Presburger-Arithmetik II - Coopers Algorithmus

Erik Sturzenhecker

30. April 2020

Um Quantoren-Elimination für die Presburger-Arithmetik zu beweisen und einen Algorithmus dafür zu erhalten, brauchen wir (nur) eine Quantoren-Elimination für $\exists x.P[x]$, wobei $P[x]$ eine quantorenfreie Formel in NNF ist, deren Literale folgende Form haben können:

- $0 = x + a$
- $\neg(0 = x + a)$
- $0 < x + a$
- $0 < -x + a$
- $d \mid x + a$
- $\neg(d \mid x + a)$
- literal without x.

Da die ganzen Zahlen diskret sind und jede nach unten beschränkte Menge ganzer Zahlen ein minimales Element hat, gilt $\exists x.P[x]$ genau dann, wenn entweder (i) beliebig große negative x existieren sodass $P[x]$ gilt, oder (ii) ein minimales x mit $P[x]$ existiert. Wir erhalten also folgende Äquivalenz und werden für beide Fälle der rechten Seite quantorenfreie Äquivalente finden:

$$(\exists x.P[x]) \Leftrightarrow (\forall y.\exists x.x < y \wedge P[x]) \vee (\exists x.P[x] \wedge \forall y.y < x \Rightarrow \neg P[y])$$

Beliebig große negative x

Für den Fall, dass beliebig große negative x die Formel $P[x]$ erfüllen, definieren wir $P_{-\infty}[x]$, indem wir die atomaren Formeln in $P[x]$ wie folgt ersetzen:

In $P[x]$	In $P_{-\infty}[x]$
$0 = x + a$	$\bot$
$0 < x + a$	$\bot$
$0 < -x + a$	$\top$

Die atomaren Teilbarkeits-Formeln und solche, die kein x enthalten, bleiben unverändert.

Lemma 5.6. *Für hinreichend große negative x sind $P[x]$ und $P_{-\infty}[x]$ äquivalent, d.h. $\exists y.\forall x.x < y \Rightarrow (P[x] \Leftrightarrow P_{-\infty}[x])$.*

Beweis. Wir beweisen die Aussage für atomare Formeln und verfahren dann per Induktion über den Formelaufbau von $P[x]$. Wenn $P[x]$ die Form $0 = x + a$ oder $0 < x+a$ hat, so gilt $P_{-\infty}[x] = \bot$ und $\forall x.x < -a \Rightarrow (P[x] \Leftrightarrow \bot)$, also ist $-a$ Zeuge für y. Der Fall $0 < -x + a$ ist ähnlich: $P_{-\infty}[x] = \top$ und $\forall x.x < a \Rightarrow (P[x] \Leftrightarrow \top)$. Für die restlichen atomaren Formeln ist $P_{-\infty}[x]$ das gleiche wie $P[x]$ und die Aussage ist trivial.
Im Fall $P[x] = \neg Q[x]$ ist die Induktionsannahme $\exists y.\forall x.x < y \Rightarrow (Q[x] \Leftrightarrow Q_{-\infty}[x])$, also $\exists y.\forall x.x < y \Rightarrow (\neg Q[x] \Leftrightarrow \neg Q_{-\infty}[x])$.
Ist $P[x] = Q[x] \wedge R[x]$, so haben wir $\exists y.\forall x.x < y \Rightarrow (Q[x] \Leftrightarrow Q_{-\infty}[x])$ und $\exists z.\forall x.x < z \Rightarrow (R[x] \Leftrightarrow R_{-\infty}[x])$. Gegeben y und z können wir w als deren Minimum wählen. Somit $\exists w.\forall x.x < w \Rightarrow (P[x] \Leftrightarrow P_{-\infty}[x])$.
Analog für $P[x] = Q[x] \vee R[x]$. □

