Introdução à Programação em Python
Notebook 12 - Verificação de programas

Carlos Caleiro, Jaime Ramos
Dep. Matemática, IST - 2016

5 de Fevereiro de 2018

Verificação de programas imperativos

A verificação de programas é um tópico absolutamente crucial em inúmeras aplicações cujo correcto funcionamento seja crítico, uma questão cuja importância cresce à medida que mais e mais algoritmos se vão integrando nos processos e serviços que utilizamos diariamente. Infelizmente, é bem conhecido que o problema de verificar se um dado programa termina e calcula um certo resultado é *indecidível*, um tópico que é estudado em disciplinas de Teoria da Computação.

Este facto potencia a emergência de técnicas dedicadas à análise de programas, cada vez mais sofisticadas, que requerem um razoável conhecimento por parte de quem as utiliza, incluindo técnicas algébricas, de verificação de modelos, e de demonstração automática, algumas das quais são abordadas mais tarde em disciplinas especializadas.

Neste capítulo apresenta-se e aprende-se a utilizar um sistema lógico que permite, com a ajuda da inteligência humana, demonstrar a correcção de programas.

Por simplicidade, centrar-nos-emos na verificação de pequenos programas imperativos (sem ocorrências de `break` ou mesmo de `return`), sem ciclos `for` (o que necessitaria da análise simultânea do respectivo iterador), e usando apenas tipos imutáveis (o que simplifica o controlo dos efeitos colaterais, já que a invocação de métodos sobre tipos mutáveis obrigaria mais uma vez à sua análise simultânea).

No que se segue, para facilitar a apresentação, iremos subverter as habituais convenções notacionais da linguagem *Python*. Assim, escreveremos

$$\begin{array}{c} \mathtt{P}_1;\mathtt{P}_2;\ldots;\mathtt{P}_n \\ \text{ou} \\ (\mathtt{P}_1;\mathtt{P}_2;\ldots;\mathtt{P}_n) \end{array} \qquad \text{em vez de} \qquad \begin{array}{l} \mathtt{P}_1 \\ \mathtt{P}_2 \\ \vdots \\ \mathtt{P}_n \end{array}$$

bem como, por vezes, escreveremos

$$\begin{array}{c} \mathtt{if\ C}_1 : \mathtt{P}_1\ \mathtt{elif\ C}_2 : \mathtt{P}_2\ \ldots\ \mathtt{elif\ C}_{n-1} : \mathtt{P}_{n-1}\ \mathtt{else} : \mathtt{P}_n \\ \text{ou} \\ (\mathtt{if\ C}_1 : \mathtt{P}_1\ \mathtt{elif\ C}_2 : \mathtt{P}_2\ \ldots\ \mathtt{elif\ C}_{n-1} : \mathtt{P}_{n-1}\ \mathtt{else} : \mathtt{P}_n) \end{array} \qquad \text{em vez de} \qquad \begin{array}{l} \mathtt{if\ C}_1 : \\ \quad \mathtt{P}_1 \\ \mathtt{elif\ C}_2 : \\ \quad \mathtt{P}_2 \\ \quad \vdots \\ \mathtt{elif\ C}_{n-1} : \\ \quad \mathtt{P}_{n-1} \\ \mathtt{else} : \\ \quad \mathtt{P}_n \end{array}$$

ou mesmo

`while C : P`

ou em vez de `while C : P`

(`while C : P`)

por forma a evitar sintaxe gráfica do *Python*.

Numa composição alternativa, sempre que o caso `else` é omisso, assumimos que corresponde implicitamente a `else: pass`. Assumiremos também que todas as expressões envolvidas, nomedamente em atribuições e guardas de comandos `if` ou `while`, devolvem um valor no contexto em que são avaliadas.

Cálculo de Hoare

O sistema lógico que vamos introduzir, também conhecido por *lógica de Floyd-Hoare*, resulta do trabalho pioneiro de Tony Hoare e também de Robert Floyd no final dos anos 1960s. Nesta abordagem distinguem-se dois problemas distintos:

correcção parcial: determinar se um programa, quando executado a partir de um dado estado, leva (se terminar) a um estado final apropriado;

terminação: determinar se um programa termina, quando executado a partir de um dado estado.

Quando um programa é parcialmente correcto e termina, então diz-se que é *totalmente correcto*, ou que verifica a propriedade de **correcção total**.

Começaremos por estudar o fragmento do cálculo que aborda o problema da correcção parcial, e só depois o fragmento que aborda o problema da terminação.

Correcção parcial de programas

As asserções do cálculo de Hoare para a correcção parcial de programas incluem condições Booleanas (que são verdadeiras ou falsas em cada estado da computação) e os denominados *triplos de Hoare*.

Condições Booleanas

Evidentemente, as condições Booleanas são expressões lógicas envolvendo os nomes que definem o estado do programa que pretendemos analisar, que escrevemos usando a sintaxe do *Python* bem como alguma notação matemática usual. As seguintes expressões são condições Booleanas bem formadas:

1. `True`
2. `x>0`
3. `x >= 0 and type(x) is int`
4. `nat(x)`
5. $\sum_{n=0}^{k} n$ `== 55`
6. $\forall_{\mathtt{k}=3,5,7}$ `k%2 == 1`
7. `x>1` $\Rightarrow$ `x>0`
8. `x>0` $\Rightarrow$ `x>=1`

As condições 1–3 usam apenas a sintaxe do *Pyhton*. As restantes expressões, 4–8, usam notação comum que estende a sintaxe da linguagem, nomeadamente com o predicado `nat` (`nat(x)` é equivalente à condição 3), o símbolo de somatório ($\sum$), o quantificador universal ($\forall$), ou o conectivo lógico de *implicação* ($\Rightarrow$). Vale a pena recordar que o valor lógico de $A \Rightarrow B$ é o mesmo de `(not` A`) or` B, ou seja, $A \Rightarrow B$ é verdadeira desde que A seja falsa ou B seja verdadeira, ou equivalentemente, $A \Rightarrow B$ é falsa apenas se A é verdadeira e B é falsa.

Em cada estado, cada uma destas condições é verdadeira ou falsa. No entanto, as condições 1, 6 e 7 são *válidas*, isto é, são sempre verdadeiras, seja qual for o estado onde sejam avaliadas. Não é o caso das restantes.

No que se segue, vamos necessitar de utilizar a seguinte notação. Sejam C uma condição, $\mathtt{x}_1, \ldots, \mathtt{x}_n$ nomes, e $\mathtt{e}_1, \ldots, \mathtt{e}_n$ expressões. Então, vamos denotar por $(C)^{\mathtt{x}_1, \mathtt{x}_2, \cdots, \mathtt{x}_n}_{\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_n}$ a condição que resulta de substituir todas as ocorrências de cada $\mathtt{x}_i$ em C por $\mathtt{e}_i$ para cada $i \in \{1, \ldots, n\}$.

