Hauptseminar Mathematische Logik: Entscheidbarkeit: Presburger Arithmetik

Paulina Blome

30.April 2020

1 Presburger Arithmetik

Die Presburger Arithmetik ist grob die Menge der Formeln, welche in $\mathbb{Z}$ wahr sind und ohne Multiplikation ausgedrückt werden können.

Definition 1.1.: Die Sprache der Presburger Arithmetik ist:

$$S_P = (0, 1, \neg, +, -, =, <, \leq, \geq, >, D_k \ \forall k \geq 2)\,.$$

Konkret besteht die Presburger Arithmetik aus dem folgenden Axiomensystem:

1. $\forall x.\neg(0 \equiv x + 1)$
2. $\forall x \forall y.x + 1 \equiv y + 1 \implies x \equiv y$
3. $\forall x.x + 0 \equiv x$
4. $\forall x \forall y.x + (y + 1) \equiv (x + y) + 1$
5. für alle Formeln $\phi(x_0, x_1, ..., x_{n-1}, x_n) \in L^{S_P}$:

$$\forall x_0 ... \forall x_{n-1}.\phi(x_0, x_1, ..., x_{n-1}, 0)$$
$$\wedge\ \forall x.(x > 0 \implies (\phi(x_0, x_1, ..., x_{n-1}, x) \implies \phi(x_0, x_1, ..., x_{n-1}, x + 1)))$$
$$\wedge\ \forall x.(x < 0 \implies (\phi(x_0, x_1, ..., x_{n-1}, x) \implies \phi(x_0, x_1, ..., x_{n-1}, x - 1)))$$
$$\implies \forall y.\phi(x_0, x_1, ..., x_{n-1}, y)$$

Um die Implementierung zu erleichtern, erlauben wir zusätzliche Regeln:

- Wir erlauben beliebige positive und negative ganze Zahlen als Konstanten.
- Wir erlauben die Multiplikationsfunktion nur für den Fall, dass Multiplikation mit Konstanten ausgedrückt wird.
- Wir ersetzen die Relationen D_k mit einer einzelnen zweistelligen Relation *divides* mit der Einschränkung, dass auf der linken Seite nur eine (positive) ganzzahlige Konstante stehen darf.

2 Implementierung

Erinnerung: Die Darstellung von Termen in OCaml ist :

```
type term = Var of string | Fn of string * term list
```

Wir benutzen eine bestimmte Abkürzung für den konstanten Term 0:

```
let zero = Fn("0",[]);;
```

Wir brauchen zunächst zwei Funktionen, die Terme von ganzzahligen Konstanten zu OCaml unlimited-precision numbers und andersherum umwandeln:

```
let mk_numeral n = Fn(string_of_num n, []);;

let dest_numeral t =
   match t with
     Fn(ns,[]) -> num_of_string ns
   | _ -> failwith "dest_numeral";;
```

Die nächste Funktion können wir verwenden, um zu prüfen ob ein beliebiger Term t eine ganzzahlige Konstante ist:

```
let is_numeral = can dest_numeral;;
```

Mit diesen Funktionen können wir nun beliebige einstellige oder zweistellige Operationen auf OCaml numbers umwandeln auf Operationen auf ganzzahligen Konstanten:

```
let numeral1 fn n = mk_numeral(fn(dest_numeral n));;

let numeral2 fn m n = mk_numeral(fn (dest_numeral m)(dest_numeral n));;
```

Beispiel 2.1.:
Input: Die Funktion (+/) und zwei konstante Terme 4 und 0
Output: Das Ergebnis der Funktion (+/) als Operation auf zwei Konstanten

```
# numeral2 (+/) (Fn("4", [])) (zero);;
- : term = <<|4|>>
```

3 Normalform Terme

Um die Implementierung der späteren Transformationen in der Quantoreneliminierung zu vereinfachen, formen wir alle Terme in eine kanonische Form um:

Definition 3.1.: Für einen Term $t \in T^{S_P}$ mit $var(t) = x_1, ..., x_n$ sieht die Normalform wie folgt aus:

$$c_1 \cdot x_1 + ... + c_n \cdot x_n + k$$

mit $c_i, k \in \mathbb{Z}\ \forall i \in \{1, ..n\}$.

Bemerkung: Wir achten darauf, dass:
- die c_i auch gezeigt werden, wenn sie gleich 1 sind, aber nicht gleich 0
- die k auch gezeigt wird, auch wenn es gleich 0 ist

Unser Ziel ist also einen beliebigen Term $t \in T^{S_P}$ in die kanonische Form umzuwandeln.

