LÓGICA COMPUTACIONAL VERIFICACIÓN DE PROGRAMAS Y OTRAS APLICACIONES DE LÓGICA MODAL

Francisco Hernández Quiroz

Departamento de Matemáticas
Facultad de Ciencias, UNAM
E-mail: fhq@ciencias.unam.mx
Página Web: www.matematicas.unam.mx/fhq

Facultad de Ciencias

Lógica temporal

La lógica temporal formaliza una noción de tiempo *discreta*. Existen al menos dos versiones:

- LTL lógica temporal de tiempo lineal;
- CTL lógica temporal de tiempo ramificado.

En ambas un estado en el tiempo corresponde a un mundo en un universo de Kripke.
La segunda considera la posibilidad de que el tiempo "se ramifique": puede haber varios futuros posibles y se pueden hacer afirmaciones sobre qué pasa en las otras "ramas" del tiempo.
Aquí se abordará sólo la primera lógica, por simplicidad.

Sintaxis de LTL

La sintaxis incluye operadores adicionales a los de la lógica modal:

$$\alpha ::= p_i \mid q_i \mid r_i \mid \neg\alpha \mid \cdots \mid \mathsf{X}\,\alpha \mid \mathsf{G}\,\alpha \mid \mathsf{F}\,\alpha \mid \alpha\,\mathsf{U}\,\beta.$$

Los operadores modales se leerán ahora de la siguiente forma:

$\mathsf{X}\,\alpha$	"α vale en el siguiente estado"
$\mathsf{G}\,\alpha$	"α vale en todos los estados futuros"
$\mathsf{F}\,\alpha$	"α vale en algún estado futuro"
$\alpha\,\mathsf{U}\,\beta$	"α vale en este estado y en estados futuros al menos hasta el momento en que β valga" o bien "α vale hasta que β valga".

Semántica de la lógica temporal

Para definir la semántica, se utilizará un marco de Kripke. $\to$ representa la relación de accesibilidad y $\to^*$, su cerradura transitiva y reflexiva.

$$\begin{array}{lll}
\mathscr{F}, e, w \vDash p & \text{sii} & e(p, w) = V \qquad \forall p \in P_0 \\
& \dots & \\
\mathscr{F}, e, w \vDash \mathsf{X}\,\alpha & \text{sii} & \forall v \in W.\, \text{si } w \to v \text{ entonces } \mathscr{F}, e, v \vDash \alpha \\
\mathscr{F}, e, w \vDash \mathsf{G}\,\alpha & \text{sii} & \forall v \in W.\, \text{si } w \to^* v \text{ entonces } \mathscr{F}, e, v \vDash \alpha \\
\mathscr{F}, e, w \vDash \mathsf{F}\,\alpha & \text{sii} & \exists v \in W.\, w \to^* v \text{ y } \mathscr{F}, e, v \vDash \alpha \\
\mathscr{F}, e, w \vDash \alpha\,\mathsf{U}\,\beta & \text{sii} & \exists v \in W.\, w \to^* v \text{ y } \mathscr{F}, e, v \vDash \beta \text{ y} \\
 & & \forall u \in W.\, \text{si } w \to^* u \text{ y } u \to^* v \text{ entonces} \\
 & & \mathscr{F}, e, u \vDash \alpha
\end{array}$$

Semántica alternativa

Una forma alternativa de semántica (útil sobre todo para CTL) se basa en el concepto de *camino* o *trayectoria*:

Definición 6.1

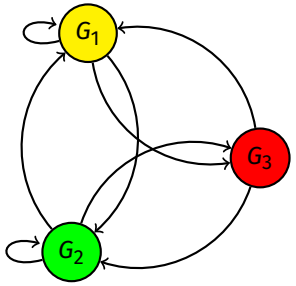
Un camino es una sucesión de mundos posibles $s_1, s_2, \ldots$ tales que $s_i \to s_{i+1}$. Usaremos π (con subíndices) para denotar caminos.
Si $\pi \equiv_{def} s_1 \to s_2 \to s_3 \to \cdots$, entonces $\pi^2 \equiv_{def} s_2 \to s_3 \to \cdots$

