Modelando e verificando o cálculo formal de Planos de Corte com Lean 4

Projeto Orientadado em Computação I - Pesquisa Científica

Bernardo Borges
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
bernardoborges@dcc.ufmg.br

***Abstract*—Esta pesquisa envolve a definição e verificação da lógica pseudo-booleana de Planos de Corte utilizando o provador de teoremas Lean 4.**

***Index Terms*—lean, cutting planes, formal methods, pseudo boolean reasoning**

I. INTRODUÇÃO

O cálculo formal de Planos de Corte [5] é uma técnica matemática utilizada na área de otimização, particularmente em Programação Linear. Neste trabalho, procuramos trazer essa lógica para os resultados formalizados em `Lean 4`, seguindo a tendência crescente no mundo da matemática e atendendo a necessidade de confiança nos resultados de solucionadores. Além da formalização, procuramos aqui disponibilizar estes resultados para a comunidade em geral, para que mais integrações entre projetos possam ser possíveis no futuro.

II. REFERENCIAL TEÓRICO

A. Satisfiabilidade

O problema da satisfabilidade booleana (SAT) é muito importante para a Ciência da Computação, sendo o primeiro demonstrado ser NP-Completo. Ele é o problema de decidir para dada expressão booleana se é possível escolher valores para as variáveis de forma que a expressão como um todo seja verdadeira. Este problema é extensivamente pesquisado na área de métodos formais [1] e existem inclusive competições para definir qual é o melhor solucionador do mundo [2].

B. Pseudo-Booleanos

Um formato comumente utilizado para representar expressões booleanas é a Forma Normal Conjuntiva (CNF), que consiste da conjunção de cláusulas, em que cada cláusula é a disjunção de variáveis ou negação de variáveis [3]. Neste trabalho, usamos uma outra representação para expressões, chamados Pseudo-Booleanos, funções estudadas desde os anos 1960 na área de pesquisa operacional, em programação inteira. Este formato consiste de um somatório do produto de um coeficiente por um literal, que é maior ou igual a uma constante natural. Este formato é exponencialmente mais compacto que o CNF, o que motiva seu uso [4].

C. Raciocínio Pseudo-Booleano

A lógica formal de Planos de Corte introduz 2 axiomas e 4 regras de inferência, nomeadamente **Adição**, **Multiplicação**, **Divisão** e **Saturação**, que permitem derivar novas inequações a partir de outras [5], o que será detalhado na próxima seção. Com essa regras, podemos manipular as equações de forma algébrica e construir uma árvore de derivação que demonstra novos resultados, unindo somatórios diferentes pela adição, e variando seus coeficientes pela multiplicação, divisão ou saturação.

D. Lean Theorem Prover

Lean é uma linguagem de programação e provador de teoremas criado por Leonardo de Moura em 2013 [6], com influência de ML, Coq e Haskell. A sua versão mais atual de 2021, Lean 4 [7], se tornou proeminente entre matemáticos, por permitir a **formalização** e **verificação** de teoremas, auxiliando o trabalho teórico. Em 2021, uma equipe de pesquisadores usou o Lean para verificar a correção de uma prova de Peter Scholze na área de matemática condensada [8], o que atraiu atenção por formalizar um resultado na vanguarda da pesquisa matemática. Em 2023, Terence Tao usou o Lean para formalizar uma prova da conjectura Polinomial de Freiman-Ruzsa [9], um resultado publicado por Tao e colaboradores no mesmo ano.

III. CONTRIBUIÇÕES

A contribuição deste trabalho está na criação de código em Lean 4 para lidar com a lógica de cutting planes. Com essa implementação conseguimos demonstrar formalmente que as regras são corretas, ao verificá-las pelo sistema de tipos de Lean. Primeiramente, as inequações foram formalmente definidas utilizando teoremas e definições da biblioteca matemática *mathlib4* [10]. Assim, as quatro regras foram definidas e formalizadas, e, em seguida, um exemplo de raciocínio apresentado por Jakob Nordström foi formalizado nessa biblioteca.

A. Definição de Inequações Pseudo-Booleanas

Uma expressão booleana na forma CNF consiste de:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n \tag{1}$$

em que,

$$C_i = a_1^i \vee a_2^i \vee \cdots \vee a_k^i \tag{2}$$

e para cada j

$$a_j^i \in \{x_1, x_2, \ldots, x_m, \neg x_1, \neg x_2, \ldots, \neg x_m\} \tag{3}$$

Um exemplo desse formato é a formula:

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_3) \tag{4}$$

Em que x_1, x_2 e x_3 são nossas variáveis booleanas e a atribuição $x_1 = T$, $x_2 = T$, $x_3 = T$ satisfaz essa fórmula.

