3
Cálculo de Hoare

3.1
Semântica Axiomática de Linguagens de Programação

As técnicas para as semânticas operacional e denotacional de linguagens de programação são fundamentadas na noção de “estado de uma máquina”. Quanto a semântica denotacional, esta não subentende um modelo computacional (formal) mas sim um modelo matemático, o qual permite especificar um modelo computacional como um tipo abstrato de dados. Por sua vez, a semântica axiomática é baseada em métodos de dedução da lógica predicativa, que provê uma base lógica para as provas de propriedades de programas. Uma das propriedades mais importantes de um programa é se ele realiza a função pretendida. É obvio que essa semântica é mais abstrata do que as semânticas citadas acima. Afinal, o significado semântico de um programa é baseado em asserções sobre relações que permanecem sendo as mesmas a cada vez que o programa é executado [5, 27].

Estas asserções que formam uma especificação semântica axiomática, em geral, não vão descrever valores particulares para cada variável, mas sim especificar certas propriedades gerais dos valores das variáveis e a relação entre elas [27]. A relação entre uma asserção inicial e uma asserção final de um trecho de código captura a essência da semântica do código [5].

De acordo com o que foi dito no parágrafo anterior, define-se abaixo a equivalência entre dois trechos de código:

Definição de Equivalência de Programas:

Dado um trecho de código **C** que implementa um algoritmo **A** e outro trecho de código **C’** que define o mesmo algoritmo ligeiramente modificado **A’**, se **C** e **C’** produzirem a mesma asserção final, **C** será semanticamente equivalente a **C’** desde que as asserções iniciais também sejam as mesmas.

As provas de que as asserções são verdadeiras não dependem de uma arquitetura de máquina particular no caso de linguagens de alto nível. Pelo

contrário, elas dependem das relações entre os valores das variáveis. Embora os valores individuais das variáveis mudem ao longo da execução do programa, certas relações existentes entre as variáveis continuam as mesmas. Estas relações invariantes formam as asserções que expressam a semântica do programa [5].

A semântica axiomática tem dois pontos iniciais: um artigo escrito por Robert Floyd [25] e uma abordagem um tanto diferente introduzida por C. A. R. Hoare [27], sendo que essa última é a utilizada neste trabalho. Tal semântica é comumente associada à prova de correção de programas usando puramente análise estática do texto do programa. Isto é possível porque programação é uma ciência exata, ou seja, todas as propriedades de um programa e todas as conseqüências de sua execução em qualquer ambiente podem, em princípio, ser encontradas utilizando-se o próprio texto do programa através de pura dedução e a semântica da linguagem de programação usada [5, 27].

Esta abordagem estática faz um contraste claro com uma abordagem dinâmica, a qual verifica um programa com base na modificação das variáveis ao longo da execução do programa. Outra aplicação dessa semântica é considerar as asserções como especificações de programas, através das quais um código de programa pode ser derivado [5].

A semântica axiomática possui algumas limitações [5, 27]:

- Não são permitidos efeitos colaterais em expressões e na avaliação de condições;
- O comando **goto** (isto é, uma instrução de salto) é difícil de ser especificado;
- Não é permitido *aliasing*; e
- Regras de escopo são difíceis de se descrever, a não ser que todos os nomes de identificadores sejam únicos.

Com relação à primeira restrição, ao se provar propriedades de programas expressas em uma linguagem permitindo efeitos colaterais, é necessário provar a ausência delas em cada caso antes de se aplicar a técnica de prova adequada. Embora seja considerada uma limitação, pode-se argumentar que é duvidoso dizer que o uso da notação funcional para chamar procedimentos com efeitos colaterais seja realmente uma vantagem, considerando-se que o principal propósito de uma linguagem de alto nível é auxiliar na construção e verificação de programas corretos [27]. E o mesmo pode-se dizer com relação ao comando **goto**.

Apesar destas limitações, a semântica axiomática é uma tecnologia atrativa por causa de seu efeito potencial em desenvolvimento de *software* [5]:

- O desenvolvimento de algoritmos “livres de *bugs*” que foram provados corretos; e
- A geração automática de código de programa baseado em especificações.

3.2 Correção de Programas

Quando se prova a correção de um programa, a lógica predicativa de primeira ordem com igualdade é a utilizada. Nela, as variáveis individuais correspondem a variáveis do programa e os símbolos funcionais incluem todas as operações que ocorrem nas expressões. Portanto, as expressões aritméticas podem ser vistas como termos da lógica predicativa cujos valores são determinados pelos valores correntes das variáveis individuais da linguagem lógica [5].

