Quantorenelimination in der Sprache der Polynome über einem algebraisch abgeschlossenem Körper mit Charakteristik 0 in OCaml

Thomas v. Campenhausen

May 14, 2020

1 Pseudo-Division

Eine Pseudo-Division des Polynoms s durch p. Das Ziel der Divison sind drei Polynome c, q und r, so dass

$$\forall x \in \mathbb{C} : c(x) \cdot s(x) = p(x) \cdot q(x) + r(x)$$

Wobei c in x (aber nicht unbedingt in den anderen Variablen) konstant ist, und r niedrigeren Grad (in x) als p hat.

Bei Polynomen nur in x mit rationalen Koeffizienten ließe sich $c = 1$ garantieren, aber diese Voraussetzungen lassen sich hier nicht garantieren.

Dafür ist für unsere Zwecke die genaue Berechnung von q nicht notwendig.

Satz 1. *Für zwei Polynome s und p gibt es Polynome q, r und $k \in \mathbb{N}$ s. d. $a^k s(x) = p(x)q(x) + r(x)$, mit a dem führenden Koeffizienten von p und $deg(r) < deg(p)$.*

Beweis. Sei $n := deg(p), m := deg(s)$. Induktion über m.

Für $m < n$, so ist $q, k = 0$ eine Lösung.

Sonst ($n \leq m$): Wir teilen p, s auf, s. d. $p(x) = ax^n + p_0(x), s(x) = bx^m + s_0(x)$, wobei p_0, s_0 jeweils niedrigeren Grad als p, s haben.

$$\begin{aligned} as(x) &= abx^m + as_0(x) + bx^{m-n}(p(x) - ax^n - p_0(x)) \\ &= abx^m + as_0(x) + bx^{m-n}p(x) - abx^m - bx^{m-n}p_0(x) \\ &= bx^{m-n}p(x) + s'(x) \end{aligned}$$

mit $s'(x) = as_0(x) - bx^{m-n}p_0(x)$, und $deg(s') < m$, also gibt es laut Induktionshypothese k', q' und r', s. d. $a^{k'}s'(x) = p(x)q'(x) + r'(x)$ und $deg(r') < deg(p)$. Also gilt

$$\begin{aligned} a^{k'+1}s(x) &= a^{k'}(bx^{m-n}p(x) + s'(x)) \\ &= a^{k'}bx^{m-n}p(x) + a^{k'}s'(x) \\ &= a^{k'}bx^{m-n}p(x) + p(x)q'(x) + r'(x) \\ &= (a^{k'}bx^{m-n} + q'(x))p(x) + r'(x) \end{aligned}$$

Mit $q(x) = a^{k'}bx^{m-n} + q'(x)$, $k = k'+1$ und $r = r'$ ist also die Aussage des Satzes erfüllt. □

Die Induktion im obigen Satz ist auch die Grundlage der Rekursion des folgenden Algorithmus. Verbessert wird diese, indem im Fall $a = b$ damit gekürzt, und ein gesonderter Abbruch im Fall $s = 0$ eingeführt wird.

```
let pdivide =
   let shift1 x p = Fn("+",[zero; Fn("*",[Var x; p])]) in
   let rec pdivide_aux vars a n p k s =
      if s = zero then (k,s) else
      let b = head vars s and m = degree vars s in
      if m < n then (k,s) else
      let p' = funpow (m - n) (shift1 (hd vars)) p in
      if a = b then pdivide_aux vars a n p k (poly_sub vars s p')
      else pdivide_aux vars a n p (k+1)
          (poly_sub vars (poly_mul vars a s) (poly_mul vars b p')) in
   fun vars s p -> pdivide_aux vars (head vars p) (degree vars p) p 0 s;;
```

2 Signs

Der Typ `sign` dient der Erfassung des Vorzeichens von (in x konstanten) Polynomen mittels assoziativer Listen.

```
type sign = Zero | Nonzero | Positive | Negative;;
```

Da die genaue Unterscheidung zwischen positiven und negativen Vorzeichen in den komplexen Zahlen nicht sinnvoll ist, sind hier nur die "Vorzeichen" Null und Nicht-Null relevant.

```
let assertsign sgns (p,s) =
  if p = zero then if s = Zero then sgns else failwith "assertsign" else
  let p',swf = monic p in
  let s' = swap swf s in
  let s0 = try assoc p' sgns with Failure _ -> s' in
  if s' = s0 or s0 = Nonzero & (s' = Positive or s' = Negative)
  then (p',s')::(subtract sgns [p',s0]) else failwith "assertsign";;
```