Por exemplo, $(\mathtt{x} > \mathtt{y})^{\mathtt{x}, \mathtt{y}}_{\mathtt{y+1}, \mathtt{0}}$ resulta na condição `y+1>0`.

Triplos de Hoare

Os triplos de Hoare são asserções da forma

$$\{C_1\}\ \mathtt{P}\ \{C_2\}$$

onde C_1 e C_2 são condições Booleanas, ditas respectivamente *pré-condição* e *pós-condição*, e `P` é um programa imperativo. Sendo asserções de correcção parcial, o significado de um triplo de Hoare é intuitivo: a asserção é válida se sempre que o programa `P` é executado a partir de um estado que satisfaz a pré-condição C_1, se a execução terminar levará a um estado que satisfaz a pós-condição C_2. As seguintes asserções são triplos de Hoare bem formados:

1. {`x==0`} `x=x+1` {`x==1`}
2. {`x < 5`} `x = x ** 2` {`x < 25`}
3. {`len(w)==1`} `w=w[1:]` {`len(w)==0`}
4. {`x != 0`} `if x==0: x=1 else: x=2` {`x==2`}
5. {`x<y`} `while x<y: x=x+1` {`x==y`}
6. {`x<y and type(x) is int and type(y) is int`} `while x<y: x=x+1` {`x==y`}
7. {`True`} `while True: pass` {`False`}
8. {`False`} `P` {C}

Os triplos 1, 3–4 e 6–8 são válidos. Vale a pena, em particular, compreender bem os últimos dois casos. O programa `while True: pass` certamente não termina, qualquer que seja o estado de partida, pelo que o triplo seria válido qualquer que fosse a pós-condição. Já a pré-condição `False` não é satisfeita por nenhum estado inicial possível, pelo que a asserção é vacuosamente válida. Por outro lado, é fácil constatar que os triplos 2 e 5 não são válidos.

Regras de inferência

Para além da nossa capacidade de análise semântica das asserções, condições ou triplos de Hoare, o cálculo de Hoare fornece-nos um conjunto de regras de inferência que permitem construir, estruturadamente, demonstrações de validade (para as asserções válidas, obviamente).

O cálculo é constituído por regras da forma

$$\frac{\text{Premissa}_1\ \ldots\ \text{Premissa}_n}{\text{Conclusão}}$$

que permitem concluir da validade da asserção correspondente à Conclusão, se assumirmos a validade das asserções $\text{Premissa}_1, \ldots, \text{Premissa}_n$ correspondentes às premissas. Quando a regra não tem premissas, a sua conclusão diz-se um *axioma*. Tendo em conta que uma premissa de uma regra pode ser a conclusão de outra regra, é possível construir demonstrações formais de validade, sempre terminando em axiomas.

As regras do cálculo, apresentadas na Figura 1 certificam condições Booleanas válidas **(Rval)**, permitem fortalecer a pré-condição **(Rpre)** ou enfraquecer a pós-condição **(Rpos)** de um triplo de Hoare, e são de resto estruturais nos programas que queiramos analisar. Nomeadamente, as regras **(Rpass, Ratr, Rseq, Ralt, Riter)**, correspondem respectivamente ao programa `pass`, a atribuições, e à composição sequencial, alternativa e iterativa de programas.

As regras **(Rpass)** e **(Ratr)** não têm premissas pois dizem respeito a programas atómicos. **(Rseq)**, **(Ralt)** e **(Riter)** têm premissas respeitantes à correcção parcial dos subprogramas que são compostos. Todas as regras capturam de forma simples o significado dos programas ou construções a que diz respeito. As duas regras menos intuitivas são **(Ratr)** e **(Riter)**, que passamos a explicar mais detalhadamente.

A regra das atribuições **(Ratr)** diz-nos que após uma atribuição $\mathtt{x_1}, \ldots, \mathtt{x_n} = \mathtt{e_1}, \ldots, \mathtt{e_n}$ chegamos sempre a um estado que satisfaz uma condição C (que envolve os nomes $\mathtt{x_1}, \ldots, \mathtt{x_n}$) desde que tenhamos partido de um estado que satisfaz a condição semelhante $(C)^{\mathtt{x_1}\ \mathtt{x_2}\ \cdots\ \mathtt{x_n}}_{\mathtt{e_1},\mathtt{e_2},\ldots,\mathtt{e_n}}$. Vejamos alguns exemplos:

- $\{\mathtt{y>5}\}\ \mathtt{x = y}\ \{\mathtt{x>5}\}$

  A condição $(\mathtt{x} > 5)^{\mathtt{x}}_{\mathtt{y}} \equiv \mathtt{y} > 5$ é a pré-condição óbvia necessária para que se tenha $\mathtt{x} > 5$ após executar $\mathtt{x = y}$.

- $\{\mathtt{x+1==1}\}\ \mathtt{x = x + 1}\ \{\mathtt{x==1}\}$

  É claro que a condição $(\mathtt{x == 1})^{\mathtt{x}}_{\mathtt{x+1}} \equiv \mathtt{x + 1 == 1}$ é equivalente a $\mathtt{x == 0}$, a pré-condição óbvia necessária para que se tenha $\mathtt{x == 1}$ após executar $\mathtt{x = x + 1}$.

- $\{\mathtt{y+1+x-1==0}\}\ \mathtt{x,y=y+1,x-1}\ \{\mathtt{x+y==0}\}$

  A condição $(\mathtt{x + y == 0})^{\mathtt{x}\ \mathtt{y}}_{\mathtt{y+1},\mathtt{x-1}} \equiv \mathtt{y + 1 + x - 1 == 0}$ é equivalente à pós-condição $\mathtt{x + y == 0}$, o que é precisamente o necessário já que a atribuição $\mathtt{x, y = y + 1, x - 1}$ não altera o valor da soma $\mathtt{x + y}$.

A regra da composição iterativa **(Riter)** é muito mais interessante, e envolve o conceito fundamental de *condição invariante* de um ciclo. A regra

$$\textbf{(Riter)}\ \frac{\{C_{\text{inv}}\ \mathtt{and\ G}\}\ \mathtt{P}\ \{C_{\text{inv}}\}}{\{C_{\text{inv}}\}\ \mathtt{while\ G : P}\ \{C_{\text{inv}}\ \mathtt{and\ (not\ G)}\}}$$

diz-nos que se a condição C_{inv} for mantida ao longo da execução do ciclo (daí chamar-se invariante) então, começando num estado que a satisfaça, o ciclo, se terminar, termina num estado que ainda a satisfaz, para além de obviamente tornar a guarda do ciclo `G` falsa.