Dies ist die "Minus-Unendlich"-Transformation, geschrieben in OCaml (unter der Annahme, dass wir bereits kanonische Form hergestellt haben):

```
let rec minusinf x fm =
  match fm with
    Atom(R("=",[Fn("0",[]); Fn("+",[Fn("*",[Fn("1",[]);y]);a])]))
        when y = x -> False
  | Atom(R("<",[Fn("0",[]); Fn("+",[Fn("*",[pm1;y]);a])])) when y = x ->
        if pm1 = Fn("1",[]) then False else True
  | Not(p) -> Not(minusinf x p)
  | And(p,q) -> And(minusinf x p,minusinf x q)
  | Or(p,q) -> Or(minusinf x p,minusinf x q)
  | _ -> fm;;
```

Die nächste Beobachtung ist, dass alle Teilbarkeits-Formeln $d \mid x + a$ unverändert bleiben, wenn man x um ein Vielfaches von d verändert. Mit folgendem Code finden wir das (positive) kleinste gemeinsame Vielfache D aller ds, die in Teilformel $d \mid c \cdot x + a$ auftreten (tatsächlich wissen wir, dass zu diesem Zeitpunkt überall $c = 1$ ist):

```
let rec divlcm x fm =
  match fm with
    Atom(R("divides",[d;Fn("+",[Fn("*",[c;y]);a])])) when y = x ->
        dest_numeral d
  | Not(p) -> divlcm x p
  | And(p,q) | Or(p,q) -> lcm_num (divlcm x p) (divlcm x q)
  | _ -> Int 1;;
```

Nun sind alle atomaren Teilbarkeits-Formeln invariant, wenn x durch $x \pm kD$ ersetzt wird. Außerdem sind im Fall von $P_{-\infty}[x]$ die Teilbarkeiten die einzigen Teilformeln, die noch ein x enthalten, also gilt $P_{-\infty}[x \pm kD] \Leftrightarrow P_{-\infty}[x]$. Daher können wir für unsere Zielformel ein einfacheres Äquivalent finden:

Satz 5.7. *Für jedes quantorenfreie $P[x]$ in NNF gilt*

$$(\forall y.\exists x.x < y \wedge P[x]) \Leftrightarrow \bigvee_{i=1}^{D} P_{-\infty}[i].$$

Beweis. Nach Lemma 5.6 sind $P[x]$ und $P_{-\infty}[x]$ für hinreichend negative x äquivalent, also also ist die linke Seite äquivalent zu $\forall y.\exists x.x < y \wedge P_{-\infty}[x]$. Da $P_{-\infty}[x]$ invariant ist unter Veränderungen von x um Vielfache von D, ist dies äquivalent zu $\exists x.P_{-\infty}[x]$. Aus dem gleichen Grund ist dies schließlich äquivalent zu $\bigvee_{i=1}^{D} P_{-\infty}[i]$, denn jedes x ist kongruent zu einem dieser i modulo D. □

Ein minimales x

Nun betrachten wir die Möglichkeit, dass es ein minimales x gibt, das $P[x]$ erfüllt. In diesem Fall gilt $P[x]$, aber nicht $P[x - D]$. Da Teilbarkeits-Formeln unter Translation mit D invariant sind, muss muss eines der übrigen Literale der NNF von wahr auf falsch wechseln, wenn man x durch eine niedrigere Zahl ersetzt. Für jedes solche Literal finden wir einen "Grenzpunkt" b, sodass das Literal wahr ist für $x = b + 1$, aber falsch für $x = b$.

Wir nennen die Menge aller Grenzpunkte der relevanten Literale die B-Menge der jeweiligen Formel. Also definieren wir $B = B(P)$ zuerst für Literale

Literal $P[x]$	$B(P)$
$0 = x + a$	$\{-(a+1)\}$
$\neg(0 = x + a)$	$\{-a\}$
$0 < x + a$	$\{-a\}$
$0 < -x + a$	$\emptyset$
$d \mid x + a$	$\emptyset$
$\neg(d \mid x + a)$	$\emptyset$
Literale ohne x	$\emptyset$

und dann für eine zusammengesetzte Formel $P[x]$ in NNF:

