5
Implementação do Corretor de Programas

5.1
Introdução

Para validar as idéias expostas anteriormente, faz-se necessário implementar um corretor de programas com as heurísticas apresentadas. Desta forma, é possível mostrar que o uso do cálculo de Hoare para linguagens imperativas é viável para *proof-carrying-code*. Ou seja, que a geração "automática" de provas gera provas corretas e que o número das que forem geradas raramente será muito grande. Sendo que, com as heurísticas pesquisadas, é sempre possível gerar apenas uma prova correta quando o programa satisfizer a sua especificação.

A implementação foi feita em Prolog [9], por ser uma linguagem com a qual é fácil implementar a análise léxica e sintática de um trecho de código escrito em uma determinada linguagem de programação e de expressões booleanas dadas como asserções. Além disso, nela é fácil se tratar termos e expressões, e montar provas, recursos necessários na escrita do corretor de programas. Afinal, esta é uma linguagem lógica e que possui facilidades para a manipulação simbólica.

Por fim, os invariantes de *loops* são sempre dados pelo usuário.

5.2
A Sintaxe da Linguagem e o Formato das Asserções

A linguagem definida pela BNF abaixo é bem simples, contendo apenas os tipos de dados booleano, inteiro e vetores dos dois tipos anteriores, não possuindo blocos, procedimentos, funções nem declarações de variáveis. Trechos de código que não respeitem esta sintaxe não são aceitos pelo corretor de programas. Um trecho de código é uma lista de comandos (**cmds**).

```
< cmds > ::= < cmd > | < cmd > ; < cmds >
< cmd > ::= < var > := < exp > | skip
         | if < bexp > then < cmds > fi
         | if < bexp > then < cmds > else < cmds > esle_fi
```

```
        | while < bexp > do < cmds > od
< exp > ::= < bexp > | < iexp >
< iexp > ::= < iterm > | < iterm > < fraco > < iexp >
< iterm > ::= < ielem > | < ielem > < forte > < iterm >
< ielem > ::= < num > | < var > | - ( < iexp > )
          | ( < iexp > )
< bexp > ::= < bterm > | < bterm > or < bexp >
< bterm > ::= < belem > | < belem > and < bterm >
< belem > ::= true | false | < var > | not ( < bexp > )
         | ( < bexp > ) | < comp >
< comp > ::= < iexp > < rel > < iexp >
< fraco > ::= + | -
< forte > ::= * | / | mod
< rel > ::= <= | < | = | >= | <>
< var > ::= < end > | < end >[ < vetor > ]
< end > ::= < letra > | < end > < letra > | < end > _
         | < end > < digito >
< vetor > ::= < end > | < end >[ < vetor > ]
< num > ::= < digito > | < digito > < num >
< letra > ::= A | ... | Z | a | ... | z
< digito > ::= 0 | ... | 9
```

Abaixo, pode-se ver o formato das asserções, que são as pré-condições, as pós-condições e os invariantes dos laços **while**. Basicamente, trata-se da lógica predicativa, com a igualdade e com termos aritméticos. Uma asserção é uma condição (**cond**). Nesta BNF, só foram colocadas as definições não contidas na apresentada acima:

```
< cond > ::= true | false | < var >
< var > ::= < end > | ‘< maiuscula >< end >’
           | vet( < end >, var )
           | vet( ‘< maiuscula >< end >’, var )
           | < cond > and < cond > | < cond > or < cond >
           | < cond > => < cond > | < cond > <=> < cond >
           | < cond > = < cond > | < cond > <> < cond >
           | not < cond > | ( < cond > )
           | ( < Exp > = < Exp > ) | ( < Exp > <> < Exp > )
           | ( < Exp > < < Exp > ) | ( < Exp > > < Exp > )
           | ( < Exp > <= < Exp > ) | ( < Exp > >= < Exp > )
           | ex( < end >, < cond > )
           | ex( ‘< maiuscula >< end >’, < cond > )
           | all( < end >, < cond > )
           | all( ‘< maiuscula >< end >’, < cond > )
< Exp > ::= < num > | < var > | ‘< var >’ | - < Exp >
          | < Exp > + < Exp > | < Exp > - < Exp >
          | < Exp > * < Exp > | < Exp > / < Exp >
          | < Exp > mod < Exp >
< maiuscula > ::= A | ... | Z
```

5.3
Implementação das Regras do Cálculo de Hoare

Para aplicar as regras, criou-se um predicado chamado **fazerHoare( )**, o qual possui os seguintes parâmetros, nesta ordem:

- **Invars**: uma lista de pares **< *loop*, invariante >**, para cada laço **while** dado no programa sendo verificado formalmente;
- **PreCond**: a pré-condição para o trecho de programa sendo corrigido;
- **Cmds**: o trecho de código sendo corrigido;
- **PosCond**: a pós-condição para o trecho de programa sendo corrigido;
- **Prova**: a árvore de prova resultante da aplicação da regra;
- **SentsAnt**: sentenças a serem demonstradas antes de se aplicar a regra; e
- **SentsDep**: sentenças a serem demonstradas depois da aplicação da regra.