Als ersten Schritt konstruieren wir zwei Hauptoperationen auf Termen in der oben genannten kanonischen Form:

- Multiplikation mit einer Konstanten:

```
let rec linear_cmul n tm =
   if n =/ Int 0 then zero else
   match tm with
    Fn("+", [Fn("*",[c;x]);r]) ->
       Fn("+", [Fn("*", [numeral1(( */ ) n) c; x]); linear_cmul n r])
   | k -> numeral1(( */ ) n) k;;
```

- Addition:

```
let rec linear_add vars tm1 tm2 =
   match (tm1, tm2) with
    (Fn("+", [Fn("*", [c1; Var x1]); r1]),
     Fn("+", [Fn("*", [c2; Var x2]); r2]) ->
        if x1 = x2 then
          let c = numeral2 (+/) c1 c2 in
          if c = zero then linear_add vars r1 r2
          else Fn("+", [Fn("*", [c; Var x1]); linear_add vars r1 r2])
        else if earlier vars x1 x2 then
          Fn("+", [Fn("*", [c1; Var x1]); linear_add vars r1 tm2])
        else
          Fn("+", [Fn("*", [c2; Var x2]); linear_add vars tm1 r2])
   | (Fn("+", [Fn("*", [c1; Var x1]); r1]) , k2) ->
        Fn("+", [Fn("*", [c1; Var x1]); linear_add vars r1 k2])
   | (k1, Fn("+", [Fn("*", [c2; Var x2]); r2])) ->
        Fn("+", [Fn("*", [c2; Var x2]); linear_add vars k1 r2])
   | _ -> numeral2(+/) tm1 tm2;;
```

Beispiel 3.2.:
Input: Die Variablenliste $[x; y]$ und die beiden Terme $4 \cdot x + 2 \cdot y$ und $3 \cdot y + 1$ in kanonischer Form
Output: Die Summe der beiden Input-Terme als kanonischer Term

```
# linear_add (["x"; "y"]) (<<|4 * x + 2 * y + 0|>>) (<<|3 * y + 1|>>);;
- : term = <<|4 * x + 5 * y + 1|>>
```

Mit diesen Basisfunktionen können wir auch eine Negations-, eine Subtraktions- und eine allgemeine Multiplikationsfunktion auf Termen in der kanonischen Form definieren:

- Negation:

```
let linear_neg tm = linear_cmul (Int(-1)) tm;;
```

- Subtraktion:

```
let linear_sub vars tm1 tm2 = linear_add vars tm1 (linear_neg tm2);;
```

- Multiplikation:

```
let linear_mul tm1 tm2 =
   if is_numeral tm1 then linear_cmul (dest_numeral tm1) tm2
   else if is_numeral tm2 then linear_cmul (dest_numeral tm2) tm1
   else failwith "linear_mul: nonlinearity";;
```

Um nun einen Term $t \in T^{S_P}$ in die oben definierte Normalform umzuwandeln, benutzen wir die folgende Funktion:

```
let rec lint vars tm =
   match tm with
     Var(_) -> Fn("+", [Fn("*", [Fn("1", []); tm]); zero])
   | Fn("-", [t]) -> linear_neg (lint vars t)
   | Fn("+", [s;t]) -> linear_add vars (lint vars s) (lint vars t)
   | Fn("-", [s;t]) -> linear_sub vars (lint vars s) (lint vars t)
   | Fn("*", [s;t]) -> linear_mul (lint vars s) (lint vars t)
   | _ -> if is_numeral tm then tm else failwith "lint: unknown term";;
```

Beispiel 3.3.:
Input: Die Liste $[x; y]$ und der Term $3 \cdot y + 2 \cdot x + 4 \cdot x$
Output: Den Term $6 \cdot x + 3 \cdot y + 0$

```
# lint (["x";"y"]) (Fn("+", [Fn("+", [Fn("*", [Fn("3", []); Var "y"]); Fn("*", [Fn("2", []);
Var "x"])]); Fn("*", [Fn("4", []); Var "x"])]));;
- : term = <<|6* x + 3 * y +0|>>
```

Beispiel 3.4.: [im pretty-printing Format für Terme]
Input: Die Liste $[y; z; x]$ und der Term $x + 4 \cdot y + 0 \cdot z + 4$
Ouput: Den Term $4 \cdot y + 1 \cdot x + 4$

```
# lint (["y";"z";"x"]) (<<| x + 4 * y + 0 * z + 4|>>);;
- : term = <<|4 * y + 1 * x + 4|>>
```

4 Normalform Formeln

Wir wollen auch für eine beliebige Formel $\phi \in L^{S_P}$ eine kanonische Form entwickeln, um die spätere Quantorenelimination zu vereinfachen. Diese soll die folgenden Eigenschaften haben:

1. Wir wollen für Gleichheiten und Ungleichheiten 0 auf der linken Seite stehen haben.
2. Wir wollen "$<$" als einzige Ungleichheitsrelation.
3. In Teilbarkeitsformeln soll die linke Seite eine positive ganze Zahl sein.
4. Wir wollen, dass "$<$" nur in nicht-negierter Form vorkommt.