$$B(P) = \bigcup_{L \text{ ist Literal von } P} B(L)$$

```
let rec bset x fm =
  match fm with
    Not(Atom(R("=",[Fn("0",[]); Fn("+",[Fn("*",[Fn("1",[]);y]);a])])))
        when y = x -> [linear_neg a]
  | Atom(R("=",[Fn("0",[]); Fn("+",[Fn("*",[Fn("1",[]);y]);a])]))
        when y = x -> [linear_neg(linear_add [] a (Fn("1",[])))]
  | Atom(R("<",[Fn("0",[]); Fn("+",[Fn("*",[Fn("1",[]);y]);a])]))
        when y = x -> [linear_neg a]
  | Not(p) -> bset x p
  | And(p,q) -> union (bset x p) (bset x q)
  | Or(p,q) -> union (bset x p) (bset x q)
  | _ -> [];;
```

Satz 5.8. *Es sei D das kgV aller relevanten Teiler in einer quantorenfreien NNF-Formel $P[x]$ ohne logisch negierte Ungleichungs-Literale. B sei ihre B-Menge und x sei derart, dass $P[x]$ gilt, nicht aber $P[x-D]$. Dann gibt es $b \in B$ und $1 \leq j \leq D$, sodass $x = b + j$.*

Beweis. Induktion über den NNF-Formelaufbau: Ist $P[x]$ ein Literal, folgt aus der Annahme des Satzes, dass $P[x]$ einen Grenzpunkt hat, also von der Form $0 = x+a$, $\neg(0 = x + a)$ oder $0 < x + a$ ist. Im ersten Fall folgt, da $P[x]$ gilt, dass $x = -a$ ist. Die B-Menge des Literals ist $\{-(a + 1)\}$ und tatsächlich $x = -a = -(a + 1) + j$ für $j = 1$. Im zweiten Fall folgt aus $\neg P[x - D]$, dass $0 = (x - D) + a$ gilt, also $x = -a + j$, wobei $j = D$ und $B = \{-a\}$. Im dritten Fall gilt $0 < x + a$, aber nicht $0 < (x - D) + a$, also $-a + 1 \leq x \leq -a + D$, d.h. auch hier ist $x = -a + j$ mit $1 \leq j \leq D$ und $B = \{-a\}$. Das beweist die Aussage des Satzes für alle Literale.

Nun sei entweder $P[x] = Q[x] \wedge R[x]$ oder $P[x] = Q[x] \vee R[x]$. In beiden Fällen folgt aus $P[x]$ und $\neg P[x - D]$, dass entweder $Q[x]$ gilt und $Q[x - D]$ nicht, oder aber $R[x]$ gilt und $R[x-D]$ nicht. Da die B-Menge von $P[x]$ die von $Q[x]$ und $R[x]$ enthält, folgt die Behauptung des Satzes aus der Induktionshypothese. □

Hauptttheorem über die Quantoren-Elimination

Korollar 5.9. *$P[x]$ sei quantorenfrei und in NNF mit B-Menge B und D sei das kgV aller relevanten Teiler in der Formel. Dann gilt folgende Äquivalenz:*

$$(\exists x.P[x]) \Leftrightarrow \bigvee_{j=1}^{D} \left(P_{-\infty}[j] \vee \bigvee_{b \in B} P[b + j] \right)$$

Beweis. Nach einer einfachen Umformung bleibt zu zeigen:

$$(\exists x.P[x]) \Leftrightarrow \left(\bigvee_{j=1}^{D} P_{-\infty}[j] \right) \vee \left(\bigvee_{j=1}^{D} \bigvee_{b \in B} P[b + j] \right)$$

Angenommen, $\exists x.P[x]$ gilt. Wie zuvor gilt entweder $\forall y.\exists x.x < y \wedge P[x]$ (es gibt beliebig große negative x mit $P[x]$) oder $\exists x.P[x] \wedge \forall y.y < x \Rightarrow \neg P[y]$ (es gibt ein minimales x mit $P[x]$. Im ersten Fall folgt $\bigvee_{j=1}^{D} P_{-\infty}[j]$ aus Satz 5.7. Im zweiten Fall gibt es ein x mit $P[x]$, aber $\neg P[x - D]$. Aus Satz 5.8 folgt dann $x = b + j$ für ein $b \in B$ und $1 \leq j \leq D$, also gilt $\bigvee_{j=1}^{D} \bigvee_{b \in B} P[b + j]$.

