3 Semântica axiomática

3.1 A linguagem IMP

A base do nosso estudo de semântica axiomática (e de semântica denotacional) é a linguagem IMP, um núcleo pequeno de uma linguagem imperativa. Ela consiste de

- expressões aritméticas inteiras com operações aritméticas comuns $(+, -, \times)$;
- identificadores de variáveis;
- expressões booleanas com operações lógicas $(\wedge, \vee, \neg)$ e comparações entre expressões aritméticas $(<, \leq, >, \geq, =)$;
- quatro comandos simples: atribuição, seqüência, condicional e o comando `while`. Além desses, um comando `skip` que não faz nada, mas que simplifica a sintaxe e a semântica (por exemplo, com `skip` precisamos só uma forma de condicional - `if then else`).

Este núcleo *não* contém

- divisão de números inteiros (/);
- variáveis boleanas: a memória contém somente números inteiros;
- outros tipos de estruturas de dados (como vetores, registros, etc);
- apontadores,
- entrada e saída.

IMP – conjuntos sintáticos (119)

- Int – números inteiros : `1`,`42`,`4711`
- Bool – booleanos: `true`, `false`
- Ident – identificadores: `x`,`y`,`z`,...
- Aexp – expressões aritméticas: `1+2×3`, `x+5`
- Bexp – expressões booleanas: `x=y∧true`, `¬false`
- Com – comandos: `x := 5` e `if true then x:=1 else x:=0`

Meta-variáveis (120)

$$\begin{array}{lcl} \mathbf{n}, \mathbf{n}', \mathbf{n_1}, \mathbf{n_2} \ldots & \in & \text{Int} \\ \mathbf{t}, \mathbf{t_1}, \mathbf{t_2}, \ldots & \in & \text{Bool} \\ \mathbf{x}, y, z, \ldots & \in & \text{Ident} \\ \mathbf{b}, \mathbf{b}', \mathbf{b_1}, \mathbf{b_2}, \ldots & \in & \text{Bexp} \\ \mathbf{a}, \mathbf{a}', \mathbf{a_1}, \mathbf{a_2}, \ldots & \in & \text{Aexp} \\ \mathbf{c}, \mathbf{c}', \mathbf{c_1}, \mathbf{c_2}, \ldots & \in & \text{Com} \end{array}$$

IMP – sintaxe abstrata (121)

$$\begin{array}{llll} \text{Aexp} & \mathbf{a} & ::= & \mathbf{x} \mid \mathbf{n} \mid \mathbf{a_1} + \mathbf{a_2} \mid \mathbf{a_1} - \mathbf{a_2} \mid \mathbf{a_1} * \mathbf{a_2} \\ \text{Bexp} & \mathbf{b} & ::= & \mathbf{t} \mid \mathbf{a_1} = \mathbf{a_2} \mid \mathbf{a_1} < \mathbf{a_2} \mid \neg\mathbf{b} \mid \mathbf{b_1} \wedge \mathbf{b_2} \mid \mathbf{b_1} \vee \mathbf{b_2} \\ \text{Com} & \mathbf{c} & ::= & \texttt{skip} \\ & & \mid & \mathbf{x} \texttt{:=} \mathbf{a} \\ & & \mid & \mathbf{c_1} ; \mathbf{c_2} \\ & & \mid & \texttt{if } \mathbf{b} \texttt{ then } \mathbf{c_1} \texttt{ else } \mathbf{c_2} \\ & & \mid & \texttt{while } \mathbf{b} \texttt{ do } \mathbf{c} \end{array}$$

A gramática acima é abstrata mas ainda guarda elementos relacinados a uma gramática concreta. Ela pode ser redefinida de forma a ser mais abstrata ainda.

IMP – observações (122)

- Variáveis não são declaradas.
- As regras de tipos são, na verdade, embutidas na definição da sintaxe (por exemplo: `while 5 do x:= 2` não é sintaticamente correto)
- Expressões não têm efeitos colaterais

Exercício 3.1 Defina uma gramática abstrata para IMP com um grau de abstração maior do que a gramática abstrata dada. Construa árvores de sintaxe abstratas pertencentes ao conjunto definido por essa gramática.

Exercício 3.2 Implemente a gramática abstrata para IMP, definida no exercício acima, em OCAML e/ou Scala e construa valores em OCAML/objetos em Scala correpondentes a árvores de sintaxe abstrata definidas por essa gramática.

