `safe(V)` bereits in den Vorbedingungen steht, kann `safe(V)` entfallen und `perm(V),{1..n})` mit Lemma B.1.13.13 vereinfacht werden zu `{1..n}⊆range(V)`. Übrig bleibt damit nur

  ```
  if {1..n}⊆range(V)  then {V}  else ∅
  ```

- Im anderen Teil können wir mit Lemma B.1.15.6

  $\{ f[x,\, g[x,y]\,] \mid y \in S \wedge h[\, g[x,y]\,]\}$

  den geschachtelten Mengenausdruck vereinfachen. Dabei wird jeweils `W` durch `V·i` ersetzt und wir erhalten

  ```
  { queens_gs(n,V·i) | i∈{1..n} ∧ nodups(V·i) ∧ safe(V·i)}
  ```

  Nun können wir wiederum die Vorbedingung verwenden und Lemmata über `nodups`, `append` und beschränkte Allquantoren einsetzen (B.2.24.6 / B.1.11.2)

  ```
  nodups(V·i)  ⇔  nodups(V) ∧ i∉range(V)
  safe(V·i)    ⇔  safe(V) ∧ ∀k<|V|.  |V[k]-i|≠ |V|+1-k
  ```

  Man beachte, daß `V·i[|V|+1]=i` und `V·i[k]=V[k]` ist (Lemma B2.14.6/5).

Da `nodups(V)` und `safe(V)` bereits im Kontext der Vorbedingungen stehen, können wir uns diese aufwendigen Berechnungen ersparen und erhalten

$$\{\texttt{queens}_{gs}\texttt{(n,V·i)} \mid \texttt{i}\in\{\texttt{1..n}\} \wedge \texttt{i}\notin\texttt{range(V)} \wedge \forall\texttt{k<|V|.}\ \texttt{|V[k]-i|}\neq\texttt{|V|+1-k}\}$$

Die Hilfsfunktion lautet nun

$$\begin{array}{l}\texttt{FUNCTION queens}_{gs}\texttt{(n:}\mathbb{Z}\texttt{,V:Seq(}\mathbb{Z}\texttt{)):Set(Seq(}\mathbb{Z}\texttt{))}\\ \quad\texttt{WHERE n}\geq\texttt{1} \wedge \texttt{range(V)}\subseteq\{\texttt{1..n}\} \wedge \texttt{nodups(V)} \wedge \texttt{safe(V)}\\ \quad\texttt{RETURNS }\{\texttt{nq}\mid\texttt{perm(nq,\{1..n\})} \wedge \texttt{V}\sqsubseteq\texttt{nq} \wedge \texttt{safe(nq)}\}\\ \equiv\quad \texttt{if }\{\texttt{1..n}\}\subseteq\texttt{range(V)}\ \texttt{ then }\{\texttt{V}\}\ \texttt{ else }\emptyset\\ \quad\cup\bigcup\{\texttt{queens}_{gs}\texttt{(n,V·i)} \mid \texttt{i}\in\{\texttt{1..n}\} \wedge \texttt{i}\notin\texttt{range(V)} \wedge \forall\texttt{k<|V|.}\ \texttt{|V[k]-i|}\neq\texttt{|V|+1-k}\}\end{array}$$

- An dieser Stelle bietet sich an, endliche Differenzierung anzuwenden. Der Ausdruck $\texttt{i}\in\{\texttt{1..n}\} \wedge \texttt{i}\notin\texttt{range(V)}$ stellt eine ständig wiederkehrende iterative Berechnung dar, die besser inkrementell berechnet wird. Gleiches gilt für die Berechnung von `|V|`. Um dies vorzubereiten vereinfachen wir weiter

  Mit den Lemmata $x\in M \wedge x\notin M' \equiv x\in M\backslash M'$ und $M\subseteq M' \equiv M\backslash M'=\emptyset$ können wir in beiden Teilen der Hilfsfunktion den gleichen Mengendifferenzausdruck $\{1..n\}\backslash\text{range(V)}$ einführen, den wir dann für das endliche Differenzieren verwenden. Wir erhalten

$$\begin{array}{l}\texttt{FUNCTION queens}_{gs}\texttt{(n:}\mathbb{Z}\texttt{,V:Seq(}\mathbb{Z}\texttt{)):Set(Seq(}\mathbb{Z}\texttt{))}\\ \quad\texttt{WHERE n}\geq\texttt{1} \wedge \texttt{range(V)}\subseteq\{\texttt{1..n}\} \wedge \texttt{nodups(V)} \wedge \texttt{safe(V)}\\ \quad\texttt{RETURNS }\{\texttt{nq}\mid\texttt{perm(nq,\{1..n\})} \wedge \texttt{V}\sqsubseteq\texttt{nq} \wedge \texttt{safe(nq)}\}\\ \equiv\quad \texttt{if }\{\texttt{1..n}\}\backslash\texttt{range(V)}=\emptyset\ \texttt{ then }\{\texttt{V}\}\ \texttt{ else }\emptyset\\ \quad\cup\bigcup\{\texttt{queens}_{gs}\texttt{(n,V·i)} \mid \texttt{i}\in\{\texttt{1..n}\}\backslash\texttt{range(V)} \wedge \forall\texttt{k<|V|.}\ \texttt{|V[k]-i|}\neq\texttt{|V|+1-k}\}\end{array}$$

