Estratégias da Teoria de Reescrita em PVS: considerações sobre especificação e aplicações

André Luiz Galdino*

Departamento de Matemática
Campus de Catalão
Universidade Federal de Goiás
Professor Assistente

&

Grupo de Teoria da Computação
Departamento de Matemática
Universidade de Brasília
Estudante de Doutorado

Orientador: Mauricio Ayala-Rincón

* Parcialmente Financiado pelo CNPq

08 de dezembro de 2006

Plano de Apresentação

- A Linguagem de Especificação PVS
- Uma Especificação de Sistemas Abstratos de Reescrita
- Porque Estratégias em PVS são Importantes
- Conclusão e Trabalho Futuro

A Linguagem de Especificação PVS

- O *PVS* (Prototype Verification System) é uma linguagem de especificação baseada sobre lógica de ordem superior e rica em sistema de tipos.
- A *especificação* em PVS é organizada como uma coleção de *Teorias*. Uma *Teoria* é essencialmente feita de:
  - *Declarações de Tipos*: Introduz novos nomes de tipos e/ou tipos;
    `T:TYPE, par: TYPE = {x: T | EXISTS(y: T): x = 2*y}`
  - *Declarações de Constantes e variáveis*: Introduz novas constantes e/ou variáveis;
    `x,y,z: VAR nat`
  - *Declarações de Formulas*: Introduz axiomas (AXIOM) e/ou *Teoremas* (THEOREM, LEMMA ou OBLIGATION).

A Linguagem de Especificação PVS

- O *PVS* (Prototype Verification System) é uma linguagem de especificação baseada sobre lógica de ordem superior e rica em sistema de tipos.
- A *especificação* em PVS é organizada como uma coleção de *Teorias*. Uma *Teoria* é essencialmente feita de:
  - *Declarações de Tipos*: Introduz novos nomes de tipos e/ou tipos;
    `T:TYPE, par:TYPE={x:T|EXISTS(y:T):x=2*y}`
  - *Declarações de Constantes e variáveis*: Introduz novas constantes e/ou variáveis;
    `a,b:T, z:VAR par`
  - *Declarações de Fórmulas*: Introduz *Axiomas* (`AXIOM`) e/ou *Teoremas* (`CLAIM`, `LEMMA` ou `THEOREM`)

A Linguagem de Especificação PVS

- O *PVS* (Prototype Verification System) é uma linguagem de especificação baseada sobre lógica de ordem superior e rica em sistema de tipos.
- A *especificação* em PVS é organizada como uma coleção de *Teorias*. Uma *Teoria* é essencialmente feita de:
  - *Declarações de Tipos*: Introduz novos nomes de tipos e/ou tipos;
    `T:TYPE, par:TYPE` $= \{$`x:T | EXISTS(y:T): x = 2*y`$\}$
  - *Declarações de Constantes e variáveis*: Introduz novas constantes e/ou variáveis;
    `a,b:T, z:VAR par`
  - *Declarações de Fórmulas*: Introduz *Axiomas* (`AXIOM`) e/ou *Teoremas* (`CLAIM`, `LEMMA` ou `THEOREM`)

A Linguagem de Especificação PVS

- O *PVS* (Prototype Verification System) é uma linguagem de especificação baseada sobre lógica de ordem superior e rica em sistema de tipos.
- A *especificação* em PVS é organizada como uma coleção de *Teorias*. Uma *Teoria* é essencialmente feita de:
  - *Declarações de Tipos*: Introduz novos nomes de tipos e/ou tipos;
    `T:TYPE, par:TYPE` $= \{$`x:T | EXISTS(y:T): x = 2*y`$\}$
  - *Declarações de Constantes e variáveis*: Introduz novas constantes e/ou variáveis;
    `a,b:T, z:VAR par`
  - *Declarações de Fórmulas*: Introduz *Axiomas* (`AXIOM`) e/ou *Teoremas* (`CLAIM`, `LEMMA` ou `THEOREM`)

A Linguagem de Especificação PVS

- O *PVS* (Prototype Verification System) é uma linguagem de especificação baseada sobre lógica de ordem superior e rica em sistema de tipos.
- A *especificação* em PVS é organizada como uma coleção de *Teorias*. Uma *Teoria* é essencialmente feita de:
  - *Declarações de Tipos*: Introduz novos nomes de tipos e/ou tipos;
    `T:TYPE, par:TYPE` $= \{$`x:T | EXISTS(y:T): x = 2*y`$\}$
  - *Declarações de Constantes e variáveis*: Introduz novas constantes e/ou variáveis;
    `a,b:T, z:VAR par`
  - *Declarações de Fórmulas*: Introduz *Axiomas* (`AXIOM`) e/ou *Teoremas* (`CLAIM, LEMMA` ou `THEOREM`)