Note-se que um mesmo ciclo tem em geral muitas condições invariantes. A escolha da condição invariante adequada em cada circunstância depende da condição final que se pretende que o ciclo garanta, que terá necessariamente de ser consequência de C_{inv} `and` (`not G`), de acordo com a regra **(Rpos)**. Vejamos um par de exemplos:

- $\{\mathtt{x==N\ and\ r==0}\}\ \mathtt{while\ x!=y : (x = x + 1; r = r + 1)}\ \{\mathtt{r==y-N}\}$

  É claro neste caso que o programa não terminará sempre, o que é algo que não nos deve preocupar de momento. No entanto, caso termine, o valor de `x` progredirá de N até igualar `y`, sendo o progresso acumulado registado em `r`. A condição invariante que devemos considerar, neste caso, deve reflectir exactamente isto:

  $$C_{\text{inv}} \equiv \mathtt{r == x - N}.$$

  Constataremos, já de seguida, que de facto esta condição é invariante do ciclo, para além de ser garantida pela pré-condição $\mathtt{x == N\ and\ r == 0}$, e de juntamente com a negação da guarda `not(x != y)` garantir a pós-condição $\mathtt{r == y - N}$.

se C é uma condição válida $\dfrac{}{C}$ **(Rval)**

$$\frac{A \Rightarrow B \qquad \{B\}\ \mathtt{P}\ \{C\}}{\{A\}\ \mathtt{P}\ \{C\}}\textbf{(Rpre)} \qquad\qquad \frac{\{A\}\ \mathtt{P}\ \{B\} \qquad B \Rightarrow C}{\{A\}\ \mathtt{P}\ \{C\}}\textbf{(Rpos)}$$

$$\frac{}{\{C\}\ \mathtt{pass}\ \{C\}}\textbf{(Rpass)} \qquad\qquad \frac{}{\{(C)^{\mathtt{x_1}\ \mathtt{x_2}\ \cdots\ \mathtt{x_n}}_{\mathtt{e_1},\mathtt{e_2},\ldots,\mathtt{e_n}}\}\ \mathtt{x_1},\ldots,\mathtt{x_n} = \mathtt{e_1},\ldots,\mathtt{e_n}\ \{C\}}\textbf{(Ratr)}$$

$$\frac{\{C_0\}\ \mathtt{P}_1\ \{C_1\} \qquad \{C_1\}\ \mathtt{P}_2\ \{C_2\} \quad \ldots \quad \{C_{n-1}\}\ \mathtt{P}_n\ \{C_n\}}{\{C_0\}\ \mathtt{P}_1; \mathtt{P}_2; \ldots; \mathtt{P}_n\ \{C_n\}}\textbf{(Rseq)}$$

$$\frac{\{A\ \mathtt{and}\ \mathtt{C}_1\}\ \mathtt{P}_1\ \{B\} \qquad \{A\ \mathtt{and}\ (\mathtt{not}\ \mathtt{C}_1)\ \mathtt{and}\ \mathtt{C}_2\}\ \mathtt{P}_2\ \{B\} \quad \ldots \quad \{A\ \mathtt{and}\ (\mathtt{not}\ \mathtt{C}_1)\ \mathtt{and} \ldots \mathtt{and}\ (\mathtt{not}\ \mathtt{C}_{n-1})\}\ \mathtt{P}_n\ \{B\}}{\{A\}\ \mathtt{if}\ \mathtt{C}_1 : \mathtt{P}_1\ \mathtt{elif}\ \mathtt{C}_2 : \mathtt{P}_2\ \ldots\ \mathtt{elif}\ \mathtt{C}_{n-1} : \mathtt{P}_{n-1}\ \mathtt{else} : \mathtt{P}_n\ \{B\}}\textbf{(Ralt)}$$

$$\frac{\{C_{\text{inv}}\ \mathtt{and}\ \mathtt{G}\}\ \mathtt{P}\ \{C_{\text{inv}}\}}{\{C_{\text{inv}}\}\ \mathtt{while}\ \mathtt{G} : \mathtt{P}\ \{C_{\text{inv}}\ \mathtt{and}\ (\mathtt{not}\ \mathtt{G})\}}\textbf{(Riter)}$$

Figura 1: Regras de inferência do cálculo de Hoare para correcção parcial de programas.

- $\{\texttt{type(x) is int and x>=0}\}\ \texttt{while x} > \texttt{0 : x} = \texttt{x} - \texttt{1}\ \{\texttt{x==0}\}$

  No caso 2, é claro que o programa terminará sempre se iniciado num estado que satisfaça a pré-condição. Obviamente, nesse caso terminará de facto com o valor de `x` a `0`. Isto só é verdade porque o valor de `x` se manterá um inteiro positivo ao longo do ciclo, pelo que é apropriado considerar a condição invariante

  $$C_{\text{inv}} \equiv (\texttt{type(x) is int and x} >= \texttt{0}) \equiv \texttt{nat(x)}$$

  já que, juntamente com a negação da guarda do ciclo, `not(x>0)`, garante a pós-condição pretendida.

Exemplos de aplicação

Consideremos alguns exemplos simples de aplicação.

- $\{\texttt{x} == \texttt{3}\}\ \texttt{y} = \texttt{x} + \texttt{1}; \texttt{x} = \texttt{2} * \texttt{y}\ \{\texttt{x} == \texttt{8}\}$

  A demonstração, na Figura 2, é muito simples e segue por aplicação das regras **(Rseq)** (o programa é uma composição sequencial), **(Ratr)** (a composição sequencial consiste de duas atribuições), **(Rpre)** para acerto da pré-condição obtida naturalmente das regras da atribuição, e finalmente **(Rval)** para confirmar a validade da expressão lógica obtida.

- $\{x == 4\}\ \texttt{if x\%2} == \texttt{0 : y} = \texttt{x} - \texttt{1 else : y} = \texttt{x} + \texttt{1}\ \{\texttt{y} == \texttt{3}\}$

  A demonstração, na Figura 3, usa de forma essencial a regra **(Ralt)** (o programa é uma composição alternativa), resultando em asserções cuja demonstração é semelhante às que vimos acima.

- $\{\texttt{x==N and r==0}\}\ \texttt{while x!} = \texttt{y : (x} = \texttt{x} + \texttt{1; r} = \texttt{r} + \texttt{1)}\ \{\texttt{r==y-N}\}$

  A demonstração, na Figura 4, começa por usar **(Rpre)** e **(Rpos)** por forma a introduzir a condição invariante previamente obtida. Isto resulta em duas condições Booleanas cuja validade é facilmente garantida pela regra **(Rval)**. Após a correcta introdução da condição invariante, a demonstração faz uso essencial da regra **(Riter)** (o programa é uma composição iterativa), resultando numa asserção cuja demonstração é semelhante às anteriores.

- $\{\texttt{x==5}\}\ \texttt{while x} > \texttt{0 : x} = \texttt{x} - \texttt{1}\ \{\texttt{x==0}\}$

  A demonstração, na Figura 5, tem uma estrutura semelhante à anterior, tirando partido da condição invariante previamente obtida.

Vale a pena notar que, em todos os casos, é necessário justificar a validade da condição Booleana sempre que se aplica a regra **(Rval)**.