O formato pseudo-booleano consiste de:

$$\sum_i a_i l_i \geq A \tag{5}$$

em que,

$$\begin{gathered} A, a_i \in \mathbb{N} \\ l_i \in \{x_i, \overline{x}_i\}, \qquad x_i + \overline{x}_i = 1 \end{gathered} \tag{6}$$

A mesma fórmula acima nesse formato é expressa por:

$$x_1 + x_2 + \overline{x}_3 \geq 2 \tag{7}$$

Lean é uma linguagem de tipos dependentes, ou seja, é possível definir tipos que tem relações com valores concretos. Um caso útil é o tipo `Fin n`, que contém os números naturais até `n`. Para modelar os Pseudo-Booleanos utilizamos o tipo `Fin 2`, que contém os valores 0 e 1.

Para os coeficientes `Coeff`, utilisamos a estrutura `FinVec`, que permite definir uma lista com exatamente `n` elementos, expressa por uma função que toma um `Fin n` e retorna o elemento, em que cada um será um par de dois números naturais, o tipo $\mathbb{N}$ em Lean:

```
abbrev Coeff (n : ℕ) := Fin n → (ℕ × ℕ)
```

Esse par consiste do coeficiente de x_i no primeiro elemento e o coeficiente de $\overline{x}_i$ no segundo elemento.

Com essa definição, definimos o *PBSum*, que, com os coeficientes cs e os valores 0-1 xs, utiliza o *BigOperator* de somatório ($\sum$) para somar os elementos da lista iterando no índice i:

```
def PBSum (cs : Coeff n) (xs : Fin n → Fin 2) :=
  ∑ i, let (p,n) := cs i;
    if xs i = 1 then p else n
```

E com esse somatório podemos agora criar o *PBIneq* que é a verificação que o *PBSum* é maior ou igual à constante $const$:

```
def PBIneq (cs : Coeff n) (xs : Fin n → Fin 2)
    (const : ℕ) :=
  PBSum cs xs ≥ const
```

Para criar uma expressão desse tipo basta fornecer as listas de coeficientes, pseudo-booleanos e a constante, e em seguida provar que a propriedade vale:

```
example : PBIneq ![(1,0),(2,0)] ![0,1] 2 := by
  -- Change goal to 1 * 0 + 2 * 1 ≥ 2
  reduce
  -- Prove 1 * 0 + 2 * 1 ≥ 2
  exact Nat.le_refl 2
  done
```

O exemplo acima prova a expressão $x_1 + 2x_2 \geq 2$, quando $x_1 = 0$ e $x_2 = 1$.

B. Prova da Regra Multiplicação

A primeira regra a ser formalizada é a multiplicação, que diz que dada uma inequação pseudo-booleana, podemos obter outra inequação válida ao multiplicar os coeficientes por um escalar natural $c \in \mathbb{N}^+$, ao mesmo tempo que multiplicamos a constante pelo mesmo valor.

$$\frac{\sum_i a_i l_i \geq A}{\sum_i c a_i l_i \geq cA} \tag{8}$$

O teorema *Multiplication* implementa esse comportamento em Lean:

```
theorem Multiplication
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  (c : ℕ)
  : PBIneq (c • as) xs (c • A)
```

Podemos usar o fato `nsmul_le_nsmul_right` que, se $a \leq b$, então $c \cdot a \leq c \cdot b$, aplicamos isso em nossa premissa para obter:

$$c \cdot \sum_i a_i l_i \geq c \cdot A \tag{9}$$

Como a muliplicação por escalares já está definida na *mathlib4* para os `FinVecs` pelo teorema `Finset.sum_nsmul` da biblioteca, que diz:

$$\sum_{x \in s} n \cdot f(x) = n \cdot \sum_{x \in s} f(x) \tag{10}$$

Podemos aplicá-la da direita para esquerda para obter:

$$\sum_i c \cdot a_i l_i \geq c \cdot A \tag{11}$$

A prova completa se encontra no Apêndice B.