O Provador PVS

- O *Provador* PVS é usado interativamente para construir a prova de um teorema e ele usa a apresentação de prova no estilo cálculo sequente para mostrar o atual objetivo de prova da prova em progresso;

```
 [-1] A
 [-2] D
|-------
 [1] B
 [2] C
 [3] E
```

- O provador mantém uma *árvore de prova* para o teorema que está sendo provado, e cada nó da árvore de prova é um objetivo de prova que resulta da aplicação de um *passo de prova* ao nó raiz.

O Provador PVS

- O *Provador* PVS é usado interativamente para construir a prova de um teorema e ele usa a apresentação de prova no estilo cálculo sequente para mostrar o atual objetivo de prova da prova em progresso;

```
  [-1] A
  [-2] D
|-------
  [1] B
  [2] C
  [3] E
```

- O provador mantém uma *árvore de prova* para o teorema que está sendo provado, e cada nó da árvore de prova é um objetivo de prova que resulta da aplicação de um *passo de prova* ao nó raiz.

Regras e Estratégias (Comandos de Prova)

- **Regra:** Executa um passo atômico na prova.
  - `skolem!`: introduz automaticamente constantes de Skolem para quantificadores universais no consequente ou uma testemunha para quantificadores existenciais no antecedente;
  - `flatten`: separa os componentes de um antecedente que é uma conjunção ou um consequente que é uma disjunção.
- **Estratégia:** Combina aplicações de regras e/ou outras estratégias.
  - `skosimp*`: introduz automaticamente constantes de Skolem e aplica a simplificação disjuntiva.

Regras e Estratégias (Comandos de Prova)

- **Regra:** Executa um passo atômico na prova.
  - `skolem!`: introduz automaticamente constantes de Skolem para quantificadores universais no consequente ou uma testemunha para quantificadores existenciais no antecedente;
  - `flatten`: separa os componentes de um antecedente que é uma conjunção ou um consequente que é uma disjunção.
- **Estratégia:** Combina aplicações de regras e/ou outras estratégias.
  - `skoSimp*`: introduz automaticamente constantes de Skolem e aplica a simplificação disjuntiva.

Regras e Estratégias (Comandos de Prova)

- **Regra:** Executa um passo atômico na prova.
  - `skolem!`: introduz automaticamente constantes de Skolem para quantificadores universais no consequente ou uma testemunha para quantificadores existenciais no antecedente;
  - `flatten`: separa os componentes de um antecedente que é uma conjunção ou um consequente que é uma disjunção.
- **Estratégia:** Combina aplicações de regras e/ou outras estratégias.
  - `skosimp*`: introduz automaticamente constantes de Skolem e aplica a simplificação disjuntiva.

Regras e Estratégias (Comandos de Prova)

- **Regra:** Executa um passo atômico na prova.
  - `skolem!`: introduz automaticamente constantes de Skolem para quantificadores universais no consequente ou uma testemunha para quantificadores existenciais no antecedente;
  - `flatten`: separa os componentes de um antecedente que é uma conjunção ou um consequente que é uma disjunção.
- **Estratégia:** Combina aplicações de regras e/ou outras estratégias.
  - `skosimp*`: introduz automaticamente constantes de Skolem e aplica a simplificação disjuntiva.

Exemplos de Estratégias Disponíveis

- *Manip*: fornece estratégias para manipulação algébrica;
  - `swap`: comuta dois termos
- *Field*: fornece estratégias de simplificação aritmética (corpo dos números reais)
  - `cancel-by`: cancela termos comuns em igualdades e desigualdades.

Exemplos de Estratégias Disponíveis

- *Manip*: fornece estratégias para manipulação algébrica;
  - `swap`: comuta dois termos
- *Field*: fornece estratégias de simplificação aritmética (corpo dos números reais)
  - `cancel-by`: cancela termos comuns em igualdades e desigualdades.