Consideremos agora um exemplo mais interessante, nomeadamente o programa imperativo de cálculo do factorial que recordamos de seguida.

$$\dfrac{\dfrac{}{\mathtt{x == 3} \Rightarrow \mathtt{2 * (x + 1) == 8}}\ (\mathbf{Rval}) \quad \dfrac{}{\{\mathtt{2 * (x + 1) == 8}\}\ \mathtt{y = x + 1}\ \{\mathtt{2 * y == 8}\}}\ (\mathbf{Ratr})}{\{\mathtt{x == 3}\}\ \mathtt{y = x + 1}\ \{\mathtt{2 * y == 8}\}}\ (\mathbf{Rpre}) \quad \dfrac{}{\{\mathtt{2 * y == 8}\}\ \mathtt{x = 2 * y}\ \{\mathtt{x == 8}\}}\ (\mathbf{Ratr})$$
$$\dfrac{\{\mathtt{x == 3}\}\ \mathtt{y = x + 1}\ \{\mathtt{2 * y == 8}\} \quad \{\mathtt{2 * y == 8}\}\ \mathtt{x = 2 * y}\ \{\mathtt{x == 8}\}}{\{\mathtt{x == 3}\}\ \mathtt{y = x + 1; x = 2 * y}\ \{\mathtt{x == 8}\}}\ (\mathbf{Rseq})$$

Figura 2: Demonstração de $\{\mathtt{x == 3}\}\ \mathtt{y = x + 1; x = 2 * y}\ \{\mathtt{x == 8}\}$.

$$\dfrac{\dfrac{}{(\mathtt{x == 4\ and\ x\%2 == 0}) \Rightarrow \mathtt{x - 1 == 3}}\ (\mathbf{Rval}) \quad \dfrac{}{\{\mathtt{x - 1 == 3}\}\ \mathtt{y = x - 1}\ \{\mathtt{y == 3}\}}\ (\mathbf{Ratr})}{\{\mathtt{x == 4\ and\ x\%2 == 0}\}\ \mathtt{y = x - 1}\ \{\mathtt{y == 3}\}}\ (\mathbf{Rpre}) \quad \dfrac{\dfrac{}{(\mathtt{x == 4\ and\ x\%2 \neq 0}) \Rightarrow \mathtt{x + 1 == 3}}\ (\mathbf{Rval}) \quad \dfrac{}{\{\mathtt{x + 1 == 3}\}\ \mathtt{y = x + 1}\ \{\mathtt{y == 3}\}}\ (\mathbf{Ratr})}{\{\mathtt{x == 4\ and\ x\%2 \neq 0}\}\ \mathtt{y = x + 1}\ \{\mathtt{y == 3}\}}\ (\mathbf{Rpre})$$
$$\dfrac{\{\mathtt{x == 4\ and\ x\%2 == 0}\}\ \mathtt{y = x - 1}\ \{\mathtt{y == 3}\} \quad \{\mathtt{x == 4\ and\ x\%2 \neq 0}\}\ \mathtt{y = x + 1}\ \{\mathtt{y == 3}\}}{\{\mathtt{x == 4}\}\ \mathtt{if\ x\%2 == 0 : y = x - 1\ else : y = x + 1}\ \{\mathtt{y == 3}\}}\ (\mathbf{Ralt})$$

Figura 3: Demonstração de $\{\mathtt{x == 4}\}\ \mathtt{if\ x\%2 == 0 : y = x - 1\ else : y = x + 1}\ \{\mathtt{y == 3}\}$.

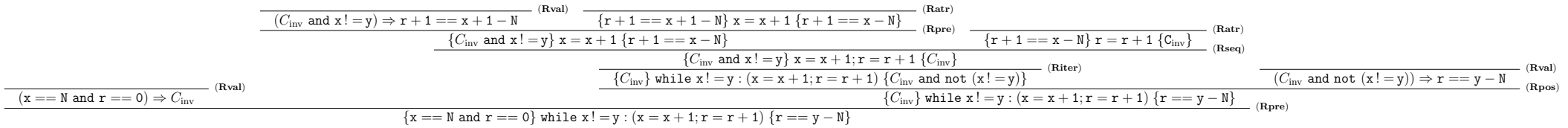

Figura 4: Demonstração de $\{\texttt{x} == \texttt{N and r} == 0\}$ while $\texttt{x} != \texttt{y} : (\texttt{x} = \texttt{x} + 1; \texttt{r} = \texttt{r} + 1)\ \{\texttt{r} == \texttt{y} - \texttt{N}\}$ com $C_{\text{inv}} \equiv \texttt{r} == \texttt{x} - \texttt{N}$.

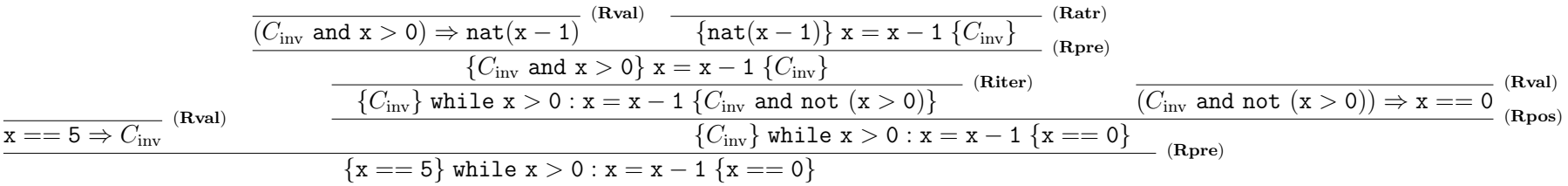

Figura 5: Demonstração de $\{\texttt{x} == 5\}$ while $\texttt{x} > 0 : \texttt{x} = \texttt{x} - 1\ \{\texttt{x==0}\}$ com $C_{\text{inv}} \equiv \texttt{nat(x)}$.

```
In [1]: def factorial(n):
            i=n
            r=1
            while i!=0:
                r=r*i
                i=i-1
            return r
```

O corpo (FACT) da definição, sobre o qual pretendemos raciocinar, corresponde a

$$\texttt{FACT} \left\{ \begin{array}{l} \left.\begin{array}{l} \texttt{i} = \texttt{n} \\ \texttt{r} = 1 \end{array}\right\} \texttt{INIC} \\ \left.\begin{array}{l} \texttt{while i} \,!= 0: \\ \quad \left.\begin{array}{l} \texttt{r} = \texttt{r} * \texttt{i} \\ \texttt{i} = \texttt{i} - 1 \end{array}\right\} \texttt{PASSO} \end{array}\right\} \texttt{CICLO} \end{array} \right.$$

onde, para facilitar, distinguimos a inicialização (INIC), o ciclo (CICLO), e o passo do ciclo (PASSO).

Pretende-se demonstrar a asserção

$$\{\texttt{True}\}\ \texttt{FACT}\ \{\texttt{r} == \prod_{1 \leq \texttt{k} \leq \texttt{n}} \texttt{k}\}$$

que garante que, ao terminar, o valor de `r` corresponde de facto ao factorial de `n`.

