Derivación de Programas

Notas de clase de Algoritmos y Estructuras de Datos I (borrador)

Franco Luque

Índice

Introducción

En este documento se encuentra todo el contenido teórico y práctico de la materia Algoritmos y Estructuras de Datos I de la FAMAF, UNC (Universidad Nacional de Córdoba), Argentina.

Todo este contenido fue creado con el apoyo de la **Universidad Pública y Gratuita** a través de la UNC, y del **Sistema Nacional de Ciencia y Tecnología** a través del CONICET.

Un poco de historia

En los inicios de las computadoras, la programación era una tarea subestimada, y los programas eran escritos con el **"método" de prueba y error** (***code & fix***). No existían metodologías y procesos estructurados para el desarrollo de software. En cambio, los programadores iban directo a escribir el código que debía resolver su problema o necesidad. Una vez que el programa estaba escrito, era cuestión de probarlo y corregirlo hasta que funcione de la manera esperada.

Esta manera de programar resultaba muy efectiva ya que las computadoras no eran en ese momento muy poderosas, y por lo tanto no eran muy complejos los problemas que podían resolver.

Con el aumento de la potencia de las computadoras y por ende de la complejidad de los programas, empezó a ser evidente que el *code & fix* no era suficiente para el desarrollo de proyectos exitosos. El mundo entró en lo que se denominó la **Crisis del software**, ya que muchos grandes proyectos empezaron a fracasar, provocando pérdidas millonarias y hasta vidas humanas.

La crisis del software dio lugar a la creación de la **Ingeniería del Software** como una nueva disciplina que estudia el **proceso de creación y mantenimiento del software**. En este marco se desarrollaron investigaciones y experiencias que llevaron a la creación de numerosas metodologías para todos los aspectos que comprende el proceso de desarrollo del software.

Ingeniería del Software: Programación a gran escala

La Ingeniería del Software trata del **proceso completo de desarrollo del software** desde el mismo surgimiento de la necesidad o problema a resolver hasta las últimas etapas de gestión y mantenimiento del software ya terminado y en uso. Por lo tanto, no abarca solamente cuestiones relacionadas con la técnica, las computadoras y la programación, sino que también abarca aspectos sociales como la comunicación con el cliente y el manejo de equipos enteros de personas.

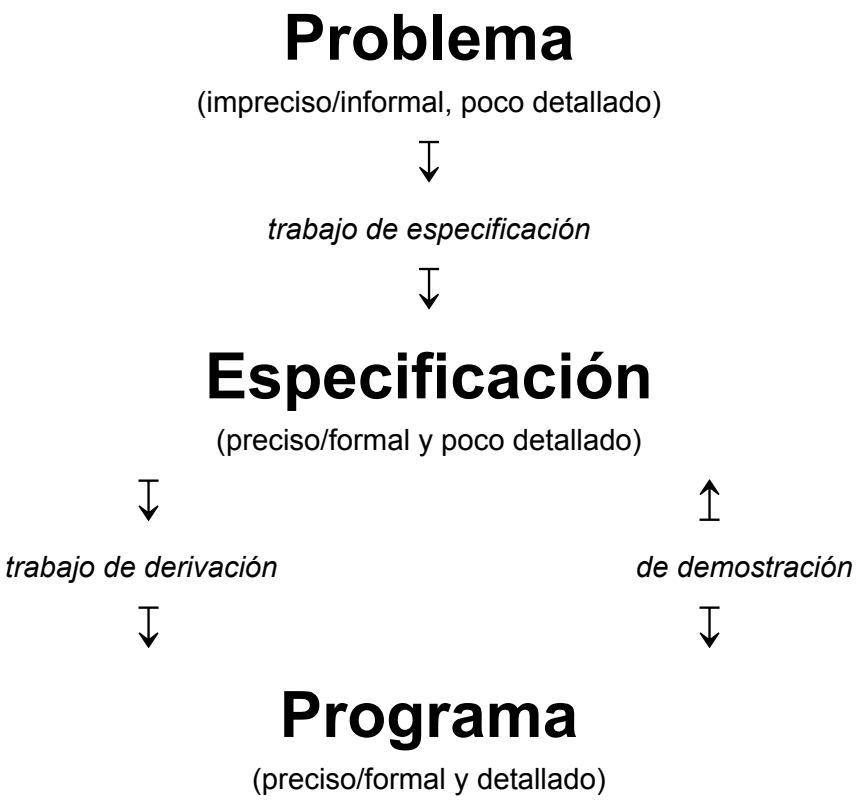
Hay infinidad de libros y artículos publicados sobre Ingeniería del Software, y existen procesos de todo tipo para ser aplicados en proyectos de diferentes características.

La aplicación cuidadosa de un proceso de desarrollo se vuelve fundamental sobre todo en la gestión de proyectos grandes, ambiciosos, complejos y/o de alto riesgo, lo que podríamos llamar **"programación a gran escala"**[1]. En contraste, podemos llamar **"programación a pequeña escala"** a la creación de pequeñas piezas de software o programas, a veces para resolver problemas sencillos y otras veces como parte de un proceso más grande de desarrollo.

Algoritmos I: Programación a pequeña escala ("in the small")

En esta materia nos vamos a ocupar de aprender programación a pequeña escala, esto es, la escritura de pequeños algoritmos para la resolución de problemas relativamente simples. Estos problemas serán siempre problemas de cálculo de uno o más resultados a partir de unos datos de entrada. No habrá entonces ningún tipo de interacción o interactividad de nuestro programa.