Uma Especificação de Sistemas Abstratos de Reescrita: ARS.pvs

- Fecho de uma Relação: Usamos as definições dadas por Alfons Geser (.../*lib*/*sets_aux*);
- Terminologia: Redutível, forma normal, juntabilidade, confluência e comutatividade;
- Resultados:
  1. Confluência: Propriedade do Diamante ⇒ Fortemente Confluente
  2. Forma Normal: Confluente e Normalizado ⇔ todo elemento possui forma normal única
  3. Comutação: Duas reduções que comutam fortemente comutam

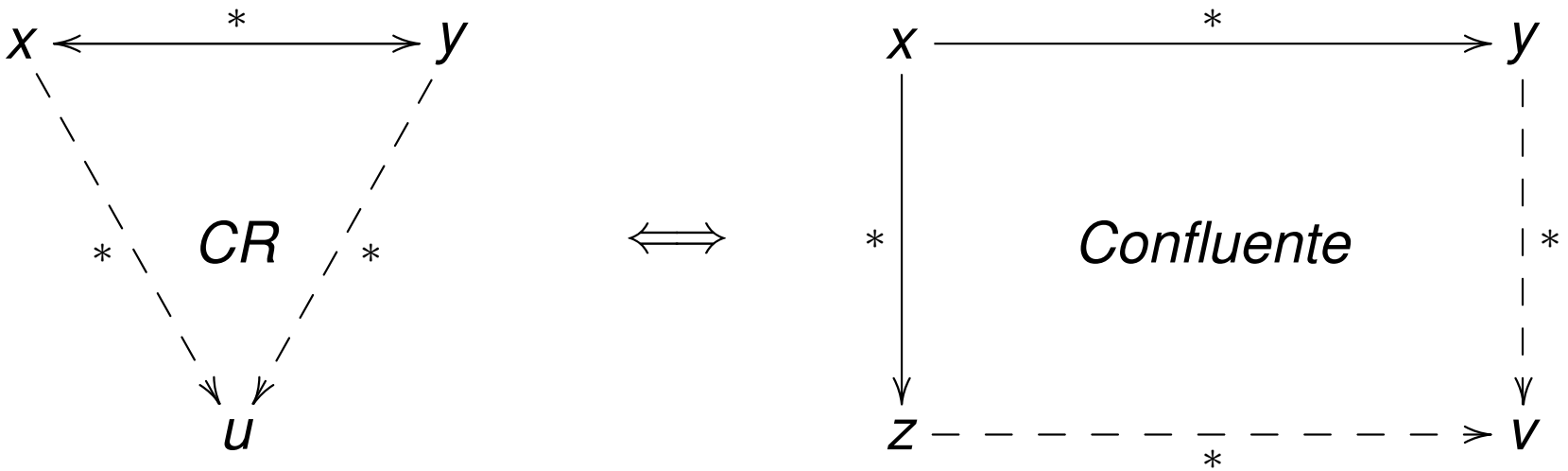
Uma Especificação de Sistemas Abstratos de Reescrita: ARS.pvs

- Fecho de uma Relação: Usamos as definições dadas por Alfons Geser (.../*lib*/*sets_aux*);
- Terminologia: Redutível, forma normal, juntabilidade, confluência e comutatividade;
- Resultados:
  1. Confluência: Propriedade do Diamante $\Rightarrow$ Fortemente Confluente
  2. Forma Normal: Confluente e Normalizado $\Leftrightarrow$ todo elemento possui forma normal única
  3. Comutação: Duas reduções que comutam fortemente comutam

Uma Especificação de Sistemas Abstratos de Reescrita: ARS.pvs

- Fecho de uma Relação: Usamos as definições dadas por Alfons Geser (.../*lib*/*sets_aux*);
- Terminologia: Redutível, forma normal, juntabilidade, confluência e comutatividade;
- Resultados:
  1. Confluência: Propriedade do Diamante $\Rightarrow$ Fortemente Confluente
  2. Forma Normal: Confluente e Normalizado $\Leftrightarrow$ todo elemento possui forma normal única
  3. Comutação: Duas reduções que comutam fortemente comutam