Es gelte nun die rechte Seite. Falls $\bigvee_{j=1}^{D} P_{-\infty}[j]$ gilt, existieren nach Satz 5.7 beliebig große negative x mit $P[x]$, insbesondere gilt $\exists x.P[x]$. Im anderen Fall impliziert $\bigvee_{j=1}^{D} \bigvee_{b \in B} P[b + j]$ sofort $\exists x.P[x]$. □

Um das Haupttheorem anzuwenden, müssen wir Substitutions-Instanzen wie $P[b + j]$ bilden und dabei die kanonische Form beibehalten. Die folgende Funktion ersetzt die Variable x durch einen Term t (der kein x enthält). Durch die Verwendung von `linear_cmul` und `linear_add` bleibt die kanonische Form erhalten:

```
let rec linrep vars x t fm =
  match fm with
    Atom(R(p,[d; Fn("+",[Fn("*",[c;y]);a])])) when y = x ->
        let ct = linear_cmul (dest_numeral c) t in
    Atom(R(p,[d; linear_add vars ct a]))
  | Not(p) -> Not(linrep vars x t p)
  | And(p,q) -> And(linrep vars x t p,linrep vars x t q)
  | Or(p,q) -> Or(linrep vars x t p,linrep vars x t q)
  | _ -> fm;;
```

Für den gesamten inneren Quantoren-Eliminations-Schritt, führen wir die Transformation entsprechend Korollar 5.9 aus:

```
let cooper vars fm =
  match fm with
    Exists(x0,p0) ->
        let x = Var x0 in
        let p = unitycoeff x p0 in
        let p_inf = simplify(minusinf x p) and bs = bset x p
        and js = Int 1 --- divlcm x p in
        let p_element j b =
        linrep vars x (linear_add vars b (mk_numeral j)) p in
        let stage j = list_disj                          (* ⋁ ... *)
        (linrep vars x (mk_numeral j) p_inf ::           (* P_-∞[j] *)
        map (p_element j) bs) in                         (* (P[b + j])_b∈B *)
        list_disj (map stage js)                         (* ⋁_j=1^D *)
  | _ -> failwith "cooper: not an existential formula";;
```

Der vollständige Algorithmus

Wenn aus einer geschlossenen Formel alle Quantoren eliminiert wurden, enthält das Ergebnis überhaupt keine Variablen mehr und jede atomare Formel kann ausgewertet werden als wahr (z.B. $0 < 5$ oder $2 \mid 4$) bzw. falsch (z.B. $0 = 7$). Eine solche Auswertungsfunktion kann auch in Zwischenschritten für sinnvolle Vereinfachungen verwendet werden, da z.B. $0 < -4 \wedge P$ direkt durch $\bot$ ersetzt und P ignoriert werden kann. Die folgende Hilfsfunktion assoziiert atomare Formeln mit den entsprechenden Operationen für rationale Zahlen:

```
let operations =
  ["=",(=/); "<",(</); ">",(>/); "<=",(<=/); ">=",(>=/);
   "divides",(fun x y -> mod_num y x =/ Int 0)];;
```

Wir definieren nun die Auswertungsfunktion, wobei die Aufrufe von `dest_numeral` scheitern werden, wenn `s` oder `t` kein Zahlenwert ist. In diesem Fall bleibt die atomare Formel unverändert.

```
let evalc = onatoms
  (fun (R(p,[s;t]) as at) ->
      (try if (assoc p operations) (dest_numeral s) (dest_numeral t)
           then True else False
       with Failure _ -> Atom at));;
```

Die gesamte Quantoren-Elimination wird wie üblich mit `lift_qelim afn nfn qfn` definiert. Hierbei ist `afn` eine Hilfsfunktion, die Atome vereinfacht, `nfn` stellt die Normalform her und `qfn` eliminiert einen einzelnen Quantor. In unserem Fall:

```
let integer_qelim =
  simplify ** evalc **
  lift_qelim linform (cnnf posineq ** evalc) cooper;;
```