El proceso de construcción de programas



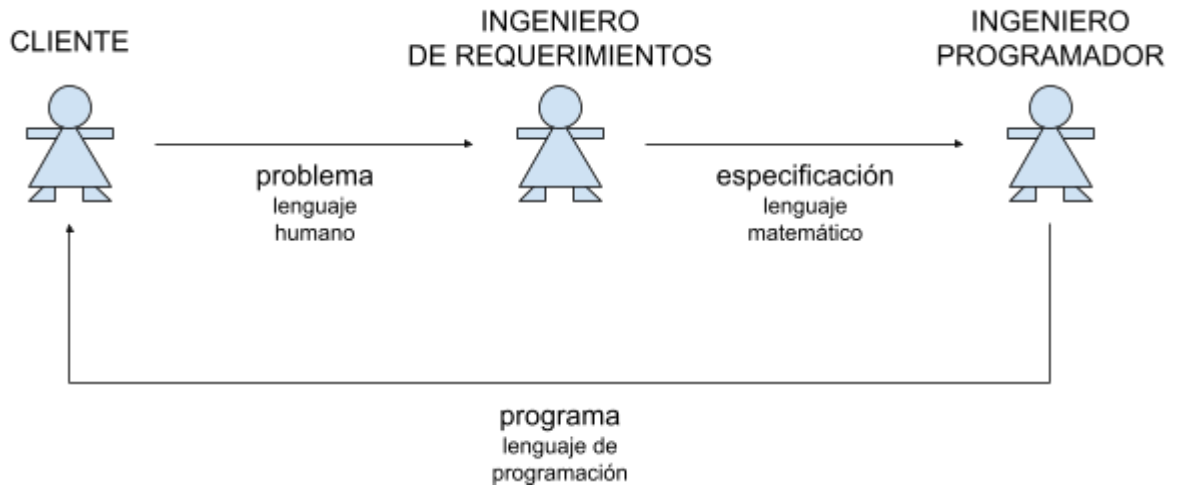
[1] https://en.wikipedia.org/wiki/Programming_in_the_large_and_programming_in_the_small

Problemas: Son tareas que se desean resolver usando computadoras. En el marco de esta materia, expresaremos los problemas en **lenguaje humano escrito**, y los problemas van a tratar de obtener ciertos valores de resultado que se quieren obtener a partir de ciertos datos iniciales o de entrada.

Especificaciones: Son formas precisas y formales de expresar qué debería cumplir un programa para ser solución del problema que se desea solucionar. Para especificar se usan herramientas matemáticas que permiten expresar muchas “fórmulas” o “cuentas” de manera simple y breve. Una especificación siempre va a mencionar el programa solución diciendo **qué hace el programa**, **pero nunca cómo lo hace.** En general no será posible que una computadora entienda o ejecute una especificación.

Programas: Los programas son “textos” que una computadora puede entender y ejecutar, por lo que vamos a usarlos para solucionar problemas. Un programa está escrito en un lenguaje de programación que tiene una sintaxis y una semántica. La sintaxis es la forma de escribir los programas o, dicho de otra manera, el conjunto de reglas que define qué textos son programas válidos y cuáles no. La semántica define qué significan los programas, esto es, qué sucede cuando se ejecutan, y cómo las distintas partes del texto se traducen en acciones de la computadora.

Roles básicos en un proyecto de software



Cliente:

- persona común y corriente (tu tía comerciante)
- tiene un problema recurrente que quiere resolver

Ingeniero de requerimientos:

- genio de las matemáticas
- hábil comunicador
- (casi) no sabe programar

Ingeniero programador:

- genio de las computadoras y de la programación
- no tantas habilidades sociales

Estructura del curso

La materia está dividida en tres partes:

1. Cuantificación general
2. Programación funcional
3. Programación imperativa

Cuantificación general: En esta parte presentamos una herramienta matemática muy útil para escribir **especificaciones** precisas y formales de muchos problemas. Esta herramienta se suma a otras herramientas ya conocidas como la lógica proposicional, la lógica de primer orden ($\forall$, $\exists$) y muchos otros recursos matemáticos que podrán ser usados a la hora de especificar.

Programación funcional: En esta parte presentamos un lenguaje de programación de tipo funcional, esto es, centrado en la definición de funciones y cuyo modelo computacional es el del cálculo/reducción de expresiones. Además, presentamos una forma de especificar problemas de manera funcional, y técnicas de derivación que permiten obtener programas funcionales a partir de especificaciones.

Programación imperativa: En esta parte presentamos un lenguaje de programación de tipo imperativo, cuyo modelo computacional es la definición de un estado y la modificación del mismo a través de la ejecución de sentencias. Presentamos también una forma de especificar problemas de manera imperativa, y técnicas de derivación que permiten obtener programas imperativos a partir de especificaciones.

Preliminares

En esta sección se mencionan brevemente contenidos previos que son necesarios para comprender la materia. Todos estos contenidos corresponden a materias previamente cursadas, especialmente a Introducción a los Algoritmos.

Cálculo proposicional (lógica de orden cero)

Repasar, tomando las cosas que más nos sirven (en particular: Leibniz 2).

Ver sección de “Cálculo Proposicional” del digesto.pdf

Qué cosas de Intro no vamos a usar:

- **Regla dorada. ¡No sirve para nada!**

Cálculo de predicados (lógica de primer orden)

Repasar para todo (∀) y existe (∃).

Tipos básicos

- Booleanos: Tipo **Bool**. Valores: True y False. Operaciones: …
- Enteros: Tipo **Int**.
- Naturales: Tipo **Nat**.
- Decimales: Tipo **Float**.
- Números en general: Tipo **Num**. Lo usamos para decir que pueden ser **Int, Nat o Float**.
- Caracteres: Tipo **Char**. Valores: ‘a’, ‘6’, ‘#’, etc.

Listas: funciones y propiedades

listas.pdf

Qué cosas de Intro no vamos a usar para nada:

- Función “pertenece” para listas

Demostraciones

Escribiremos demostraciones usando una notación que nos permite explicar qué estamos haciendo en cada paso: qué propiedades, axiomas, teoremas y/o hipótesis estamos aplicando.