Um diese Forderungen umzusetzen, betrachten wir die folgenden Äquivalenzen:

1. Für jedes Relationssymbol $R \in \{=, <, \leq, \geq, >\}$ gilt: $\forall s, t \in T^{S_P} : s \ R \ t \iff 0 \ R \ (t - s)$
2. $\forall s, t \in T^{S_P} : s \leq t \iff 0 < (t + 1) - s$
3. Für alle $k \in \mathbb{Z}$ gilt: $\forall t \in T^{S_P} : k|t \iff -k|t$
4. $\forall t \in T^{S_P} : \neg(0 < t) \iff 0 < 1 - t$

Definition 4.1: Eine Formel $\phi \in L^{S_P}$ ist in der gewünschten Normalform, falls sie aus einer Konjunktion oder Disjunktion von den folgenden atomaren Formeln besteht: $0 = t, \neg(0 = t), 0 < t, d|t, \neg(d|t)$ für ein $d \in \mathbb{N}$.

Dafür brauchen wir als erstes eine Helferfunktion, um Terme zu linearisieren und so in eine atomare Formel einzufügen, dass auf der linken Seite 0 steht:

```
let mkatom vars p t = Atom(R(p, [zero; lint vars t]));;
```

Die folgende Funktion ist die Hauptfunktion, welche die Input-Formel in die kanonische Form umwandelt. Diese wird später in `lift_qelim` als Parameter `afn` eingesetzt.

```
let linform vars fm =
   match fm with
     Atom(R("divides", [c;t])) ->
         Atom(R("divides", [numeral1 abs_num c; lint vars t]))
   | Atom(R("=", [s;t])) -> mkatom vars "="(Fn("-", [t;s]))
   | Atom(R("<", [s;t])) -> mkatom vars "<"(Fn("-", [t;s]))
   | Atom(R(">", [s;t])) -> mkatom vars "<"(Fn("-", [s;t]))
   | Atom(R("<=", [s;t])) ->
         mkatom vars "<"(Fn("-", [Fn("+", [t;Fn("1", [])]);s]))
   | Atom(R(">=", [s;t])) ->
         mkatom vars "<"(Fn("-", [Fn("+", [s;Fn("1", [])]);t]))
   | _ -> fm;;
```

Die ersten 3 der oben genannten Forderungen sind hiernach erfüllt.

Beispiel 4.2.:
Input: Eine Liste der vorkommenden Variablen $[x]$ und die atomare Formel $4 \leq 2 \cdot x$
Output: Die Formel in Normalform $0 < 2 \cdot x - 3$

```
# linform (["x"]) (Atom(R("<=",[Fn("4", []); Fn("*", [Fn("2", []); Var "x"])])));;
- : fol formula = <<0 < 2 * x + -3>>
```

Wir betrachten noch den Fall, dass "$<$" negiert vorkommt. Dafür definieren wir die folgende Funktion:

```
let rec posineq fm =
   match fm with
   | Not(Atom(R("<", [Fn("0", []); t]))) ->
         Atom(R("<", [Fn("0", []); linear_sub [] (Fn("1", [])) t]))
   | _ -> fm;;
```

Beispiel 4.3: [im pretty-printing Format für Formeln]
Input: Die negierte atomare Formel $\neg(0 < 3 \cdot x + 1)$
Output: Die atomare Formel $0 < -3 \cdot x + 0$

```
# posineq (<<~(0 < 3 * x + 1)>>;;
- : fol formula = <<0 < -3 * x + 0>>
```

5 Der Algorithmus von Cooper

Presburgers Original-Algorithmus folgt einem typischen Muster der Quantoreneliminierung, indem er nur dem Spezialfall einer Formel der folgenden Form behandelt:

$$\Phi = \exists x. \bigwedge L_i \text{ , wobei die } L_i \text{ Literale sind.}$$

Wir gucken uns allerdings den Algorithmus von Cooper (1972) genauer an, welcher den Vorteil hat, einen existentiellen Quantor aus jeder Formel $\Phi = \exists x.p$, wobei p eine quantorenfreie Formel in NNF ist, eliminieren zu können.

