Formalizando Bit-blasting com Pseudo-booleanos e verificando no projeto Carcara

Projeto Orientado em Computação II - 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 aplicação da teoria de Pseudo-booleanos para realizar o procedimento de bit-blasting, utilizado para se raciocinar sobre vetores de bits e operações que os envolvem.**

***Index Terms*—formal methods, pseudo boolean reasoning, bit blasting, proof checking**

I. Introdução

HUH

II. Referencial Teórico

A. Aritmética de Bit-Vectors

A aritmética de Bit-Vectors é amplamente utilizada em áreas como verificação formal, circuitos digitais e design de hardware, pois permite modelar o comportamento de sistemas que operam diretamente em representações binárias. Um vetor de bits é uma sequência de bits (0s e 1s) de comprimento fixo, onde cada bit representa um dígito binário. Operações como soma, subtração, multiplicação e deslocamento são realizadas diretamente sobre esses vetores, respeitando restrições como largura fixa e ocorrência de overflow.

B. Bit Blasting

Uma forma que solucionadores SMT utilizam para raciocinar sobre tais Bit-Vectors é o bit-blasting, uma representação que utiliza-se de uma variável proposicional para cada bit, o que então pode ser repassado para outras regras de raciocínio.

O bit-blasting converte operações aritméticas e lógicas em vetores de bits para fórmulas proposicionais que podem ser resolvidas por solucionadores SAT (Satisfiability). Nesse processo, cada bit de um vetor é tratado como uma variável proposicional independente, e as operações sobre os vetores são traduzidas em expressões lógicas que representam o comportamento bit a bit. Por exemplo, uma operação de soma em vetores de bits pode ser reduzida a uma série de expressões que modelam o comportamento de somadores completos e meio-somadores, incluindo a propagação de carry (ou transporte) entre os bits. Assim, uma operação que ocorre em um nível mais abstrato, como $A + B$, é traduzida em uma série de restrições lógicas que descrevem a interação entre os bits individuais de A e B.

C. 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 [**?**]. 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 [**?**].

D. Bit Blasting Pseudo Booleano

Aproveitando sua estrutura, os Pseudo-Booleanos oferecem uma abordagem alternativa para o processo de Bit Blasting, permitindo uma representação mais direta da semântica associada a vetores de bits. Essa semântica está relacionada ao significado numérico ou lógico dos vetores, que pode ser expressa utilizando coeficientes que ponderam a contribuição de cada bit em cálculos aritméticos ou restrições lógicas

E. Formato de Prova Alethe

A corretude é uma preocupação central em solucionadores SMT, dado que eles são amplamente utilizados em verificação formal, onde a precisão é essencial para garantir que sistemas críticos funcionem conforme especificado. No entanto, provar a corretude desses solucionadores é um desafio devido à vasta base de código e às constantes inovações que introduzem alterações frequentes. Quando um solucionador SMT retorna um resultado 'SAT', é relativamente simples verificar de forma independente se o modelo gerado satisfaz todas as condições impostas. Contudo, no caso de um resultado 'UNSAT', a verificação da corretude requer um *certificado*, ou seja, um registro detalhado dos passos de raciocínio que levaram à conclusão de insatisfiabilidade. Nesse sentido, foi desenvolvido o formato Alethe [1], inspirado na linguagem SMT-LIB, que representa de forma flexível e padronizada as provas geradas por solucionadores SMT, que podem ser verificados de forma independente.

F. Verificador de prova Carcara

Carcara [2] é um verificador de prova desenvolvido em Rust pelo laboratório SMITE da UFMG, sob a liderança do professor Haniel Barbosa, projetado para verificar certificados de prova no formato Alethe. Este projeto permite a identificação de erros lógicos nos sistemas que geram esses certificados, aumentando a confiança nos resultados apresentados.

III. Contribuições

A contribuição deste trabalho está na definição dos passos tomados para realizar bit-blasting pseudo booleano para cada operação suportada entre bit-vectors, e então a implementação do *checker* no projeto Carcara que permite verificar a correta aplicação de tais regras no formato Alethe.

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`, utilizamos 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 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 multiplicação 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 (Apêndice F). Aqui podemos ver como usar as regras, com auxílio da tática `apply`.

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] ALETHE PAPER HERE, "Am I watching enough cult movies?", Novembro de 2023.

[2] CARCARA PAPER HERE