Nun können wir geschlossene Formeln bestätigen oder widerlegen

```
# integer_qelim <<forall x y. ~(2 * x + 1 = 2 * y)>>;;
- : fol formula = <<true>>
# integer_qelim <<forall x. exists y. 2 * y <= x /\ x < 2 * (y + 1)>>;;
- : fol formula = <<true>>
# integer_qelim <<exists x y. 4 * x - 6 * y = 1>>;;
- : fol formula = <<false>>
# integer_qelim <<forall x. ~divides(2,x) /\ divides(3,x-1) <=>
                            divides(12,x-1) \/ divides(12,x-7)>>;;
- : fol formula = <<true>>
```

und in Formeln mit freien Variablen die Quantoren eliminieren:

```
# integer_qelim <<forall x. b < x ==> a <= x>>;;
- : fol formula = <<~0 < 1 * a + -1 * b + -1>>          a − b ≤ 1
```

Die Effizienz von Coopers Algorithmus lässt sich an mehreren Stellen verbessern. Eine Möglichkeit, die schon in Coopers urpsrünglichem Paper betrachtet wurde, besteht darin ggf. eine "Plus-Unendlich"-Variante anstelle von "Minus-Unendlich" zu verwenden und die B-Menge durch eine entrsprechende A-Menge zu ersetzen. Außerdem ergeben sich in der Praxis Optimierungsmöglichkeiten dadurch, dass die auftretenden Probleme nur einen kleinen Teil der Presburger-Arithmetik darstellen. So beobachtete Pratt (1977), dass meist nur Ungleichungen der Form $x \leq y + c$ vorkommen.

Die natürlichen Zahlen

Die Quantoren-Elimination für die ganzen Zahlen kann verwendet werden, um eine für die natürlichen Zahlen zu erhalten. Wir können $\mathbb{N} = \{x \in \mathbb{Z} \mid 0 \leq x\}$ bzw. $\mathbb{N} = \{x \in \mathbb{Z} \mid 0 < x\}$ identifizieren. Soll nun eine Formel in $\mathbb{N}$ interpretiert werden, können wir eine entsprechende Formel erhalten, deren Bedeutung in $\mathbb{Z}$ die gleiche ist, indem wir alle Quantoren systematisch *relativieren*:

$$\begin{array}{lcl} \forall x.P[x] & \longrightarrow & \forall x.0 \leq x \Rightarrow P[x], \\ \exists x.P[x] & \longrightarrow & \exists x.0 \leq x \wedge P[x] \end{array}$$

Die Relativierung bezüglich einer allgemeinen Bedingung `r` kann wie folgt implementiert werden:

```
let rec relativize r fm =
  match fm with
    Not(p) -> Not(relativize r p)
  | And(p,q) -> And(relativize r p,relativize r q)
  | Or(p,q) -> Or(relativize r p,relativize r q)
  | Imp(p,q) -> Imp(relativize r p,relativize r q)
  | Iff(p,q) -> Iff(relativize r p,relativize r q)
  | Forall(x,p) -> Forall(x,Imp(r x,relativize r p))
  | Exists(x,p) -> Exists(x,And(r x,relativize r p))
  | _ -> fm;;
```

Wir wenden sie auf den Fall $0 \leq x$ an, um eine Quantoren-Elimination für die natürlichen Zahlen zu erhalten:

```
let natural_qelim =
  integer_qelim ** relativize(fun x -> Atom(R("<=",[zero; Var x])));;
```

Der Unterschied wird an einer Instanz des Satzes von Bézout deutlich. Wir können die Version für die natürlichen Zahlen so verstehen, dass man jeden Geldbetrag aus 3- und 5-Cent-Briefmarken bilden könnte, was falsch ist:

```
# integer_qelim <<forall d. exists x y. 3 * x + 5 * y = d>>;;
- : fol formula = <<true>>
# natural_qelim <<forall d. exists x y. 3 * x + 5 * y = d>>;;
- : fol formula = <<false>>
# natural_qelim <<forall d. d > 7 ==> exists x y. 3 * x + 5 * y = d>>;;
- : fol formula = <<true>>
```