De esta forma se define la satisfacción por medio de caminos:

$$\begin{array}{lll}
\mathcal{F}, e, \pi \vDash p & \text{sii} & \mathcal{F}, e, s_1 \vDash p \\
\ldots & & \\
\mathcal{F}, e, \pi \vDash \mathsf{X}\,\alpha & \text{sii} & \mathcal{F}, e, \pi^2 \vDash \alpha \\
\mathcal{F}, e, \pi \vDash \mathsf{G}\,\alpha & \text{sii} & \forall\, i.\, 1 \leq i \Rightarrow \mathcal{F}, e, \pi^i \vDash \alpha \\
\mathcal{F}, e, \pi \vDash \mathsf{F}\,\alpha & \text{sii} & \exists\, i.\, 1 \leq i \wedge \mathcal{F}, e, \pi^i \vDash \alpha \\
\mathcal{F}, e, \pi \vDash \alpha\,\mathsf{U}\,\beta & \text{sii} & \exists\, i.\, \mathcal{F}, e, \pi^i \vDash \beta \wedge \forall\, j.\, j \leq i \Rightarrow \mathcal{F}, e, \pi^j \vDash \alpha
\end{array}$$

Una aplicación

Las redes genéticas suelen representarse de manera simplificada por diagramas:



Cada nodo en la gráfica representa un gen y las flechas indican interacciones. Por ejemplo, el gen G_1 envía una señal al gen G_2.
Los genes pueden estar activos o inactivos en un estado y cambiar en el siguiente.

Tablas auxiliares I

- Sin embargo, las interacciones suelen ser más complejas de los que tales diagramas indican.
- Por ejemplo, el gen G_2 está influido por los genes G_1 y G_3, pero dicha interacción se refleja mejor en la fórmula boolena

$$G_1 \wedge \neg G_3$$

que puede leerse como

"una señal de G_1 y la ausencia de señal de G_3 activarán a G_2".

- Por esta razón, los diagramas suelen ir acompañados de tablas como esta:

Tablas auxiliares II

Edo	G_1	G_2	G_3	$G_1' = G_1 \vee G_2 \vee G_3$	$G_2' = G_1 \wedge \neg G_3$	$G_3' = \neg G_1 \vee \neg G_2 \vee G_3$	Edo sig
s_1	1	1	1	1	0	1	s_3
s_2	1	1	0	1	1	0	s_2
s_3	1	0	1	1	0	1	s_3
s_4	1	0	0	1	1	1	s_1
s_5	0	1	1	1	0	1	s_3
s_6	0	1	0	1	0	1	s_3
s_7	0	0	1	1	0	1	s_3
s_8	0	0	0	0	0	1	s_7

- 1 equivale a activo y 0, a inactivo;
- G_i' designa el siguiente estado del gen G_i;
- "Edo" es una abreviatura de estado y "sig", de siguiente.

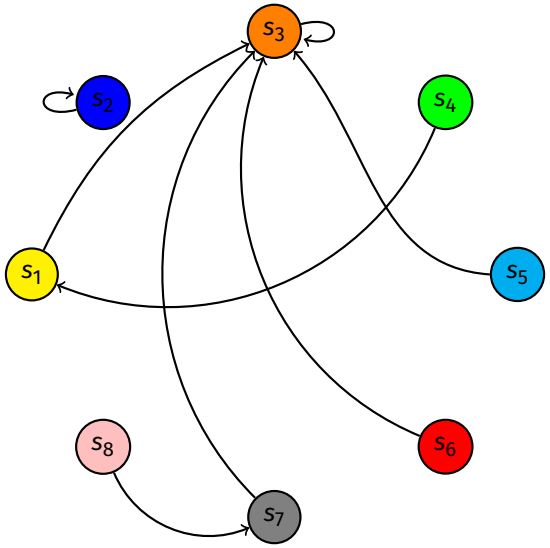
Marcos de Kripke, de nuevo I

Si consideramos

- las fórmulas atómicas G_1, G_2 y G_3;
- los estados s_1–s_8 como mundos posibles;
- los valores de los estados como codificaciones de una evaluación;
- y que esta evaluación contempla todas las combinaciones de valores de verdad posibles para las fórmulas atómicas;
- y que la primera y la segunda columna definen implícitamente una relación de accesibilidad;

obtenemos una gráfica del marco y la evaluación:

Marcos de Kripke, de nuevo II



Verificación de modelos

Podemos ahora verificar la satisfacción de algunas fórmulas por el modelo anterior.
Sean $\pi_1 = s_1 \to s_3 \to s_3 \to \cdots$ y $\pi_8 = s_8 \to s_7 \to s_3 \to s_3 \to \cdots$. Entonces:

- $\pi_1 \vDash G_2$, pues G_2 está activo en s_1.
- $\pi_1 \vDash \mathrm{X}\,\neg G_2$, pues $\pi_1^2 \vDash \neg G_2$, ya que G_2 está inactivo en s_3.
- $\pi_8 \vDash \mathrm{G}\,\neg G_2$, pues $\forall\, i,\ \pi_8^i \vDash \neg G_2$, ya que G_2 permanece inactivo en todos los estados de la trayectoria π_8.
- $\pi_8 \vDash \neg G_2\ \mathrm{U}\ G_1$, porque $\pi_8^3 \vDash G_1$ y $\pi_8, \pi_8^2 \vDash \neg G_2$.