C. Prova da Regra Saturação

A regra da Saturação permite substituir os coeficientes de uma inequação pelo *mínimo* desse número com a constante A:

$$\frac{\sum_i a_i l_i \geq A}{\sum_i \min(a_i, A) \cdot l_i \geq A} \tag{12}$$

O teorema *Saturation* implementa esse comportamento ao aplicar *map* na lista de coeficientes com a função `mapBoth (min A)`, que transforma ambos elementos do par no mínimo entre eles e a constante A:

```
theorem Saturation
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  : PBIneq (map (mapBoth (min A)) as) xs A
```

Este teorema (Apêndice C) envolveu mais passos, pois não havia o mesmo suporte nativo da *mathlib4*. O lema que provamos, chamado `le_sum_min_of_le_sum` é o caso mais simples, onde trabalhamos com uma lista de naturais e desejamos mostrar que a relação menor-ou-igual se mantém ao aplicar o mínimo de A:

```
lemma le_sum_min_of_le_sum {n A : ℕ}
  {as : Fin n → ℕ}
  (h : A ≤ ∑i, as i)
  : A ≤ ∑i, min A (as i)
```

Em alto nível, podemos provar isso por casos:

1) Todos os elementos de as são menores-ou-iguais a A. Nesse caso $\min(A, as_i) = as_i$, para todo i, logo temos a mesma lista. Então a afirmação vale pela hipótese.
2) Caso contrário, existe ao menos um índice k da lista as, em que $as_k > A$. Podemos dividir o somatório em $\sum_{i \neq k} \min(A, as_i) + \min(A, as_k)$, separando esse índice em específico. Como $as_k > A$, substituímos $\min(A, as_k)$ por A. Como queremos mostrar que $A \leq \sum_{i \neq k} \min(A, as_i) + A$, terminamos a prova com o teorema `Nat.le_add_left A`, que diz $A \leq B + A$, para qualquer $B \in \mathbb{N}$.

D. Prova da Regra Divisão

A regra Divisão nos permite fazer o caminho inverso da Multiplicação, dividindo por um escalar $c \in \mathbb{N}^+$, com a diferença que divisões não exatas serão arrendondadas para cima:

$$\frac{\sum_i a_i l_i \geq A}{\sum_i \lceil \frac{a_i}{c} \rceil l_i \geq \lceil \frac{A}{c} \rceil} \tag{13}$$

O teorema *Division* implementa esse comportamento ao aplicar *map* na lista de coeficientes com a função `mapBoth (ceildiv c)`, que divide ambos elementos do par por c, arredondando para cima:

```
theorem Division
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  (c : ℕ)
  : PBIneq (map (mapBoth (ceildiv c)) as)
    xs (ceildiv c A)
```

Essa prova foi mais complexa, pois precisamos mostrar o comportamento para a lista toda, não sendo suficiente apenas mostrar para algum elemento em particular, como no caso da Saturação. Dispondo de ajuda pelo *Zulip do Lean*, chegamos em dois lemas que permitem provar a propriedade.

Mostramos primeiramente que, para dois elementos em isolados, a divisão com teto da soma é menor-ou-igual à soma das divisões com teto:

```
theorem Nat.add_ceildiv_le_add_ceildiv
    (a b c : ℕ)
  : (a + b) ⌈/⌉ c ≤ (a ⌈/⌉ c) + (b ⌈/⌉ c)
```

Com esse teorema, agora podemos criar uma prova por indução que vai valer para listas de qualquer tamanho:

```
theorem Finset.ceildiv_le_ceildiv {α : Type*}
    (as : α → ℕ) (s : Finset α) (c : ℕ)
  : (∑i in s, as i) ⌈/⌉ c
  ≤ ∑i in s,(as i ⌈/⌉ c)
```

O último detalhe que tivemos que provar é que podemos distribuir o *ceildiv* sobre a expressão `if-then-else`:

```
lemma ceildiv_ite (P : Prop) [Decidable P]
    (a b c : ℕ)
  : (if P then b else c) ⌈/⌉ a
  = if P then (b ⌈/⌉ a) else (c ⌈/⌉ a)
```

Com isso concluímos a prova.