3.2 Pré e pós-condições

Uma especificação de um programa é o pre-requisito mais importante para uma implementação. Sem conhecer as entradas possíveis e sem saber as saídas desejadas uma implementação é correta só por acaso. Sem especificação

- é impossível saber quando a implementação está pronta;
- é impossível saber se a implementação está correta;
- a depuração do programa se torna muito mais complicada.

Se aceitamos a necessidade de uma especificação, ainda temos que escolher a forma da especificação. Temos ao menos três possibilidades:

- Uma especificação informal: a sintaxe e a semântica da especificação não são (totalmente) definidas. Um exemplo é uma especificação em linguagem natural.
- Uma especificação semi-formal: a sintaxe da especificação é definida, mas não a semântica. Um exemplo é uma especificação em UML.
- Uma especificação formal: ambas, sintaxe e semântica são definidas. Nós vamos estudar esse tipo de especificação. Um exemplo é o cálculo de Hoare que será descrito a seguir.

Verificação de Programas (123)

- Converter descrição informal D em uma especificação formal do problema como uma fórmula ϕ_D de alguma lógica.
- Escrever um programa P com o propósito de resolver o problema.
- Provar que o programa P satisfaz a fórmula ϕ_D.

Exemplo (124)

- Se a descrição informal D diz que o programa deve

  computar um número y cujo quadrado é menor do que a entrada x

- a especificaçao formal da propriedade que deve ser satisfeita pelo programa após a sua execução poderia então ser:

$$y.y < x$$

Exemplo (125)

- Mas e se a entrada for -4?
- Não existe um número cujo quadrado é menor do que um número negativo, logo não é possível escrever o programa de tal forma que ele funcione para todas as entradas possíveis.
- Podemos revisar a especificação informal para:

  Se a entrada x é maior do que zero computar um número cujo quadrado é menor do que x

Pré e pós-condições (126)

- Deve ser possível falarmos sobre o estado antes e depois do programa executar:

$$\{\phi\}P\{\psi\}$$

- Significado: *se o programa P executa em um estado que satisfaz ϕ então o estado depois da sua execução irá satisfazer ψ.*

Pré e pós-condições (127)

- Para o problema acima a especificação fica:

$$\{x > 0\} \quad P \quad \{y.y < x\}$$

- Observe que a especificaçao não diz nada sobre o que acontece caso a entrada seja menor ou igual a zero.
- O programador pode fazer o que ele quiser neste caso.

Pré e pós condições (128)

- A forma $\{\phi\}P\{\psi\}$ é chamada de *tripla de Hoare* (C. A. R. Hoare)
- φ é a *pré-condição* e
- ψ é chamada de *pós-condição.*

Pré e pós condições (129)

- Note que a tripla $\{x > 0\} \quad P \quad \{y.y < x\}$ pode ser satisfeita por diversos programas como esse por exemplo:

```
y := 0;
```

- assim como:

```
y:= 0;
while (y * y < x) (
          y := y + 1;
        )
y := y - 1;
```

3.3 Correção Parcial e Total

Correção parcial e total (130)

- dizemos que a tripla $\{\phi\}P\{\psi\}$ é *satisfeita sob correção parcial* se, para todos os estados que satisfazem ϕ, o estado resultante da execução de P satisfaz ψ, **caso o programa P termine**. Neste caso escrevemos:

$$\models_{par} \{\phi\}\ P\ \{\psi\}$$

Correção parcial só diz o que deve acontecer *se* o programa termina.

Correção Parcial e total (131)

- Dizemos que a tripla $[\phi]\ P[\psi]$ é *satisfeita sob correcção total* se, para todos os estados que satisfazem ϕ, **o programa termina** e o estado resultante da execução de P satisfaz ψ. Neste caso escrevemos;

$$\models_{tot} [\phi]\ P\ [\psi]$$

- Provar correção total usualmente envolve provar correção parcial e depois provar que o programa termina.

Exemplo (132)

- Seja Sucessor o seguinte programa

```
a := x + 1;
if (a - 1 =0) {
    y := 1;}
else {
    y := a;}
```

- O programa Sucessor satisfaz a especificação:

$$\{\top\}\ \mathsf{Sucessor}\ \{y = (x + 1)\}$$

- tanto sob correção parcial como sob correção total.