Uma Especificação de Sistemas Abstratos de Reescrita: ARS.pvs

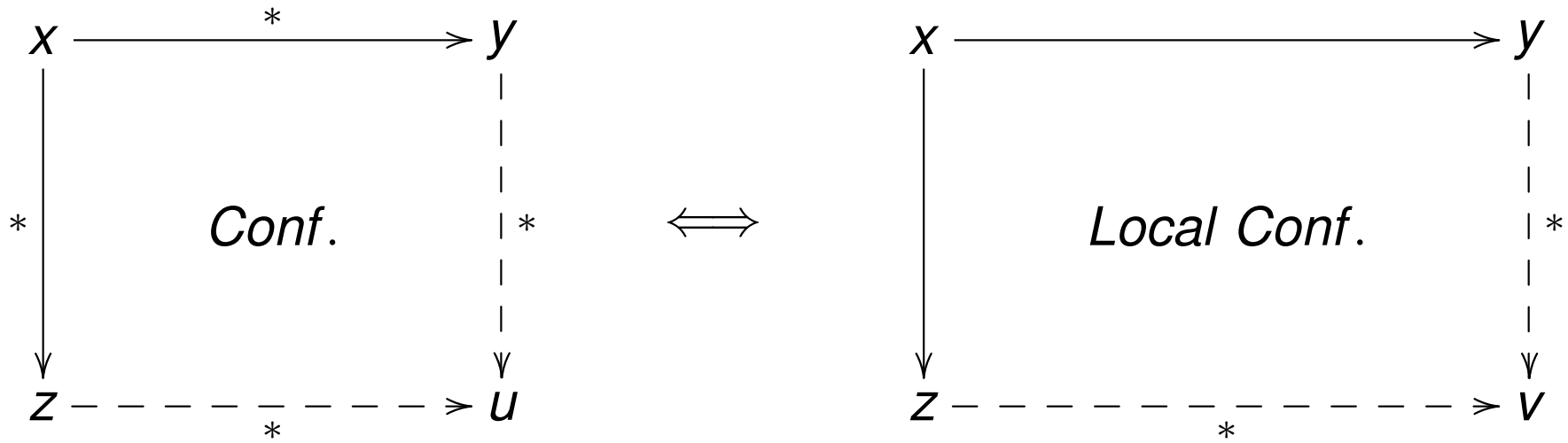


1. $\longrightarrow^* = \bigcup_{n \geq 0} \longrightarrow^n$
2. $\longrightarrow^{n+1} = \longrightarrow^n \circ \longrightarrow = iterate(\longrightarrow, n)$

Uma Especificação de Sistemas Abstratos de Reescrita: ARS.pvs

(Lema de Newman)

Se $\longrightarrow$ é terminante (Noetheriana), então



Relação Noetheriana: noetherian(R) = well-founded(converse(R))

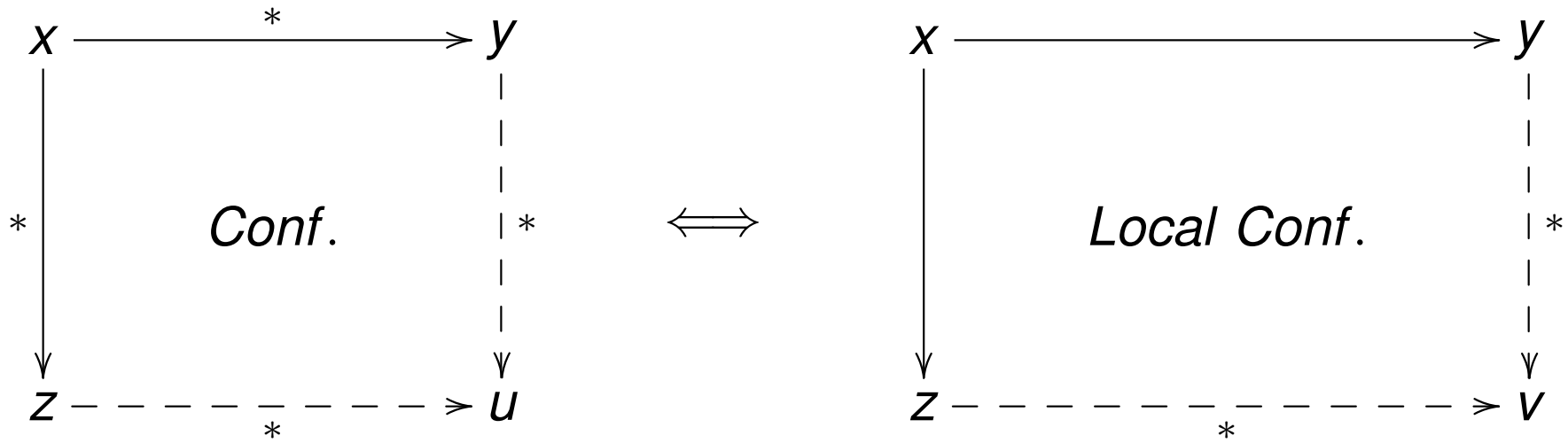
Indução Noetheriana: Seja $(A, \rightarrow)$ um sistema de redução terminante.

$$(\forall x \in A,\ (\forall y \in A,\ x \rightarrow^+ y \ \Rightarrow\ P(y)) \ \Rightarrow\ P(x)) \Rightarrow (\forall x \in A,\ P(x))$$