E. Prova da Regra Adição

Adição foi a última prova demonstrada, e se tornou a mais difícil, pois, além de somar duas inequações, dois literais de polaridades opostas se aniquilam, o que chamamos aqui de *Redução*:

$$\frac{\sum_i a_i l_i \geq A \qquad \sum_i b_i l_i \geq B}{\sum_i (a_i + b_i) l_i \geq (A + B)} \tag{14}$$

Primeiro implementamos uma regra *Addition'*, que realiza a adição diretamente sem a redução:

```
theorem Addition'
  (xs : Fin n → Fin 2)
  (as : Coeff n) (A : ℕ) (ha : PBIneq as xs A)
  (bs : Coeff n) (B : ℕ) (hb : PBIneq bs xs B)
  : PBIneq (as + bs) xs (A + B)
```

Os teoremas da *mathlib4* deram bom suporte à prova, o que `Finset.sum_add_distrib` resolveu em um passo.

Um lema usado para seguir adiante foi `ite_eq_bmul`, que nos permite transitar da notação `if-then-else` para a multiplicação dos termos pseudo-booleanos:

```
lemma ite_eq_bmul (x y : ℕ) (b : Fin 2)
  : (if b = 1 then x else y)
  = (x * b + y * (1 - b))
```

Em seguida, implementamos a regra *Reduction*, que toma uma inequação e aniquila coeficientes em que ambos elementos do par são maiores que 0. Quando isso acontece, essa diferença "*slack*" deve ser subtraída da constante:

```
def ReductionProp (xs : Fin n → Fin 2)
    (ks : Coeff n) (K : ℕ) : Prop :=
  let pos := λ i => ks i |>.1
  let neg := λ i => ks i |>.2
  let slack := (∑i, min (pos i) (neg i))
  let rs := λ i => (pos i - neg i,neg i - pos i)
  PBIneq rs xs (K - slack)

theorem Reduction
  (xs : Fin n → Fin 2)
  (ks : Coeff n) (K : ℕ) (ha : PBIneq ks xs K)
  : ReductionProp xs ks K
```

Com esses dois teoremas, definimos *Addition* compondo-os:

```
theorem Addition
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  {bs : Coeff n} {B : ℕ} (hb : PBIneq bs xs B)
  : ReductionProp xs (as + bs) (A + B) := by
  have hk := Addition' xs as A ha bs B hb
  exact Reduction xs (as + bs) (A + B) hk
  done
```

F. Implementação do Exemplo

Com todas as regras necessárias, prosseguimos para um exemplo retirado da apresentação (Apêncide F). Aqui podemos ver como usar as regras, com auxílio da tática `apply`.

$$\text{Divide by 3}\ \dfrac{\text{Add}\ \dfrac{\text{Add}\ \dfrac{\text{Multiply by 2}\ \dfrac{w+2x+y\geq 2}{2w+4x+2y\geq 4} \qquad w+2x+4y+2z\geq 5}{3w+6x+6y+2z\geq 9} \qquad \dfrac{\overline{z}\geq 0}{2\overline{z}\geq 0}\ \text{Multiply by 2}}{3w+6x+6y \qquad \geq 7}}{w+2x+2y\geq 3}$$

O leitor fica convidado a visitar o repositório público em `github.com/bernborgess/lean-cutting-planes` e tentar por conta própria.

CONCLUSÕES

Nós demonstramos a lógica de planos de corte como método correto para trabalhar com pseudo-booleanos. Com nossa biblioteca `lean-cutting-planes` agora podemos utilizar o sistema de tipos de Lean para validar com confiança passos dessa lógica. Com a documentação convidamos novos pesquisadores e matemáticos a usar os resultados em verificadores.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] D. Le Berre, “Sat Live! keep up to date with research on the satisfiability problem”, Acesso em http://www.satlive.org/.

[2] M. Heule, M. Jävisalo, M. Suda, “The International SAT Competition Web Page”, Acesso em https://satcompetition.github.io/.

[3] “Conjunctive normal form”, Encyclopedia of Mathematics, EMS Press, 2001.

[4] J. Nordström, “Pseudo-Boolean Solving and Optimization”, Fevereiro de 2021.

[5] J. Nordström, “A Unified Proof System for Discrete Combinatorial Problems”, Novembro de 2023.

[6] L. de Moura, S. Kong, J. Avigad, F. van Doorn, J. von Raumer “The Lean Theorem Prover”, 25th International Conference on Automated Deduction (CADE-25), Berlin, Germany, 2015. Acesso em https://lean-lang.org/papers/system.pdf.

[7] L. de Moura, S. Ullrich “The Lean 4 Theorem Prover and Programming Language”, 28th International Conference on Automated Deduction (CADE-28), Pittsburgh, USA, 2021. Acesso em https://lean-lang.org/papers/lean4.pdf.