Exemplo de Programa IMP (133)

- Programa Fat para computar o fatorial de **x**:

```
y := 1;
z := 0;
while (z <> x) {
  z := z + 1;
  y := y * z;
}
```

Exemplo (134)

- O programa para calcular o fatorial termina somente se o valor inicial de x for não-negativo. Para correção total deveríamos poder provar

$$\models_{tot} [x \geq 0]\ \mathsf{Fat}\ [y = x!]$$

- Contudo a afirmação mais forte

$$\models_{tot} [\top]\ \mathsf{Fat}\ [y = x!]$$

  não pode ser provada!! (por que ?)

Exemplo (135)

- Note que, considerando correção parcial podemos provar tanto:

$$\models_{par} \{x \geq 0\}\ \mathsf{Fat}\ \{y = x!\}$$

como

$$\models_{par} \{\top\}\ \mathsf{Fat}\ \{y = x!\}$$

Cálculo para correção parcial (136)

$$\frac{\{\phi\}C_1\ \ \{\eta\} \qquad \{\eta\}\ c_2\ \{\psi\}}{\{\phi\}\ c_1; c_2\ \{\psi\}} \quad (\textsc{Comp})$$

$$\{\psi[a/x]\}\ x := a\ \{\psi\} \quad (\textsc{Atrib})$$

$$\frac{\{\phi \wedge b\}\ C_1\ \{\psi\} \qquad \{\phi \wedge \neg b\}\ C_2\ \{\psi\}}{\{\phi\}\ \texttt{if}\ \ b\ \texttt{then}\ \{c_1\}\ \texttt{else}\ \ \{c_2\}\ \{\psi\}} \quad (\textsc{If})$$

$$\frac{\{\psi \wedge b\}\ C\ \{\psi\}}{\{\psi\}\ \mathsf{while\ b}\ \{c\}\ \{\psi \wedge \neg b\}} \quad (\textsc{While})$$

$$\frac{\vdash \phi' \rightarrow \phi \qquad \{\phi\}\ c\ \{\psi\} \qquad \vdash \psi \rightarrow \psi'}{\{\phi'\}\ c\ \{\psi'\}} \quad (\textsc{Implic})$$

Tableuax (137)

- As regras do slide anterior constituem a semântica axiomática da linguagem IMP,
- Uma forma mais prática de utilizar as regras é através do método de tableux visto a seguir

Tableaux (138)

- Podemos pensar em programas como uma seqüência

$$
\begin{array}{c}
c_1; \\
c_2; \\
\vdots \\
c_n
\end{array}
$$

- onde cada c_i é um comando da linguagem (atribuições, ifs e whiles).

Tableaux (139)

- Para provar $\vdash_{par} \{\phi_0\}\;\; c_1; c_2; \ldots c_n \;\; \{\phi_n\}$ encontrar fórmulas $\phi_1, \phi_2, \ldots \phi_{n-1}$ tais que

$$
\begin{array}{l}
\quad\{\phi_0\} \\
c_1; \\
\quad\{\phi_1\} \\
c_2; \\
\quad\{\phi_2\} \\
\quad\vdots \\
\{\phi_{n-1}\} \\
c_n; \\
\quad\{\phi_n\}
\end{array}
$$

Tableaux (140)

- Essas fórmulas são *condições intermediárias* que devem ser verdadeiras no ponto em que são colocadas.
- Para obtê-las, começamos com ϕ_n e, usando c_n, tentar obter ϕ_{n-1} e assim por diante.
- Cada fórmula intermediária a ser encontrada deve ser a *mais fraca* possível, ou seja, ela deve ser exatamente o suficiente para, depois da execução do comando, fazer com que a pós-condição seja satisfeita.
- A seguir será examinado como a pré-condição mais fraca é obtida para cada comando:

Atribuição (141)

- O axioma da atribuição é facilmente adaptado para trabalhar com *tableaux*

$$\begin{array}{lll} & \{\psi\{a/x\}\} & \\ x := a & & \\ & \{\psi\} & \text{Atribuição} \end{array}$$

Implicação (142)

- Essa regra, na forma de *tableaux*, permite que duas fórmulas intermediárias sejam escritas uma após a outra sem comando entre elas.