É fundamental formular a condição invariante deste ciclo correctamente, nomeadamente face à inicialização e ao objectivo de garantir que $\texttt{r} == \prod_{1 \leq \texttt{k} \leq \texttt{n}} \texttt{k}$ no final da execução do programa. Tendo em conta que o ciclo progride de forma a que `i` percorra os valores de `n` até `1`, acumulando `r` o produto dos números já visitados, é relativamente simples chegar à seguinte proposta:

$$C_{\text{inv}} \equiv \texttt{r} == \prod_{\texttt{i}+1 \leq \texttt{k} \leq \texttt{n}} \texttt{k}$$

que não só é assegurada pela inicialização (um produto vazio é necessariamente 1), como garante o objectivo final quando conjugada com a negação da guarda do ciclo (`not i != 0`).

A demonstração, na Figura 6, está apresentada em duas partes, para facilitar a sua escrita e visualização. A primeira parte resultante da aplicação da regra **(Rseq)** (FACT é a composição sequencial INIC;CICLO) com introdução de C_{inv} como condição intermédia, inclui a demonstração da primeira premissa $\{\texttt{True}\}\ \texttt{INIC}\ \{C_{\text{inv}}\}$. Na segunda parte demonstra-se a segunda premissa da aplicação de **(Rseq)**, $\{C_{\text{inv}}\}\ \texttt{CICLO}\ \{\texttt{r} == \prod_{1 \leq \texttt{k} \leq \texttt{n}} \texttt{k}\}$.

Vale a pena atentar na estrutura geral da demonstração da correcção parcial de um ciclo inicializado.

$$\texttt{PROG} \left\{ \begin{array}{l} \texttt{INIC} \\ \left.\begin{array}{l} \texttt{while G}: \\ \quad \texttt{PASSO} \end{array}\right\} \texttt{CICLO} \end{array} \right.$$

Para demonstrar

$$\{A\}\ \texttt{PROG}\ \{B\}$$

depois de determinada a condição invariante C_{inv}, a demonstração pode sempre tomar a forma seguinte

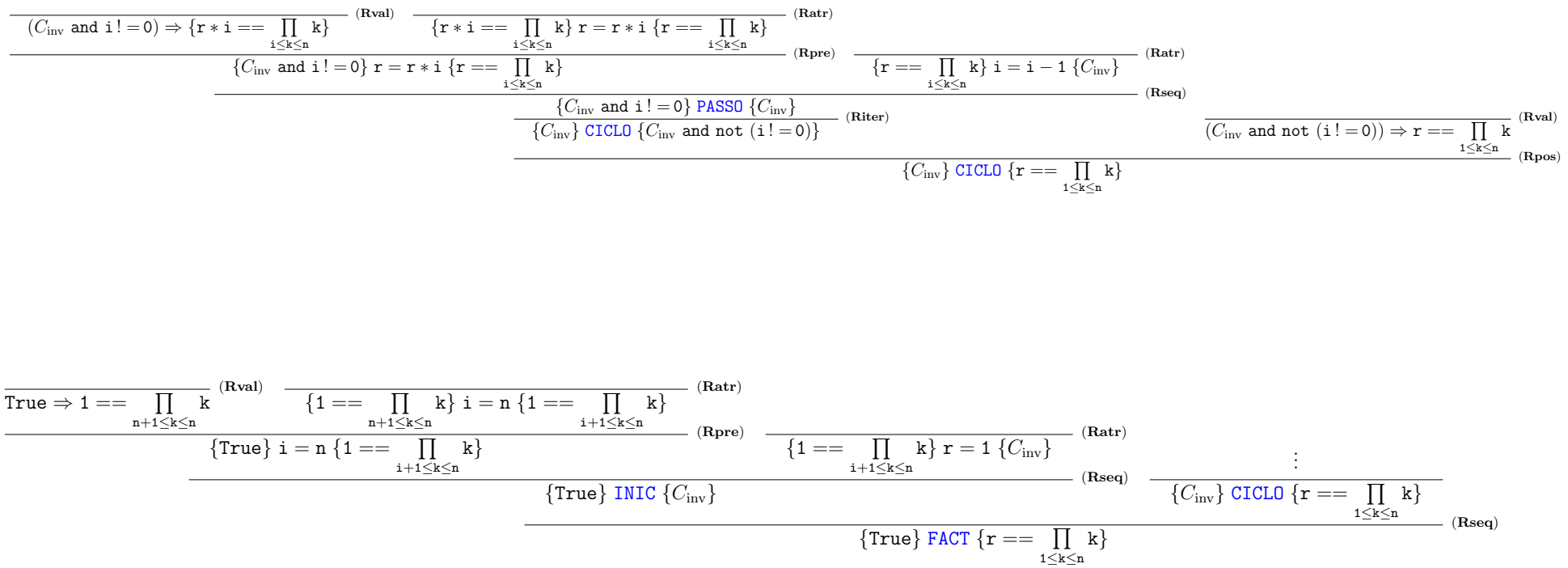

Figura 6: Demonstração de $\{\texttt{True}\}\ \texttt{FACT}\ \{\texttt{r} == \prod_{1 \le \texttt{k} \le \texttt{n}} \texttt{k}\}$ com $C_{\text{inv}} \equiv \texttt{r} == \prod_{\texttt{i}+1 \le \texttt{k} \le \texttt{n}} \texttt{k}$.

$$\dfrac{\dfrac{\vdots}{\{A\}\ \texttt{INIC}\ \{C_{\text{inv}}\}} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\{C_{\text{inv}}\ \texttt{and G}\}\ \texttt{PASSO}\ \{C_{\text{inv}}\}}}{\{C_{\text{inv}}\}\ \texttt{CICLO}\ \{C_{\text{inv}}\ \texttt{and not G}\}}\ (\textbf{Riter}) \qquad \dfrac{\vdots}{(C_{\text{inv}}\ \texttt{and not G}) \Rightarrow B}}{\{C_{\text{inv}}\}\ \texttt{CICLO}\ \{B\}}\ (\textbf{Rpos})}{\{A\}\ \texttt{PROG}\ \{B\}}\ (\textbf{Rseq})$$

bastando para a completar encontrar demonstrações de

- $\{A\}$ INIC $\{C_{\text{inv}}\}$ - o invariante é garantido pela inicialização;
- $\{C_{\text{inv}}$ and G$\}$ PASSO $\{C_{\text{inv}}\}$ - C_{inv} é de facto invariante do ciclo; e
- $(C_{\text{inv}}$ and not G$) \Rightarrow B$ - o invariante e a negação da guarda garantem o objectivo.