Uma asserção é uma fórmula lógica construída usando variáveis individuais, constantes individuais e símbolos funcionais no cálculo predicativo. Quando cada variável em uma asserção recebe um valor (determinado pelo valor da variável do programa correspondente), a asserção se torna verdadeira (**true**) ou falsa (**false**) utilizando uma interpretação padrão das constantes e das funções na linguagem lógica [5].

Tipicamente, asserções consistem em uma conjunção de proposições elementares descrevendo as propriedades lógicas das variáveis do programa, seja afirmando que uma variável recebe valores de um determinado conjunto ou definindo uma relação entre variáveis. Porém, todos os conectivos booleanos além do **and** podem ser utilizados para expressar diferentes propriedades. Em algumas instâncias, as asserções não usam apenas o que é expressável em lógica booleana, utilizando também os quantificadores existências e universais [5]. Um exemplo típico em que os quantificadores geralmente são utilizados é quando se deseja expressar propriedades sobre vetores.

Para os propósitos da semântica axiomática, um programa se reduz ao significado de um comando, que na sintaxe abstrata inclui uma seqüência de comandos. A semântica de um programa é descrita colocando-se asserções que são sempre válidas quando o controle do programa atinge os pontos das mesmas. No caso do cálculo de Hoare, o significado ou correção de um comando (um programa) é descrito colocando-se uma asserção, chamada de

pré-condição, antes do mesmo e uma outra, chamada de pós-condição, depois do comando, da seguinte forma:

$$\{ P \} C \{ Q \},$$

onde **P** é a pré-condição, **C** é o comando e **Q** é a pós-condição [5].

Logo, pode-se dizer que o significado do comando **C** compreende um par ordenado **<P, Q>**, chamado de especificação de **C**. Diz-se que o comando **C** é correto com respeito a sua especificação dada pela pré-condição e pela pós-condição se o comando é executado com valores que fazem a pré-condição verdadeira, se o mesmo pára e se os valores resultantes fazem a pós-condição verdadeira. Estendendo esta noção a um programa inteiro, proporciona-se um significado da correção do programa e uma semântica a programas em uma dada linguagem [5].

Um programa é parcialmente correto com respeito a uma pré-condição e uma pós-condição se o programa for executado com valores que tornam a pré-condição verdadeira e os valores resultantes fizerem a pós-condição verdadeira sempre que o programa parar (se isso acontecer). Se também for possível mostrar que o programa termina quando executado com todos os valores que satisfazem a pré-condição, o programa é dito (totalmente) correto [5]. Ou seja:

$$\text{Correção parcial} = \text{pré-condição} \wedge \text{terminação} \rightarrow \text{pós-condição}$$
$$\text{Correção total} = \text{Correção parcial} \wedge \text{terminação}$$

O foco deste trabalho é na correção parcial de programas, mesmo porque não se pode automatizar a prova de que um programa é totalmente correto (se isso pudesse ser feito, o problema da parada estaria resolvido). Portanto, as regras e axiomas do cálculo de Hoare mostrados a seguir não dão base para a prova de que um programa termina com sucesso. Além de um *loop* infinito, um programa pode não terminar devido a uma violação de um limite de implementação, como por exemplo, o intervalo dos operandos numéricos, o espaço de armazenamento ou um limite de tempo do sistema operacional [27].

Com a exceção de provas da inexistência de *loops* infinitos, é provavelmente melhor provar a correção parcial de um programa e confiar em uma implementação para dar um alerta se a execução precisar ser abandonada devido ao resultado de um limite estourado [5, 27].

O objetivo da semântica axiomática é prover axiomas e regras de prova que capturem o significado pretendido de cada comando em uma linguagem de programação. Essas regras são construídas de modo que uma especificação para um dado comando possa ser deduzida, tendo como conseqüência a correção parcial do mesmo relativa à especificação. Tal dedução consiste de uma seqüência finita de asserções, sendo cada uma delas uma pré-condição, um axioma associado a um comando de programa, ou uma regra de inferência cujas premissas já foram estabelecidas [5].

