Cálculo de Correção parcial ($\mathcal{H}$)

$[skip_p\ ]$

$$\{\varphi\}\,\texttt{skip}\,\{\varphi\}$$

$[ass_p\ ]$

$$\{\varphi[E/x]\}\,x \leftarrow E\,\{\varphi\}$$

$[comp_p\ ]$

$$\frac{\{\varphi\}\,C_1\,\{\eta\} \qquad \{\eta\}\,C_2\,\{\psi\}}{\{\varphi\}\,C_1; C_2\,\{\psi\}}$$

$[if_p\ ]$

$$\frac{\{\varphi\ \wedge\ B\}\,C_1\,\{\psi\} \qquad \{\varphi\ \wedge\ \neg B\}\,C_2\,\{\psi\}}{\{\varphi\}\,\texttt{if}\,B\,\texttt{then}\,C_1\,\texttt{else}\,C_2\,\{\psi\}}$$

$[while_p\ ]$

$$\frac{\{\psi\ \wedge\ B\}\,C\,\{\psi\}}{\{\psi\}\,\texttt{while}\,B\,\texttt{do}\,C\,\{\psi\ \wedge\ \neg B\}}$$

$[cons_p\ ]$

$$\frac{\vdash \varphi' \to \varphi \qquad \{\varphi\}\,C\,\{\psi\} \qquad \vdash \psi \to \psi'}{\{\varphi'\}\,C\,\{\psi'\}}$$

Mecanização da construção de derivações na lógica de Hoare

De um modo geral, dado um triplo de Hoare ($\{\varphi\}C\{\psi\}$) aplicamos as regras a partir da conclusão, assumindo que as condições auxiliares se verificam.

- Se todas as condições auxiliares se verificarem então construímos uma demonstração;
- Se alguma das condições auxiliares não se verifica, a árvore construída não constitui uma dedução válida, mas será possível construir uma outra árvore que o seja?

Existe uma estratégia para construir as árvores de forma a poder concluir (caso algumas das condições auxiliares não se verifique) que não existe uma derivação para o triplo dado.

Mecanização da lógica de Hoare

A maior parte das regras do cálculo de Hoare têm a *propriedade de sub-fórmula:*

> *todas as asserções que ocorrem nas premissas de uma regra também ocorrem na sua conclusão.*

As excepções são:

- A regra *comp*, que requer uma condição intermédia;
- A regra *cons*, onde a pré-condição e a pós-condição têm que ser "adivinhadas".

Outra propriedade desejável é que não seja ambígua a escolha das regras:

- A regra *cons*, pode ser aplicada para qualquer triplo de Hoare.

Versão da lógica de Hoare sem *cons*: sistema $\mathcal{H}_g$

$$\frac{}{\{\varphi\}\,\mathsf{skip}\,\{\psi\}}\ \text{se} \models \varphi \to \psi$$

$$\frac{}{\{\varphi\}\,x \leftarrow E\,\{\psi\}}\ \text{se} \models \varphi \to \psi[E/x]$$

$$\frac{\{\varphi\}\,C_1\,\{\eta\} \qquad \{\eta\}\,C_2\,\{\psi\}}{\{\varphi\}\,C_1; C_2\,\{\psi\}}$$

$$\frac{\{\varphi\ \wedge\ B\}\,C_1\,\{\psi\} \qquad \{\varphi\ \wedge\ \neg B\}\,C_2\,\{\psi\}}{\{\varphi\}\,\mathtt{if}\,B\,\mathtt{then}\,C_1\,\mathtt{else}\,C_2\,\{\psi\}}$$

$$\frac{\{\eta\ \wedge\ B\}\,C\,\{\eta\}}{\{\varphi\}\,\mathtt{while}\,B\,\mathtt{do}\,\{\eta\}C\,\{\psi\}}\ \text{se} \models \varphi \to \eta \text{ e} \models \eta\ \wedge\ \neg B \to \psi$$

Sistema $\mathcal{H}_g$

É fácil de demonstrar que a regra *cons* é derivável em $\mathcal{H}_g$.

Lema 23.1. *Se* $\Gamma \vdash_{\mathcal{H}_g} \{\varphi\}C\{\psi\}$ *e* $\models \varphi' \to \varphi$, $\models \psi \to \psi'$, *então* $\Gamma \vdash_{\mathcal{H}_g} \{\varphi'\}C\{\psi'\}$.

Demonstração: Por indução sobre a derivação $\Gamma \vdash_{\mathcal{H}_g} \{\psi\}C\{\varphi\}$. Vamos ver os casos para o `skip` e para a sequência.