  Eis a prova de que $\vdash_{par} \{y = 5\} \;\; x := y + 1 \;\; \{x = 6\}$

$$\begin{array}{lll} & \{y = 5\} & \\ & \{y + 1 = 6\} & \text{Implicação} \\ \texttt{x := y +1;} & & \\ & \{x = 6\} & \text{Atribuição} \end{array}$$

Tableuax (143)

- A prova é construída de baixo para cima.
- Começamos com $\{x = 6\}$ e, usando o axioma da atribuição, substituimos todas as ocorrências de x por $y + 1$, resultando em $\{y + 1 = 6\}$.
- Depois, comparamos com a pré-condição que foi dada inicialmente.
- A pré-condição dada $\{y = 5\}$ implica $\{y + 1 = 6\}$

Exemplo (144)

- Prova de $\vdash_{par} \{y < 3\} \;\; y := y + 1 \;\; \{y < 4\}$

$$\begin{array}{lll} & \{y < 3\} & \\ & \{y + 1 < 4\} & \text{Implicação} \\ \mathsf{y := y + 1;} & & \\ & \{y < 4\} & \text{Atribuição} \end{array}$$

- E novamente, $y < 3$ implica $y + 1 < 4$.

Exemplo (145)

- Prova de que $\vdash_{par} \{\top\}$ `z := x; z := z + y; u := z` $\{u = x + y\}$.

$$
\begin{array}{lll}
 & \{\top\} & \\
 & \{x + y = x + y\} & \textsf{Implicação} \\
\texttt{z} := \texttt{x}; & & \\
 & \{z + y = x + y\} & \textsf{Atribuição} \\
\texttt{z} := \texttt{z} + \texttt{y}; & & \\
 & \{z = x + y\} & \textsf{Atribuição} \\
\texttt{u} := \texttt{z}; & & \\
 & \{u = x + y\} & \textsf{Atribuição}
\end{array}
$$

- E, obviamente, $\top \to (x + y = x + y)$ é verdadeiro.

Exercícios (146)

- **Exercício:** Use as regras Atribuição e Implicação para mostrar que:
  1. $\vdash_{par} \{x > 0\}$ `y := x + 1` $\{y > 1\}$
  2. $\vdash_{par} \{\top\}$ `y := x; y := x + x + y` $\{y = 3.x\}$
  3. $\vdash_{par} \{x > 1\}$ `a:=1; y:=x; y:=y-a` $\{y > 0 \wedge x > y\}$

Condicional (147)

- Queremos obter a pré-condição mais fraca ϕ tal que

$$\{\phi\} \;\; \texttt{if b thenc}_1\texttt{else c}_2 \;\; \{\psi\}$$

- A fórmula ϕ pode ser obtida da seguinte forma:
  1. "empurre"ψ para cima de c_1, resultando em ϕ_1
  2. "empurre"ψ para cima de c_2, resultando em ϕ_2
  3. ϕ passa a ser $(b \to \phi_1) \wedge (\neg b \to \phi_2)$

Exemplo (148)

- Considere o programa Sucessor:

```
a := x + 1;
if (a - 1 = 0) {
    y :=1;
} else {
    y := a;
}
```

- Queremos provar que $\vdash_{par} \{\top\}$ Sucessor $\{y := x + 1\}$.

Exemplo (149)

```
     {⊤}
     {(x+1−1 = 0 → 1 = x+1) ∧ (¬(x+1−1 = 0) → x+1 = x+1)}   Impl.
a := x + 1;
     {(a−1 = 0 → 1 = x+1) ∧ (¬(a−1 = 0) → a = x+1)}   Atr.
if (a − 1 = 0){
     {1 = x+1}   If
  y = 1;
     {y = x+1}   Atribuição
} else{
     {a = x+1}   If
  y = a;
     {y = x+1}}   Atribuição
     {y = x+1}   If
```

Exemplo (150)

- Ainda resta provar

$$\top \to ((x+1-1=0 \to 1=x+1) \wedge (\neg(x+1-1=0) \to x+1=x+1))$$

- que obviamente é verdadeiro.

Exercícios (151)

- Use as regras vistas até o momento e prove o seguinte:
  $\vdash_{par} \{\top\}$ `P` $\{z = min(x,y)\}$, onde $min(x,y)$ é o menor número entre x e y e `P` é o seguinte programa:

```
if (x > y)  {z := y} else  {z := x}
```

While (152)

- A regra para o comando `while` é a seguinte:

$$\frac{\{\eta \wedge b\}\ c\ \{\eta\}}{\{\eta\}\ \mathsf{while}\ b\ \ c\ \{\eta \wedge \neg b\}} \qquad (\textsc{While})$$

- a fórmula η é chamada de *invariante* do laço.