  Nun differenzieren wir über die Ausdrücke $\{1..n\}\backslash\text{range(V)}$ (Variable `pool`) und `|V|` (Variable `vs`), was zu folgendem veränderten Funktionenpaar führt

$$\begin{array}{l}\texttt{FUNCTION queens(n:}\mathbb{Z}\texttt{):Set(Seq(}\mathbb{Z}\texttt{)) WHERE n}\geq\texttt{1}\\ \quad\texttt{RETURNS }\{\texttt{nq}\mid\texttt{perm(nq,\{1..n\})} \wedge \texttt{safe(nq)}\}\\ \equiv\ \texttt{queens}_{gs}\texttt{(n,[],\{1..n\},0)}\end{array}$$

$$\begin{array}{l}\texttt{FUNCTION queens}_{gs}\texttt{(n:}\mathbb{Z}\texttt{,V:Seq(}\mathbb{Z}\texttt{),pool:Seq(}\mathbb{Z}\texttt{),vs:}\mathbb{Z}\texttt{):Set(Seq(}\mathbb{Z}\texttt{))}\\ \quad\texttt{WHERE n}\geq\texttt{1} \wedge \texttt{range(V)}\subseteq\{\texttt{1..n}\} \wedge \texttt{nodups(V)} \wedge \texttt{safe(V)} \wedge \texttt{pool=}\{\texttt{1..n}\}\backslash\texttt{range(V)} \wedge \texttt{vs=|V|}\\ \quad\texttt{RETURNS }\{\texttt{nq}\mid\texttt{perm(nq,\{1..n\})} \wedge \texttt{V}\sqsubseteq\texttt{nq} \wedge \texttt{safe(nq)}\}\\ \equiv\quad \texttt{if pool=}\emptyset\ \texttt{ then }\{\texttt{V}\}\ \texttt{ else }\emptyset\\ \quad\cup\bigcup\{\texttt{queens}_{gs}\texttt{(n,V·i,pool}\backslash\{\texttt{i}\}\texttt{,vs+1)} \mid \texttt{i}\in\texttt{pool} \wedge \forall\texttt{k<vs.}\ \texttt{|V[k]-i|}\neq\texttt{vs+1-k}\}\end{array}$$

- Nun bietet sich an, die Fallunterscheidung `if pool=∅ ..` über beide Teile zu distributieren und dann den Ausdruck weiter zu vereinfachen

$$\begin{array}{l}\texttt{if pool=}\emptyset\\ \quad\texttt{then }\{\texttt{V}\}\cup\bigcup\{\texttt{queens}_{gs}\texttt{(n,V·i,pool}\backslash\{\texttt{i}\}\texttt{,vs+1)} \mid \texttt{i}\in\texttt{pool} \wedge \forall\texttt{k<vs.}\ \texttt{|V[k]-i|}\neq\texttt{vs+1-k}\}\\ \quad\texttt{else }\ \emptyset\ \cup\bigcup\{\texttt{queens}_{gs}\texttt{(n,V·i,pool}\backslash\{\texttt{i}\}\texttt{,vs+1)} \mid \texttt{i}\in\texttt{pool} \wedge \forall\texttt{k<vs.}\ \texttt{|V[k]-i|}\neq\texttt{vs+1-k}\}\end{array}$$

  Im ersten Fall verwenden wir das Lemma $\bigcup\{f(\texttt{i}) \mid \texttt{i}\in\emptyset\} = \emptyset$ um den zweiten Teil der Vereinigung zu $\emptyset$ zu vereinfachen. Mit $S\cup\emptyset = \emptyset = \emptyset\cup S$ ergibt sich

$$\begin{array}{l}\texttt{if pool=}\emptyset\ \texttt{ then }\{\texttt{V}\}\\ \quad\texttt{else }\bigcup\{\texttt{queens}_{gs}\texttt{(n,V·i,pool}\backslash\{\texttt{i}\}\texttt{,vs+1)} \mid \texttt{i}\in\texttt{pool} \wedge \forall\texttt{k<vs.}\ \texttt{|V[k]-i|}\neq\texttt{vs+1-k}\}\end{array}$$