Concurrencia

- La mayoría de los sistemas de cómputo actuales no se basan en la ejecución *secuencial* de instrucciones.
- En general contienen subsistemas que actúan de manera simultánea y que intercambian información. Por eso se conocen como *sistemas concurrentes*.
- Hay dos modelos fundamentales de cómo se realiza el intercambio de información:
  1. memoria compartida;
  2. envío de mensajes.
- En el primero, hay un área de la memoria (variables, buffers, etc.) a la que varios sistemas concurrentes tienen acceso para guardar o leer datos.
- El envío de mensajes se realiza por un medio abierto (difusión) o por medio de canales específicos.

Communicating Concurrent Systems

En los 80, Robin Milner propuso un lenguaje de especificación para procesos concurrentes: ccs (Communicating Concurrent Systems). Dicho lenguaje consta de:

- Un conjunto de valores (enteros, por ejemplo) que se envían o reciben a través de canales α, β, etc.
- Operadores para la composición secuencial o paralela de procesos.
- Un operador de "elección"
- Operaciones para reetiquetar canales y restringir el acceso a éstos.

Sintaxis I

Para comenzar, las acciones básicas de comunicación entre procesos:

$$\lambda ::= \tau \mid \alpha!m \mid \alpha?m$$

τ es la acción nula, $\alpha!m$ es el envío de m por el puerto de salida del canal α y $\alpha?m$ es la recepción de m por el puerto de entrada del mismo canal. Los procesos de ccs se definen así:

$$p ::= nil \mid \lambda.p \mid \sum_{i \in I} p_i \mid p \parallel p' \mid p\backslash L \mid p[f] \mid P$$

En términos informales, la sintaxis contiene los siguientes elementos:

nil es un proceso que no realiza ninguna acción y se considera primitivo;

λ es una acción;

$\sum$ es la elección arbitraria de un proceso tomado del conjunto de procesos indexado por I;

Sintaxis II

$\parallel$ es la composición paralela de procesos (intuitivamente, ambos procesos pueden realizarse simultáneamente);

L es un conjunto de canales (cuyo acceso se restringirá a otros procesos, como se verá más adelante);

P designa a un proceso definido por medio de una expresión del tipo

$$P \equiv_{def} p$$

que puede tener carácter recursivo.

Semántica operacional estructural I

- La forma más común de definir el significado de las construcciones de un lenguaje es por medio de explicaciones en una lengua natural (inglés, generalmente).
- Sin embargo, el lenguaje natural se presta a las ambigüedades y éstas, a los errores o a las divergencias entre las diferentes implementaciones de un mismo lenguaje.
- Para evitar estos problemas, aquí se usarán *reglas de semántica operacional estructural* o SOE, para abreviar.
- Una regla de inferencia tiene la forma siguiente:

$$\frac{p_1, \ldots, p_n}{q}$$

donde $p_1, \ldots, p_n$ son las premisas y q es la conclusión.

Semántica operacional estructural II

- Las reglas de SOE son un tipo particular de reglas de inferencia en las que las premisas y la conclusión son *transiciones*.
- Una *transición* tiene la forma

$$\alpha \rightarrow \beta$$

- La forma concreta de las expresiones α y β, así como las transiciones aceptables, dependen del lenguaje de programación específico.