Equivalencia entre dos predicados

Para demostrar P ≡ Q:

```
  P
≡ { paso 1 }
  P'
≡ { paso 2 }
  …
≡ { paso n }
  Q
```

Igualdad entre dos expresiones

Cuando tenemos expresiones que no son predicados usamos “=” en lugar de “≡”.
Para demostrar E = F:

```
  E
= { paso 1 }
  E'
= { paso 2 }
  …
= { paso n }
  F
```

Implicaciones

Para demostrar P ⇒ Q en general vamos a **suponer P como hipótesis** y vamos a demostrar que vale Q:

```
  Q
≡ { paso 1 (se puede usar hipótesis P) }
  Q'
≡ { paso 2 (se puede usar hipótesis P) }
  …
≡ { paso n }
  True
```

Otra forma (menos común) de demostrar la implicación es partiendo de P y llegando a Q donde algunos pasos se conectan con ≡ y otros con ⇒:

P
≡ { paso 1 }
P'
⇒ **{ paso 2: en este paso vale P' ⇒ P'' }**
P''
≡ { paso 3 }
…
≡ { paso n }
Q

Análisis por casos

A veces necesitamos dividir la demostración en varios casos posibles. Cada caso es una rama de la demostración.
Por ejemplo, para P ≡ Q:

P
≡ { paso 1 }
P'

- Caso 1: vale condición C_1
  ≡ { paso 1.2 (puedo usar nueva hipótesis C_1)}
  …
  ≡ { paso 1.n }
  Q
- Caso 2: vale condición C_2
  ≡ { paso 2.2 (puedo usar nueva hipótesis C_2)}
  …
  ≡ { paso 2.n }
  Q

Diferencia entre igualdad (=) y equivalencia (≡)

Equivalencia (≡): Se usa solamente para **predicados**, es decir, expresiones booleanas. Solamente se puede escribir "A ≡ B" si tanto A como B son predicados.

Igualdad (=): Se puede usar en **cualquier caso**. "A = B" se puede escribir siempre que A y B sean expresiones del mismo tipo. Sin embargo, si sabemos que A y B son predicados preferimos usar la equivalencia (≡) ya que se habilitan las propiedades específicas de la equivalencia.

Cuantificación General

Introducción

TODO:

- sumatoria y productoria como en álgebra I

Sintaxis

$$\langle \oplus\ i : R.i : T.i \rangle$$

- $\oplus$ representa la operación que está siendo cuantificada.
  Observaciones:
  1. Siempre se trata de operaciones binarias: **Tipo A -> A -> A**.
  (toma dos cosas de tipo A y devuelve una de tipo A).
  2. La operación debe ser asociativa y conmutativa.
  Ejemplos: suma (+ ó $\sum$), producto (* ó $\prod$), conjunción ($\wedge$ ó $\forall$), disyunción ($\vee$ ó $\exists$), máximo, mínimo, etc.
  Ejemplos que no: negación (no es binaria), resta (no es conmutativa), división (no es conmutativa).
- i: Nombres de las **variables cuantificadas**. (ponemos sólo "i" pero pueden ser varias). Cada variable va a tener un tipo asociado. En general no vamos a escribir los tipos explícitamente sino que se van a poder inferir por el contexto en el que se usan las variables.
- R.i : **El rango**. Es un predicado (tipo Bool) sobre las variables cuantificadas. Define el conjunto de valores posibles que van a tomar las variables al ser aplicadas al término.
- T.i : **El término**. Es una expresión de tipo A (a donde A es el tipo de la operación). Define las expresiones que luego serán aplicadas a los valores definidos por el rango para luego ser "juntadas" (operadas) con la operación cuantificada $\oplus$, para finalmente obtener un único valor de tipo A.

Lectura Operacional

¿Cómo calcular el resultado de una expresión cuantificada?

0. Identificar los tipos de las variables y el **universo de cuantificación.**
   La variable cuantificada i es de tipo A.
1. Calcular el conjunto de valores posibles para las variables cuantificadas de **acuerdo al rango R**.
   Los valores posibles para i son $\{ v_1, v_2, \dots, v_n \}$
2. Aplicar el término T a todos los elementos del conjunto de valores posibles.
   $T.v_1$

$T.v_2$
. . .
$T.v_n$

3. “Juntar todo” con la operación cuantificada:

$$T.v_1 \oplus T.v_2 \oplus \ldots \oplus T.v_n$$

Ejemplos

Ejemplo:

$$\langle \prod i : 0 \leq i < 5 : i * 2 + 1 \rangle$$

0. El tipo de i es Int (y también el universo de cuantificación es Int).
1. El rango es: i ∈ { 0, 1, 2, 3, 4 }.
2. Los términos aplicados son:
   0 * 2 + 1
   1 * 2 + 1
   2 * 2 + 1
   3 * 2 + 1
   4 * 2 + 1
3. Al operar (“juntar todo”), nos queda:
   (0 * 2 + 1) * (1 * 2 + 1) * (2 * 2 + 1) * (3 * 2 + 1) * (4 * 2 + 1)

Otro ejemplo:

$$\langle \prod i : 0 \leq i < 5 \wedge 2 < i \leq 10 : 37 \rangle$$

1. El rango es: i ∈ { 3, 4 }
2. Los términos aplicados son:
   37
   37
3. Al operar (“juntar todo”), nos queda:
   37 * 37

Ejemplos con dos variables cuantificadas

Se usan tuplas para definir el conjunto de valores posibles.

Ejemplo: “Todos los elementos de la lista xs son distintos.”

$$\langle \forall i , j : 0 \leq i < \#xs \wedge 0 \leq j < \#xs \wedge i \neq j : xs!i \neq xs!j \rangle$$

Lectura operacional con xs = [2, 7, -1, 7]:

1. El rango es: (i, j) ∈ { (0, 1), (0, 2), (0, 3),
   (1, 0), (1, 2), (1, 3),
   (2, 0), (2, 1), (2, 3),
   (3, 0), (3, 1), (3, 2) }

2. Son 12 términos (ejercicio: aplicarlos y resolver)

i \ j	0	1	2	3
0	~~(0, 0)~~	(0, 1)	(0, 2)	(0, 3)
1	(1, 0)	~~(1, 1)~~	(1, 2)	(1, 3)
2	(2, 0)	(2, 1)	~~(2, 2)~~	(2, 3)
3	(3, 0)	(3, 1)	(3, 2)	~~(3, 3)~~

Hay muchos términos repetidos: (0,1) y (1,0) dicen lo mismo.