Ainda com relação a especificações para programas, sabe-se que é difícil descrever especificações precisas de algoritmos, ou seja, um par **<P, Q>** que descreve corretamente o comportamento esperado de um determinado trecho de código. Assim, ao utilizar o cálculo de Hoare para PCC, tem-se o problema da dificuldade de se cobrir tudo o que se entende por segurança, como esperado. Obviamente, uma prova de correção de um programa só é útil ao programador se a mesma for feita utilizando-se especificações corretas [5].

Porém, se as intenções do usuário puderem ser descritas rigorosamente através de asserções sobre os valores das variáveis ao final (ou em pontos intermediários) da execução de um programa, então as regras e axiomas descritos abaixo podem ser usados para provar a correção parcial de um programa, desde que a implementação da linguagem de programação esteja em conformidade com as mesmas. Este fato também pode ser estabelecido utilizando-se dedução, através um conjunto de axiomas que descrevam as propriedades lógicas dos circuitos do *hardware* [27].

Quando a correção de um programa, seu compilador e o *hardware* do computador tiverem sido estabelecidas com rigor matemático, será possível ter grande confiança nos resultados do programa e predizer suas propriedades com a confiança limitada apenas pela eletrônica [27].

3.3
Regras do Cálculo de Hoare

As regras aqui explicadas são reproduzidas do cálculo descrito em [5].

O primeiro comando a ser descrito é o que não faz nada, chamado aqui de **skip**. Como ele não altera nenhuma variável, sua pré-condição deve ser igual a pós-condição, ou seja, **P** = **Q**. Portanto, sua regra é a seguinte:

$$\frac{}{\{ P \}\ \texttt{skip}\ \{ P \},} \quad (1)$$

a qual é um teorema.

Já para o comando de atribuição, assume-se que não há expressões com efeitos colaterais. Portanto, só a variável que recebe o novo valor é alterada. Neste comando, substitui-se a expressão do lado direito da atribuição por toda ocorrência da variável que recebe o novo valor na pós-condição, derivando, então, a pré-condição, seguindo o princípio de que tudo o que for verdadeiro para essa variável depois da atribuição deverá ser verdadeiro sobre a expressão antes da atribuição .

Dada uma atribuição da forma **V := E** e uma pós-condição **P**, a notação **P(V / E)** é utilizada para indicar a substituição consistente de **E** no lugar de cada ocorrência livre de **V** em **P**. Esta notação permite que se dê uma definição axiomática ao comando da atribuição como [27, 29]:

$$\frac{}{\{\ P(\ V\ /\ E\ )\ \}\ V\ :=\ E\ \{\ P\ \},} \quad (2)$$

a qual também é um teorema.

A operação de substituição deve ser definida com cuidado, já que as fórmulas do cálculo predicativo permitem a ocorrência de variáveis tanto livres quanto ligadas. Para se evitar problemas, deve-se usar a técnica de substituição de variáveis utilizada no λ-calculus (por exemplo, substituição explícita) [5, 11].

Se as asserções forem vistas como predicados, ou seja, expressões booleanas com um parâmetro, o axioma da substituição pode ser expressado da seguinte forma:

$$\frac{}{\{\ P(\ E\ )\ \}\ V\ :=\ E\ \{\ P(\ V\ )\ \}} \quad (2')$$

O axioma que especifica **V := E** define, em essência, que se é possível provar uma propriedade sobre **E** antes da atribuição, a mesma vale para **V** depois da atribuição.

Para as especificações axiomáticas dos demais comandos, introduz-se regras de inferência que possuem a seguinte forma:

$$\frac{H_1,\ H_2,\ \ldots,\ H_n}{C}$$

Tal notação pode ser interpretada da seguinte forma: se $\mathbf{H_1, H_2, \ldots, H_n}$ podem ser provadas, então é possível concluir que **C** também pode ser provada.

Um programa, em geral, consiste de uma seqüência de comandos que são executados um após o outro. Logo, a próxima regra é a da seqüência de dois grupos de comandos [27, 29]:

$$\frac{\{ P \}\ C_1\ \{ Q \}\quad \{ Q \}\ C_2\ \{ R \}}{\{ P \}\ C_1 ; C_2\ \{ R \}} \qquad (3)$$

A regra diz que, se, começando com a pré-condição **P**, pode-se provar **Q** após a execução de $\mathbf{C_1}$, e, começando com **Q**, pode-se provar **R** depois de $\mathbf{C_2}$ então pode-se concluir que, começando com a pré-condição **P**, **R** é verdade depois de se executar $\mathbf{C_1 ; C_2}$. Observe-se que a asserção do meio **Q** é "esquecida" na conclusão.