While (153)

- Isso significa que, contanto que b seja verdadeiro, se η é verdadeiro antes de começar o comando c, e c termina, então η também é verdadeiro no final.
- Isso é expresso pela premissa $\{\eta \wedge b\}\ \ c\ \ \{\eta\}$.

While (154)

- `while` inicia em estado que satisfaz η.
- se b é falso já de início, c não é executado e nada afeta o valor-verdade de η e o `while` termina com $\{\eta \wedge \neg b\}$ verdadeiro.
- Se b é verdadeiro quando o `while` inicia, c é executado. Pela premissa da regra, η é verdadeiro no fim de c:
- se b é agora falso, a execução para com $\eta \wedge \neg b$ verdadeiro
- se b é verdadeiro, c é executado novamente e η é novamente *reestabelecida* (não importa quantas vêzes c é executado η será *reestabelecido* ao final de cada iteração).

While (155)

- A regra, tal como é formulada, nos permite provar coisas da forma

  $$\{\eta\}\quad \texttt{while b}\quad c\quad \{\eta \wedge \neg b\}$$

  ou seja triplas nas quais a pós-condição é igual a pré-condição mais $\neg b$. Mas, em geral, temos que provar triplas da forma

  $$\{\phi\}\quad \texttt{while b c}\quad \{\psi\}$$

While (156)

- Como usar a regra `While` nesses casos?
  1. *descubra* uma fórmula η que, espera-se, será um invariante apropriado
  2. prove que $\vdash \eta \wedge \neg b \rightarrow \psi$. Isso prova que η é forte o suficiente para implicar a pós-condição.
  3. "empurre"η para cima via c (isso implica usar as outras regras já vista). Uma fórmula η' "aparecerá"em cima.
  4. Prove que $\eta \wedge b \rightarrow \eta'$ (isso conclui a prova de que η é um invariante apropriado).
  5. Escreva η acima do `while`.

Invariante (157)

- **Descobrir um invariante** η apropriado requer criatividade e não pode ser automatizado.
- Para encontrá-lo pode ser útil construir um *trace* da execução do programa.

Exemplo (158)

- Considerando o programa Fat abaixo provar

$$\{\top\} \quad \mathsf{Fat} \quad \{y = x!\}$$

```
y:=1;
z:=0;
while (z <> x) {
    z := z + 1;
    y := y * z;
}
```

Exemplo (159)

- Eis um trace para Fat começando com $x = 6$:

iteração	z	y	b
0	0	1	true
1	1	1	true
2	2	2	true
3	3	6	true
4	4	24	true
5	5	120	true
6	6	720	false

Exemplo (160)

```
   {T}
    {1=0!}   (Implic.)
y = 1;
    {y=0!}   (Atrib.)
z = 0;
    {y = z!} (Atrib)
while (z <> x) {
    {y=z! &  z <> x}      Invar. e B.
    {y.(z+1) = (z+1)!}     (Implic.)
   z = z+1;
    {y.z = z!} (Atrib)
   y =y*z;
    {y=z!}   (Atrib)
}   {y=z! & ~ (z <> x)} (While)
    {y=x!} (Implic)
```

Variáveis lógicas e de programas (I) (161)

- variáveis das especificações vistas até então são variáveis de programas
- algumas vezes é necessário usar em especificações outras variáveis que não as do programas (variáveis lógicas)

Variáveis lógicas e de programas (II) (162)

- Outra versão do programa fatorial Fac2:

```
y := 1;
   while (x <> 0) {
      y := y * x;
      x := x - 1;
   }
```

- se o programa termina, o valor de x, no estado final, é diferente do valor que x possuía no estado inicial !!!
- A seguinte afirmação portanto não é verdadeira

$$\{x \geq 0\} \;\; \mathsf{Fac2} \;\; \{y = x!\}$$

Variáveis lógicas e de programas (III) (163)

- precisamos lembrar o valor inicial da variável x que é "destruído"pelo programa
- isso é feito com o uso de uma variável lógica x_0:

$$\{x = x_0 \;\wedge\; x \geq 0\} \quad \mathsf{Fac2} \;\; \{y = x_0!\}$$

- *para todo x_0, se $x = x_0$, $x \geq 0$ e o programa* Fac2 *termina, então o estado resultante satisfaz $y = x_0!$*