Otra forma quitando términos repetidos: Ponemos i < j en lugar de i ≠ j.

$$\langle\ \forall\ i\ ,\ j\ :\ 0 \le i < \#xs\ \wedge\ 0 \le j < \#xs\ \wedge\ \underline{\mathbf{i < j}}\ :\ xs!i \neq xs!j\ \rangle$$

Equivalente, más corto y más bonito:

$$\langle\ \forall\ i\ ,\ j\ :\ 0 \le i < j < \#xs\ :\ xs!i \neq xs!j\ \rangle$$

i \ j	0	1	2	3
0	~~(0, 0)~~	**(0, 1)**	**(0, 2)**	**(0, 3)**
1	~~(1, 0)~~	~~(1, 1)~~	**(1, 2)**	**(1, 3)**
2	~~(2, 0)~~	~~(2, 1)~~	~~(2, 2)~~	**(2, 3)**
3	~~(3, 0)~~	~~(3, 1)~~	~~(3, 2)~~	~~(3, 3)~~

Quedan 6 términos.

Otro ejemplo:

$$\langle\ \prod i\ ,\ j\ \ :\ 0 \le i < 5\ \wedge\ \ i \bmod 2 = 0\ \ \wedge\ |\ j - i\ |\ = 1\ :\ \ i + j\ \rangle$$

0. Ambas variables son Int, el universo de cuantificación es Int x Int (o sea pares de enteros).
1. El rango es: (i, j) ∈ { (0, -1), (0, 1) , (2, 1) , (2, 3) , (4, 3) , (4, 5) } (6 elementos)
2. Los términos aplicados son: **ejercicio!!**
3. Resultado: **Ejercicio!!**

Axiomas y Teoremas de Cuantificadores

Digesto

digesto.pdf

A1: Rango vacío

$$\langle \oplus\ i : False : T.i \rangle = e$$

- donde e es el neutro $\oplus$ ($e \oplus X = X$)

¿Qué pasa cuando no hay ningún valor posible para las variables cuantificadas que satisfaga el rango?
En la lectura operacional: El rango es el conjunto vacío.

Ejemplo:
⟨ Max i : i mod 2 = 0 ∧ i > 10 ∧ i < 12 : i * 2 ⟩
= { propiedad de los enteros que me dice que i > 10 ∧ i < 12 es lo mismo que i = 11 }
⟨ Max i : i mod 2 = 0 ∧ i = 11 : i * 2 ⟩
= { reemplazo de iguales por iguales en la conjunción (Leibniz 2) }
⟨ Max i : 11 mod 2 = 0 ∧ i = 11 : i * 2 ⟩
= { 11 mod 2 es 1 }
⟨ Max i : 1 = 0 ∧ i = 11 : i * 2 ⟩
= { lógica }
⟨ Max i : False ∧ i = 11 : i * 2 ⟩
= { lógica (absorvente de ∧) }
⟨ Max i : False : i * 2 ⟩
= { rango vacío (A1), ya que - infinito es el neutro de max }
- infinito

A2: Rango unitario

$$\langle \oplus\ i : i = C : T.i \rangle = T.C$$

¿Qué pasa cuando la variable cuantificada tiene sólo un valor posible?

En la lectura operacional: El rango es un conjunto de un solo elemento.

Ejemplo (casi igual al anterior, pero con i impar):
⟨ Max i : i mod 2 = 1 ∧ i = 11 : i * 2 ⟩
= { reemplazo de iguales por iguales en la conjunción (Leibniz) }
⟨ Max i : 11 mod 2 =1 ∧ i = 11 : i * 2 ⟩
= { 11 mod 2 es 1 }

⟨ Max i : 1 = 1 ∧ i = 11 : i * 2 ⟩
= { lógica }
⟨ Max i : True ∧ i = 11 : i * 2 ⟩
= { lógica (neutro de ∧) }
⟨ Max i : i = 11 : i * 2 ⟩
= { rango unitario (A2) }
11 * 2

Ejemplo:

⟨ ∑ i , j , k : i + j = 10 ∧ j + 2 * k = 10 ∧ i + k = 9 : i * (j + 1) * (k - 1) ⟩

Lectura operacional:
0. El universo de cuantificación es Int x Int x Int (triplas o ternas o tuplas de 3 elementos).
1. El rango es: (i, j, k) ∈ { (6, 4, 3) } (ejercicio: chequear! es un sistema de tres ecuaciones con tres incógnitas)
2. Aplicar el término una sola vez:

   6 * (4 + 1) * (3 - 1)

3. Juntar todo (hay una sola cosa):

   6 * (4 + 1) * (3 - 1)

Demostración:
⟨ ∑ i , j , k : i + j = 10 ∧ j + 2 * k = 10 ∧ i + k = 9 : i * (j + 1) * (k - 1) ⟩
= { resuelvo el sistema de ecuaciones, reemplazo predicados equivalentes }
⟨ ∑ i , j , k : i = 6 ∧ j = 4 ∧ k = 3 : i * (j + 1) * (k - 1) ⟩
= { armo la tupla para que tenga la forma de rango unitario }
⟨ ∑ i , j , k : (i , j , k) = (6 , 4 , 3) : i * (j + 1) * (k - 1) ⟩
= { rango unitario }
6 * (4 + 1) * (3 - 1)
= { aritmética }
60

A3: Partición de rango

⟨ ⊕ i : R.i ∨ S.i : T.i ⟩
= ⟨ ⊕ i : R.i : T.i ⟩⊕ ⟨ ⊕ i : S.i : T.i ⟩
siempre que suceda la menos una de estas dos cosas:
- ⊕ es idempotente
- R y S son disjuntos

Ejemplo:
⟨ ∀ i : **(-1 ≤ i < 1) ∨ (8 ≤ i < 11)** : T.i ⟩ (A)
¿es esto igual que ….
⟨ ∀ i : (-1 ≤ i < 1) : T.i ⟩ ∧ ⟨ ∀ i : (8 ≤ i < 11) : T.i ⟩ (B)

? ¡Sí! Veamos la lectura operacional:

(A): El rango es { -1, 0, 8, 9, 10 }. El resultado es:

(T.(-1) ∧ T.0 ∧ T.8 ∧ T.9 ∧ T.10)

que es lo mismo que:

(T.(-1) ∧ T.0) ∧ (T.8 ∧ T.9 ∧ T.10)

(B): Tengo dos expresiones cuantificadas conjugadas entre sí.

La primera es (T.(-1) ∧ T.0)

La segunda es (T.8 ∧ T.9 ∧ T.10)

Luego me queda exactamente lo mismo que (A):

(T.(-1) ∧ T.0) ∧ (T.8 ∧ T.9 ∧ T.10)

Otro ejemplo:

〈 ∀ i : **(-1 ≤ i < 2) ∨ (0 ≤ i < 4)** : T.i 〉 (A)

¿es esto igual que ….