- Para $C \equiv \texttt{skip}$, temos $\Gamma \vdash_{\mathcal{H}_g} \{\varphi\}\texttt{skip}\{\psi\}$, se $\models \varphi \to \psi$. Temos $\models \varphi' \to \varphi$, $\models \varphi \to \psi$ e $\models \psi \to \psi'$, logo $\models \varphi' \to \psi'$, o que significa que temos $\Gamma \vdash_{\mathcal{H}_g} \{\varphi'\}\texttt{skip}\{\psi'\}$.
- Para $C \equiv C_1; C_2$, temos $\Gamma \vdash_{\mathcal{H}_g} \{\varphi\}C_1; C_2\{\psi\}$, se $\Gamma \vdash_{\mathcal{H}_g} \{\varphi\}C_1\{\eta\}$ e $\Gamma \vdash_{\mathcal{H}_g} \{\eta\}C_2\{\psi\}$. Mas então por H.I. temos $\Gamma \vdash_{\mathcal{H}_g} \{\varphi'\}C_1\{\eta\}$ (uma vez que $\models \varphi' \to \varphi$ e $\models \eta \to \eta$) e $\Gamma \vdash_{\mathcal{H}_g} \{\eta\}C_2\{\psi'\}$ (uma vez que $\models \eta \to \eta$ e $\models \psi \to \psi'$), logo $\Gamma \vdash_{\mathcal{H}_g} \{\varphi'\}C_1; C_2\{\psi'\}$.

Exercício 23.1. *Completa a demonstração anterior.*

Equivalência $\mathcal{H}$ e $\mathcal{H}_g$

$\Gamma \vdash_{\mathcal{H}} \{\varphi\}C\{\psi\}$ se e só se $\Gamma \vdash_{\mathcal{H}_g} \{\varphi\}C\{\psi\}$

($\Rightarrow$) Por indução sobre a derivação $\Gamma \vdash_{\mathcal{H}} \{\psi\}C\{\varphi\}$, usando o lema anterior. Vamos ver os casos para atribuição e para a regra da consequência.

- Temos $\Gamma \vdash_{\mathcal{H}} \{\varphi[E/x]\}x \leftarrow E\{\varphi\}$ e $\models \varphi[E/x] \to \varphi[E/x]$, logo $\Gamma \vdash_{\mathcal{H}_g} \{\varphi[E/x]\}x \leftarrow E\{\varphi\}$
- Pela regra da consequência temos $\Gamma \vdash_{\mathcal{H}} \{\varphi\}C\{\psi\}$, se $\Gamma \vdash_{\mathcal{H}} \{\varphi'\}C\{\psi'\}$ e $\models \varphi \to \varphi'$, $\models \psi' \to \psi$.
  Por H.I. temos $\Gamma \vdash_{\mathcal{H}_g} \{\varphi'\}C\{\psi'\}$, logo pelo lema anterior temos $\Gamma \vdash_{\mathcal{H}_g} \{\varphi\}C\{\psi\}$.

($\Leftarrow$) Por indução sobre a derivação $\Gamma \vdash_{\mathcal{H}_g} \{\psi\}C\{\varphi\}$. Vamos ver os casos para a atribuição e para o condicional.

- Temos $\Gamma \vdash_{\mathcal{H}_g} \{\psi\}x \leftarrow E\{\varphi\}$ se $\models \psi \to \varphi[E/x]$. Como $\Gamma \vdash_{\mathcal{H}} \{\varphi[E/x]\}x \leftarrow E\{\varphi\}$ e $\models \psi \to \varphi[E/x]$ e $\models \psi \to \psi$, então pela regra da consequência, temos $\Gamma \vdash_{\mathcal{H}} \{\psi\}x \leftarrow E\{\varphi\}$.
- Temos $\Gamma \vdash_{\mathcal{H}_g} \{\psi\}\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2\, \{\varphi\}$, se $\Gamma \vdash_{\mathcal{H}_g} \{\psi \wedge B\}C_1\{\varphi\}$ e $\Gamma \vdash_{\mathcal{H}_g} \{\psi \wedge \neg B\}C_2\{\varphi\}$.
  Por H.I. $\Gamma \vdash_{\mathcal{H}} \{\psi \wedge B\}C_1\{\varphi\}$ e $\Gamma \vdash_{\mathcal{H}} \{\psi \wedge \neg B\}C_2\{\varphi\}$, logo $\Gamma \vdash_{\mathcal{H}} \{\psi\}\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2\, \{\varphi\}$

Exercício 23.2. *Completa a demonstração anterior.*

Pós e Contras

Vantagens de $\mathcal{H}_g$:

- Eliminamos a ambiguidade provocada pela regra *cons*.
- Eliminamos uma das regras sem a propriedade de sub-fórmula.

No entanto, ainda é necessário "adivinhar" pré-condições intermédias para *comp*.

A estratégia de pré-condição mais fraca

Queremos construir uma derivação para um triplo de Hoare $\{\varphi\}C\{\psi\}$, onde φ pode ou não ser conhecido (nesse caso escrevemos $\{?\}C\{\psi\}$).