Bemerkung: Wir gehen davon aus, dass in p nur die folgenden Literale vorkommen:

$$0 = t, \neg(0 = t), 0 < t, d|t, \neg(d|t)$$

Da wir nun eine Formel in NNF betrachten, wollen wir eine weitere Normalisierung einführen, um den Fall, dass die quantifizierte Variable in mehreren atomaren Formeln vorkommt, zu behandeln:

Um den Fall, dass x in mehreren Formeln mit unterschiedlichen Koeffizienten vorkommt, zu behandeln, suchen wir das kleinste gemeinsame Vielfache aller Koeffizienten von x:

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

Beispiel 5.1.: $\phi = \exists x.0 = x - y \wedge 0 < 3 \cdot x + 1 \vee 0 = 2 \cdot x$
Input: Die existentiell quantifizierte Variable x und der innere Teil der Formel in Normalform: $0 = 1 \cdot y - 1 \cdot x + 0 \wedge 0 < 3 \cdot x + 1 \vee 0 = 2 \cdot x + 0$
Output: Das kleinste gemeinsame Vielfache der Koeffizienten von x als OCaml number: 6

```
# formlcm (Var "x") (<<0 = 1 * x - 1 * y + 0 ∧ 0 < 3 * x +1 ∨ 0 = 2 * x + 0>>);;
- : num = 6
```

Setze l gleich dem gefundenen kgV der Koeffizienten von x.

Wir wollen als nächsten Schritt alle Koeffizienten von x zu $\pm l$ umrechnen.
Dafür multiplizieren wir rekursiv alle atomaren Formel mit $c \cdot x + z$ auf der rechten Seite mit einem angemessenen m.
Dabei setzen wir $m = \frac{l}{c}$, außer für den Fall einer Ungleichheitsformel. Hier soll $m = |\frac{l}{c}|$ sein.

```
let rec adjustcoeff x l fm =
  match fm with
    Atom(R(p, [d; Fn("+", [Fn("*", [c;y]);z])])) when y = x ->
        let m = l // dest_numeral c in
        let n = if p = "<" then abs_num(m) else m in
        let xtm = Fn("*", [mk_numeral (m // n); x]) in
        Atom(R(p, [linear_cmul (abs_num m) d;
          Fn("+", [xtm; linear_cmul n z])]))
  | Not(p) -> Not(adjustcoeff x l p)
  | And(p,q) -> And(adjustcoeff x l p, adjustcoeff x l q)
  | Or(p,q) -> Or(adjustcoeff x l p, adjustcoeff x l q)
  | _ -> fm;;
```

Beispiel 5.2.: $\phi = \exists x.0 = x - y \wedge 0 < 3 \cdot x + 1 \vee 0 = 2 \cdot x$
Input: Die existentiell quantifizierte Variable, das bereits gefundene kleinste gemeinsame Vielfache der Koeffizienten von x und der innere Teil der Formel in Normalform.
Output: Die Formel mit ± 1 als Koeffizienten von x.

```
# adjustcoeff (Var "x") (Int 6) (<<0 = 1 * x - 1 * y + 0 ∧ 0 < 3 * x + 1 ∨ 0 = 2 * x + 0>>);;
-: fol formula = <<0 = 1 * x - 1 * y + 0 ∧ 0 < 1 * x + 2 ∨ 0 = 1 * x + 0 >>
```

In der obigen Funktion geben wir die Formel sogar mit $\pm 1 \cdot x$ aus anstelle von $\pm l \cdot x$.
Um diesen Schritt zu rechtfertigen, betrachten wir die folgende Äquivalenz:

$$(\exists x.P[l \cdot x]) \iff (\exists x.l|x \wedge P[x])$$

und binden eine neue Teilbarkeitsformel ein.

Die komplette Transformation der Formeln ist in der nächsten Funktion eingebunden. Dabei werden zunächst die Koeffizienten von x auf 1 reduziert und danach ein zusätzliches Literal hinzugefügt:

```
let unitycoeff x fm =
  let l = formlcm x fm in
  let fm' = adjustcoeff x l fm in
  if l =/ Int 1 then fm' else
  let xp = Fn("+", [Fn("*", [Fn("1", []); x]); zero]) in
  And(Atom(R("divides", [mk_numeral l; xp])), adjustcoeff x l fm);;
```

Beispiel 5.3.: $\phi = \exists x.0 = x - y \wedge 0 < 3 \cdot x + 1 \vee 0 = 2 \cdot x$
Input: Die existentiell quantifizierte Variable und der innere Teil von ϕ in Normalform.
Output: Die Formel mit ± 1 als Koeffizienten von x und zusätzlicher Teilbarkeitsformel.

```
# unitycoeff (Var "x") (<<0 = 1 * x - 1 * y + 0 ∧ 0 < 3 * x + 1 ∨ 0 = 2 * x + 0>>);;
-: fol formula =
<<divides(6,1 * x + 0) ∧ (0 = 1 * x - 1 * y + 0 ∧ 0 < 1 * x + 2 ∨ 0 = 1 * x + 0 )>>
```