Quanto ao comando **if-then-else**, há uma escolha entre duas alternativas. Dois caminhos emergem do mesmo; logo, se for possível provar que cada um deles é correto dado o valor apropriado da expressão booleana **B**, o comando inteiro é correto. Eis aqui a regra:

$$\frac{\{ P \wedge B \}\ C_1\ \{ Q \}\quad \{ P \wedge \neg B \}\ C_2\ \{ Q \}}{\{ P \}\ \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi } \{ Q \}} \qquad (4)$$

Note-se que a expressão booleana **B** é usada como parte das asserções nas premissas da regra. A definição axiomática para o comando **if-then** é semelhante, exceto que para o ramo do **false** é necessário apenas mostrar que a asserção final pode ser derivada diretamente da asserção inicial **P** quando a condição **B** é falsa. Adiante será explicado que é fácil mostrar que a regra abaixo se reduz a anterior se o comando $\mathbf{C_2}$ for substituído pelo **skip**:

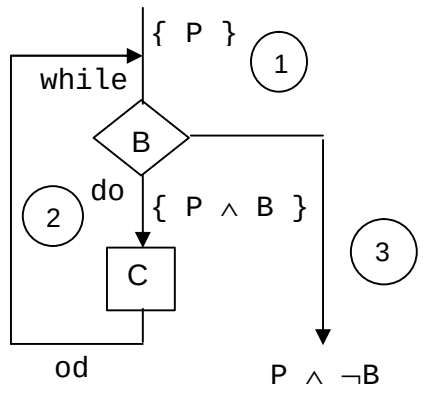
$$\frac{\{ P \wedge B \}\ C\ \{ Q \}\quad P \wedge \neg B \rightarrow Q}{\{ P \}\ \text{if } B \text{ then } C \text{ fi } \{ Q \}} \qquad (5)$$

Já para o comando **while**, a regra é a seguinte [27, 29]:

$$\frac{\{ P \wedge B \}\ C\ \{ P \}}{\{ P \}\ \text{while } B \text{ do } C \text{ od } \{ P \wedge \neg B \}} \qquad (6)$$

Nesta regra, **P** é chamado de invariante do *loop*. Esta asserção captura a essência do **while**: ela deve ser verdadeira inicialmente, ser preservada depois da execução do corpo do *loop* e, combinada com a condição de saída, implicar na asserção que vem depois do *loop*. Obviamente, **P** será verdadeira depois de

qualquer número de iterações da seqüência de comandos **C** (mesmo que não haja nenhuma iteração). Maiores detalhes sobre invariantes de *loops* serão dados na próxima seção. A figura a seguir ilustra a situação aqui descrita [27]:



1. **Inicialização**: Mostra que o invariante do *loop* é válido inicialmente.
2. **Preservação**: Verifica que o invariante do *loop* vale toda vez que o corpo do *loop* é executado.
3. **Finalização**: Prova que o invariante do *loop* e a condição de saída implicam na asserção final.

Figura 2 - Estrutura da regra do **while**

O propósito do passo de preservação é verificar a premissa para a regra do **while** mostrada anteriormente. Já os passos de inicialização e finalização são usados para amarrar o *loop* **while** ao código em volta do mesmo e às asserções.

Além das regras específicas para cada comando, há algumas regras gerais que se aplicam a todos eles. Algumas vezes, o resultado provado é mais forte do que o desejado. De acordo com a definição de força dada em [49], tem-se que:

Definição de força:

Se **A** ® **B** mas Ø **(B** ® **A)**, então diz-se que **A** é mais forte do que **B** ou **B** é mais fraco do que **A**.

Então, é possível enfraquecer a pós-condição buscando uma outra mais fraca [29]:

$$\frac{\{ P \}\ C\ \{ Q \} \quad Q \rightarrow R}{\{ P \}\ C\ \{ R \}} \qquad (7)$$

Isso é possível porque se a execução de um trecho de código **C** assegura a verdade da asserção **Q**, então ele também garante a validade de qualquer asserção lógica implicada por **Q** [27].