Um Fälle zu unterscheiden nutzen wir die Funktion

```
let split_zero sgns pol cont_z cont_n =
  try let z = findsign sgns pol in
      (if z = Zero then cont_z else cont_n) sgns
  with Failure "findsign" ->
      let eq = Atom(R("=",[pol; zero])) in
      Or(And(eq,cont_z (assertsign sgns (pol,Zero))),
         And(Not eq,cont_n (assertsign sgns (pol,Nonzero))));;
```

Die mathematische Rechtfertigung für den letzten Fall (das Vorzeichen von `pol` ist noch nicht festgelegt) ist folgendes Lemma:

Lemma 1. *Sei P eine Formel in der Spache der komplexen Polynome. Gibt es Formeln $P_{a=0}$ und $P_{a\neq 0}$ s.d. $a = 0 \Rightarrow (P \Leftrightarrow P_{a=0})$ und $a \neq 0 \Rightarrow (P \Leftrightarrow P_{a\neq 0})$, so ist $P \Leftrightarrow (a = 0 \wedge P_{a=0}) \vee (a \neq 0 \wedge P_{a\neq 0})$.*

Beweis.

$$\begin{aligned} P \Leftrightarrow P \wedge (a = 0 \vee a \neq 0) \Leftrightarrow \quad & (P \wedge a = 0) \vee (P \wedge a \neq 0) \\ \Leftrightarrow & (P_{a=0} \wedge a = 0) \vee (P_{a\neq 0} \wedge a \neq 0) \end{aligned}$$

□

3 Main Algorithm

3.1 nonzero

Um festzustellen, ob ein Polynom $p(x) = a_n x^n + \cdots + a_0$ nicht Null ist, wäre es möglich einfach die Formel $a_n \neq 0 \vee \cdots \vee a_0 \neq 0$ zu verwenden. Da aber jedes "oder" die Laufzeit verdoppeln könnte, bietet es sich an Abkürzungen über den Vorzeichen-Kontext zu machen.

Also teilt man die Koeffizienten in die Entschiedenen (`dcs`) und Unentschiedenen (`ucs`) (bzw. bekannt/unbekannt) ein. Gibt es einen Koeffizienten, der entschieden nicht Null ist, so ist auch das gesamte Polynom nicht null. Gibt es jedoch keinen solchen Koeffizienten, aber die Liste/Menge der unentschiedenen Koeffizienten ist leer, so sind alle Koeffizienten Null und demnach ist das Polynom Null. Im letzten Fall sind alle entschiedenen Koeffizienten Null, demnach muss man nur noch die unentschiedenen Koeffizienten betrachten, über die man dann eine Kette von "oder"-s erzeugt (wie oben beschrieben) und zurückgibt.

```
let poly_nonzero vars sgns pol =
  let cs = coefficients vars pol in
  let dcs,ucs = partition (can (findsign sgns)) cs in
  if exists (fun p -> findsign sgns p <> Zero) dcs then True
  else if ucs = [] then False else
  end_itlist mk_or (map (fun p -> Not(mk_eq p zero)) ucs);;
```

Da der Algorithmus auch eine Formel erzeugen muss, die Äquivalent dazu ist, dass ein Polynom ein anderes nicht teilt, wird folgende Funktion definiert:

```
let rec poly_nondiv vars sgns p s =
  let _,r = pdivide vars s p in poly_nonzero vars sgns r;;
```

Sie formuliert die verneinte Teilbarkeit von zwei Polynomen als die Frage, ob der Rest der Division nicht Null ist.

Mit allen Hilfsfunktionen kann man nun die zentrale Funktion der Quantoreneliminination definieren.

Der Input sind die Variablen `vars`, die Listen `eqs` und `neqs`, die die Polynome beinhalten, von denen x (nicht) Nullstelle sein soll, und `sgns`, eine assoziative Liste von konstanten Polynomen und Vorzeichen als Kontext.

```
let rec cqelim vars (eqs,neqs) sgns =
  try let c = find (is_constant vars) eqs in
     (try let sgns' = assertsign sgns (c,Zero)
          and eqs' = subtract eqs [c] in
          And(mk_eq c zero,cqelim vars (eqs',neqs) sgns')
      with Failure "assertsign" -> False)
  with Failure _ ->
     if eqs = [] then list_conj(map (poly_nonzero vars sgns) neqs) else
     let n = end_itlist min (map (degree vars) eqs) in
     let p = find (fun p -> degree vars p = n) eqs in
     let oeqs = subtract eqs [p] in
     split_zero sgns (head vars p)
       (cqelim vars (behead vars p::oeqs,neqs))
       (fun sgns' ->
          let cfn s = snd(pdivide vars s p) in
          if oeqs <> [] then cqelim vars (p::(map cfn oeqs),neqs) sgns'
          else if neqs = [] then True else
          let q = end_itlist (poly_mul vars) neqs in
          poly_nondiv vars sgns' p (poly_pow vars q (degree vars p)));;
```