Semántica de ccs I

El el caso de ccs, las transiciones tienen la forma siguiente

$$P \xrightarrow{\lambda} P'$$

donde P y P son procesos de ccs y λ es una acción básica. Intuitivamente, la regla quiere decir "El proceso P se transforma en el proceso P' después de ejecutar la acción λ". Las siguientes reglas SOE define cómo se comportan los procesos:

Procesos resguardados

$$\lambda.p \xrightarrow{\lambda} p$$

Sumas

Semántica de ccs II

$$\frac{p_j \xrightarrow{\lambda} q}{\sum_{i \in I} p_i \xrightarrow{\lambda} q} \quad j \in I$$

Composición paralela

$$\frac{p_0 \xrightarrow{\lambda} p_0'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0' \parallel p_1} \qquad \frac{p_1 \xrightarrow{\lambda} p_1'}{p_0 \parallel p_1 \xrightarrow{\lambda} p_0 \parallel p_1'} \qquad \frac{p_0 \xrightarrow{\lambda} p_0' \quad p_1 \xrightarrow{\bar{\lambda}} p_1'}{p_0 \parallel p_1 \xrightarrow{\tau} p_0' \parallel p_1'}$$

Restricción

$$\frac{p \xrightarrow{\lambda} q}{p \backslash L \xrightarrow{\lambda} q \backslash L} \quad \lambda \notin L \cup \bar{L}$$

Reetiquetamiento

Semántica de ccs III

$$\frac{p \xrightarrow{\lambda} q}{p[f] \xrightarrow{f(\lambda)} q[f]}$$

Identificadores

$$\frac{p \xrightarrow{\lambda} q}{P \xrightarrow{\lambda} q} \quad \text{donde } P \equiv_{def} p.$$

Ejemplo

Veamos un ejemplo muy sencillo: un *buffer* de capacidad 1.

- Un *buffer* es un dispositivo de almacenamiento con capacidad finita en el que un proceso (*el productor*) puede guardar información a la espera de que otro proceso (*el consumidor*) lea esta información.
- La lectura *destruye* la información guardada.
- Es importante que la información se guarde y se consuma de acuerdo con los permisos de los procesos.
- El siguiente proceso de ccs se comporta de acuerdo con la descripción anterior:

$$B \equiv_{def} ((\alpha?m.\,\beta!m.\,nil) \parallel (\beta?m.\,\gamma!m.\,B))\backslash\{\beta\}$$

Una lógica para ccs

Ahora definiremos un lenguaje simple de especificación de propiedades por parte de programas en ccs. La sintaxis es muy simple:

$$D ::= V \mid F \mid \neg D \mid D \vee D \mid D \wedge D \mid \langle\lambda\rangle D \mid \langle\cdot\rangle D \mid \mu X.\,D$$

y las siguientes abreviaturas

$$[\lambda]D \equiv_{def} \neg\langle\lambda\rangle\neg D \qquad [\cdot]D \equiv_{def} \neg\langle\cdot\rangle\neg D$$

Semántica

La relación de satisfacción se define de manera inductiva a partir de la estructura del lenguaje L(ccs):

$$\begin{array}{lcll}
p & \models & V & \forall p \in \text{ccs} \\
p & \not\models & F & \forall p \in \text{ccs} \\
p & \models & \langle\lambda\rangle D & \text{sii} \quad \exists p'.\, p \xrightarrow{\lambda} p' \wedge p' \models D \\
p & \models & \langle\cdot\rangle D & \text{sii} \quad \exists p', \lambda.\, p \xrightarrow{\lambda} p' \wedge p' \models D \\
p & \models & [\lambda]D & \text{sii} \quad \forall p'.\, p \xrightarrow{\lambda} p' \Rightarrow p' \models D \\
p & \models & [\cdot]D & \text{sii} \quad \forall p', \lambda.\, p \xrightarrow{\lambda} p' \Rightarrow p' \models D \\
p & \models & \mu X.\,D & \text{sii} \quad p \models D_{[X:=\mu X.D]}
\end{array}$$

Ejemplos I

El programa $(\alpha?m.\,nil) \parallel (\beta!n.\,nil)$ satisface la fórmula $\langle\alpha?m\rangle[\cdot]V$ pues una posible transición es la siguiente

$$(\alpha?m.\,nil) \parallel (\beta!n.\,nil) \xrightarrow{\alpha?m} nil \parallel (\beta!n.\,nil)$$

y este último programa sólo puede ejecutar una acción, por lo que la única transición posible es

$$nil \parallel (\beta!n.\,nil) \xrightarrow{\beta!n} nil \parallel nil$$

y $nil \parallel nil \models V$. En cambio, el mismo programa no satisface la fórmula $[\cdot]\langle\alpha?m\rangle V$, pues si tenemos primero la transición

$$(\alpha?m.\,nil) \parallel (\beta!n.\,nil) \xrightarrow{\alpha?m} nil \parallel (\beta!n.\,nil)$$