Variáveis lógicas e de programas (IV) (164)

- Considere o programa Sum

```
z := 0;
while (x > 0) {
   z := z + x;
   x := x   1;
}
```

- ele satisfaz a seguinte especificação:

$$\{x = x_0 \;\wedge\; x \geq 0\} \quad \mathsf{Sum} \;\; \{z = \frac{x_0(x_0+1)}{2}\}$$

Correção Total (I) (165)

- mesmas regras usadas para correção parcial
- única regra diferente é a regra para `while`
- identificar uma expressão inteira cujo valor diminui a cada repetição, mas que é sempre não negativo
- se existe tal expressão, chamada de *variante*, o `while` termina

Correção Total (II) (166)

- eis a regra para `while` para provar correção total

$$\frac{[\eta \ \wedge \ b \ \wedge \ 0 \leq e = v] \ \ c \ \ [\eta \ \wedge \ 0 \leq e < v]}{[\eta \ \wedge \ 0 \leq e] \ \ \texttt{while} \ b \ \{c\} \ \ [\eta \ \wedge \ \neg b]}$$

- nessa regra e é a expressão cujo valor diminui a cada repetição
- para provar correção total: provar correção parcial e provar terminação.

Correção Total (III) (167)

- provar $\vdash_{tot} [x \geq 0]$ Fac1 $[y = x!]$
- onde Fac1 é o seguinte programa

```
y := 1;
z := 0;
while (x <> z) {
  z := z + 1;
  y := y * z;
}
```

3.4 Propriedades

Nesta seção vamos formalizar a sintaxe e a semântica de asserções e também o significado de triplas de Hoare para correção parcial e total. Também serão discutidas (sem prova) propriedades importantes relativas a semântica axiomática.
Abaixo segue a gramática da linguagem lópgica usada na escrita das pré e pós-condições. A linguagem utilizada é lógica de primeira ordem.

Linguagem das Asserções (168)

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid n \mid e_1 + e_2 \mid e_1 * e_2 \\
\varphi & ::= & \texttt{true} \mid e_1 = e_2 \mid e_1 \leq e_2 \\
 & \mid & \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \forall v.\varphi \mid \exists v.\varphi
\end{array}
$$

- as expressões aritméticas de IMP estão incluidas na gramática para termos da linguagem das asserções
- as expressões booleanas de IMP também são fórmulas da lógica de predicados
- há duas categorias de variáveis: programa (x) e lógicas (v)
- quantificação só é feita sobre variáveis lógicas

Semântica das Asserções (169)

- Vimos a sintaxe da linguagem das asserções
- Precisamos definir precisamente o significado
- A notação $\sigma \models_{\mathcal{I}} \varphi$ quer dizer que φ é verdadeira em um dado estado σ dada uma valoração $\mathcal{I}$ para variáveis lógicas
- A semântica é definida indutivamente na estrutura sintática das fórmulas

Note que tanto a grámatica como a semântica abaixo tem o própósito de ilustrar como a linguagem de asserções pode ser definida. A linguagem pode ser evidentemente aumentada com novas constantes, símbolos predicativos e funcionais (com os respectivos acréscimos na gramática e nas cláusulas da definição da semântica). Em muitos exemplos e exercícios já aparecem algumas extensões.

Semântica das Asserções (II) (170)

$$
\begin{array}{ll}
\sigma \models_{\mathcal{I}} \texttt{true} & \textit{sempre} \\
\sigma \models_{\mathcal{I}} e_1 = e_2 & \textit{sse } T[\![e_1]\!]\mathcal{I}\sigma = T[\![e_2]\!]\mathcal{I}\sigma \\
\sigma \models_{\mathcal{I}} e_1 \leq e_2 & \textit{sse } T[\![e_1]\!]\mathcal{I}\sigma \leq T[\![e_2]\!]\mathcal{I}\sigma \\
\sigma \models_{\mathcal{I}} \varphi_1 \wedge \varphi_2 & \textit{sse } \sigma \models_{\mathcal{I}} \varphi_1 \textit{ e } \sigma \models_{\mathcal{I}} \varphi_2 \\
\sigma \models_{\mathcal{I}} \varphi_1 \vee \varphi_2 & \textit{sse } \sigma \models_{\mathcal{I}} \varphi_1 \textit{ ou } \sigma \models_{\mathcal{I}} \varphi_2 \\
\sigma \models_{\mathcal{I}} \forall v.\varphi & \textit{sse } \forall n \in \mathbb{Z}.\ \sigma \models_{\mathcal{I}[v \mapsto n]} \varphi \\
\sigma \models_{\mathcal{I}} \exists v.\varphi & \textit{sse } \exists n \in \mathbb{Z}.\ \sigma \models_{\mathcal{I}[v \mapsto n]} \varphi
\end{array}
$$