5.3.1
Comando skip

Neste comando, pode acontecer duas situações. Na primeira, a pré-condição e a pós-condição são determinadas e ambas são iguais, ou uma delas não foi determinada ainda, mas, como se sabe, devem ser iguais para se ter um axioma, portanto, a asserção não determinada passa a ser idêntica a determinada. Em Prolog, tem-se o seguinte:

```
1 fazerHoare( _Invars, Cond, [ skip ], Cond, Prova, Sents, Sents ) :-
2    Prova = [ [ Cond, [ skip ], Cond ] ]                          ,
3    !                                                             .
```

No outro caso, as asserções são diferentes. Portanto, deve-se realizar ou o fortalecimento da pré-condição ou o enfraquecimento da pós-condição para que as duas sejam idênticas, o que gera uma sentença a ser demonstrada. Na implementação, foi escolhida a primeira opção, onde tem-se que:

```
1 fazerHoare( _Invars, PreCond, [ skip ], PosCond, Prova, SentsAnt, SentsDep ) :-
2    Prova = [ [ PreCond, [ skip ], PosCond ], [ PreCond => PosCond ],
3              [ [ PosCond, [ skip ], PosCond ] ] ]                          ,
4    SentsDep = [ PreCond => PosCond | SentsAnt ]                           ,
5    !                                                                      .
```

Seria possível fazer uma escolha não-determinística entre as duas opções, o que infelizmente tornaria a implementação muito mais complexa. Adicionalmente, com a geração determinística de provas, uma mesma prova

será gerada para cada entrada ao corretor de programas. Logo, para essa e outras regras, a implementação não possui não-determinismos.

5.3.2
Comando de atribuição

Se a variável que estiver sendo modificada for a posição de um vetor e a pós-condição for conhecida, se acontecer de a pós-condição depois da substituição ser igual a nova pré-condição encontrada e o endereço do vetor aparecer nela, é porque a propriedade sendo descrita é com relação a todo vetor e não ao elemento sendo modificado. Logo, é preciso encontrar uma nova pós-condição que implique na antiga através de enfraquecimento, a qual vai conter uma referência específica à posição sendo modificada. A nova pós-condição é encontrada nas linhas 12 e 13 na chamada recursiva ao corretor, produzindo na variável **NovaPosCond** a nova pós-condição. Já na linha 17, a implicação da nova pós-condição na antiga é adicionada à lista de sentenças a serem demonstradas. Neste caso, a pré-condição deve estar definida para o programa não entrar em *loop*. Isso é feito aplicando-se o cálculo de Hoare da seguinte maneira:

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                   SentsDep ) :-
3    nonvar( PosCond )                                                          ,
4    converter( Exp, EquivExp )                                                 ,
5    serIgual( CondTemp, Var, EquivExp, PosCond )                               ,
6    CondTemp == PosCond                                                        ,
7    Var = vet( End, _Idx )                                                     ,
8    acharVar( End, PosCond )                                                   ,
9    ( nonvar( PreCond )                                                        ;
10     ( !                                                                      ,
11       fail ) )                                                               ,
12   fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], NovaPosCond,
13                                ProvaFilha, SentsAnt, SentsTemp )             ,
14   NovaPosCond \= ex( _VarLig, _CondLig )                                     ,
15   ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                   ,
16   Prova = [ ProvaPai, ProvaFilha, [ NovaPosCond => PosCond ] ]               ,
17   SentsDep = [ NovaPosCond => PosCond | SentsTemp ]                          ,
18   !                                                                          .
```

Caso contrário, para se saber a pré-condição é necessário somente fazer a substituição da variável que recebe o novo valor pela expressão atribuída na pós-condição, ou seja, corrigir a atribuição de trás para frente (heurística apresentada na seção 4.5), o que é feito na linha 5. Se já houver uma pré-condição determinada, diferente da encontrada pela substituição, é necessário realizar o fortalecimento da pré-condição. Assim:

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                   SentsDep ) :-
3    nonvar( PosCond )                                                          ,
4    converter( Exp, EquivExp )                                                 ,
5    serIgual( CondTemp, Var, EquivExp, PosCond )                               ,
6    ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                   ,
7    ( var( PreCond )                                                           ->
8      PreCond = CondTemp                                                       ;
9      true )                                                                   ,
10   ( PreCond == CondTemp                                                      ->
11     ( Prova = [ ProvaPai ]                                                   ,
12       SentsDep = SentsAnt )                                                  ;
13     ( Prova = [ ProvaPai, [ PreCond => CondTemp ],
14                 [ [ CondTemp, [ atribui( Var, Exp ) ], PosCond ] ] ]         ,
15       SentsDep = [ PreCond => CondTemp | SentsAnt ] ) )                      ,
16   !                                                                          .
```