- Als Endprodukt aller Optimierungen erhalten wir

```
FUNCTION queens(n:ℤ):Set(Seq(ℤ)) WHERE n≥1
   RETURNS {nq | perm(nq,{1..n}) ∧ safe(nq)}
≡ queens_gs(n,[],{1..n},0)

FUNCTION queens_gs(n:ℤ,V:Seq(ℤ),pool:Seq(ℤ),vs:ℤ):Set(Seq(ℤ))
  WHERE n≥1 ∧ range(V)⊆{1..n} ∧ nodups(V) ∧ safe(V) ∧ pool={1..n}\range(V) ∧ vs=|V|
  RETURNS {nq | perm(nq,{1..n}) ∧ V⊑nq ∧ safe(nq)}
≡    if pool=∅  then {V}
   else ⋃{ queens_gs(n,V·i,pool\{i},vs+1) | i∈pool ∧ ∀k<vs. |V[k]-i|≠ vs+1-k}
```

Zum Abschluß könnte man nun noch Datentypverfeinerung durchführen, was – wie bei den Costas-Arrays – dazu führt `V` als umgekehrt verkettete Liste und `pool` als Bitvektor zu implementieren.

Lösung 6.2

6.2–a Welche 5 prinzipiellen Möglichkeiten einer automatischen Unterstützung für formales Beweisen gibt es?
Siehe Seite 183/184: Proof Checking, Proof Editoren, Taktiken, Entscheidungsprozeduren, Theorembeweiser-Strategien

6.2–b Welche Doppelrolle spielt die Sprache ML für interaktiver Beweissysteme?
Siehe Seite 185/186: formale Metasprache des Kalküls und Programmiersprache für Implementierung

6.2–c Schreiben Sie ein ML-Programm `prim`, welches einen Primzahltest durchführt.

```
let prim x =
letrec check y = if y*y>x then true else
           if divides x y then false else check (y+1)
   in check 2;;
```

6.2–d Durch welche Datenstruktur werden Sequenzenbeweise in ML repräsentiert? Nennen sie auch die wichtigsten Zugriffs- und Manipulationsoperatoren.
Siehe Seite 191/192: abstrakter Datentyp – i.w. rekursive Baumstruktur. Manipulation NUR durch `refine`. Zugriffe auf Deklarationen, Konklusion, Regel und Nachfolger.

6.2–e Welche Besonderheit besitzt der Termeditor von NuPRL und welche Vorteile ergeben sich daraus?
Siehe Seite 194: Strukturediteor: kein parser nötig, flexible Syntax.

6.2–f Welcher Mechanismus ist erforderlich, um konservative Erweiterungen der Typentheorie durch Abstraktionen fehlerfrei zu ermöglichen?
Siehe Seite 198/199: Substitution und Matching zweiter Stufe

6.2–g Erklären Sie die besondere Rolle von Taktiken bei der Beweisführung.
Siehe Seite 200f: flexible, benutzerdefinierte Erweiterung des Inferenzsystems ohne Sicherheitsprobleme

6.2–h Wodurch unterscheiden sich Verfeinerungs- und Transformationstaktiken?
Siehe Seite 202/203 – eine längere Erklärung

6.2–i Warum sind Beweise, die mit Taktiken geführt werden, immer korrekt?
Siehe Seite 210: sie können Beweise NUR mithilfe der festen Regeln der Theorie manipulieren.

6.2–j Welche Voraussetzungen müssen erfüllt sein, um eine Entscheidungsprozedur in einem Beweissystem einzusetzen?
Siehe Seite 211/212: entscheidbare Teiltheorie, maschinennahe Charakterisierung der Gültigkeit, Konsistenz mit Rest der Theorie... dabei Entscheidungsprozeduren als solche erklären.

6.2–k Warum gibt es keine Entscheidungsprozedur für die gesamte Arithmetik?
Siehe Seite 213: Theorie der rekursiven Funktionen ist darin enthalten.

6.2–l Welche Art von Problemen entscheidet die Prozedur `arith`?
Siehe Seite 214: elementar-arithmetische Ausdrücke – genauer erklären.

6.2–m Erklären Sie die grundsätzliche Arbeitsweise der Prozedur `arith`.
Siehe Seite 218

6.2–n Erklären Sie die grundsätzliche Arbeitsweise der Prozedur `equality`.
Siehe Seite 220

6.2–o Welche Probleme besitzen Beweissysteme, die sich ausschließlich auf Entscheidungsprozeduren und Benutzerinteraktion stützen?
Siehe Seite 223: längere Erklärung

6.2–p Welche Arten von Beweisführung gibt es für die Logik erster Stufe und wie funktionieren sie?
Siehe Kapitel 15, Folie 1 und folgende

6.2–q Beschreiben Sie die Funktionsweise von JProver.
Siehe Kapitel 15, Folien 9 ff

Lösung 6.3

Die Antworten befinden sich meist direkt auf den Folien