$$
\begin{array}{lcl}
T[\![v]\!]\mathcal{I}\sigma & = & \mathcal{I}(v) \\
T[\![x]\!]\mathcal{I}\sigma & = & \sigma(x) \\
T[\![n]\!]\mathcal{I}\sigma & = & n \\
T[\![e_1 + e_2]\!]\mathcal{I}\sigma & = & T[\![e_1]\!]\mathcal{I}\sigma + T[\![e_2]\!]\mathcal{I}\sigma \\
T[\![e_1 * e_2]\!]\mathcal{I}\sigma & = & T[\![e_1]\!]\mathcal{I}\sigma * T[\![e_2]\!]\mathcal{I}\sigma
\end{array}
$$

Semântica das Asserções (III) (171)

- Agora é possível definir formalmente a semântica de asserções de correção parcial:

$$\begin{array}{l} \models_{\mathcal{I}} \{\varphi\}\ c\{\psi\} \quad \textit{é verdadeiro sse} \\ \quad \forall \sigma.\sigma'.\ \ \sigma \models_{\mathcal{I}} \varphi \quad \rightarrow \quad ([\![c]\!]\sigma = \sigma' \rightarrow \sigma' \models_{\mathcal{I}} \psi) \end{array}$$

- e o significado de asserções de correção total

$$\begin{array}{l} \models [\varphi]\ c\ [\psi] \quad \textit{é verdadeiro sse} \\ \quad \models_{\mathcal{I}} \{\varphi\}\ c\ \{\psi\} \\ \qquad \wedge \\ \quad \forall \sigma.\exists \sigma'.\ \ \sigma \models_{\mathcal{I}} \varphi \quad \rightarrow \quad [\![c]\!]\sigma = \sigma' \end{array}$$

Semântica das Asserções (IV) (172)

- Também podemos fazer essa definição recorrendo a semântica operacional *big-step*:

$$\begin{array}{l} \models_{\mathcal{I}} \{\varphi\}\ c\{\psi\} \quad \textit{é verdadeiro sse} \\ \quad \forall \sigma.\sigma'.\ \ \sigma \models_{\mathcal{I}} \varphi \quad \rightarrow \quad nnc\sigma \Downarrow \sigma' \rightarrow \sigma' \models_{\mathcal{I}} \psi) \end{array}$$

- e o significado de asserções de correção total

$$\begin{array}{l} \models [\varphi]\ c\ [\psi] \quad \textit{é verdadeiro sse} \\ \quad \models_{\mathcal{I}} \{\varphi\}\ c\ \{\psi\} \\ \qquad \wedge \\ \quad \forall \sigma.\exists \sigma'.\ \ \sigma \models_{\mathcal{I}} \varphi \quad \rightarrow \quad c, \sigma \Downarrow \sigma' \end{array}$$

Propriedades (173)

- Temos uma linguagem para especificação de propriedades de programas (asserções ou triplas de Hoare)
- Sabemos quando uma asserção é verdadeira (semântica formal da linguagens das asserções)
- Temos um método simbólico para derivar asserções (método do *tableuax* para utilizar as regras)
- Propriedades de interesse sobre a Semântica Axiomática (Lógica de Hoare)?

Propriedades Semântica Axiomática (174)

- Segurança:
  $$Se \ \vdash \{\varphi\}\ c\ \{\psi\}\ então \ \models \{\varphi\}\ c\ \{\psi\}$$
  A lógica de Hoare é segura.
- Completeza:
  $$Se \ \models \{\varphi\}\ c\ \{\psi\}\ então \ \vdash \{\varphi\}\ c\ \{\psi\}$$
  A lógica de Hoare não é completa. Por que?