Ejemplos II

(recuérdese que el operador $[\cdot]$ nos obliga a considerar todas las transiciones posibles) tendremos que

$$nil \parallel (\beta!n.\, nil) \not\models \langle\alpha?m\rangle V.$$

Considérense ahora los programas y proposiciones siguientes:

$$\begin{aligned} P &\equiv_{def} (\alpha?n.\, nil) \parallel (\alpha?n.\, \beta!n.\, P) \\ P_1 &\equiv_{def} \beta!n.\, P \\ \Psi &\equiv_{def} \mu X.\, \langle\cdot\rangle\langle\beta!n\rangle V \vee \langle\cdot\rangle X \\ \Phi &\equiv_{def} \mu X.\, [\cdot]\langle\beta!n\rangle V \vee [\cdot]\langle\cdot\rangle X \end{aligned}$$

Para saber si $P \models \Psi$ tenemos que saber si

$$P \models (\langle\cdot\rangle\langle\beta!n\rangle V \vee \langle\cdot\rangle X)_{[X:=\Psi]}$$

Ejemplos III

Es decir, si $P \models \langle\cdot\rangle\langle\beta!n\rangle V$ o $P \models \langle\cdot\rangle\Psi$. De acuerdo con las definiciones correspondientes, existen P' y λ tales que $P \xrightarrow{\lambda} P'$ y $P' \models \langle\beta!n\rangle V$, a saber, $P' = P_1$ y $\lambda = \alpha?n$. Entonces $P \models \Psi$.
Por otro lado, $P \models \Phi$ si y sólo si

$$P \models ([\cdot]\langle\beta!n\rangle V \wedge [\cdot]\langle\cdot\rangle X)_{[X:=\Phi]}$$

lo que se reduce a $P \models [\cdot]\langle\beta!n\rangle V$ o $P \models [\cdot]\langle\cdot\rangle\Phi$. Pero el operador $[\cdot]$ nos obliga a considerar todas las transiciones posibles y, en particular, tenemos el caso en que $P \xrightarrow{\alpha?n} nil$ y $nil \not\models \langle\beta!n\rangle V$ y $nil \not\models \langle\cdot\rangle\Phi$. Por tanto, $P \not\models \Phi$.

Lenguaje de programación IMP

Se trata de un CFL. Empezaremos con las expresiones aritméticas *EA*:

$$A \to n \mid X \mid (A + A) \mid (A - A) \mid (A \times A)$$

donde $n \in \mathbb{Z}$ y $X \in \text{Loc}$.
Tenemos ahora las expresiones booleanas *EB*:

$$B \to V \mid F \mid (A = A) \mid (A < A) \mid \neg B \mid (B \wedge B) \mid (B \vee B)$$

donde $A \in EA$. Finalmente, los comandos del lenguaje se definen así:

$$C \to \textbf{skip} \mid X := A \mid (C\,;\,C) \mid (\textbf{if } B \textbf{ then } C \textbf{ else } C) \mid (\textbf{while } B \textbf{ do } C).$$

donde $A \in EA$ y $B \in EB$.

¿Qué es la programación imperativa?

- IMP es un lenguaje imperativo típico: el comando básico es la asignación de valor a las localidades de la memoria de la computadora.
- Para definir el significado de una asignación se parte de la existencia de una *máquina virtual* que posee una memoria con localidades con nombre:
$$\text{Loc} = \{X, Y, Z, X_0, \ldots\}$$
- Las localidades pueden tomar valores de $\mathbb{Z}$.

Estados de la memoria

- Un *estado de la memoria* es una función $\sigma : \text{Loc} \to \mathbb{Z}$.
- La expresión

$$\sigma(X)$$

nos dice qué valor contiene la localidad X en el estado σ.

- La alteración del valor de una sola localidad de la memoria cuando se encuentra en el estado σ se representa con la función

$$\sigma_{[X:=m]},$$

donde X es la localidad modificada y m es el nuevo valor.

- Formalmente:

$$\sigma_{[X:=m]}(Y) = \begin{cases} \sigma(Y) & \text{si } X \neq Y \\ m & \text{en caso contrario.} \end{cases}$$

- El conjunto de estados es Σ.