〈 ∀ i : (-1 ≤ i < 2) : T.i 〉 ∧ 〈 ∀ i : (0 ≤ i < 4) : T.i 〉 (B)

? ¡Si! Veamos:

Acá, para (A) el rango es { -1, 0, 1, 2, 3}. El resultado es:

(T.(-1) ∧ T.0 ∧ T.1 ∧ T.2 ∧ T.3)

Para (B) tengo dos rangos: { -1, 0, 1 } y { 0, 1, 2, 3 }. Luego. el resultado de toda la expresión es:

(T.(-1) ∧ T.0 ∧ T.1) ∧ (T.0 ∧ T.1 ∧ T.2 ∧ T.3)

¿Es esto lo mismo que A? Sí, pero sólo gracias a que la conjunción es idempotente. Me queda:

(T.(-1) ∧ T.0 ∧ T.1 ∧ T.2 ∧ T.3)

Otro ejemplo:

〈 ∑ i : **(-1 ≤ i < 2) ∨ (0 ≤ i < 4)** : T.i 〉 (A)

¿es esto igual que ….

〈 ∑ i : (-1 ≤ i < 2) : T.i 〉 + 〈 ∑ i : (0 ≤ i < 4) : T.i 〉 (B)

? ¡No! Ya que (A) es

(T.(-1) + T.0 + T.1 + T.2 + T.3)

Y (B) es

(T.(-1) + T.0 + T.1) + (T.0 + T.1 + T.2 + T.3)

T.0 y T.1 aparecen repetidos pero no se pueden eliminar porque la suma no es idempotente.

¿En qué caso sí se podría aplicar la partición de rango si el operador no es idempotente? En el caso en el que no aparecen términos repetidos, es decir que los predicados R y S son disjuntos entre sí (no hay ningún valor que satisfaga ambos al mismo tiempo).

A4: Regla del término

〈 ⊕ i : R.i : T.i ⊕ U.i 〉= 〈 ⊕ i : R.i : T.i 〉⊕〈 ⊕ i : R.i : U.i 〉

En el término tengo dos sub-términos operados entre sí.

En lectura operacional:
La expresión de la izquierda es:

$$(T.v_1 \oplus U.v_1) \oplus \ldots. \oplus (T.v_n \oplus U.v_n)$$

La expresión de la derecha es:

$$(T.v_1 \oplus \ldots. \oplus T.v_n) \oplus (U.v_1 \oplus \ldots. \oplus U.v_n)$$

¿Son equivalentes estas dos expresiones? Sí. ¿Por qué? Porque es sólo un reacomodo de los términos usando asociatividad y conmutatividad.

Ejemplo:
$\langle \prod i : \mathbf{i \bmod 2 = 1} \wedge \underline{\mathbf{| i - 10 | \leq 3}} : i * (10 - i) \rangle$
= { regla del término (A4) con T.i = i , U.i = 10 - i }
$\langle \prod i : i \bmod 2 = 1 \wedge | i - 10 | \leq 3 : i \rangle * \langle \prod i : i \bmod 2 = 1 \wedge | i - 10 | \leq 3 : (10 - i) \rangle$

Lectura operacional:

1. Rango: $i \in \{ 7, 9, 11, 13 \}$
2. (y 3.). Aplicamos término y operamos:
   $(7 * (10 - 7)) * (9 * (10 - 9)) * (11 * (10 - 11)) * (13 * (10 - 13))$
   $= 7 * 3 * 9 * 1 * 11 * (-1) * 13 * (-3)$
   $= \ldots.$

Ejercicio: hacer la lectura operacional de la expresión que queda luego de aplicar el axioma.

A5: Término constante

$\langle \oplus i : R.i : C \rangle = C$

El término de la expresión cuantificada no menciona a las variables cuantificadas.
¿Cuándo vale esta propiedad? **Sólo vale cuando el operador $\oplus$ es idempotente en el valor C.**
Además, **el rango debe ser no vacío.** Si no, quedaría el elemento neutro.

En lectura operacional: Si el rango es $\{ v_1 , \ldots , v_n \}$, tengo los siguientes n términos:

$$C \oplus C \oplus \ldots \oplus C$$

(n veces)
¿Cuándo sucede que esto es lo mismo que C? Confirmamos: cuando $\oplus$ es idempotente en ese valor específico C.

Ejemplo 1:
$\langle \forall xs : \#xs = 2 \wedge (xs!0) * (xs!1) = 1 : \underline{xs = [\,] \vee xs \neq [\,]} \rangle$
≡ { tercero excluido }
$\langle \forall xs : \#xs = 2 \wedge (xs!0) * (xs!1) = 1 : True \rangle$
≡ { término constante }
True

Lectura operacional: sólo para seguir jugando con los rangos.
0. Universo de cuantificación: Es el de las listas de números enteros.
    1. Rango: xs ∈ { [1, 1] , [-1 , -1] }
    2. (y 3.) Aplicamos término y operamos: True ∧ True

Ejemplo 2 (ejemplo que no funciona):
⟨ ∏ xs : #xs = 2 ∧ (xs!0) * (xs!1) = 1 : #xs ⟩
= { sustitución: por el rango, sé que #xs = 2 }
⟨ ∏ xs : #xs = 2 ∧ (xs!0) * (xs!1) = 1 : 2 ⟩
= { si valiera término constante, podría decir: }
2
Pero no ya que da (haciendo lectura operacional): 2 * 2 = 4.

Si hubiera tenido:
⟨ ∏ xs : #xs = 2 ∧ (xs!0) * (xs!1) = 1 : 1 ⟩
Ahí sí se puede aplicar término constante ya que * es idempotente en el valor 1.
Quedaría: 1.