1. Se φ for conhecido, então aplicamos a única regra possível de $\mathcal{H}_g$. Se C for $C_1; C_2$, então construímos uma sub-derivação da forma $\{?\}C_2\{\psi\}$. Eventualmente quando concluírmos esta derivação podemos prosseguir com $\{\varphi\}C_1\{\theta\}$, com θ obtido da sub-derivação anterior.

2. Se φ é desconhecido, a construção procede da mesma forma, excepto que no caso das regras `skip`, atribuição e ciclos, com uma condição auxiliar $\varphi \to \theta$, tomamos a pré-condição φ como θ.

Uma Arquitectura para Verificação de Programas

Dado um triplo de Hoare $\{\varphi\}C\{\psi\}$ e uma teoria $\mathcal{T}$:

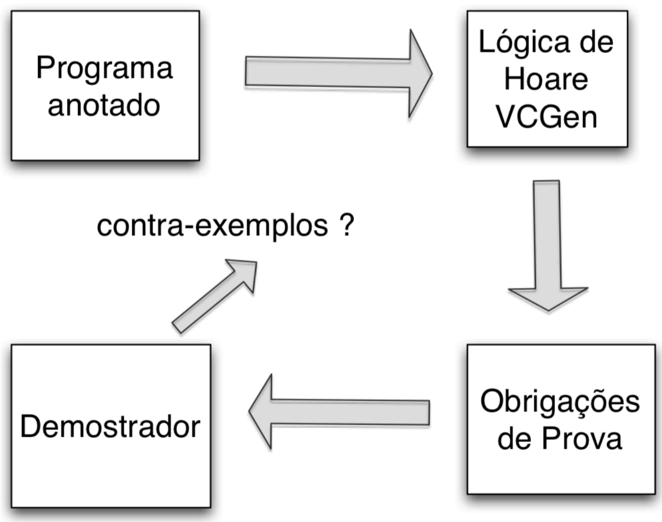
1. Aplicamos os princípios apresentados anteriormente para construir uma derivação com conclusão $\{\varphi\}C\{\psi\}$, assumindo que todas as condições auxiliares geradas no processo se verificam.

2. Cada fórmula de primeira ordem gerada como condição auxiliar (chamada neste contexto de *condição de verificação* (VC)) tem que ser verificada numa ferramenta de prova.

3. Se todas as condições de verificação são classificadas como $\mathcal{T}$-válidas, então $\mathcal{T} \vdash_{\mathcal{H}_g} \{\varphi\}C\{\psi\}$.

Nota: como não existe ambiguidade na construção das árvores, podemos eliminar essa parte do processo e simplesmente gerar as VC usando um *Gerador de condições de verificação*(VCGen).

Duas fases para a verificação



Um algoritmo VCGen: cálculo das pré-condições mais fracas (wp)

Dado um programa C e uma pós-condição ψ, podemos calcular $wp(C,\psi)$ tal que $\{wp(C,\psi)\}C\{\psi\}$ é válida e se $\{\varphi\}C\{\psi\}$ é válida para algum φ então $\varphi \rightarrow wp(C,\psi)$.

$$
\begin{array}{rcl}
wp(\mathbf{skip},\psi) & = & \psi \\
wp(x \leftarrow E,\psi) & = & \psi[E/x] \\
wp(C_1;C_2,\psi) & = & wp(C_1,wp(C_2,\psi)) \\
wp(\mathtt{if}\, B\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2,\psi) & = & (B \rightarrow wp(C_1,\psi)) \\
 & & \wedge(\neg B \rightarrow wp(C_2,\psi)) \\
wp(\mathbf{while}\, B\, \mathbf{do}\, \{\eta\}C,\psi) & = & \eta
\end{array}
$$

Algoritmo VCGen

Primeiro calcula as VC não considerando as pré-condições

$$
\begin{array}{rcl}
VC(\mathtt{skip},\psi) & = & \emptyset \\
VC(x \leftarrow E,\psi) & = & \emptyset \\
VC(C_1;C_2,\psi) & = & VC(C_1,wp(C_2,\psi)) \cup VC(C_2,\psi) \\
VC(\mathtt{if}\, B\, \mathtt{then}\, C_1\, \mathtt{else}\, C_2,\psi) & = & VC(C_1,\psi) \cup VC(C_2,\psi) \\
VC(\mathtt{while}\, B\, \mathtt{do}\, \{\eta\}C,\psi) & = & \{(\eta\ \wedge\ B) \rightarrow wp(C,\eta)\} \cup \\
 & & \{(\eta\ \wedge\ \neg B) \rightarrow \psi\} \cup VC(C,\eta)
\end{array}
$$

A pré-condição é tomada em consideração:

$$VCG(\{\varphi\}C\{\psi\}) = \{\varphi \to wp(C, \psi)\} \cup VC(C, \psi)$$

Exemplo

Seja fact o seguinte programa:

$f \leftarrow 1; i \leftarrow 1;$
while $i \leq n$ **do**
Ensure: $f = (i-1)! \wedge i \leq n+1$
$f \leftarrow f * i;$
$i \leftarrow i + 1;$

Vamos calcular

$$\text{VCG}(\{n \geq 0\}\mathsf{fact}\{f = n!\})$$

com $\theta = f = (i-1)! \wedge i \leq n+1$ e $C_w = f \leftarrow f * i; i \leftarrow i+1$

$$\begin{aligned}
& VC(\texttt{fact}, f = n!) \\
= \;& VC(f \leftarrow 1; i \leftarrow 1, wp(\textbf{while } i \leq n \textbf{ do}\{\theta\}C_w, f = n!)) \\
& \cup VC(\textbf{while } i \leq n \texttt{ do}\{\theta\}C_w, f = n!) \\
= \;& VC(f \leftarrow 1; i \leftarrow 1, \theta) \cup \{\theta \wedge i \leq n \to wp(C_w, \theta)\} \\
& \cup \{\theta \wedge i > n \to f = n!\} \cup VC(C_w, \theta) \\
= \;& VC(f \leftarrow 1, wp(i \leftarrow 1, \theta)) \cup VC(i \leftarrow 1, \theta) \\
& \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \to wp(f \leftarrow f * i; i \leftarrow i+1, \theta)\} \\
& \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i > n \to f = n!\} \\
& \cup VC(f = f * i, wp(i \leftarrow i+1, \theta)) \cup VC(i \leftarrow i+1, \theta) \\
= \;& \emptyset \cup \emptyset \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \\
& \qquad\qquad \to wp(f \leftarrow f * i, f = (i+1-1)! \wedge i+1 \leq n+1)\} \\
& \cup \{f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \to f = n!\} \cup \emptyset \cup \emptyset \\
= \;& \{f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \to f * i = (i+1-1)! \wedge i+1 \leq n+1, \\
& f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \to f = n!\}
\end{aligned}$$

$$
\begin{aligned}
& VCG(\{n \geq 0\}\mathsf{fact}\{f = n!\}) \\
= \ & \{n \geq 0 \rightarrow wp(\mathsf{fact}, f = n!)\} \cup VC(\mathsf{fact}, f = n!) \\
= \ & \{n \geq 0 \rightarrow wp(f \leftarrow 1; i \leftarrow 1; wp(\mathbf{while}\ i \leq n\ \mathbf{do}\{\theta\}C_w, f = n!), \\
& f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow f * i = (i+1-1)! \wedge i+1 \leq n+1, \\
& f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow f = n!\} \\
= \ & \{n \geq 0 \rightarrow wp(f \leftarrow 1; i \leftarrow 1; \theta), \\
& f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow f * i = (i+1-1)! \wedge i+1 \leq n+1, \\
& f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow f = n!\}
\end{aligned}
$$

Chegamos às seguintes obrigações de prova:

1. $n \geq 0 \rightarrow 1 = (1-1)! \wedge 1 \leq n+1$
2. $f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow f * i = (i+1-1)! \wedge i+1 \leq n+1)$
3. $f = (i-1)! \wedge i \leq n+1 \wedge i \leq n \rightarrow f = n!$

Arrays (*aliases*)

Se tivermos uma variável indexada $a[]$ a regra da atribuição não pode ser directamente aplicada a um elemento:

$$\{\varphi[E_2/a[E_1]]\}a[E_1] \leftarrow E_2\{\varphi\}$$

porque as modificações em $a[E_1]$ podem alterar outras referências a a que ocorram em φ ou em E_2.

Por exemplo, o triplo $\{a[j] > 100\}a[i] := 10\{a[j] > 100\}$ seria derivado pelo axioma acima, mas não é válido: por exemplo se for avaliado num estado em que $i = j$.

A solução de T. Hoare foi considerar os *Arrays* como monolíticos, e uma atribuição

$$a := a[E_1 \leftarrow E_2]$$

que significa que a passou a ser um *array* igual ao anterior mas em que a posição E_1 passou a valer E_2.

Assim no caso anterior o valor de $a[i]$ e de $a[j]$ mudam os dois...porque muda o próprio array...

Arrays

Lógica de Hoare

$[array_p\ ]$

$$\{\psi[a[E_1 \leftarrow E_2]/a]\}\, a[E_1] \leftarrow E_2\, \{\psi\}$$

onde E_1 é um inteiro.