Já em outras ocasiões, a pré-condição dada é mais forte do que a necessária para completar a prova [29]:

$$\frac{P \rightarrow Q \quad \{ Q \}\ C\ \{ R \}}{\{ P \}\ C\ \{ R \}} \qquad (8)$$

Nesse caso, se **Q** é uma pré-condição de um trecho de código **C** para produzir o resultado **R**, então também é uma pré-condição qualquer outra asserção que implica **Q** logicamente [27].

A figura a seguir mostra que ao aplicar a regra do enfraquecimento da pós-condição (7) à regra do **if-then-else** (4) com o **skip** como o comando do ramo **else**, obtém-se as mesmas premissas e a mesma conclusão da regra **if-then** (5). Portanto, aplicando-se a noção de equivalência definida na seção 3.1, obtém-se a equivalência entre os comandos **if-then** e **if-then-else** com o **skip** no ramo do **else**.

$$\frac{\dfrac{\{ P \wedge \neg B \}\ \text{skip}\ \{ P \wedge \neg B \} \quad P \wedge \neg B \rightarrow Q}{\{ P \wedge B \}\ C\ \{ Q \} \qquad \{ P \wedge \neg B \}\ \text{skip}\ \{ Q \}}}{\{ P \}\ \text{if B then C else skip fi}\ \{ Q \}}$$

Figura 3 - Redução da regra do **if-then** a do **if-then-else**

As regras acima descritas são apenas as que foram implementadas no corretor de programas que será descrito mais adiante neste trabalho. O conjunto delas pode ser facilmente estendido para uma linguagem de programação com mais comandos e mais complexa (contendo funções, blocos, recursão e outras construções sintáticas).

É possível, ainda, relacionar asserções pelas relações lógicas da conjunção e da disjunção:

$$\frac{\{ P_1 \} C \{ Q_1 \} \quad \{ P_2 \} C \{ Q_2 \}}{\{ P_1 \wedge P_2 \} C \{ Q_1 \wedge Q_2 \}} \qquad (9)$$

$$\frac{\{ P_1 \} C \{ Q_1 \} \quad \{ P_2 \} C \{ Q_2 \}}{\{ P_1 \vee P_2 \} C \{ Q_1 \vee Q_2 \}} \qquad (10)$$

É importante enfatizar que nas regras do **if-then**, do fortalecimento e do enfraquecimento, uma sentença em lógica de primeira ordem deve ser demonstrada, ou seja, ser um teorema em uma dada teoria (neste caso, na dos tipos de dados envolvidos). Caso contrário, a prova que está sendo construída não está correta. Logo, em uma prova de correção de um programa, várias sentenças devem ser demonstradas para que a prova seja correta [2].

Infelizmente, demonstrar sentenças em lógica predicativa é indecidível. Portanto, o ideal é que se tente provar apenas as sentenças demonstráveis, pois sabe-se que há uma prova para elas. Para evitar a tentativa de demonstrações de sentenças inválidas durante a construção da prova de correção para programas que obedecem a sua especificação dada pela sua pré-condição e pela sua pós-condição, neste trabalho são estudadas heurísticas que procurem gerar provas apenas com sentenças demonstráveis em sua construção. Isto é, são procuradas heurísticas que dificultem ao máximo a geração de provas com sentenças inválidas.

Por fim, para que uma prova tenha significado, as pós-condições que nela aparecem não podem possuir contradições, pois isso implicaria que se pode provar qualquer propriedade a partir do trecho onde a contradição apareceu, o que não é algo desejado, como se pode ver pela dedução abaixo [2]:

$$\frac{\{ P \} C \{ \bot \} \quad \bot \rightarrow Q,}{\{ P \} C \{ Q \}}$$

isso acontece porque ^ ® **Q** é sempre verdade, independentemente de **Q**. Porém, saber se uma asserção é uma contradição (uma sentença que é sempre falsa) também é indecidível.

Logo, a não ser no caso do **if** (que será explicado abaixo), para que não apareçam contradições nas pós-condições, as provas são geradas de tal maneira que isso nunca aconteça. Ainda, para que uma contradição não apareça na pós-condição, deve-se assumir que ela não apareça em nenhuma pré-condição, pois, se assim fosse, ela poderia ser propagada para as pós-condições. Ou seja, supõe-se que o usuário não entra a pré-condição de um

programa contendo uma contradição. Afinal, se ele o fizer, vai ser capaz de obter o que quiser como pós-condição.