Ejemplo 3 (otro que no funciona):
⟨ ∑ i : 3 ≤ i ≤ 6 : 24 ⟩
¿Cuánto da?
Lectura operacional:
1. Rango: i ∈ { 3, 4, 5, 6 }.
2. Aplico el término a los valores del rango:
   24
   24
   24
   24
   (lo que estoy haciendo es sustituir en la expresión del término cada mención de i por su valor) (lo que pasa es que i no se menciona en el término, entonces queda siempre lo mismo).
3. Juntamos todo con la suma:
   24 + 24 + 24 + 24 = 96

   Luego, **no vale la regla de término constante (no dió 24).**

A6: Distributividad a derecha

⟨ ⊕ i : R.i : T.i ⓧ C ⟩= ⟨ ⊕ i : R.i : T.i ⟩ⓧ C

Requisitos:
- C es constante (las variables cuantificadas no aparecen mencionadas en C)
- ⊕ y ⓧ son dos operaciones de tipo **A** →**A** →**A**.
- ⓧ distribuye con ⊕ a derecha: (x ⓧ y) ⊕ (z ⓧ y) = (x ⊕ z) ⓧ y
- R es no vacío, o bien el neutro de ⊕ es absorbente para ⓧ.

Lectura operacional:

La expresión de la izquierda es: si el rango es $\{ v_1 , \dots , v_n \}$:

$$(T.v_1 \otimes C) \oplus \dots \oplus (T.v_n \otimes C)$$

La expresión de la derecha es:

$$(T.v_1 \oplus \dots \oplus T.v_n) \otimes C$$

¿Cuándo sucede que estas dos expresiones son equivalentes? Confirmamos que debe darse la siguiente **propiedad de distributividad**:

$(x \otimes y) \oplus (z \otimes y) = (x \oplus z) \otimes y$ "saco factor común y"

Formalmente, estamos diciendo que $\otimes$ distribuye con $\oplus$ **a derecha**.

¿Qué pasa si tengo rango vacío?

La expresión de la izquierda me queda el elemento neutro de la operación $\oplus$ (llamemoslo $e_\oplus$):

$$e_\oplus$$

La expresión de la derecha me queda:

$$e_\oplus \otimes C$$

¿qué debe suceder para que estas dos cosas sean iguales? $e_\oplus = e_\oplus \otimes C$

Si pasa esto, quiere decir que el neutro de $\oplus$ ($e_\oplus$) es absorbente para la operación $\otimes$.

Conclusión: si tengo rango vacío, para poder aplicar distributividad debe suceder que el neutro de $\oplus$ es absorbente para $\otimes$.

Ejemplo (ejercicio 12c):

$\langle \forall i : \; i = 0 \vee 4 > i \geq 1 \; : \; \neg f.i \vee \neg f.n \; \rangle$

≡ { $\oplus$ es $\wedge$, $\otimes$ es $\vee$, C es $\neg$ f.n , luego puedo aplicar distributividad ya que se cumple todos los requisitos }

$\langle \forall i : \; i = 0 \vee 4 > i \geq 1 \; : \; \neg f.i \rangle \vee \neg f.n$

Ejemplo (ejercicio 12d):

$\langle$Max i : $0 \leq i < \#xs$: <u>k + xs!i</u> $\rangle$

= { conmutatividad }

$\langle$Max i : $0 \leq i < \#xs$: xs!i + k $\rangle$

= { $\oplus$ es max , $\otimes$ es + , C es k, ¿distribuyen?

(x + y) max (z + y) = (x max z) + y . Sí vale, **siempre.**

¿es el rango no vacío? no sabemos, si xs = [ ], el rango es vacío.

luego, debe suceder que el neutro de max (-infinito) es absorbente para + (la suma), o sea que **(-infinito) + x = - infinito .**

En esta materia vamos a asumir que esto vale, así que vamos a permitir distribuir.

}

$\langle$Max i : $0 \leq i < \#xs$: xs!i $\rangle$ + k

Observación importante acerca de los rangos: Si xs es vacío, podemos ver que:

$0 \leq i < \#xs$

≡ { sustituyo xs por [ ] }

$0 \leq i < \#[\,]$

≡ { def de # }

$0 \leq i < 0$

≡ { lógica (no existe i tal que es al mismo tiempo ≥ 0 y < 0) }
False

MUCHO OJO CON LAS DESIGUALDADES: **MENOR/MAYOR ESTRICTO (<, >) Y MENOR/MAYOR IGUAL (≤, ≥) SON COSAS MUY DISTINTAS, NUNCA "DA LO MISMO" USAR UNA U OTRA.**

A6': Distributividad a izquierda

Igual que A6 pero con la distributividad al revés:

$\langle \oplus i : R.i : C \otimes T.i \rangle = C \otimes \langle \oplus i : R.i : T.i \rangle$

A7: Anidado

$\langle \oplus i , j : R.i \wedge S.i.j : T.i.j \rangle = \langle \oplus i : R.i : \langle \oplus j : S.i.j : T.i.j \rangle \rangle$

Aclaraciones:
- R.i no menciona a "j", puede mencionar a "i" pero no obligatoriamente
- S.i.j puede mencionar a "i" y "j", pero no obligatoriamente.

Ejemplo:
$\langle \sum i , j : 7 \leq i < 10 \wedge i \bmod j = 0 \wedge j > 0 : T.i.j \rangle$
= { R.i es $7 \leq i < 10$, S.i.j es $i \bmod j = 0 \wedge j > 0$, aplicamos anidado }
$\langle \sum i : 7 \leq i < 10 : \langle \sum j : i \bmod j = 0 \wedge j > 0 : T.i.j \rangle \rangle$

Lectura operacional del ejemplo:
El lado izquierdo:
- El rango es: $(i, j) \in \{ (7, 1), (7, 7), (8, 8), (8, 1), (8, 2), (8, 4), (9, 1), (9, 9), (9, 3)\}$
- La expresión queda:

$$T.7.1 + T.7.7 + T.8.8 + T.8.1 + \ldots..+ T.9.3$$

El lado derecho:
- El rango de la sumatoria de afuera ($7 \leq i < 10$): $i \in \{ 7, 8, 9 \}$
- Luego, debo calcular el término para cada uno de estos valores posibles:
  1. $\langle \sum j : 7 \bmod j = 0 \wedge j > 0 : T.7.j \rangle +$
  2. $\langle \sum j : 8 \bmod j = 0 \wedge j > 0 : \mathbf{T.8.j} \rangle +$
  3. $\langle \sum j : 9 \bmod j = 0 \wedge j > 0 : T.9.j \rangle$
  1. El rango es: $j \in \{ 1, 7 \}$. Me queda:
  $$T.7.1 + T.7.7$$
  2. El rango es: $j \in \{ 1, 2, 4, 8 \}$. Me queda:
  $$T.8.1 + T.8.2 + T.8.4 + T.8.8$$
  3. El rango es: $j \in \{ 1, 3, 9 \}$. Me queda:
  $$T.9.1 + T.9.3 + T.9.9$$
- Resultado del lado derecho:
  $$(T.7.1 + T.7.7) + (T.8.1 + T.8.2 + T.8.4 + T.8.8) + (T.9.1 + T.9.3 + T.9.9)$$

¿Quedó igual que el lado izquierdo?
Sí, sólo reorganizando los términos. Se verifica la regla.