Considere-se o exemplo seguinte de um programa que calcula `x ** e` de forma eficiente.

```
In [1]: def sqrmult(x,e):
            r=1
            while e!=0:
                if e%2==0:
                    x=x**2
                    e=e/2
                else:
                    r=r*x
                    e=e-1
            return r
```

O corpo (SQM) da definição tem a estrutura de um ciclo inicializado.

$$\texttt{SQM}\left\{\begin{array}{l} \left.\begin{array}{l} \texttt{r = 1} \\ \texttt{while e != 0 :} \end{array}\right\}\texttt{INIC} \\ \left.\begin{array}{l} \left.\begin{array}{l} \texttt{if e\%2 == 0 :} \\ \quad \texttt{x = x ** 2} \\ \quad \texttt{e = e/2} \end{array}\right\}\texttt{PASSO} \\ \texttt{else :} \\ \quad \texttt{r = r * x} \\ \quad \texttt{e = e - 1} \end{array}\right\}\texttt{CICLO}$$

Pretende-se demonstrar a asserção

$$\{\texttt{x == A and e == N}\}\ \texttt{SQM}\ \{\texttt{r == A ** N}\} \quad .$$

Analisando o programa, não é difícil de encontrar a condição invariante

$$C_{\text{inv}} \equiv \texttt{r} * (\texttt{x ** e}) == \texttt{A ** N}$$

podendo a demonstração da correcção parcial do programa ser facilmente construída de acordo com a estrutura geral delineada acima, faltando apenas demonstrar:

- $\{\texttt{x == A and e == N}\}\ \texttt{r = 1}\ \{C_{\text{inv}}\}$;
- $\{C_{\text{inv}}\ \texttt{and e != 0}\}$ PASSO $\{C_{\text{inv}}\}$; e
- $(C_{\text{inv}}\ \texttt{and not (e != 0)}) \Rightarrow \texttt{r == A ** N}$.

As demonstrações da primeira e última asserções são em tudo semelhantes a outras que fizemos acima. A primeira segue muito facilmente, usando a regra **(Ratr)** com $(C_{\text{inv}})_1^{\mathtt{r}} \equiv \mathtt{1} * (\mathtt{x} \mathbin{**} \mathtt{e}) == \mathtt{A} \mathbin{**} \mathtt{N}$ e as regras **(Rval)** e **(Rpre)**. A última é uma condição válida, facilmente justificável com a regra **(Rval)**. A segunda das asserções é decerto a mais interessante, mas reduz-se por aplicação de **(Ralt)** a duas asserções de demonstração simples:

- $\{C_{\text{inv}}\ \mathtt{and\ e} != \mathtt{0}\ \mathtt{and\ e}\%\mathtt{2} == \mathtt{0}\}\ \mathtt{x} = \mathtt{x} \mathbin{**} \mathtt{2}; \mathtt{e} = \mathtt{e}/\mathtt{2}\ \{C_{\text{inv}}\}$; e
- $\{C_{\text{inv}}\ \mathtt{and\ e} != \mathtt{0}\ \mathtt{and\ not}\ (\mathtt{e}\%\mathtt{2} == \mathtt{0})\}\ \mathtt{r} = \mathtt{r} * \mathtt{x}; \mathtt{e} = \mathtt{e} - \mathtt{1}\ \{C_{\text{inv}}\}$.

Deixam-se os detalhes como exercício.

Correcção total de programas

As asserções do cálculo de Hoare para a correcção total de programas incluem asserções de terminação, para além de condições Booleanas e triplos de Hoare.

Asserções de terminação

As asserções de terminação são da forma

$$\Omega(C, \mathtt{P})$$

onde C é uma condição Booleana (*pré-condição*), e `P` é um programa imperativo. O significado de uma tal asserção é intuitivo: a asserção é válida se a execução do programa `P` termina sempre que é iniciada num estado que satisfaz a condição C. A escolha da notação Ω também é reveladora, sendo a última letra do alfabeto grego, e veiculando portanto a intuição de que o programa chega ao fim.

Fica claro que, no contexto em que trabalhamos, a única fonte de não-terminação de que precisamos de cuidar são os ciclos. Se é verdade em geral que a avaliação de expressões (em atribuições, ou guardas Booleanas em comandos `if` ou `while`) pode também ser problemática, caso as expressões resultem por exemplo da invocação de definições auxiliares que possam não terminar, vale a pena notar que assumimos que tal não acontece nos programas que nos propomos analisar.

Seguem-se exemplos de asserções de terminação bem formadas.

1. $\Omega(\mathtt{True}, \mathtt{while\ x} != \mathtt{0} : \mathtt{x} = \mathtt{x} - \mathtt{1})$
2. $\Omega(\mathtt{nat(x)}, \mathtt{while\ x} != \mathtt{0} : \mathtt{x} = \mathtt{x} - \mathtt{1})$
3. $\Omega(\mathtt{False}, \mathtt{P})$

Claramente, a asserção 1 não é válida, ao contrário da asserção 2 onde o reforço da pré-condição exclui todos os possíveis estados iniciais que levariam à não-terminação do ciclo. A asserção 3 é válida, vacuosamente, independentemente da estrutura do programa `P`.

Regras de inferência

As regras do cálculo de Hoare para a correcção total de programas, apresentadas na Figura 7, estendem o cálculo original com regras relativas à terminação de programas. A regra **(Tpre)** fortalece a pré-condição numa asserção de terminação. As regras relativas à estrutura dos programas reflectem uma vez mais, de forma quase imediata, o significado de cada um dos programas/mecanismos de composição. As regras **(Tpass)** e **(Tatr)** garantem que `pass` e atribuições terminam sempre. **(Tseq)**, **(Talt)** e **(Titer)** têm premissas respeitantes à terminação dos subprogramas que são compostos.

A única regra menos intuitiva, como seria de esperar, e que vale a pena compreender com cuidado, é **(Titer)**. A regra

$$\frac{A \Rightarrow B \qquad \Omega(B,\, \mathtt{P})}{\Omega(A,\, \mathtt{P})}\,\textbf{(Tpre)}$$

$$\frac{}{\Omega(C,\, \mathtt{pass})}\,\textbf{(Tpass)} \qquad\qquad \frac{}{\Omega(C,\, \mathtt{x_1}, \ldots, \mathtt{x_n} = \mathtt{e_1}, \ldots, \mathtt{e_n})}\,\textbf{(Tatr)}$$

$$\frac{\Omega(C_0,\, \mathtt{P}_1) \qquad \{C_0\}\ \mathtt{P}_1\ \{C_1\} \qquad \Omega(C_1,\, \mathtt{P}_2) \qquad \{C_1\}\ \mathtt{P}_2\ \{C_2\} \quad \ldots \quad \{C_{n-2}\}\ \mathtt{P}_{n-1}\ \{C_{n-1}\} \qquad \Omega(C_{n-1},\, \mathtt{P}_n)}{\Omega(C_0,\, \mathtt{P}_1; \mathtt{P}_2; \ldots; \mathtt{P}_n)}\,\textbf{(Tseq)}$$