[8] J. Commelin, P. Scholze “Liquid Tensor Experiment”. Acesso em https://math.commelin.net/files/LTE.pdf

[9] T. Tao, “The Polynomial Freiman-Ruzsa Conjecture”, Novembro de 2023. Acesso em https://teorth.github.io/pfr/.

[10] The mathlib Community. 2020. “The lean mathematical library”. In CPP 2020. 367-381. https://doi.org/10.1145/3372885.3373824

Apêndice

A. Definição de Inequações Pseudo-Booleanas

```
import Mathlib.Data.Fin.Tuple.Reflection

namespace PseudoBoolean

open FinVec BigOperators

abbrev Coeff (n : ℕ) := Fin n → (ℕ × ℕ)

def PBSum (cs : Coeff n) (xs : Fin n → Fin 2) :=
  ∑ i, let (p,n) := cs i;
    if xs i = 1 then p else n

def PBIneq (cs : Coeff n) (xs : Fin n → Fin 2) (const : ℕ) :=
  PBSum cs xs ≥ const

example : PBIneq ![(1,0)] ![1] 1 := by
  reduce                  -- Expand the goal to 1 * 1 ≥ 1
  exact Nat.le_refl 1     -- Prove that 1 * 1 ≥ 1
  done

example : PBIneq ![(1,0),(2,0)] ![0,1] 2 := by
  reduce                  -- Change goal to 1 * 0 + 2 * 1 ≥ 2
  exact Nat.le_refl 2     -- Prove 1 * 0 + 2 * 1 ≥ 2
  done

example : PBIneq ![(3,0),(4,0)] ![0,1] 2 := by
  reduce
  simp
  done

def mapBoth (f : α → β) (t : α × α) : β × β := Prod.map f f t

end PseudoBoolean
```

B. Prova da Regra Multiplicação

```
import <<LeanCuttingPlanes>>.Basic

namespace PseudoBoolean

open BigOperators FinVec

theorem Multiplication
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  (c : ℕ)
  : PBIneq (c • as) xs (c • A) := by
  unfold PBIneq PBSum at *
  simp only [Fin.isValue, ge_iff_le, nsmul_eq_smul, smul_eq_mul] at *
  apply nsmul_le_nsmul_right at ha
  specialize ha c
  rw [←Finset.sum_nsmul] at ha
  simp only [smul_eq_mul, Fin.isValue, mul_ite] at ha
  exact ha
  done

example
  (ha : PBIneq ![(1,0)] xs 3)
  : PBIneq ![(2,0)] xs 6 := by
  apply Multiplication ha 2
  done

end PseudoBoolean
```

C. Prova da Regra Saturação

```
import <<LeanCuttingPlanes>>.Basic

namespace PseudoBoolean
open FinVec Matrix BigOperators Finset

-- @collares
lemma split_summation (n : ℕ) (as : Fin n → ℕ) (k : Fin n) :
    (∑i with i≠k, as i) + as k = (∑i, as i) := by
  have : (∑i with i=k, as i) = as k := by rw [Finset.sum_eq_single_of_mem] <;> simp
  rw [← this, ← Finset.sum_filter_add_sum_filter_not Finset.univ (· ≠ k)]
  simp only [ne_eq, Decidable.not_not]

lemma le_sum_min_of_le_sum {n A : ℕ} {as : Fin n → ℕ}
  (h : A ≤ ∑i, as i)
  : A ≤ ∑i, min A (as i) := by
  by_cases ha : ∀i, as i ≤ A
  . -- Assume all elements of as are ≤ A
    simp_rw [@Nat.min_eq_right A (as _) (ha _)]
    -- rewrite min A (as i) to (as i)
    exact h

  . -- Otherwise, ∃k, (as k) > A
    simp only [not_forall, not_le] at ha
    obtain ⟨k,hk⟩ := ha

    rw [←split_summation]
    -- Split goal from  ⊢ A ≤  ∑i, min A (as i)
    -- to               ⊢ A ≤ (∑i with i ≠ k, min A (as i)) + min A (as k)

    -- min A (as k) = A
    rw [min_eq_left_of_lt hk]

    -- ⊢ A ≤ (∑i,min A (as i) - A) + A
    exact Nat.le_add_left A _

theorem Saturation
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  : PBIneq (map (mapBoth (min A)) as) xs A := by
  unfold PBIneq PBSum FinVec.map mapBoth at *
  simp only [Fin.isValue, ge_iff_le, Prod_map, seq_eq] at *
  have h := le_sum_min_of_le_sum ha
  simp_rw [apply_ite (min A) ((xs _ = 1)) ((as _).1) ((as _).2)] at h
  exact h
  done

example
  (ha : PBIneq ![(3,0),(4,0)] xs 3)
  : PBIneq ![(3,0),(3,0)] xs 3 := by
  apply Saturation ha
  done

end PseudoBoolean
```