Todavia, ao corrigir o comando **if**, pode ser necessário usar o absurdo como pós-condição, no caso da "não" execução de um dos ramos na hora de se construir a prova. Isso acontece quando a pré-condição **P** possui informações que permitem avaliar o valor do teste **B**. Então, um entre **P** Ù **B** e **P** Ù Ø**B** será falso, e pode ser desejado que essa contradição se propague por todo o ramo, como será mostrado no próximo capítulo.

3.4
Invariantes de *Loops*

Descobrir o invariante de um *loop* requer perspicácia, e fazê-lo pode ser um tanto complicado para programas não-triviais. Mais ainda, a construção de invariantes de *loops* para comandos **while** em um programa é o principal desafio quando se prova a correção de programas em linguagens imperativas [5, 18].

O invariante de um *loop* envolve uma relação entre variáveis que permanece sendo a mesma, não importando o número de vezes que o *loop* é executado. Ele também inclui a condição do **while**, modificada para incluir o caso de saída. Outras componentes dele envolvem as variáveis que mudam de valor com o resultado da execução do *loop* [5].

Apesar de não haver uma fórmula simples para resolver este problema, vários princípios gerais podem ajudar na análise da lógica do *loop* quando se tenta encontrar o seu invariante [5]:

- A construção de uma tabela de valores para as variáveis que mudam geralmente revela uma propriedade entre elas que não muda;
- A combinação do que já foi computado em algum estágio do *loop* com o que ainda tem que ser feito pode resultar em algum tipo de constante; e
- A expressão relacionada ao teste **B** do *loop* geralmente pode ser combinada com a asserção Ø**B** para produzir parte da pós-condição.

Provar que um programa é correto é um processo relativamente mecânico desde que os invariantes dos *loops* sejam conhecidos [5]. Porém, descobri-los é um desafio para qualquer tecnologia de provas de teoremas, geralmente precisando de intervenção do usuário [19, 22]. Embora haja heurísticas que permitam a descoberta de propriedades que não se alteram durante a execução

de um *loop*, nem sempre é possível descobrir o melhor invariante, sendo esse um processo indecidível.

É importante observar que, apesar de um invariante bem fraco valer para um *loop* (como por exemplo, **true**), ele pode não ser suficiente para provar a correção de um programa. Mais ainda, a força dos invariantes está ligada à força da pós-condição de um programa. Um programa com uma pós-condição forte exige um invariante que permita conclui-la.

Portanto, para programas em geral, isto representa o problema mais importante a ser resolvido, já que é o principal obstáculo na automatização da geração de provas. Por causa disso, em muitas implementações, o invariante de cada *loop* de um programa é simplesmente dado pelo usuário, como foi feito na abordagem utilizada nesta pesquisa [19, 22].

3.5
Sub-Rotinas e Lemas

O método mais rigoroso de se formular o propósito de uma sub-rotina, assim como as condições para seu uso adequado, é o de fazer asserções sobre os valores das variáveis antes e depois de sua execução. A prova de correção destas asserções pode, então, ser usada como um lema na prova de qualquer programa que chama a sub-rotina. Portanto, em um programa grande, a estrutura do todo pode ser claramente espelhada na estrutura de sua prova. Mais ainda, quando se torna necessário modificar o programa, sempre será válido substituir qualquer sub-rotina por outra que satisfaça o mesmo critério de corretude [27].

3.6
PCC Utilizando o Cálculo de Hoare

Como quem define a política de segurança de um programa é o consumidor de código, a pré-condição e a pós-condição devem ser dadas por ele, pois definem a especificação da segurança desejada. No caso de os invariantes dos *loops* serem fornecidos “manualmente”, eles devem ser dados pelo produtor porque o consumidor precisaria ver o código para defini-los (o que faria com que a técnica de PCC fosse modificada). Além disso, o invariante pode ser determinado automaticamente, embora isso seja difícil, como já foi dito na

seção 3.4. Além disso, o melhor invariante nem sempre pode ser encontrado, o que pode ser indesejável em muitas situações [2]. De qualquer maneira, os invariantes dados precisam ser tais que permitam provar a correção do programa de acordo com a pós-condição dada, pois se forem fracos demais, isso pode não ser possível.

Depois de definida a especificação, o produtor de código deve realizar a prova de correção de programas. Essa última deve ser enviada ao consumidor, juntamente com a demonstração de todas as sentenças que aparecem nela e o código.