E onde

$$\begin{array}{rcl} a[E_1 \leftarrow E_2][E_1] & = & E_2 \\ a[E_1 \leftarrow E_2][E_3] & = & a[E_3] \text{ se } E_3 \neq E_1. \end{array}$$

Exemplo

$$\begin{array}{l} \vdash_p \{a[x] = x \ \wedge \ a[y] = y\} \\ \quad \leftarrow a[x]; \\ \quad a[x] \leftarrow a[y]; \\ \quad a[y] \leftarrow r \\ \quad \{a[x] = y \ \wedge \ a[y] = x\} \end{array}$$

Nota: Actualmente nas implementações não se usa esta técnica porque é computacionalmente muito ineficiente!...

Condições de verificação para programas com arrays

Seja maxarray o seguinte programa anotado:

$max \leftarrow 0$;
$i \leftarrow 1$;
while $i < size$ **do** $\{1 \leq i \leq size \wedge 0 \leq max < i \wedge \forall a.0 \leq a < i \rightarrow u[a] \leq u[max]\}$
  if $u[i] > u[max]$ **then**
    $max \leftarrow i$
  else
    skip;
  $i \leftarrow i + 1$

Condições de verificação para programas com arrays

Queremos verificar que

$\{size \geq 1\}$ maxarray $\{0 \leq max < size \wedge \forall a.0 \leq a < size \rightarrow u[a] \leq u[max]\}$

Assumimos

$$\begin{array}{rcl} \eta & = & 1 \leq i \leq size \wedge 0 \leq max < i \wedge \forall a.0 \leq a < i \rightarrow u[a] \leq u[max] \\ C & = & \textbf{if } u[i] \ > \ u[max] \textbf{ then } max \leftarrow i \textbf{ else skip}; i \leftarrow i + 1 \\ \psi & = & 0 \leq max < size \wedge \forall a.0 \leq a < i \rightarrow u[a] \leq u[max] \end{array}$$

Extensão de VCGen para arrays

Adicionamos a seguinte regra a $\mathcal{H}_g$:

$$\frac{}{\{\varphi\}\, u[E] \leftarrow E'\, \{\psi\}} \text{ se } \models \varphi \rightarrow \psi[u[E \leftarrow E']/u]$$

extendemos wp e VC da seguinte forma:

$$\begin{array}{rcl} wp(u[E] \leftarrow E', \psi) & = & \psi[u[E \leftarrow E']/u] \\ VC(u[E] \leftarrow E', \psi) & = & \emptyset \end{array}$$

por exemplo:

$$wp(u[i] \leftarrow 10, u[j] > 100) = u[i \leftarrow 10][j] > 100$$

Exercícios

Usando o algoritmo VCGen calcula:

1. $VCG(\{u[j] > 100\}u[i] \leftarrow 10\{u[j] > 100\})$
2. $VCG(\{i \neq j \wedge u[j] > 100\}u[i] \leftarrow 10\{u[j] > 100\})$
3. $VCG(\{i = 70\}u[i] \leftarrow 10\{u[i] = 10\})$

Exercício 23.3. *Calcular as condições de verificação para o triplo* $\{size \geq 1\}maxarray\{\psi\}$ *usando o algoritmo VCGen.* $\diamond$

Propriedades de segurança

Na semântica que consideramos todas as expressões avaliam para um determinado valor e os comandos executam sem provocarem erros. Vamos considerar algumas alterações que aproximem a nossa linguagem de uma linguagem real:

- introdução de um novo valor semântico de **erro**;
- alteração da relação de avaliação para considerar avaliações de comandos que terminem com o estado **erro**;

Semântica de expressões com erros

$\mathcal{A} : \mathbf{Aexp} \rightarrow (\mathbf{State} \rightarrow (Z \cup \{\mathbf{erro}\}))$

$$
\begin{array}{rcl}
\mathcal{A}[\![n]\!]s & = & n \\
\mathcal{A}[\![x]\!]s & = & s(x) \\
\mathcal{A}[\![E_1 \odot E_2]\!]s & = & \begin{cases} \mathcal{A}[\![E_1]\!]s \odot \mathcal{A}[\![E_2]\!]s & \text{se} \mathcal{A}[\![E_1]\!]s \neq \mathbf{erro} \neq \mathcal{A}[\![E_2]\!]s \\ \mathbf{erro} & \text{caso contrário} \end{cases} \\
 & & \text{para } \odot \in \{+, -, \times\} \\
\mathcal{A}[\![E_1 \div E_2]\!]s & = & \begin{cases} \mathcal{A}[\![E_1]\!]s \div \mathcal{A}[\![E_2]\!]s & \text{se} \mathcal{A}[\![E_1]\!]s \neq \mathbf{erro} \neq \mathcal{A}[\![E_2]\!]s \\ & \text{ e } \mathcal{A}[\![E_2]\!]s \neq 0 \\ \mathbf{erro} & \text{caso contrário} \end{cases}
\end{array}
$$