Pode ser que a pós-condição contenha variáveis ligadas através de um quantificador e seus nomes estejam presentes em algumas na expressão que substituirá a variável que a recebe na pré-condição. Portanto, precisam ser renomeadas de forma que não aconteça nenhum erro durante a substituição, o que é feito na linha 5. Então, a pós-condição precisa ser enfraquecida para ter as variáveis ligadas renomeadas caso isso seja necessário. Posteriormente, deve-se aplicar a regra acima, o que acontece na chamada recursiva da linha 9. Isso está explicado neste parágrafo que foi implementado assim:

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                   SentsDep ) :-
3    nonvar( PosCond )                                                          ,
4    converter( Exp, EquivExp )                                                 ,
5    renomearLig( Var, EquivExp, PosCond, NovaPosCond )                         ,
6    PosCond \== NovaPosCond                                                    ,
7    ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                   ,
8    !                                                                          ,
9    fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], NovaPosCond,
10                               ProvaFilha, SentsAnt, SentsTemp )              ,
11   Prova = [ ProvaPai, ProvaFilha, [ NovaPosCond => PosCond ] ]               ,
12   SentsDep = [ NovaPosCond => PosCond | SentsTemp ]                          .
```

Caso não se conheça a pós-condição ou se tenha o problema já descrito com vetores, uma alternativa é tentar achar a pós-condição mais forte depois da atribuição dada a pré-condição. Quando se tem o caso **{ P } x := E { Q }** (apresentado na seção 4.4.1 e nas figuras 13 e 15), tem-se que ter o cuidado de, se **x** for uma posição de um vetor, não ter seu endereço em **P**, o que invalida a heurística, pois isso faz com que a pré-condição seja uma propriedade em relação ao vetor por inteiro. Isto é testado nas linhas 4 e 5. Já nas linhas 10, 11, 15 e 19, testa-se a pré-condição para saber qual seu valor e, assim, saber se o formato da prova será semelhante ao da figura 13 ou 15 (testes semelhantes são feitos em todas as heurísticas da atribuição):

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                   SentsDep ) :-
3    \+ acharVar( Var, PreCond )                                                ,
```

```
4    ( Var = vet( End, _Idx )                                                  ->
5      \+ acharVar( End, PreCond )                                             ;
6      true )                                                                  ,
7    converter( Exp, EquivExp )                                                ,
8    \+ acharVar( Var, EquivExp )                                              ,
9    Igualdade = ( EquivExp = EquivExp )                                       ,
10   ( ( PreCond == true                                                       ;
11       PreCond == Igualdade )                                                ->
12     PosCond = ( Var = EquivExp )                                            ;
13     PosCond = PreCond and ( Var = EquivExp ) )                              ,
14   ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                  ,
15   ( ( PreCond == true                                                       ,
16       Prova = [ ProvaPai, [ true => Igualdade ],
17                 [ [ Igualdade, [ atribui( Var, Exp ) ], PosCond ] ] ]      ,
18       SentsDep = [ true => Igualdade | SentsAnt ] )                         ;
19     ( PreCond == Igualdade                                                  ,
20       Prova = [ ProvaPai ]                                                  ,
21       SentsDep = SentsAnt )                                                 ;
22     ( Prova = [ ProvaPai, [ PreCond => ( PreCond and Igualdade ) ],
23                 [ [ PreCond and Igualdade, [ atribui( Var, Exp ) ],
24                     PosCond ] ] ]                                           ,
25       SentsDep = [ PreCond => ( PreCond and Igualdade ) | SentsAnt ] ) )    ,
26   !                                                                         .
```

Já na situação onde se tem **{ P } x := E(x) { Q }** (apresentada na seção 4.4.2 e nas figuras 18 e 19), **Q** possuirá uma expressão com o quantificador existencial. Portanto, a variável ligada deve ser diferente das que aparecem na expressão **E(x)**. Devido a este fato, é necessário encontrar uma variável diferente de todas elas. Na implementação, a variável nova é o próprio **x** concatenado com um ou mais ‘’’s. No caso de **x** ser da forma **y[i]**, a nova variável é o **y** concatenado com um ou mais ‘’’s. Isto é feito nas linhas 8 e 9. Ainda, o mesmo cuidado tomado no caso anterior com vetores é necessário aqui. Assim:

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                    SentsDep ) :-
3    \+ acharVar( Var, PreCond )                                               ,
4    ( Var = vet( End, _Idx )                                                  ->
5      \+ acharVar( End, PreCond )                                             ;
6      true )                                                                  ,
7    converter( Exp, EquivExp )                                                ,
8    acharVar( Var, EquivExp )                                                 ,
9    acharNovaVar( EquivExp, Var, NovaVar )                                    ,
10   serIgual( NovaExp, Var, NovaVar, EquivExp )                               ,
11   NovoTermo = ex( NovaVar, EquivExp = NovaExp )                             ,
12   ( ( PreCond == true                                                       ;
13       PreCond == NovoTermo )                                                ->
14     PosCond = ex( NovaVar, Var = NovaExp )                                  ;
15     PosCond = PreCond and ex( NovaVar, Var = NovaExp ) )                    ,
16   ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                  ,
17   ( ( PreCond == true                                                       ,
18       Prova = [ ProvaPai, [ true => NovoTermo ],
19                 [ [ NovoTermo, [ atribui( Var, Exp ) ], PosCond ] ] ]      ,
20       SentsDep = [ true => NovoTermo | SentsAnt ] )                         ;
21     ( PreCond == NovoTermo                                                  ,
22       Prova = [ ProvaPai ]                                                  ,
23       SentsDep = SentsAnt )                                                 ;
24     ( Prova = [ ProvaPai, [ PreCond => ( PreCond and NovoTermo ) ],
25                 [ [ PreCond and NovoTermo, [ atribui( Var, Exp ) ],
26                     PosCond ] ] ]                                           ,
27       SentsDep = [ PreCond => ( PreCond and NovoTermo ) | SentsAnt ] ) )    ,
28   !                                                                         .
```

Quanto às situações **{ P(x) } x := E { Q }** e **{ P(x) } x := E(x) { Q }** (apresentadas nas seções 4.4.3 e 4.4.4, respectivamente), deve-se tentar encontrar o valor de **x** em **P(x)** para ser possível aplicar as heurísticas apresentadas nas figuras 21 e 23, e 26 e 28, respectivamente. Na implementação, conseguiu-se fazer isso se **P(x)** for um conjunto de conjunções da forma $A_1 \wedge ... \wedge A_n$, onde um dos A_i é da forma **x = a**, **a = x**, **x** ou **not x**, o que é feito na linha 5. Nos dois últimos casos, **x** é uma variável booleana com os valores **true** e **false**, respectivamente. Como a única diferença entre estas duas situações é a necessidade de se fazer uma substituição na expressão, que no primeiro caso não modifica a expressão, ambas foram implementadas em uma só cláusula.

Na implementação, ao se obter o **P(a)**, retira-se do conjunto de conjunções as formas **a = a**, **true** ou **not false**, para não deixar redundâncias na mesma. Eis aqui a implementação:

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                  SentsDep )   :-
3    acharVar( Var, PreCond )                                                    ,
4    converter( Exp, EquivExp )                                                  ,
5    procurarIgualdade( Var, PreCond, NovaCond, Expressao )                      ,
6    nonvar( Expressao )                                                         ,
7    serIgual( ExpSubst, Var, Expressao, EquivExp )                              ,
8    ( NovaCond == true                                                          ->
9      ( NovaPreCond = ( EquivExp = ExpSubst )                                   ,
10       PosCond = ( Var = ExpSubst ) )                                          ;
11     ( serIgual( CondTemp, Var, Expressao, NovaCond )                          ,
12       NovaPreCond = ( CondTemp and ( EquivExp = ExpSubst ) )                  ,
13       PosCond = ( CondTemp and ( Var = ExpSubst ) ) ) )                       ,
14   ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                    ,
15   ProvaFilha = [ NovaPreCond, [ atribui( Var, Exp ) ], PosCond ]              ,
16   Prova = [ ProvaPai, [ PreCond => NovaPreCond ], [ ProvaFilha ] ]            ,
17   SentsDep = [ PreCond => NovaPreCond | SentsAnt ]                            ,
18   !                                                                           .
```

Caso não se consiga achar o valor de **x**, deve-se introduzir o existencial, como mostrado nas figuras 24 e 29. Portanto, é necessário achar uma variável cujo nome não apareça em nenhuma variável livre da pré-condição e da expressão (no caso de **E** depender de **x**), o que é feito nas linhas 5 a 7. A maneira de se criar um nome novo para uma variável é o mesmo já explicado anteriormente. Eis aqui a implementação:

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                  SentsDep )   :-
3    acharVar( Var, PreCond )                                                    ,
4    converter( Exp, EquivExp )                                                  ,
5    acharNovaVar( PreCond, Var, VarTemp )                                       ,
6    ( acharVar( Var, EquivExp )                                                 ->
7      acharNovaVar( EquivExp, VarTemp, NovaVar )                                ;
8      NovaVar = VarTemp )                                                       ,
9    serIgual( NovaCond, Var, NovaVar, PreCond )                                 ,
10   ( acharVar( Var, EquivExp )                                                 ->
```

```
11     ( serIgual( ExpSubst, Var, NovaVar, EquivExp )                          ,
12       NovaPreCond = ex( NovaVar, NovaCond and ( EquivExp = ExpSubst ) )     ,
13       PosCond = ex( NovaVar, NovaCond and ( Var = ExpSubst ) ) )            ;
14     ( NovaPreCond = ( ex( NovaVar, NovaCond ) and ( EquivExp = EquivExp ) ) ,
15       PosCond = ( ex( NovaVar, NovaCond ) and ( Var = EquivExp ) ) ) )      ,
16   ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                  ,
17   ProvaFilha = [ NovaPreCond, [ atribui( Var, Exp ) ], PosCond ]            ,
18   Prova = [ ProvaPai, [ PreCond => NovaPreCond ], [ ProvaFilha ] ]          ,
19   SentsDep = [ PreCond => NovaPreCond | SentsAnt ]                          ,
20   !                                                                         .
```

Quando há o problema com vetores explanado anteriormente, nenhuma das heurísticas da atribuição pode ser aplicada, então é necessário interagir com o usuário, requisitando a nova pós-condição. A partir desta, acha-se a nova pré-condição correspondente e monta-se a prova fortalecendo a pré-condição. Obviamente, o usuário deve entrar uma pós-condição contendo a posição do vetor sendo modificada explicitamente para que a substituição de variáveis possa de fato ocorrer. Adicionalmente, essa pós-condição deve ser significativa, senão, a prova não ficará correta. Desta maneira:

```
1 fazerHoare( _Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                  SentsDep )   :-
3    write( 'Pré-condição: ' )                                                  ,
4    imprimirCond( PreCond )                                                    ,
5    write( '\nComando: ' )                                                     ,
6    write( atribui( Var, Exp ) )                                               ,
7    write( '\nEntre a nova pós-condição contendo ' )                           ,
8    imprimirVetor( Var )                                                       ,
9    write( ' explicitamente: ' )                                               ,
10   read( PosCond )                                                            ,
11   checarCondicao( PosCond )                                                  ,
12   converter( Exp, EquivExp )                                                 ,
13   serIgual( NovaPreCond, Var, EquivExp, PosCond )                            ,
14   ProvaPai = [ PreCond, [ atribui( Var, Exp ) ], PosCond ]                   ,
15   ProvaFilha = [ NovaPreCond, [ atribui( Var, Exp ) ], PosCond ]             ,
16   Prova = [ ProvaPai, [ PreCond => NovaPreCond ], [ ProvaFilha ] ]           ,
17   SentsDep = [ PreCond => NovaPreCond | SentsAnt ]                           ,
18   !                                                                          .

1 fazerHoare( Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova, SentsAnt,
2                                                                  SentsDep )   :-
3    write( '\nNova pós-condição inválida! Tente de novo!\n' )                  ,
4    fazerHoare( Invars, PreCond, [ atribui( Var, Exp ) ], PosCond, Prova,
5                                SentsAnt, SentsDep )                          ,
6    !                                                                          .
```

Percebe-se que todas as regras acima utilizam o corte (**!**). Isso tem a utilidade de impedir que a regra seguinte da atribuição que está sendo aplicada seja utilizada, evitando, assim, o *backtracking* e, conseqüentemente, a geração de mais de uma prova quando uma correta já foi encontrada.

5.3.3
Comandos if-then e if-then-else

As regras destes comandos foram implementadas levando em conta que a pré-condição é sempre determinada, algo garantido pela implementação da regra para a seqüência de comandos. Quando a pós-condição é determinada, a regra utilizada é a primeira ou a terceira, para o **if-then** ou o **if-then-else**, respectivamente, quando o corpo do **if** será corrigido de trás para frente (heurística apresentada na seção 4.5, na figura 31). Já quando esta asserção não é determinada, a pós-condição é a disjunção das pós-condições, como explicado na seção 4.6 e mostrado nas figuras 32 e 33. Neste caso, a segunda ou a quarta regra será utilizada.

Portanto, a implementação das regras do **if-then** e do **if-then-else** ficam assim, respectivamente:

```
fazerHoare( Invars, PreCond, [ if( Teste, Entao ) ], PosCond, Prova, SentsAnt,
                                                          SentsDep )    :-
    nonvar( PosCond )                                                   ,
    !                                                                   ,
    converter( Teste, EquivTeste )                                      ,
    fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond, ProvaFilha,
                                                   SentsAnt, SentsTemp ) ,
    SentsDep = [ ( PreCond and not EquivTeste ) => PosCond | SentsTemp ]     ,
    Prova = [ [ PreCond, [ if( Teste, Entao ) ], PosCond ], ProvaFilha,
              [ ( PreCond and not EquivTeste ) => PosCond ] ]               .

fazerHoare( Invars, PreCond, [ if( Teste, Entao ) ], PosCond, Prova, SentsAnt,
                                                          SentsDep )    :-
    converter( Teste, EquivTeste )                                      ,
    fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond_1, ProvaNeta,
                                                   SentsAnt, SentsTemp ) ,
    PosCond_2 = ( PreCond and not EquivTeste )                          ,
    PosCond = ( PosCond_1 or PosCond_2 )                                ,
    ProvaFilha = [ [ PreCond and EquivTeste, Entao, PosCond ], ProvaNeta,
                   [ ( PosCond_1 => PosCond ) ] ]                       ,
    SentsDep = [ ( PosCond_2 => PosCond ), ( PosCond_1 => PosCond )
                                       | SentsTemp ]                    ,
    Prova = [ [ PreCond, [ if( Teste, Entao ) ], PosCond ], ProvaFilha,
              [ PosCond_2 => PosCond ] ]                                .

fazerHoare( Invars, PreCond, [ if( Teste, Entao, Senao ) ], PosCond, Prova,
                                             SentsAnt, SentsDep )       :-
    nonvar( PosCond )                                                   ,
    !                                                                   ,
    converter( Teste, EquivTeste )                                      ,
    fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond, ProvaFilha_1,
                                                   SentsAnt, SentsTemp ) ,
    fazerHoare( Invars, PreCond and not EquivTeste, Senao, PosCond,
                        ProvaFilha_2, SentsTemp, SentsDep )             ,
    Prova = [ [ PreCond, [ if( Teste, Entao, Senao ) ], PosCond ],
              ProvaFilha_1, ProvaFilha_2 ]                              .

fazerHoare( Invars, PreCond, [ if( Teste, Entao, Senao ) ], PosCond, Prova,
                                             SentsAnt, SentsDep )       :-
    converter( Teste, EquivTeste )                                      ,
    !                                                                   ,
    fazerHoare( Invars, PreCond and EquivTeste, Entao, PosCond_1, ProvaNeta_1,
                                            SentsAnt, SentsTemp_1 )     ,
    fazerHoare( Invars, PreCond and not EquivTeste, Senao, PosCond_2,
                        ProvaNeta_2, SentsTemp_1, SentsTemp_2 )         ,
    PosCond = ( PosCond_1 or PosCond_2 )                                ,
    ProvaFilha_1 = [ [ PreCond and EquivTeste, Entao, PosCond ], ProvaNeta_1,
```

```
11                    [ ( PosCond_1 => PosCond ) ] ]                          ,
12    ProvaFilha_2 = [ [ PreCond and not EquivTeste, Senao, PosCond ],
13                     ProvaNeta_2, [ ( PosCond_2 => PosCond ) ] ]            ,
14    SentsDep = [ ( PosCond_1 => PosCond ), ( PosCond_2 => PosCond )
15                                          | SentsTemp_2 ]                   ,
16    Prova = [ [ PreCond, [ if( Teste, Entao, Senao ) ], PosCond ],
17              ProvaFilha_1, ProvaFilha_2 ]                                  .
```

5.3.4
Comando while

A implementação da regra deste comando (cuja estrutura de prova foi apresentada na seção 4.2, na figura 10) foi separada em três partes, que devem ser colocadas estritamente na ordem apresentada aqui. Na primeira, a pré-condição já é igual ao invariante ou ainda não foi determinada, e a pós-condição é igual a conjunção do invariante com a negação do teste do **while** ou ainda não foi determinada. Tem-se a certeza de que é o invariante certo que está sendo usado pela consulta à tabela de invariantes para cada *loop*. A implementação desta parte fica assim:

```
1 fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ], Invar and not EquivTeste,
2                                          Prova, SentsAnt, SentsDep )     :-
3    procurarInvar( Invars, while( Teste, Corpo ), Invar )                  ,
4    converter( Teste, EquivTeste )                                         ,
5    !                                                                      ,
6    fazerHoare( Invars, Invar and EquivTeste, Corpo, Invar, ProvaFilha,
7                                                 SentsAnt, SentsDep )    ,
8    Prova = [ [ Invar, [ while( Teste, Corpo ) ], Invar and not EquivTeste ],
9              ProvaFilha ]                                                .
```

Na segunda, a pré-condição já é igual ao invariante ou ainda não foi determinada e a pós-condição é diferente da conjunção do invariante com a negação do teste do **while**. Portanto, é necessário enfraquecer a pós-condição para torná-la igual ao desejado para a correção formal deste comando. Neste caso, a implementação fica assim:

```
1 fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ], PosCond, Prova, SentsAnt,
2                                                                SentsDep )  :-
3    procurarInvar( Invars, while( Teste, Corpo ), Invar )                  ,
4    converter( Teste, EquivTeste )                                         ,
5    !                                                                      ,
6    fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ],
7                Invar and not EquivTeste, ProvaFilha, SentsAnt, SentsTemp ) ,
8    SentsDep = [ ( Invar and not EquivTeste ) => PosCond | SentsTemp ]     ,
9    Prova = [ [ Invar, [ while( Teste, Corpo ) ], PosCond ], ProvaFilha,
10             [ ( Invar and not EquivTeste ) => PosCond ] ]                 .
```

Por fim, no último caso, a pré-condição não é igual ao invariante. Logo, precisa-se fortalecer a pré-condição para torná-la igual ao invariante. Nesta situação, a implementação fica assim:

```
1 fazerHoare( Invars, PreCond, [ while( Teste, Corpo ) ], PosCond, Prova,
2                                                        SentsAnt, SentsDep ) :-
3    procurarInvar( Invars, while( Teste, Corpo ), Invar )                     ,
4    !                                                                         ,
5    fazerHoare( Invars, Invar, [ while( Teste, Corpo ) ], PosCond, ProvaFilha,
6                                              SentsAnt, SentsTemp )          ,
7    SentsDep = [ PreCond => Invar | SentsTemp ]                               ,
8    Prova = [ [ PreCond, [ while( Teste, Corpo ) ], PosCond ],
9              [ PreCond => Invar ], ProvaFilha ]                              .
```

Note-se os cortes em cada uma das partes da implementação da regra, para que as demais não sejam tentadas por *backtracking*.

5.3.5
Seqüência de Comandos

Caso o último comando não seja um **if-then** ou um **if-then-else** (o que é testado nas linhas 7 e 8), pode-se começar a corrigir o trecho de código pelo último comando, desde que a pós-condição seja determinada (pode acontecer de ela não ser determinada se esse conjunto de comandos estiver no corpo de um **if**). Nesta situação, a implementação é a seguinte:

```
1 fazerHoare( Invars, PreCond, Comandos, PosCond, Prova, SentsAnt, SentsDep )  :-
2    length( Comandos, TamCmds )                                                ,
3    TamCmds > 1                                                                ,
4    nonvar( PosCond )                                                          ,
5    last( Ultimo, Comandos )                                                   ,
6    append( RestCmds, [ Ultimo ], Comandos )                                   ,
7    Ultimo \= if( _Teste, _Entao )                                             ,
8    Ultimo \= if( _Teste, _Entao, _Senao )                                     ,
9    !                                                                          ,
10   fazerHoare( Invars, CondTemp, [ Ultimo ], PosCond, ProvaFilha_2, SentsAnt,
11                                                    SentsTemp )               ,
12   fazerHoare( Invars, PreCond, RestCmds, CondTemp, ProvaFilha_1, SentsTemp,
13                                                              SentsDep ) ,
14   Prova = [ [ PreCond, Comandos, PosCond ], ProvaFilha_1, ProvaFilha_2 ]     .
```

Senão, deve-se começar pelo primeiro comando desde que a pré-condição seja determinada. Nessa situação, tem-se que:

```
1 fazerHoare( Invars, PreCond, [ Cmd | Cmds ], PosCond, Prova, SentsAnt,
2                                                          SentsDep )      :-
3    length( Cmds, TamCmds )                                                  ,
4    TamCmds > 0                                                              ,
5    nonvar( PreCond )                                                        ,
6    !                                                                        ,
7    fazerHoare( Invars, PreCond, [ Cmd ], CondTemp, ProvaFilha_1, SentsAnt ,
8                                                            SentsTemp ) ,
9    fazerHoare( Invars, CondTemp, Cmds, PosCond, ProvaFilha_2, SentsTemp,
10                                                          SentsDep )    ,
11  Prova = [ [ PreCond, [ Cmd | Cmds ], PosCond ], ProvaFilha_1,
12                                                 ProvaFilha_2 ]           .
```

5.3.6
Fortalecimento da Pré-Condição e Enfraquecimento da Pós-Condição

A implementação direta dessas regras não foi feita, já que ela só é aplicada nos axiomas, no comando **while** e no **if**. Portanto, quando são necessários, são feitos diretamente nas implementações das regras que os usam. Embora isso torne estas regras mais complexas, o enfraquecimento e o fortalecimento não devem ser implementados separadamente para não permitir o uso deles em outro lugar que não seja nos axiomas, **while** e **if**, o que aumentaria o espaço de busca e, conseqüentemente, diminuiria a eficiência do corretor de programas.

5.4
Substituição de Variáveis

Na regra do comando de atribuição, é necessário se fazer a substituição de uma variável por uma expressão em uma asserção lógica. Quando essa asserção possuir variáveis ligadas através do quantificador existencial, essa substituição tem que ser feita com cuidado para que a semântica da condição não seja modificada. Nesta seção, será mostrado como foi feita a implementação nas situações onde a pós-condição possui variáveis ligadas. A idéia foi a de implementar o que é utilizado no λ-calculus [5].

O predicado que realiza a substituição de variáveis é o **serIgual( )**, que possui os seguintes parâmetros, nesta ordem:

- **PreCond**: a pré-condição da atribuição;
- **Var**: a variável que será substituída;
- **Exp**: a expressão que substituirá a variável; e
- **PosCond**: a pós-condição da atribuição.

Quando a atribuição for da forma **x := E** e a pós-condição possuir um termo da forma $**x R(x)** ou "**x R(x)**, como **x** não se encontra livre no mesmo, nada tem que ser substituído. O mesmo ocorre quando este termo for da forma **R**, ou seja, não depender de **x**. Logo, tem-se que:

```
1 serIgual( Cond, Var, _Exp, Cond ) :-
2    \+ var( Cond )                  ,
3    \+ acharVar( Var, Cond )        ,
4    !                               .
```

onde **acharVar( )** verifica se uma variável está livre em uma expressão ou não.

Já quando a pós-condição possuir um termo da forma $**y R(x, y)** ou "**y R(x, y)**, a substituição pode ocorrer recursivamente sem a renomeação de **y** se esta variável não ocorrer (livremente) em **E**. Assim:

```
1 serIgual( PreCond, Var, Exp, PosCond )          :-
2    ( ( PreCond = ex( VarLig, PreCondLig )       ,
3        PosCond = ex( VarLig, PosCondLig ) )     ;
4      ( PreCond = all( VarLig, PreCondLig )      ,
5        PosCond = all( VarLig, PosCondLig ) ) )  ,
6    \+ acharVar( VarLig, Exp )                   ,
7    serIgual( PreCondLig, Var, Exp, PosCondLig ) .
```

Quando for necessário renomear uma variável, porque o **y** aparece em **E**, é necessário escolher uma nova variável que não apareça nem em **R** nem em **E**. Isso foi implementado em outro predicado, já que há a necessidade de enfraquecer a pós-condição $**y R(x, y)** para $**y' R(x, y')** ou "**y R(x, y)** para "**y' R(x, y')**, supondo que **y'** não apareça livre em **E** e em **R**. Este é o **renomearLig( )**, que possui os seguintes atributos, nesta ordem:

- **Var**: a variável que será substituída;
- **Exp**: a expressão que substituirá a variável;
- **Condicao**: a pós-condição da atribuição; e
- **NovaCondicao**: a nova pós-condição com as devidas variáveis renomeadas.

A implementação desse predicado quando aparecem termos com variáveis ligadas é a seguinte:

```
1 renomearLig( Var, Exp, Condicao, NovaCondicao )   :-
2    ( ( Condicao = ex( VarLig, Cond )              ,
3        NovaCondicao = ex( VarLig, NovaCond ) )    ;
4      ( Condicao = all( VarLig, Cond )             ,
5        NovaCondicao = all( VarLig, NovaCond ) ) ) ,
6    \+ acharVar( VarLig, Exp )                     ,
7    renomearLig( Var, Exp, Cond, NovaCond )        ,
8    !                                              .
```

```
1 renomearLig( Var, Exp, Condicao, NovaCondicao )          :-
2    ( ( Condicao = ex( VarLig_1, Cond )                   ,
3        NovaCondicao = ex( VarLig_2, NovaCond ) )         ;
4      ( Condicao = all( VarLig_1, Cond )                  ,
5        NovaCondicao = all( VarLig_2, NovaCond ) ) )      ,
6    acharNovaVarLig( Var, Exp, Cond, VarLig_1, VarLig_2 ) ,
7    serIgual( CondTemp, VarLig_1, VarLig_2, Cond )        ,
8    renomearLig( Var, Exp, CondTemp, NovaCond )           ,
9    !                                                     .
```

Na primeira cláusula, a variável ligada não é encontrada na expressão. Logo pode-se analisar recursivamente a pós-condição sem a necessidade de renomear a variável. Já quando o contrário acontece, é necessário achar uma

nova variável que satisfaça os requisitos já descritos, substitui-la na pós-condição e analisar recursivamente a nova pós-condição.