SOE para IMP. Expresiones aritméticas

En el caso de las expresiones aritméticas, las transiciones $\alpha \to \beta$ tiene la forma

$$\langle a, \sigma \rangle \to n$$

donde $a \in EA$, $\sigma \in \Sigma$ y $n \in \mathbb{Z}$.
Diremos que "*evaluar* operacionalmente la expresión a en el estado σ da como resultado el valor n".
Dicha evaluación se realiza de acuerdo con las siguientes reglas:

$$\langle n, \sigma \rangle \to n \qquad \langle X, \sigma \rangle \to \sigma(X)$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1 \quad n_0 \; op_{\mathbb{Z}} \; n_1 = n}{\langle a_0 \; \mathbf{op} \; a_1, \sigma \rangle \to n}$$

En esta regla **op** puede ser +, − o × y $op_{\mathbb{Z}}$, $+_{\mathbb{Z}}$, $-_{\mathbb{Z}}$ o $\times_{\mathbb{Z}}$ (ambos operadores deben coincidir).

SOE para IMP. Expresiones booleanas I

En las expresiones booleanas, una transición es $\langle b, \sigma \rangle \to T$, con $b \in EB$, $\sigma \in \Sigma$ y $T \in \{V, F\}$. Las reglas operacionales son:

$$\langle V, \sigma \rangle \to V \qquad \langle F, \sigma \rangle \to F$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to V} \quad sii \quad n \text{ y } m \text{ son iguales}$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to F} \quad sii \quad n \text{ y } m \text{ no son iguales}$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 < a_1, \sigma \rangle \to V} \quad sii \quad n \text{ es menor que } m$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 < a_1, \sigma \rangle \to F} \quad sii \quad n \text{ no es menor que } m$$

SOE para IMP. Expresiones booleanas II

$$\frac{\langle b, \sigma \rangle \to V}{\langle \neg b, \sigma \rangle \to F} \qquad \frac{\langle b, \sigma \rangle \to F}{\langle \neg b, \sigma \rangle \to V}$$

$$\frac{\langle b_0, \sigma \rangle \to T_0 \quad \langle b_1, \sigma \rangle \to T_1}{\langle b_0 \wedge b_1, \sigma \rangle \to T} \quad \begin{array}{l} \text{con } T = V \text{ si } T_0, T_1 = V \\ \text{y } T = F \text{ en caso contrario} \end{array}$$

$$\frac{\langle b_0, \sigma \rangle \to T_0 \quad \langle b_1, \sigma \rangle \to T_1}{\langle b_0 \wedge b_1, \sigma \rangle \to T} \quad \begin{array}{l} \text{con } T = V \text{ si } T_0 = V \text{ o } T_1 = V \\ \text{y } T = F \text{ en caso contrario} \end{array}$$

SOE para IMP. Comandos I

Las reglas de transición de los comandos son así:

$$\langle c, \sigma \rangle \to \sigma',$$

donde $c \in \text{IMP}$ y $\sigma, \sigma' \in \Sigma$.
El valor de σ' se infiere de este modo:

Comandos atómicos

$$\langle \textbf{skip}, \sigma \rangle \to \sigma \qquad \frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma_{[X:=m]}}$$

Composición secuencial

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0 \,;\, c_1, \sigma \rangle \to \sigma'}$$

SOE para IMP. Comandos II

Condicionales

$$\frac{\langle b, \sigma \rangle \to V \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma'} \qquad \frac{\langle b, \sigma \rangle \to F \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma'}$$

Ciclos

$$\frac{\langle b, \sigma \rangle \to F}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to V \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma'' \rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma'}$$

Ejemplo

Dadas las reglas anteriores, no es difícil ver que el programa de IMP siguiente calcula el factorial del número n (si $n \geq 0$) y lo guarda en la localidad Y:

$X := 1;$
$Y := 1;$
while $X < n$ **do**
  $(X := X + 1;$
  $Y := Y \times X)$

Verificación de programas contra *debugging*

- Una práctica muy extendida consiste realizar ensayos (“pruebas”) de un programa para ver si realmente cumple la tarea para la que fue diseñado.
- Esta práctica tiene una limitación seria: un fracaso es evidencia de que existía un error, pero la ausencia de fracaso no constituye una prueba de que no existan errores: un ensayo no puede explorar todas las situaciones posibles en las que se ejecuta un programa.
- Una alternativa consiste en la *demostración de programas*: la aplicación de alguna técnica matemática (basada en algún lenguaje lógico) para demostrar formalmente que un programa cumple con su tarea.