Semântica das expressões Booleanas

$\mathbf{T} = \{\mathbf{V}, \mathrm{F}\}$, $\mathcal{B} : \mathbf{Bexp} \rightarrow (\mathbf{State} \rightarrow (\mathbf{T} \cup \{\mathbf{erro}\}))$

$$
\begin{array}{rcl}
\mathcal{B}[\![\mathtt{true}]\!]s & = & \mathbf{V} \\
\mathcal{B}[\![\mathtt{false}]\!]s & = & \mathrm{F} \\
\mathcal{B}[\![\neg b]\!]s & = & \begin{cases} \mathbf{V} & \text{se } \mathcal{B}[\![b]\!]s = \mathrm{F} \\ \mathrm{F} & \text{se } \mathcal{B}[\![b]\!]s = \mathbf{V} \\ \mathbf{erro} & \text{se } \mathcal{B}[\![b]\!]s = \mathbf{erro} \end{cases} \\
\mathcal{B}[\![E_1 \odot E_2]\!]s & = & \begin{cases} \mathcal{A}[\![E_1]\!]s \odot \mathcal{A}[\![E_2]\!]s & \text{se } \mathcal{A}[\![E_1]\!]s \neq \mathbf{erro} \neq \mathcal{A}[\![E_2]\!]s \\ \mathbf{erro} & \text{se caso contrário} \end{cases} \\
 & & \text{para } \odot \in \{=, <, \leq\}. \\
\mathcal{B}[\![b_1 \ \wedge \ b_2]\!]s & = & \begin{cases} \mathrm{F} & \text{se } \mathcal{B}[\![b_1]\!]s = \mathrm{F} \\ \mathbf{erro} & \text{se } \mathcal{B}[\![b_1]\!]s = \mathbf{erro} \\ \mathcal{B}[\![b_1]\!]s & \text{caso contrário} \end{cases}
\end{array}
$$

Semântica operacional natural com erros (*big-step*)

$$\langle \texttt{skip}, s\rangle \longrightarrow s$$

$$\langle x \leftarrow E, s\rangle \longrightarrow \begin{cases} s[\mathcal{A}[\![E]\!]s/x] & \text{se } \mathcal{A}[\![E]\!]s \neq \textbf{erro} \\ \textbf{erro} & \text{caso contrário} \end{cases}$$

$$\frac{\langle C_1, s\rangle \longrightarrow \textbf{erro}}{\langle C_1; C_2, s\rangle \longrightarrow \textbf{erro}}$$

$$\frac{\langle C_1, s\rangle \longrightarrow s' \ , \ \langle C_2, s'\rangle \longrightarrow s''}{\langle C_1; C_2, s\rangle \longrightarrow s''} \text{ se } s' \neq \textbf{erro}$$

$$\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \longrightarrow \textbf{erro} \text{ se } \mathcal{B}[\![B]\!]s = \textbf{erro}$$

Semântica operacional natural com erros (*big-step*)

$$\frac{\langle C_1, s\rangle \longrightarrow s'}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \longrightarrow s'} \text{ se } \mathcal{B}[\![B]\!]s = \textbf{V}$$

$$\frac{\langle C_2, s\rangle \longrightarrow s'}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, s\rangle \longrightarrow s'} \text{ se } \mathcal{B}[\![B]\!]s = \text{F}$$

$$\langle \texttt{while } B \texttt{ do } C, s\rangle \longrightarrow \textbf{erro} \text{ se } \mathcal{B}[\![B]\!]s = \textbf{erro}$$

$$\frac{\langle C, s\rangle \longrightarrow \textbf{erro}}{\langle \texttt{while } B \texttt{ do } C, s\rangle \longrightarrow \textbf{erro}} \text{ se } \mathcal{B}[\![B]\!]s = \textbf{V}$$

$$\frac{\langle C, s\rangle \longrightarrow s', \langle \texttt{while } B \texttt{ do } C, s'\rangle \longrightarrow s''}{\langle \texttt{while } B \texttt{ do } C, s\rangle \longrightarrow s''} \text{ se } \mathcal{B}[\![B]\!]s = \textbf{V}, s' \neq \textbf{erro}$$

$$\langle \texttt{while } B \texttt{ do } C, s\rangle \longrightarrow s \text{ se } \mathcal{B}[\![B]\!]s = \text{F}$$

