Lógica Computacional 117366
Descrição do Projeto
Verificação de MDC via Aritmética Modular

17 de Outubro de 2016

Prof. Mauricio Ayala-Rincón & Prof. Flávio L. C. de Moura

O monitor Matheus Schmitz (matheus81653100@gmail.com) dará suporte aos alunos no desenvolvimento do projeto. Laboratórios do LINF têm instalado o *software* necessário (PVS 6.0 com as bibliotecas PVS da NASA).

1 Introdução

A aritmética modular está implementada no núcleo de diversos sistemas computacionais. Em particular, operadores como mdc (gcd em inglês) podem ser implementados eficientemente utilizando técnicas desta aritmética, via mecanismos introduzidos originalmente por Euclides de Alexandria há mais de dois milênios.

O objetivo do presente projeto é introduzir os mecanismos básicos de manuseio de tecnologias de verificação e formalização que utilizam técnicas dedutivas lógicas, como as estudadas na disciplina, para garantir que objetos computacionais são logicamente corretos.

2 Descrição do Projeto

Este projeto aborda questões apresentadas nos arquivos de especificação e prova (gcd.pvs e gcd.prf) de algoritmos para computar gcd "corretos", onde por correto entendem-se algoritmos que para uma entrada consistente de dois números, computam um natural que os divide e que é máximo, conforme a definição matemática de gcd:

Definição 1 (gcd) *O gcd de m, n, inteiros não simultaneamente nulos é um natural k, tal que $k|m$, $k|n$ (k divide m e k divide n) e tal que para qualquer outro divisor l de m e n, $k \geq l$.*

A tecnologia de Euclides para computar gcd utiliza o seguinte operador.

Definição 2 (Módulo ou Resíduo da Divisão Inteira) *Para qualquer natural positivo b e inteiro n, seja $rem(b)(m)$ definido como o resíduo da divisão inteira de m por b:*

$$0 \leq rem(b)(m) < b \text{ tal que existe inteiro } j \text{ com } m = j \cdot b + rem(b)(m)$$

Assim, por exemplo, $rem(6)(20) = 2$ e $rem(6)(-20) = 4$. A técnica de Euclides (eficiente) está justificada por o seguinte resultado.

Teorema 1 (Euclides ca. 300 AC) *Para qualquer natural positivo b e inteiro n,*

$$gcd(b, m) = gcd(rem(b)(m), m)$$

Os arquivos de especificação e prova estão disponíveis na página da disciplina, especificados na linguagem do assistente de demonstração PVS (pvs.csl.sri.com) executável em plataformas Unix/Linux. Os alunos deverão formalizar propriedades relacionadas com a *correção* das respostas computadas por um algoritmo para computar o gcd, que é baseado no resultado acima mencionado.

2.1 Algoritmos para cálculo de gcd

No arquivo do prelúdio de PVS, a saber `prelude.pvs` (consulte via comando `M-x view-prelude-file`), estão disponíveis diversos resultados da aritmética modular que precisarão ser aplicados, como por exemplo:

```
rem_mod2: LEMMA 0 <= x AND x < b IMPLIES rem(b)(x) = x
rem_zero: LEMMA rem(b)(0) = 0
rem_self: LEMMA rem(b)(b) = 0
rem_sum: LEMMA
     rem(b)(x) = rem(b)(y) AND rem(b)(z) = rem(b)(t)
        IMPLIES rem(b)(x + z) = rem(b)(y + t)
```

Esses e outros resultados podem ser aplicados utilizando os comandos de demonstração `lemma` e `rewrite`, o primeiro com a instanciação adequada.

Três diferentes funções para o computo de gcd estão especificadas no arquivo `gcd.pvs`: `gcd`, `gcd_eff` e `gcd_sw`:

```
gcd(n, m) : RECURSIVE nat =
                  IF abs(n) = abs(m) THEN abs(n)
                  ELSE IF (n = 0 OR m = 0) THEN abs(n+m)
                       ELSE IF (abs(n) > abs(m)) THEN
                                          gcd(abs(n)-abs(m), abs(m))
                            ELSE gcd(abs(m)-abs(n),abs(n))
                            ENDIF
                       ENDIF
                  ENDIF
MEASURE abs(n)+abs(m)

gcd_eff(n,m) : RECURSIVE nat =
                  IF abs(n) = abs(m) THEN abs(n)
                  ELSE IF (n = 0 OR m = 0) THEN abs(n+m)
                       ELSE IF (abs(n) > abs(m)) THEN
                               gcd_eff(rem(abs(m))(abs(n)), abs(m))
                            ELSE gcd_eff(rem(abs(n))(abs(m)),abs(n))
                            ENDIF
                       ENDIF
                  ENDIF
MEASURE abs(n)+abs(m)

gcd_sw(m : nat, n : posnat) : RECURSIVE nat =
                  IF m = 0 THEN n
                  ELSE IF m < n THEN gcd_sw(n, m)
                       ELSE gcd_sw(m - n, n)
                       ENDIF
                  ENDIF
MEASURE lex2(n,m)
```

A correção destas funções pode ser provada de diferentes formas. Por exemplo, a correção de `gcd` é provada de forma *direta*, ou seja, constrói-se uma prova para o seguinte teorema:

```
gcd_is_correct : COROLLARY
            (m /= 0 OR n /=0) => divides(gcd(m,n),m) AND
                                 divides(gcd(m,n),n) &
                                 FORALL (k) : (divides(k,m) &
                                               divides(k,n) =>
                                               k <= gcd(m,n))
```

A ideia de prova direta aqui está relacionada ao fato de que apenas foram utilizadas propriedades de aritmética modular.