D. Prova da Regra Divisão

```
import <<LeanCuttingPlanes>>.Basic
import Mathlib.Algebra.Order.Floor.Div

namespace PseudoBoolean
open Finset FinVec BigOperators

def ceildiv (c : ℕ) (a : ℕ) := a ⌈/⌉ c

lemma ceildiv_le_ceildiv_right {a b : ℕ} (c : ℕ) (hab : a ≤ b)
  : a ⌈/⌉ c ≤ b ⌈/⌉ c := by
  repeat rw [Nat.ceilDiv_eq_add_pred_div]
  apply Nat.div_le_div_right
  apply Nat.sub_le_sub_right
  apply Nat.add_le_add_right
  exact hab
  done

-- @kbuzzard
theorem Nat.add_ceildiv_le_add_ceildiv (a b c : ℕ)
  : (a + b) ⌈/⌉ c ≤ (a ⌈/⌉ c) + (b ⌈/⌉ c) := by
  -- deal with c=0 separately
  obtain (rfl | hc) := Nat.eq_zero_or_pos c
  · simp
  -- 0 < c case
  -- First use the "Galois connection"
  rw [ceilDiv_le_iff_le_smul hc, smul_eq_mul]
  rw [mul_add]
  -- now use a standard fact
  gcongr <;> exact le_smul_ceilDiv hc
  done

-- @Ruben-VandeVelde
theorem Finset.ceildiv_le_ceildiv {α : Type*} (as : α → ℕ) (s : Finset α) (c : ℕ)
  : (∑i in s, as i) ⌈/⌉ c ≤ ∑i in s,(as i ⌈/⌉ c) := by
  classical
  induction s using Finset.cons_induction_on with
  | h₁ => simp
  | @h₂ a s ha ih =>
    rw [sum_cons, sum_cons]
    have h := Nat.add_ceildiv_le_add_ceildiv (as a) (∑x ∈ s, as x) c
    exact le_add_of_le_add_left h ih
    done

lemma ceildiv_ite (P : Prop) [Decidable P] (a b c : ℕ)
  : (if P then b else c) ⌈/⌉ a = if P then (b ⌈/⌉ a) else (c ⌈/⌉ a) := by
  split_ifs <;> rfl
  done
```

```
theorem Division
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  (c : ℕ)
  : PBIneq (map (mapBoth (ceildiv c)) as) xs (ceildiv c A) := by
  unfold PBIneq PBSum ceildiv mapBoth at *
  simp only [Fin.isValue, ge_iff_le, gt_iff_lt,
             Prod_map, map_eq, Function.comp_apply] at *
  apply ceildiv_le_ceildiv_right c at ha
  apply le_trans ha
  simp only [←ceildiv_ite]
  apply Finset.ceildiv_le_ceildiv
  done

example
  (ha : PBIneq ![(3,0),(4,0)] xs 3)
  : PBIneq ![(2,0),(2,0)] xs 2 := by
  apply Division ha 2
  done

end PseudoBoolean
```