Em geral, um programador não quer que os usuários de seu código tenham acesso ao código fonte, disponibilizando a eles apenas o código binário. Portanto, o código geralmente será enviado no formato de código de máquina. Ainda, como a prova de correção, no caso desta pesquisa, é feita sobre o código fonte, é possível que o usuário extraia o programa a partir da mesma. Logo, pode ser interessante que a prova seja codificada de tal forma que o consumidor não possa entendê-la. Porém, esse não é o intuito deste trabalho, pois deseja-se que o usuário veja a prova para que assim possa entender o funcionamento do programa.

Por fim, após obter a prova, o consumidor deve checá-la, o que é um processo fácil e rápido, pois é apenas uma verificação sintática sobre a estrutura da prova. Estando ela correta, o programa é seguro de ser executado.

É vantajoso utilizar cálculo de Hoare para linguagens imperativas para *proof-carrying code* pelos seguintes motivos:

- É possível olhar a prova e entender por que o programa garante a segurança (ou não) mais facilmente do que no caso binário;
- As propriedades que se podem checar são de mais alto nível em comparação com o caso binário;
- É fácil verificar que uma prova respeita as regras do cálculo de Hoare e possui somente sentenças demonstráveis, ou seja, com provas corretas para cada uma delas. No entanto, verificar a ausência de contradições nas pós-condições é um processo indecidível;
- Embora o uso de um provador de teoremas seja inadequado para uma prova que não possua todas as sentenças demonstráveis (ele pode não conseguir mostrar que uma sentença não é válida, podendo entrar em *loop* se for automático, pois a determinação da validade de uma sentença é um problema indecidível), é possível realizar a prova “manualmente”;

- Se um programa PCC for modificado, não pode acontecer de a prova resultante ser válida e não ser uma prova de segurança. Isso acontece porque o que garante a segurança são as pré e pós-condições, que continuarão sendo as mesmas em ambas versões do programa;
- Aparentemente, as provas não ficam tão grandes em relação as obtidas com cálculo de Hoare para código binário. Afinal, cada instrução de alto nível pode gerar muitas instruções de máquina, o que resulta em um código de máquina muito mais extenso do que o seu equivalente em alto nível;
- Como as provas são feitas através da utilização do código fonte, elas são independentes da arquitetura onde o programa será executado [5]. Portanto, o produtor de código não precisa gerar várias provas, uma para cada arquitetura diferente, como acontece no caso binário; e
- No caso binário, tem-se que tratar saltos, o que é difícil de ser especificado pela semântica axiomática. Portanto, isto torna a estrutura das provas para o caso binário mais complexa do que para o caso de alto-nível.

3.7 Provando Segurança Utilizando o Cálculo de Hoare

Nesta seção será dado um exemplo do uso do cálculo de Hoare para provar a segurança de um sistema operacional extensível, conforme descrito na seção 2.3.2.

Como dito anteriormente, as aplicações não devem violar os invariantes do sistema operacional. Ou seja, determinadas propriedades que valem antes da execução do programa devem continuar valendo após a mesma. Isso pode ser expressado da seguinte maneira, utilizando a notação do cálculo de Hoare:

```
{ P } código do programa a ser executado { P }
```

onde **P** é a propriedade invariante do núcleo do sistema operacional que deve valer antes e depois da execução do programa não-confiável. Agora, tendo a prova de correção deste trecho de código, tem-se a prova de que ele é seguro de ser executado.

Para tornar este exemplo mais concreto, seja **P = (x = a)**, onde **x** é uma variável de ambiente. Claramente, o primeiro programa da figura abaixo é seguro

de ser executado, enquanto o segundo deve ser rejeitado (note-se a sentença não-demonstrável **x = c ® a = c**):

$$\dfrac{x = a \rightarrow a = a \quad \{ a = a \}\ x := b\ \{ a = a \}}{\dfrac{\{ x = a \}\ x := b\ \{ a = a \} \qquad \{ a = a \}\ x := a\ \{ x = a \}}{\{ x = a \}\ x := b\ ;\ x := a\ \{ x = a \}}}$$

$$\dfrac{x = c \rightarrow a = c \quad \{ a = c \}\ x := b\ \{ a = c \}}{\dfrac{\{ x = a \}\ x := b\ \{ a = c \} \qquad \{ a = c \}\ x := c\ \{ x = a \}}{\{ x = a \}\ x := b\ ;\ x := c\ \{ x = a \}}}$$

Figura 4 - Verificação de propriedades de segurança