$$\frac{\Omega(A \texttt{ and } \mathtt{C}_1,\, \mathtt{P}_1) \qquad \Omega(A \texttt{ and } (\texttt{not } \mathtt{C}_1) \texttt{ and } \mathtt{C}_2,\, \mathtt{P}_2) \quad \ldots \quad \Omega(A \texttt{ and } (\texttt{not } \mathtt{C}_1) \texttt{ and} \ldots \texttt{and } (\texttt{not } \mathtt{C}_{n-1})),\, \mathtt{P}_n)}{\Omega(A,\, \texttt{if } \mathtt{C}_1 : \mathtt{P}_1 \texttt{ elif } \mathtt{C}_2 : \mathtt{P}_2 \ \ldots \ \texttt{elif } \mathtt{C}_{n-1} : \mathtt{P}_{n-1} \texttt{ else} : \mathtt{P}_n)}\,\textbf{(Talt)}$$

$$\frac{C_{\text{inv}} \Rightarrow \mathtt{nat}(\tau) \qquad \Omega(C_{\text{inv}} \texttt{ and } \mathtt{G},\, \mathtt{P}) \qquad \{C_{\text{inv}} \texttt{ and } \mathtt{G} \texttt{ and } \tau == \mathtt{N}\}\ \mathtt{P}\ \{C_{\text{inv}} \texttt{ and } \tau < \mathtt{N}\}}{\Omega(C_{\text{inv}},\, \texttt{while } \mathtt{G} : \mathtt{P})}\,\textbf{(Titer)}$$

Figura 7: Regras de inferência do cálculo de Hoare para terminação de programas.

$$\frac{C_{\text{inv}} \Rightarrow \texttt{nat}(\tau) \qquad \Omega(C_{\text{inv}} \text{ and } \texttt{G}, \texttt{P}) \qquad \{C_{\text{inv}} \text{ and } \texttt{G} \text{ and } \tau == \texttt{N}\}\ \texttt{P}\ \{C_{\text{inv}} \text{ and } \tau < \texttt{N}\}}{\Omega(C_{\text{inv}}, \texttt{while G:P})} \textbf{(Titer)}$$

envolve, para além do conceito de condição invariante de um ciclo um outro conceito fulcral, agora para a terminação de programas, que é o de *expressão variante*. A expressão variante, aqui denotada por τ, é uma quantidade inteira, não negativa, que garantidamente decresça cada vez que seja executado o passo do ciclo. Como os números naturais são limitados inferiormente por 0 a ordem é bem fundada e o ciclo termina necessariamente. De facto o ciclo terá de ter uma condição invariante adequada que garanta que a quantidade τ é sempre inteira e não negativa. Vejamos um par de exemplos, para os quais tentaremos encontrar a pré-condição mais fraca que torne as asserções válidas:

- $\Omega(\_, \texttt{while x} > \texttt{0 : x} = \texttt{x} - \texttt{1})$

  É claro que o programa só terminará se o valor de `x` no estado inicial for um número natural. Como o valor de `x` é decrementado sucessivamente, igualando `0` quando o ciclo termina, é óbvio que a expressão

  $$\tau \equiv \texttt{x}$$

  é uma boa candidata a expressão variante do ciclo. A condição invariante associada, que podemos usar como pré-condição da asserção de terminação, pode então ser apenas:

  $$C_{\text{inv}} \equiv \texttt{nat(x)}.$$

- $\Omega(\_, \texttt{while x} \,!= \texttt{y : (x} = \texttt{x} + \texttt{1; r} = \texttt{r} + \texttt{1}))$

  Neste caso, é claro que o programa só terminará se o valor de $\texttt{y} - \texttt{x}$ no estado inicial for um inteiro positivo. Como o valor de `x` é incrementado sucessivamente, igualando `x` quando o ciclo termina, é óbvio que a expressão

  $$\tau \equiv \texttt{y} - \texttt{x}$$

  é uma boa candidata a expressão variante do ciclo. A condição invariante associada, que podemos usar como pré-condição da asserção de terminação, pode então ser apenas:

  $$C_{\text{inv}} \equiv \texttt{nat(y} - \texttt{x)}.$$

Exemplo de aplicação

Como exemplo de aplicação vamos usar um dos programas já considerados anteriormente. Recorde-se a definição de SQM.

```
SQM { r = 1                          } INIC
    { while e != 0 :                 }
    {   if e%2 == 0 :                } PASSO } CICLO
    {     x = x ** 2                 }
    {     e = e/2                    }
    {   else :                       }
    {     r = r * x                  }
    {     e = e - 1                  }
```

É claro que SQM termina desde que o valor inicial do expoente `e` seja um número natural. Assim sendo, para demonstrar

$$\Omega(\texttt{nat(e)}, \texttt{SQM})$$

basta tomar a expressão variante

$$\tau \equiv e$$

que de facto diminui a cada passo do ciclo, e considerar a condição invariante associada

$$C_{\text{inv}} \equiv \texttt{nat(e)}$$

para facilmente construir a demonstração da Figura 8, onde as subdemonstrações omissas indicadas por $\vdots$ são em tudo semelhantes às que lhes estão imediatamente à esquerda.

Exemplo complementar

Para reforçar os conceitos vale a pena analisar a correcção de um programa mais complexo, com dois ciclos, que soma todos os elementos de um vector de vectores.

```
In [1]: def sumall(w):
            s=0
            i=0
            while i!=len(w):
                j=0
                while j!=len(w[i]):
                    s=s+w[i][j]
                    j=j+1
                i=i+1
            return s
```

$$\text{SUM}\left\{\begin{array}{l} \texttt{s} = 0 \\ \texttt{i} = 0 \\ \texttt{while i!} = \texttt{len(w)}: \\ \quad \texttt{j} = 0 \\ \quad \texttt{while j!} = \texttt{len(w[i])}: \\ \quad\quad \texttt{s} = \texttt{s} + \texttt{w[i][j]} \\ \quad\quad \texttt{j} = \texttt{j} + 1 \\ \quad \texttt{i} = \texttt{i} + 1 \end{array}\right. \begin{array}{l} \left.\begin{array}{l} \\ \\ \\ \\ \\ \end{array}\right\}\text{INIC}_1 \\ \left.\begin{array}{l} \\ \\ \\ \end{array}\right\}\text{INIC}_2 \\ \left.\begin{array}{l} \\ \\ \end{array}\right\}\text{PASSO}_2 \\ \left.\begin{array}{l} \\ \\ \\ \end{array}\right\}\text{CICLO}_2 \\ \left.\begin{array}{l} \\ \end{array}\right\}\text{FIN}_2 \\ \left.\begin{array}{l} \\ \end{array}\right\}\text{PASSO}_1 \\ \left.\begin{array}{l} \\ \\ \\ \\ \\ \end{array}\right\}\text{CICLO}_1 \end{array}$$