Uma Especificação de Sistemas Abstratos de Reescrita: ARS.pvs

(Lema de Newman)

Se $\longrightarrow$ é terminante (Noetheriana), então



Relação Noetheriana: noetherian(R) = well-founded(converse(R))

Indução Noetheriana: Seja $(A, \rightarrow)$ um sistema de redução terminante.

$$(\forall x \in A,\ (\forall y \in A,\ x \rightarrow^{+} y \ \Rightarrow\ P(y)) \ \Rightarrow\ P(x)) \Rightarrow (\forall x \in A,\ P(x))$$

Porque Estratégias em PVS são importantes

- O desenvolvimento de estratégias é uma forma de aumentar a sua automação e pode ser:
  - de forma genérica;
  - para problemas específicos;

Porque Estratégias em PVS são importantes

- O desenvolvimento de estratégias é uma forma de aumentar a sua automação e pode ser:
  - de forma genérica:
    - `grind`: usado para tentar completar uma prova automaticamente ou para aplicar todas as simplificações óbvias até que elas não se apliquem mais.
  - para problemas especifícos:
    - `neg-formula`: nega ambos os lados de uma fórmula.

Porque Estratégias em PVS são importantes

- O desenvolvimento de estratégias é uma forma de aumentar a sua automação e pode ser:
  - de forma genérica:
    - `grind`: usado para tentar completar uma prova automaticamente ou para aplicar todas as simplificações óbvias até que elas não se apliquem mais.
  - para problemas especifícos:
    - `neg-formula`: nega ambos os lados de uma fórmula.

Porque Estratégias em PVS são importantes

- O desenvolvimento de estratégias é uma forma de aumentar a sua automação e pode ser:
  - de forma genérica:
    - `grind`: usado para tentar completar uma prova automaticamente ou para aplicar todas as simplificações óbvias até que elas não se apliquem mais.
  - para problemas especifícos:
    - `neg-formula`: nega ambos os lados de uma fórmula.

Algumas Estratégias Básicas

A lista abaixo mostra algumas estratégias usadas para definir novas regras/estratégias de prova:

(APPLY *step*)

(THEN $step_1 \cdots step_n$)

(IF *lisp−expr* $step_1$ $step_2$)

(LET ((*v*$_1$ *lisp−expr*$_1$) ··· (*v*$_n$ *lisp−expr*$_n$)) *step*)

(REPEAT *step*)

Definindo Novas Estratégias

(defstep *nome* *lista de parâmetros* (pode ser vazio)
expressão da estratégia
documentação (opcional)
formato) (opcional)

Definindo Novas Estratégias

```
(defstep   one-after-one (&rest axioms)
           (if (null axioms)
               (skip)
               (let ((rewrite-axiom `(THEN (rewrite ,(car axioms))
                                           (one-after-one$ ,@(cdr axioms)))))
                rewrite-axiom))
           ""
           "Rewriting the list of axioms one-after-one.")
```

```
(defstep   one-after-one-exh (&rest axioms)
           (if (null axioms)
               (skip)
               (let ((rewrite-axiom `(THEN (REPEAT (rewrite ,(car axioms)))
                                           (one-after-one-exh$ ,@(cdr axioms)))))
                rewrite-axiom))
           ""
           "Rewriting the first axiom of the list as much as possible
            and then the second and so on.")
```

Conclusão

- Especificamos ARS em PVS
  1. Indução Noetheriana
  2. Lema de Newman
  3. Lema de Yokouchi-Hikita
  4. Lema da Comutação

Trabalho Futuro

- Especificar *Sistema de reescrita de termos*
- Desenvolver Estratégias
  1. Específicas para ARS.pvs
  2. Teoria de Reescrita

Trabalho Futuro

- Especificar *Sistema de reescrita de termos*
- Desenvolver Estratégias
  1. Específicas para ARS.pvs
  2. Teoria de Reescrita

Referências

- Sam Owre and Natarajan Shankar. *The formal semantics of PVS*. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.
- M. Archer, B. Di Vito and C. Muñoz. *Developing User Strategies in PVS: A Tutorial*. STRATA, Rome, Italy, September 2003.
- `http://pvs.csl.sri.com/`
- F. Baader and T. Nipkow. *Term Rewriting System and All That*. Cambridge University Press, 1998.
- A. L. Galdino and M. Ayala-Rincón and C. Muñoz. *Formal Verification of an Optimal Air Traffic Conflict Resolution and Recovery Algorithm*. In Preparation 2006.