Uma outra estratégia foi utilizada para se estabelecer a correção de `gcd_eff`: inicialmente prova-se que `gcd` e `gcd_eff` são *funcionalmente equivalentes*, ou seja, que estas funções retornam valores iguais para argumentos iguais:

```
gcd_eff_same_gcd : LEMMA FORALL (m,n) : gcd_eff(m,n) = gcd(m,n)
```

Em seguida provamos um teorema similar a `gcd_is_correct`, mas a prova baseia-se na equivalência funcional, ou seja, a correção de `gcd_eff` é obtida a partir da correção de `gcd`. Neste sentido, a prova da correção de `gcd` não é direta. Analogamente, a correção de `gcd_sw` é obtida via a *equivalência funcional* desta função com `gcd`.

O projeto trata do desenvolvimento de provas diretas, isto é, sem utilizar os resultados de *equivalência funcional* entre `gcd`, `gcd_eff` e `gcd_sw`, para verificar a correção das funções `gcd_eff` e `gcd_sw`.

3 Questões

Deverão ser provados os resultados a seguir, conforme especificação dada no arquivo `gcd.pvs`.

A primeira questão do projeto consiste em demonstrar que *o resultado computado por* `gcd_eff` *é um divisor do primeiro argumento de entrada*, especificado como a conjectura abaixo. Será necessária a aplicação de indução completa no tamanho da entrada.

Questão 01

```
gcd_eff_divides_left : CONJECTURE  FORALL (n,m) : divides(gcd_eff(m,n), m)
```

A segunda questão consiste em formalizar que `gcd_eff` *cumpre o teorema de Euclides para entradas naturais*, expresso como a conjectura abaixo. Serão necessários resultados disponíveis para o gcd na teoria `gcd.pvs` e no prelúdio de PVS.

Questão 02

```
gcd_eff_is_gcd_eff_mod1 : CONJECTURE FORALL (b : posnat, m : nat) :
                  gcd_eff(m,b) = gcd_eff(rem(b)(m),b)
```

A terceira questão também requer indução, e consiste em provar a *maximalidade do divisor computado por* `gcd_eff` *para entradas naturais positivas.*

Questão 03

```
gcd_eff_of_posnats_is_maximum : LEMMA FORALL (m, n : posnat, k) :
             divides(k, m) AND divides(k,n) => k <= gcd_eff(m,n)
```

Como desafio adicional, os alunos poderão abordar a *correção* da função `gcd_sw`, também sem utilizar *equivalência funcional*.

4 Etapas do desenvolvimento do projeto

Os alunos deverão definir os grupos de trabalho limitados a **três** membros até o dia 19/10/2016. Exceto pelo dia da segunda prova, as aulas continuarão sendo realizadas no LINF.

O projeto será dividido em duas etapas como segue:

- A primeira etapa do projeto é a de Verificação das Formalizações. Os grupos deverão ter prontas as suas formalizações na linguagem do assistente de demonstração PVS e enviar via e-mail ao estagiário com cópia para o professor os arquivos de especificação e de provas desenvolvidos (`gcd.pvs` e `gcd.prf`) até às 08h da manhã do dia **21.11.2016**. Na mesma semana, ou seja, entre os dias **21.11.2016** e **24.11.2016**, durante o horário de aula, realizar-se-á a verificação do trabalho para a qual os grupos deverão, em acordo com o monitor e professor, determinar um horário (de 30 min) no qual todos membros do grupo deverão comparecer.
  Avaliação (peso 6.0):
  - Um dos membros, selecionado por sorteio, explicará os detalhes da formalização em, no máximo, 10 minutos.
  - Os quatro membros do grupo poderão complementar a explicação inicial em, no máximo, 5 minutos.
  - A formalização será testada nos 15 minutos seguintes.
- A segunda etapa do projeto consiste da apresentação dos resultados finais e conclusões do estudo do problema.
  Avaliação (peso 4.0): Cada grupo de trabalho deverá entregar um Relatório Final inédito, editado em LaTeX, limitado a 8 páginas (12 pts, A4, espaçamento simples) do projeto até o dia **28.11.2016** com o seguinte conteúdo:
  - Introdução e contextualização do problema.
  - Explicação da soluções.
  - Especificação do problema e explicação do método de solução.
  - Descrição da formalização.
  - Conclusões.
  - Referências.

Referências

[ARdM16] M. Ayala-Rincón and F.L.C. de Moura. *Applied Logic for Computer Scientists - computational deduction and formal proofs* -. Notas de aula, 2016.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms — Introduction to Design and Analysis*. Addison-Wesley, 1999.

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT press, third edition, 2009.