Caso particular: Supongamos que en el rango tengo: $0 \leq i < j < 10 \ \wedge \ i \bmod j = 0$.
Ambas partes mencionan a i y j. Pero puedo tomar:

- R.i = True
- S.i.j = $0 \leq i < j < 10 \ \wedge \ i \bmod j = 0$

Y aplicar anidado de esta manera.

T1: Cambio de variable

$\langle \oplus i : R.i : T.i \rangle = \langle \oplus j : R.(f.j) : T.(f.j) \rangle$

Requisitos:

- f es una función **biyectiva** cuya imagen es el conjunto definido por el rango R.i.
- j no aparece mencionada en R ni en T.

Ejemplo:

$\langle \sum i : 24 < i \leq 29 : i * 2 - 10 \rangle$ (A)

= { propongo usar cambio de variable con la función **f.j = j - 11** }

$\langle \sum j : 24 < (f.j) \leq 29 : (f.j) * 2 - 10 \rangle$

= { sustituyo f por lo que es }

$\langle \sum j : 24 < j - 11 \leq 29 : (j - 11) * 2 - 10 \rangle$

= { aritmética }

$\langle \sum j : 35 < j \leq 40 : (j - 11) * 2 - 10 \rangle$ (B)

¿es igual lo de arriba de todo (A) a lo de abajo de todo (B)? Sí.

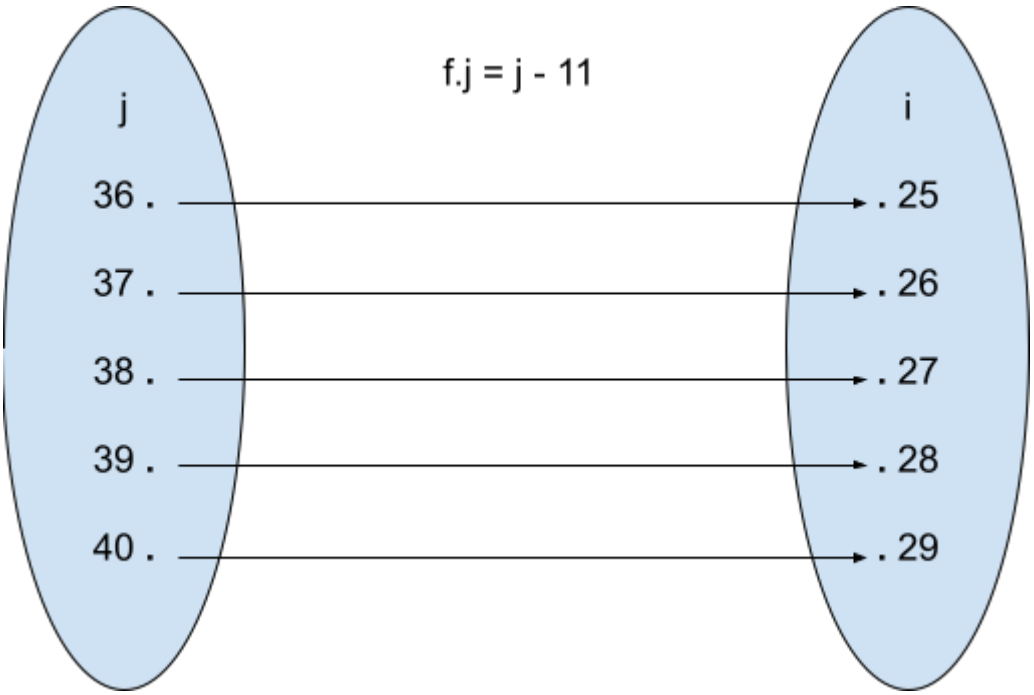
Veamos la lectura operacional:

(A) El rango es: $i \in \{ 25, 26, 27, 28, 29 \}$. La expresión queda:
(25 * 2 - 10) + (26 * 2 - 10) + (27 * 2 - 10) + (28 * 2 - 10) + (29 * 2 - 10)

(B) El rango es: $j \in \{ 36, 37, 38, 39, 40 \}$. La expresión queda:
(**(36 - 11)** * 2 - 10) + (**(37 - 11)** * 2 - 10) + ((38 - 11) * 2 - 10) + ((39 - 11) * 2 - 10) + ((40 - 11) * 2 - 10)
o sea:
(25 * 2 - 10) + (26 * 2 - 10) + (27 * 2 - 10) + (28 * 2 - 10) + (29 * 2 - 10)
que es los mismo que (A).

¿Qué pasó? Apliqué una función **cuya imagen** (el conjunto de llegada) es conjunto definido por el rango original R.i:

- Al aplicarla en el rango, cambio el conjunto original por un nuevo conjunto que es el del dominio de la función, ya no la imagen (en este ejemplo, {36,37,38,39,40}).
- Al aplicarla en el término, estoy haciendo que cada punto vuelva a tener el valor correcto que espero usar en el término (vuelvo al rango original).



¿Cuándo se rompe esto?