Lógica de Hoare com condições de segurança: sistema $\mathcal{H}_s$

$$\frac{}{\{\varphi\}\, \mathsf{skip}\, \{\psi\}} \text{ se } \varphi \to \psi$$

$$\frac{}{\{\varphi\}\, x \leftarrow E\, \{\psi\}} \text{ se } \varphi \to \mathsf{safe}(E) \text{ e } \varphi \to \psi[E/x]$$

$$\frac{\{\varphi\}\, C_1\, \{\eta\} \qquad \{\eta\}\, C_2\, \{\psi\}}{\{\varphi\}\, C_1; C_2\, \{\psi\}}$$

$$\frac{\{\varphi \ \wedge\ B\}\, C_1\, \{\psi\} \qquad \{\varphi \ \wedge\ \neg B\}\, C_2\, \{\psi\}}{\{\varphi\}\, \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2\, \{\psi\}} \text{ se } \varphi \rightarrow \mathsf{safe}(B)$$

$$\frac{\{\eta \ \wedge\ B\}\, C\, \{\eta\}}{\{\psi\}\, \texttt{while}\, B\, \texttt{do}\, \{\eta\} C\, \{\varphi\}} \text{ se } \psi \rightarrow \eta,\ \eta \rightarrow \mathsf{safe}(B) \text{ e } \eta \ \wedge\ \neg B \rightarrow \varphi$$

Um algoritmo VCGen: cálculo das pré-condições mais fracas ($\mathbf{wp}^s$)

$$\begin{array}{rcl}
wp^s(\texttt{skip}, \varphi) & = & \varphi \\
wp^s(x \leftarrow E, \varphi) & = & \mathsf{safe}(E) \wedge \varphi[E/x] \\
wp^s(C_1; C_2, \varphi) & = & wp^s(C_1, wp(C_2, \varphi)) \\
wp^s(\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2, \varphi) & = & \mathsf{safe}(B) \wedge (B \rightarrow wp^s(C_1, \varphi) \\
 & & \wedge\ (\neg B \rightarrow wp^s(C_2, \varphi) \\
wp^s(\texttt{while}\, B\, \texttt{do}\, \{\eta\} C, \varphi) & = & \eta
\end{array}$$

Algoritmo VCGen

Calcula as VC não considerando as pré-condições

$$\begin{array}{rcl}
VC^s(\texttt{skip}, \varphi) & = & \emptyset \\
VC^s(x \leftarrow E, \varphi) & = & \emptyset \\
VC^s(C_1; C_2, \varphi) & = & VC^s(C_1, wp^s(C_2, \varphi)) \cup \\
 & & VC^s(C_2, \varphi) \\
VC^s(\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2, \varphi) & = & VC^s(C_1, \varphi) \cup VC^s(C_2, \varphi) \\
VC^s(\texttt{while}\, B\, \texttt{do}\, \{\eta\} C, \varphi) & = & \{\eta \rightarrow \mathsf{safe}(B)\} \cup \\
 & & \{(\eta \ \wedge\ B) \rightarrow wp^s(C, \eta)\} \cup \\
 & & \{(\eta \ \wedge\ \neg B) \rightarrow \varphi\} \cup VC^s(C, \eta)
\end{array}$$

Definimos VCG^s como:

$$VCG^s(\{\psi\} C \{\varphi\}) = \{\psi \rightarrow wp^s(C, \varphi)\} \cup VC^s(C, \varphi)$$

A função safe para a linguagem While$^{\texttt{int}}$

$$
\begin{array}{rcl}
\mathsf{safe}(n) & = & \mathsf{true} \\
\mathsf{safe}(x) & = & \mathsf{true} \\
\mathsf{safe}(-E) & = & \mathsf{safe}(E) \\
\mathsf{safe}(E_1 \odot E_2) & = & \mathsf{safe}(E_1) \wedge \mathsf{safe}(E_2) \\
 & & \text{com } \odot \in \{+, -, \times, =, <, \leq\} \\
\mathsf{safe}(E_1 \div E_2) & = & \mathsf{safe}(E_1) \wedge \mathsf{safe}(E_2) \wedge E_2 \neq 0 \\
\mathsf{safe}(\neg B) & = & \mathsf{safe}(B) \\
\mathsf{safe}(B_1 \wedge B_2) & = & \mathsf{safe}(B_1) \wedge (B_1 \rightarrow \mathsf{safe}(B_2)) \\
\mathsf{safe}(B_1 \ \vee \ B_2) & = & \mathsf{safe}(B_1) \wedge (\neg B_1 \rightarrow \mathsf{safe}(B_2))
\end{array}
$$

Temos que

$$\mathcal{A}[\![E]\!]s \neq \texttt{erro} \text{ se e só se } [\![\mathsf{safe}(E)]\!]s = \mathsf{true}.$$

Exemplos:

$$
\begin{array}{rcl}
\mathsf{safe}((x \div y) > 2) & = & \mathsf{safe}(x) \wedge \mathsf{safe}(y) \wedge y \neq 0 \wedge \mathsf{safe}(2) \\
 & = & \mathsf{true} \wedge \mathsf{true} \wedge y \neq 0 \wedge \mathsf{true} \\
 & \equiv & y \neq 0
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{safe}(7 > x \wedge (x \div y) > 2) & = & \mathsf{safe}(7 > x) \wedge \\
 & & (\mathsf{safe}(7 > x) \rightarrow \mathsf{safe}((x \div y) > 2) \\
 & = & \mathsf{true} \wedge \mathsf{true} \wedge) \\
 & & (7 > x \rightarrow (\mathsf{true} \wedge \mathsf{true} \wedge y \neq 0 \wedge \mathsf{true})) \\
 & \equiv & 7 > x \rightarrow y \neq 0
\end{array}
$$