Lógica dinámica y verificación de programas

- Entre los muchos sistemas de demostración de programas, está la *lógica de Hoare* o semántica axiomática.
- Aquí se presentará una versión distinta *notacionalmente* a la que presentó originalmente Hoare para que parezca una lógica más de la familia de lógicas modales dinámicas.
- Una fórmula básica en esta lógica tiene la forma

$$\alpha \Rightarrow [P]\beta,$$

  donde α y β son fórmulas del cálculo de predicados de primer orden (las localidades de memoria son consideradas también términos válidos) y P es un programa de un lenguaje imperativo simple.
- La fórmula dice que si el programa P se ejecuta en un estado de la memoria σ que satisfaga α y si $\langle P, \sigma\rangle \rightarrow^* \sigma'$, entonces el estado σ' satisfará β.

Reglas de inferencia

La lógica de Hoare está formada por las siguientes reglas:

$$\textbf{skip}\quad \alpha \Rightarrow [\textbf{skip}]\alpha \qquad \textbf{asign}\quad \alpha_{[x:=a]} \Rightarrow [x := a]\alpha$$

$$\textbf{sec}\quad \frac{\alpha \Rightarrow [P]\beta \quad \beta \Rightarrow [Q]\gamma}{\alpha \Rightarrow [P\,;Q]\gamma}$$

$$\textbf{cond}\quad \frac{\alpha \wedge b \Rightarrow [P]\beta \quad \alpha \wedge \neg b \Rightarrow [Q]\beta}{\alpha \Rightarrow [\textbf{if } b \textbf{ then } P \textbf{ else } Q]\beta}$$

$$\textbf{ciclo}\quad \frac{\iota \wedge b \Rightarrow [P]\iota}{\iota \Rightarrow [\textbf{while } b \textbf{ do } P]\iota \wedge \neg b}$$

$$\textbf{cons}\quad \frac{\alpha \Rightarrow \alpha' \quad \alpha' \Rightarrow [P]\beta' \quad \beta' \Rightarrow \beta}{\alpha \Rightarrow [P]\beta}$$

Invariantes I

Las últimas dos reglas necesitan cierta explicación.

- En **ciclo**, la fórmula ι se conoce como *invariante*, pues es es verdadera antes y después de la ejecución del cuerpo del ciclo.
- ¿Para qué se necesita una fórmula que afirma que algo no cambia con la ejecución de un programa?
- La regla **ciclo** se puede combinar con **cons** para poder demostrar que un ciclo cumple con la tarea para la que fue diseñado.

Sean α y β fórmulas y P un programa. Se quiere demostrar que

$$\alpha \Rightarrow [\textbf{while } b \textbf{ do } P]\beta.$$

Invariantes II

Para esto, es necesaria una elección apropiada de una ι que cumpla lo siguiente (ι no es única):

1. ι debe ser efectivamente un *invariante*, es decir, cumplir con la premisa $\iota \wedge b \Rightarrow [P]\iota$.
2. $\alpha \Rightarrow \iota$.
3. $\iota \wedge \neg b \Rightarrow \beta$.

El requisito 1 y la regla **ciclo** nos permitirán concluir

$$\iota \Rightarrow [\textbf{while } b \textbf{ do } P]\iota \wedge \neg b.$$

Con esta conclusión, los requisitos 2 y 3, junto con la regla **cons** nos permitirán llegar a

$$\alpha \Rightarrow [\textbf{while } b \textbf{ do } P]\beta.$$

Ejemplo I

Supongamos que tenemos el programa Suma siguiente:

$X := 0;$
$Y := 0;$
while $Y \leq n$ **do**
$(X := X + Y;$
$Y := Y + 1)$

Suma calcula la sumatoria de los primeros n números naturales, aunque no es eficiente pues

$$\sum_{i=0}^{n} i = \frac{n + n^2}{2}.$$

Queremos demostrar que

$$0 \leq n \Rightarrow [\text{Suma}]X = \sum_{i=0}^{n} i$$

Ejemplo II

La regla **asign** nos da

$$0 = 0 \wedge 0 = 0 \wedge 0 \leq n \quad \Rightarrow \quad [X := 0]X = 0 \wedge 0 = 0 \wedge 0 \leq n$$
$$X = 0 \wedge 0 = 0 \wedge 0 \leq n \quad \Rightarrow \quad [Y := 0]X = 0 \wedge Y = 0 \wedge 0 \leq n$$

Y la regla **sec** nos da

$$0 = 0 \wedge 0 = 0 \wedge 0 \leq n \Rightarrow [X := 0; Y := 0]X = 0 \wedge Y = 0 \wedge 0 \leq n$$