Computabilidade (175)

- o problema da verificação de programas é indecidível.
  - o problema da validade da lógica de predicados é indecidível
  - não há algoritmo para encontrar invariante/variante

Aplicações (I) (176)

- Hoare:

  A prática de provar programas parece solucionar três dos problemas mais preementes em software e programaçao, a saber: confiabilidade, documentaçao e compatibilidade. Contudo, prova de programas será difícil mesmo para programadores de alto calibre; e pode ser aplicável somente para programas simples
- Dijkstra:

  Teste de programas pode ser usado para mostrar a presença de erros, mas nunca para mostrar sua ausência

Aplicações (II) (177)

- O projeto de definir e provar tudo formalmente não foi (ainda) bem sucedido
- Provas não substituiram testes
- Outras aplicações de semântica axiomática:
  - documentação de programas e interfaces
  - orientação no projeto e codificação

– prova da correção de descrições de hardware

3.5 Exercícios

1. Usando as regras da Semântica Axiomática para IMP (e usando o método de *tableaux* verifique a se as seguintes afirmações são verdadeiras (leve em conta correção parcial e total):

   - $\{\top\}$ succ $\{y = x + 1\}$ onde succ é o programa IMP abaixo:

     ```
     a = x + 1;
     if (a - 1 = 0) then y:= 1 else y:=a;
     ```

   - $\{\top\}$ p $\{z = min(x, y)\}$, onde $min(x, y)$ é o menor número entre x e y e p é o seguinte programa:

     ```
     if (x > y)  then z := y else z := x
     ```

   - $\{x \geq 0\}$ fac1 $\{y = x!\}$ e onde fac1 é

     ```
     y := 1;
     z := 0;
     while (z <> x) do
        (z := z + 1;
         y := y * z)
     ```

   - $\{\top\}$ fat1 $\{y = x!\}$
   - $\{x \geq 0\}$ fac2 $\{y = x!\}$ onde fat2 é a seguinte versão de fat1

     ```
     y := 1;
     while (x <> 0) do
         (y := y * x;
          x := x - 1)
     ```

   - $\{x = x_0 \wedge x \geq 0\}$ fac2 $\{y = x_0!\}$
   - $\{x = x_0 \wedge x \geq 0\}$ sum $\{z = \frac{x_0(x_0+1)}{2}\}$ onde *sum* é

     ```
     z := 0;
     while (x > 0) do
         (z := z + x;
          x := x - 1)
     ```

   - $\{x \geq 0\}$ copy1 $\{x = y\}$ onde copy1 é

     ```
     a := x;
             y := 0;
     ```

```
while (a <> 0) do
   (y := y + 1;
    a := a - 1)
```

- $\{y \geq 0\}$ mult1 $\{z = x.y\}$ onde mult1 é

```
 a := 0;
z := 0;
while (a <> y) do
   (z := z + x;
    a := a + 1)
```

- $\{y = y_0 \wedge y \geq 0\}$ mult2 $\{z = x.y_0\}$ onde mult2 é

```
z := 0;
while (y <> 0) do
   (z := z + x;
    y := y - 1)
```

- $\{x \geq 0\}$ downfac $\{y = x!\}$ onde downfac é

```
a := x;
          y := 1;
          while (a > 0) do
             (y := y * a;
              a := a - 1)
```

2. Explique por que correção total implica correção parcial (ou seja por que para qualquer fórmula ϕ, φ, e programa P temos que $[\phi]\ P\ [\varphi]$ implica em $\{\phi\}\ P\ \{\varphi\}$)

3. O que significam as seguintes propriedades em relação ao conjunto de regras que constituem uma semântica axiomática de uma linguagem de programação:
   - segurança
   - completeza
   - decidibilidade

4. Explique por que o seguinte problema não é decidível:
   Dadas quaisquer fórmulas φ e ϕ da lógica de predicados e dado qualquer programa P de IMP , $\{\varphi\}\ P\ \{\phi\}$

5. Verifique se o programa abaixo está parcialmente correto em relação a sua especificação:
   $\vdash_P\ [x = m \wedge y = n \wedge z = 1]$ *prog* $[z = m^n]$ onde *prog* é o seguinte programa

```
while (y <> 0) {
   while (par(y)) {
      x:= x * x;
      y:= y / 2
```

```
    };
    z:= z * x;
    y:= y - 1
}
```

- Obs. 1: y/2 é inteiro inteiro que resulta da divisão de y por 2
- Obs. 2: deixe bem claro a conclusão final (ou seja se o programa está ou não correto)