Ferramentas de verificação dedutiva de programas

- Anotação de programas
- Geração automática de obrigações de prova
- Utilização de demonstradores

  automáticos: Simplify, Yices, Alt-ergo, CVC3, Z3 etc.
  interactivos: Coq, Isabelle, Mizar, etc.
- Frameworks: Frama-C, Why3, *Dafny*

Dafny (`www.rise4fun.com/Dafty`)

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y
   ensures 0 <= x ==> x == y
   ensures x < 0 ==> y == -x
{
   if x < 0
     { return -x; }
   else
     { return x; }
}
```

Dafny:Keywords

- `requires`: pré-condição
- `ensures`: pós-condição
- `invariant`: invariante
- `decreases`: variante
- `assert`: condição que se tem de verificar sempre
- `reads`: para ser usado para indicar quais as posições de memória a que uma condição (definida por funções ou predicados) pode aceder. Corresponde ao *frame* da condição.

Dafny- Programas Imperativos

- verifica a correção de funções em relação às pós-condições anotadas
- `method` define uma sequência de código executável
- sendo dado o tipo dos argumentos
- e `returns`
- indica o tipo e a variável que retorna
- `method M(a: A, b: B, c: C) returns (x: X, y: Y, z: Y)`
- declaração de variáveis locais : `var x:T`

Funções

Definem a semântica que se pretende do programa imperativo

```
function Factorial(n: int): int
   requires 0 < = n
   ensures 1 <= Factorial(n)
  {if n == 0 then 1 else Factorial(n-1) * n}
```

Fibonacci

```
function fib(n: nat): nat
{
   if n == 0 then 0 else
   if n == 1 then 1 else
                  fib(n - 1) + fib(n - 2)
}
method ComputeFib(n: nat) returns (b: nat)
   ensures b == fib(n)
{
   if n == 0 { return 0; }
   var i: int := 1;
   var a := 0;
       b := 1;
   while i < n
      invariant 0 < i <= n
      invariant a == fib(i - 1)
      invariant b == fib(i)
   {
      a, b := b, a + b;
      i := i + 1;
   }
}
```

[fragile]Arrays

```
method Find(a: array<int>, key: int) returns (i: int)
```

- `a.Lenght` : indica sempre o tamanho do array
- Nas condições podemos usar quantificadores sobre o valor das variáveis
- `forall k: int :: 0 <= k < a.Length ==>  a[k] != key`

Procura de um valor num array

```
method Find(a: array<int>, key: int)
                 returns (index: int)
   ensures 0 <= index ==> index < a.Length
       && a[index] == key
   ensures index < 0 ==>
 forall k :: 0 <= k < a.Length ==>
              a[k] != key
{
   index := 0;
   while index < a.Length
      invariant 0 <= index <= a.Length
      invariant forall k :: 0 <= k < index ==> a[k] != key
   {
      if a[index] == key { return; }
      index := index + 1;
   }
   index := -1;
}
```

Máximo de um array

```
method maxarray(arr:array? <int>) returns(max:int)
 requires arr!=null && arr.Length > 0
ensures 0<= max < arr.Length
 ensures (forall j :int :: (j >= 0 && j < arr.Length
                    ==> arr[max] >= arr[j]))
{
  max:=0;
  var i:int :=1;
  while(i < arr.Length)
  invariant (1<=i<=arr.Length)
  invariant 0<= max < i
  invariant (forall j:int :: j>=0 && j<i ==>
     arr[max] >= arr[j]);
    decreases (arr.Length-i);
  {
    if(arr[i] > arr[max]){max := i;}
    i := i + 1;
  }
}
```

Predicados

Permitem escrever condições (pós/pré). Predicados são funções que retornam um valor booleano (e portanto é omitido na especificação).

```
predicate sorted(a: array?<int>)
   requires a != null
   reads a
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
```

Pesquisa Binária

```
method BinarySearch(a: array?<int>, value: int) returns (index: int)
   requires a != null && 0 <= a.Length && sorted(a)
   ensures 0 <= index ==> index < a.Length && a[index] == value
   ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
   var low, high := 0, a.Length;
   while low < high
      invariant 0 <= low <= high <= a.Length
      invariant forall i ::
         0 <= i < a.Length && !(low <= i < high) ==> a[i] != value
   {
      var mid := (low + high) / 2;
      if a[mid] < value
      {
         low := mid + 1;
      }
      else if value < a[mid]
      {
         high := mid;
      }
      else
      {
         return mid;
      }
   }
   return -1;
}
```