Não é difícil demonstrar a correcção parcial do programa, nomeadamente

$$\{\texttt{True}\}\ \texttt{SUM}\ \{\texttt{s} == \sum_{0\le \texttt{a}\le \texttt{len(w)}-1}\ \sum_{0\le \texttt{b}\le \texttt{len(w[a])}-1} \texttt{w[a][b]}\}$$

bastando para tal observar que o ciclo externo ($\texttt{CICLO}_1$) tem como condição invariante

$$C_{\text{inv}_1} \equiv \texttt{s} == \sum_{0\le \texttt{a}\le \texttt{i}-1}\ \sum_{0\le \texttt{b}\le \texttt{len(w[a])}-1} \texttt{w[a][b]}$$

cuja demonstração, por sua vez, necessita da condição invariante para o ciclo interno ($\texttt{CICLO}_2$) dada por

$$C_{\text{inv}_2} \equiv \texttt{s} == \sum_{0\le \texttt{a}\le \texttt{i}-1}\ \sum_{0\le \texttt{b}\le \texttt{len(w[a])}-1} \texttt{w[a][b]} + \sum_{0\le \texttt{b}\le \texttt{j}-1} \texttt{w[i][b]}.$$

A terminação do programa está assegurada para qualquer tuplo de tuplos `w`, pelo que será possível demonstrar

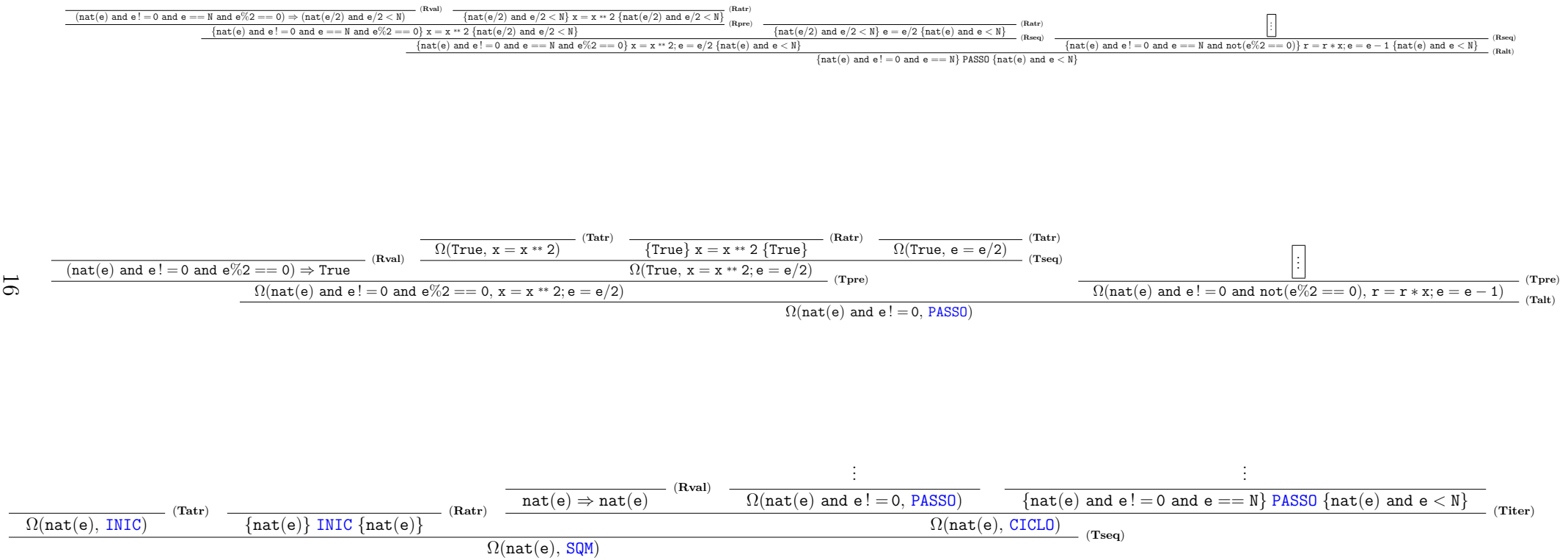

Figura 8: Demonstração de $\Omega(\texttt{nat(e)}, \texttt{SQM})$ com variante $\tau \equiv e$ e $C_{\text{inv}} \equiv \texttt{nat(e)}$.

$$\Omega(\texttt{type(w) is tuple and } (\forall_{\texttt{a=0,...,len(w)-1}}\texttt{type(w[a]) is tuple}), \texttt{SUM})$$

usando as expressões variantes

$$\tau_1 \equiv \texttt{len(w)} - \texttt{i} \qquad \text{e} \qquad \tau_2 \equiv \texttt{len(w[i])} - \texttt{j}$$

para cada um dos ciclos, agora considerando, respectivamente, as condições invariantes

$$C_{\text{inv}_1} \equiv \texttt{nat(len(w)} - \texttt{i)} \texttt{ and nat(i) and } (\forall_{\texttt{a=0,...,len(w)-1}}\texttt{type(w[a]) is tuple}), \text{ e}$$
$$C_{\text{inv}_2} \equiv \texttt{nat(len(w[i])} - \texttt{j)}.$$

Note-se que a condição invariante associada ao ciclo exterior é mais forte do que apenas `nat(len(w)-i)` já que necessita de conter informação que permita o estabelecimento da condição invariante associada ao ciclo interior. Deixam-se os detalhes das demonstrações como exercício.

Sumário

- A verificação de programas é um tópico crucial pela ubiquidade e importância dos algoritmos num número crescente de situações importantes, ou mesmo críticas. O problema não é em geral automatizável, pelo que é fundamental a intervenção de cientistas competentes no processo de demonstração.
- O cálculo de Hoare é um sistema lógico que permite raciocinar sobre a correcção de programas de forma rigorosa, em que se separa a correcção parcial do programa da sua terminação.
- O conceito fundamental subjacente à correcção parcial de um ciclo é a condição invariante, que deve ser verdadeira no início e mantida ao longo de toda a execução do ciclo.
- O conceito fundamental subjacente à terminação de um ciclo é a expressão variante, inteira e não negativa, que deve diminuir, demonstravelmente, no decurso da sua execução.

Bibliografia

Introdução à Programação em Mathematica (3a edição): J. Carmo, A. Sernadas, C. Sernadas, F. M. Dionísio, C. Caleiro, IST Press, 2014.

Think Python: How to think like a computer scientist: A. Downey, Green Tea Press, 2012.

Introduction to Computation and Programming Using Python (revised and expanded edition): J. V. Guttag, MIT Press, 2013.

The Art of Computer Programming: D. E. Knuth, Addison-Wesley (volumes 1–3, 4A), 1998.

Learning Python (fifth edition): M. Lutz, O'Reilly Media, 2013.

Programação em Python: Introdução à programação utilizando múltiplos paradigmas: J. P. Martins, IST Press, 2015.

Introdução à Programação em MatLab: J. Ramos, A. Sernadas e P. Mateus, DMIST, 2005.

Learning IPython for Interactive Computing and Data Visualization: C. Rossant, Packt Publishing, 2013.

Programação em Mathematica: A. Sernadas, C. Sernadas e J. Ramos, DMIST, 2003.