E. Prova da Regra Adição

```
import <<LeanCuttingPlanes>>.Basic

namespace PseudoBoolean

open BigOperators FinVec Matrix

theorem Addition'
  (xs : Fin n → Fin 2)
  (as : Coeff n) (A : ℕ) (ha : PBIneq as xs A)
  (bs : Coeff n) (B : ℕ) (hb : PBIneq bs xs B)
  : PBIneq (as + bs) xs (A + B) := by
  unfold PBIneq PBSum at *
  simp only [Fin.isValue, ge_iff_le] at *
  simp_rw [←ite_add_ite]
  rw [Finset.sum_add_distrib]
  exact Nat.add_le_add ha hb
  done

def ReductionProp
  (xs : Fin n → Fin 2) (ks : Coeff n) (K : ℕ)
  : Prop :=
  let pos := λ i => ks i |>.1
  let neg := λ i => ks i |>.2
  let slack := (∑i, min (pos i) (neg i))
  let rs := λ i => (pos i - neg i,neg i - pos i)
  PBIneq rs xs (K - slack)

lemma ite_eq_bmul (x y : ℕ) (b : Fin 2)
  : (if b = 1 then x else y) = (x * b + y * (1 - b)) := by
  by_cases h : b = 0
  . rw [h]
    rw [if_neg]
    . simp only [Fin.isValue, Fin.val_zero, mul_zero, tsub_zero, mul_one, zero_add]
    trivial
  . -- b = 1
    apply Fin.eq_one_of_neq_zero b at h
    rw [h]
    simp only [Fin.isValue, ↓reduceIte, Fin.val_one, mul_one, ge_iff_le, le_refl,
      tsub_eq_zero_of_le, mul_zero, add_zero]

lemma reduce_terms (p n : ℕ) (x : Fin 2)
  : p * x + n * (1 - x) = (p - n) * x + (n - p) * (1 - x) + min p n  := by
  by_cases h : x = 0
  . rw [h]
    simp only [Fin.isValue, Fin.val_zero, mul_zero, tsub_zero, mul_one, zero_add]
    rw [Nat.min_comm]
    exact Nat.sub_add_min_cancel n p |>.symm

  . -- x = 1
    apply Fin.eq_one_of_neq_zero x at h
    rw [h]
    simp only [Fin.isValue, Fin.val_one, mul_one, ge_iff_le,
                le_refl, tsub_eq_zero_of_le, mul_zero, add_zero]
    exact Nat.sub_add_min_cancel p n |>.symm
```

```lean
theorem Reduction
  (xs : Fin n → Fin 2)
  (ks : Coeff n) (K : ℕ) (ha : PBIneq ks xs K)
  : ReductionProp xs ks K := by
  unfold ReductionProp PBIneq PBSum at *
  simp only [Fin.isValue, ge_iff_le, tsub_le_iff_right] at *
  simp_rw [ite_eq_bmul] at *
  rw [←Finset.sum_add_distrib]
  simp_rw [←reduce_terms]
  exact ha
  done

def AdditionProp
  (xs : Fin n → Fin 2)
  (as : Coeff n) (A : ℕ)
  (bs : Coeff n) (B : ℕ)
  : Prop :=
  ReductionProp xs (as + bs) (A + B)

theorem Addition
  {xs : Fin n → Fin 2}
  {as : Coeff n} {A : ℕ} (ha : PBIneq as xs A)
  {bs : Coeff n} {B : ℕ} (hb : PBIneq bs xs B)
  : AdditionProp xs as A bs B := by
  have hk := Addition' xs as A ha bs B hb
  exact Reduction xs (as + bs) (A + B) hk
  done

example
  (ha : PBIneq ![(1,0),(0,0)] xs 1)
  (hb : PBIneq ![(1,0),(1,0)] xs 2)
  : PBIneq ![(2,0),(1,0)] xs 3 := by
  apply Addition ha hb
  done

-- Reduction happens automatically
example
  (ha : PBIneq ![(3,0),(0,0),(1,0)] xs 1)
  (hb : PBIneq ![(0,0),(2,0),(0,1)] xs 2)
  : PBIneq ![(3,0),(2,0),(0,0)] xs 2 := by
  apply Addition ha hb
  done

end PseudoBoolean
```

F. Implementação do Exemplo

```
import <<LeanCuttingPlanes>>

open PseudoBoolean

example
  (xs : Fin 4 → Fin 2)
  (c1 : PBIneq ![(1,0),(2,0),(1,0),(0,0)] xs 2)
  (c2 : PBIneq ![(1,0),(2,0),(4,0),(2,0)] xs 5)
  (c3 : PBIneq ![(0,0),(0,0),(0,0),(0,1)] xs 0)
  : PBIneq ![(1,0),(2,0),(2,0),(0,0)] xs 3
  := by
  let t1 : PBIneq ![(2,0),(4,0),(2,0),(0,0)] xs 4  := by apply Multiplication c1 2
  let t2 : PBIneq ![(3,0),(6,0),(6,0),(2,0)] xs 9  := by apply Addition t1 c2
  let t3 : PBIneq ![(0,0),(0,0),(0,0),(0,2)] xs 0  := by apply Multiplication c3 2
  let t4 : PBIneq ![(3,0),(6,0),(6,0),(0,0)] xs 7  := by apply Addition t2 t3
  exact Division t4 3
  done
```