Si adoptamos la convención general de que, para cualquier término t, se tiene que

$$t = t \Leftrightarrow V$$

concluimos que

$$(0 = 0 \wedge 0 = 0 \wedge 0 \leq n) \Leftrightarrow (V \wedge V \wedge 0 \leq n) \Leftrightarrow (0 \leq n).$$

Ejemplo III

La última doble implicación y la regla **cons** dan

$$0 \leq n \Rightarrow [X := 0; Y := 0]X = 0 \wedge Y = 0 \wedge 0 \leq n \qquad (1)$$

Si pudiéramos demostrar que

$$X = 0 \wedge Y = 0 \wedge 0 \leq n \Rightarrow [W]X = \sum_{i=0}^{n} i,$$

donde W es el resto de nuestro programa, a saber,

while $Y \leq n$ **do**
$(X := X + Y;$
$Y := Y + 1)$

podríamos deducir la meta original con una aplicación más de **sec**.
Para esto aplicaremos la regla **ciclo** con un invariante ι adecuado.
Recordemos que los requisitos 1–3 nos piden:

Ejemplo IV

1. $\iota \wedge Y \leq n \Rightarrow [X := X + Y; Y := Y + 1]\iota.$
2. $X = 0 \wedge Y = 0 \wedge 0 \leq n \Rightarrow \iota.$
3. $\iota \wedge Y \not\leq n \Rightarrow X = \sum_{i=0}^{n} i$

Esto nos sugiere que

$$\iota \equiv_{def} X = \sum_{i=0}^{Y-1} i \wedge Y \leq n + 1 \wedge 0 \leq n.$$

Por supuesto, este no es el único invariante que cumple con 1–3.
Hay un número infinito de otros invariantes que lo hacen, algunos triviales como

$$\iota' \equiv_{def} X = \sum_{i=0}^{Y-1} i \wedge Y \leq n + 1 \wedge 0 \leq n \wedge Z = Z,$$

Ejemplo V

donde se menciona la localidad Z, la cual ni siquiera aparece en el programa y no agrega información útil.
El propuesto es probablemente el más compacto de todos los invariantes posibles.
Verifiquemos ahora que ι cumple 1–3.
Requisito 1: tenemos que **asign** nos permite deducir las dos fórmulas

$$X + Y = \sum_{i=0}^{Y+1-1} i \land Y + 1 \leq n + 1 \land 0 \leq n \Rightarrow$$
$$[X := X + Y]X = \sum_{i=0}^{Y+1-1} i \land Y + 1 \leq n + 1 \land 0 \leq n$$
$$X = \sum_{i=0}^{Y+1-1} i \land Y + 1 \leq n + 1 \land 0 \leq n \Rightarrow [Y := Y + 1]\iota$$

Ejemplo VI

La regla **sec** nos permite concluir

$$X + Y = \sum_{i=0}^{Y+1-1} i \land Y + 1 \leq n + 1 \land 0 \leq 0 \Rightarrow [X := X + Y; Y := Y + 1]\iota.$$

Y no es difícil ver que

$$\iota \land Y \leq n \Rightarrow X + Y = \sum_{i=0}^{Y+1-1} i \land Y + 1 \leq n + 1 \land 0 \leq n,$$

y una aplicación más de **cons** nos da

$$\iota \land Y \leq n \Rightarrow [X := X + Y; Y := Y + 1]\iota,$$

con lo que queda demostrado que ι cumple 1.

Ejemplo VII

Aplicando la regla de **ciclo** podemos afirmar

$$\iota \Rightarrow [W]\iota \land Y \not\leq n. \qquad (2)$$

Requisito 2: la definición de sumatoria muestran que ι lo cumple.
Requisito 3: la aritmética implica también que

$$Y \leq n + 1 \land Y \not\leq n \Rightarrow Y = n + 1$$

2 y 3 se refieren en este caso a las afirmaciones:

$$X = 0 \land Y = 0 \land 0 \leq n \Rightarrow \iota$$
$$\iota \land Y \not\leq n \Rightarrow X = \sum_{i=0}^{n} i$$

Ejemplo VIII

Estas dos fórmulas y (2) nos permiten obtener

$$X = 0 \land Y = 0 \land 0 \leq n \Rightarrow [W]X = \sum_{i=0}^{n} i,$$

lo cual, combinado con (1) nos da

$$0 \leq n \Rightarrow [\text{Suma}]X = \sum_{i=0}^{n} i.$$