Ejemplo con función no inyectiva y no sobreyectiva:

$$f.j = j * j$$

Lado izquierdo: $\langle \sum i : 24 < i \leq 29 : i * 2 - 10 \rangle$ (A)

Lado derecho: $\langle \sum j : 24 < j * j \leq 29 : (j * j) * 2 - 10 \rangle$ (B')

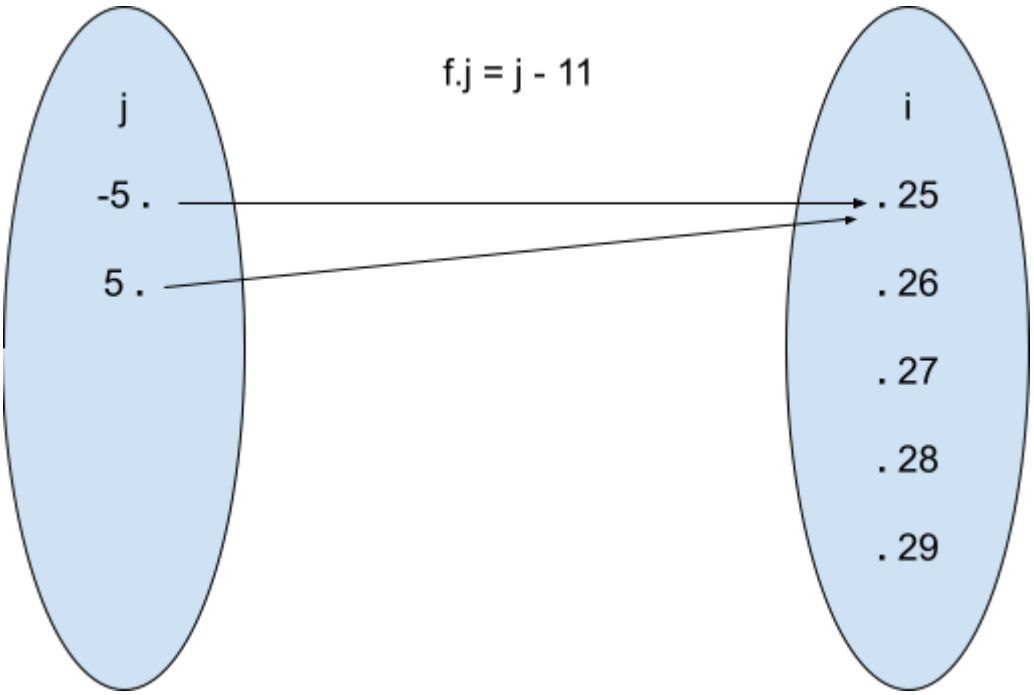
Lectura operacional para A:

$(25 * 2 - 10) + (26 * 2 - 10) + (27 * 2 - 10) + (28 * 2 - 10) + (29 * 2 - 10)$

Lectura operacional para B':

- El rango es $j \in \{ 5, -5 \}$.
- El resultado es: $((5 * 5) * 2 - 10) + (((-5) * (-5)) * 2 - 10)$
  O sea: $(25 * 2 - 10) + (25 * 2 - 10)$

Luego, se rompió todo, porque tengo un término repetido (por culpa de la no inyectividad), y tengo términos que faltan (por culpa de la no sobreyectividad).



Luego, para que la regla de cambio de variable funcione, la función que yo aplique debe ser biyectiva considerando a el conjunto definido por el rango R.i como la **imagen de la función**.

Ejemplo con función inyectiva pero no sobreyectiva:

⟨ ∑ i : 24 < i ≤ 29 : i * 2 - 10 ⟩ (A)

Cambio de variable con f.j = 2 * j quedaría:

⟨ ∑ i : 24 < 2 * j ≤ 29 : (2 * j) * 2 - 10 ⟩ (B")

Lectura operacional para A:

(25 * 2 - 10) + (26 * 2 - 10) + (27 * 2 - 10) + (28 * 2 - 10) + (29 * 2 - 10)

Lectura operacional para B":

- El rango es j ∈ { 13, 14 }.
- El resultado es: ((2 * 13) * 2 - 10) + ((2 * 14) * 2 - 10)
  = (26 * 2 - 10) + (28 * 2 - 10)

¡¡Faltan tres términos!!

Ejemplo con función sobreyectiva pero no inyectiva:

⟨ ∑ i : 24 < i ≤ 29 : i * 2 - 10 ⟩ (A)

Cambio de variable con f.j = j div 2 quedaría:

⟨ ∑ i : 24 < j div 2 ≤ 29 : (j div 2) * 2 - 10 ⟩ (B‴)

Lectura operacional para B":
El rango es j ∈ { 50, 51, 52, 53, 54, 55, 56, 57, 58, 59 }.
El resultado es: ((50 div 2) * 2 - 10) + ((51 div 2) * 2 - 10) +
((52 div 2) * 2 - 10) + ((53 div 2) * 2 - 10) + …
= (25 * 2 - 10) + (25 * 2 - 10) + (26 * 2 - 10) + (26 * 2 - 10)
¡¡Todos los términos aparecen repetidos!!

Conclusión:

- **Inyectiva:** Me asegura de que no se repiten términos al aplicar el cambio.
- **Sobreyectiva:** Me asegura de que no se pierden términos al aplicar el cambio.

Veamos el 2do requisito:

- j no aparece en R ni en T.

⟨ ∑ i : 24 < i ≤ 29 : i * 2 - 10 + j ⟩
Acá, j es variable libre y su valor no depende de la cuantificación.
Si aplico cambio de variable con f.j = j - 11, me quedaría:
⟨ ∑ j : 24 < j - 11 ≤ 29 : (j - 11) * 2 - 10 + j ⟩
Tengo una **colisión** de variables, se mezclaron dos cosas distintas.
Siempre tendremos cuidado de no pisar nombres de variables.
Usualmente, volveremos a usar el mismo nombre de variable que teníamos antes (i):
⟨ ∑ i : 24 < i ≤ 29 : i * 2 - 10 + j ⟩
= { cambio de variable con f.i = i - 11, (también solemos escribir i ← i - 11) }
⟨ ∑ i : 24 < i - 11 ≤ 29 : (i - 11) * 2 - 10 + j ⟩

Notación:

- Usualmente, volveremos a usar el mismo nombre de variable que teníamos antes (cambiamos "i" por "i - 11" y no por "j - 11").
- Muchas veces escribiremos la función de manera implícita:
  Diremos "i ← i - 11" en lugar de "f.i = i - 11"

Cambio de variable con listas

Un cambio de variable que usaremos mucho será el siguiente:

$$f.(b, bs) = b \blacktriangleright bs$$

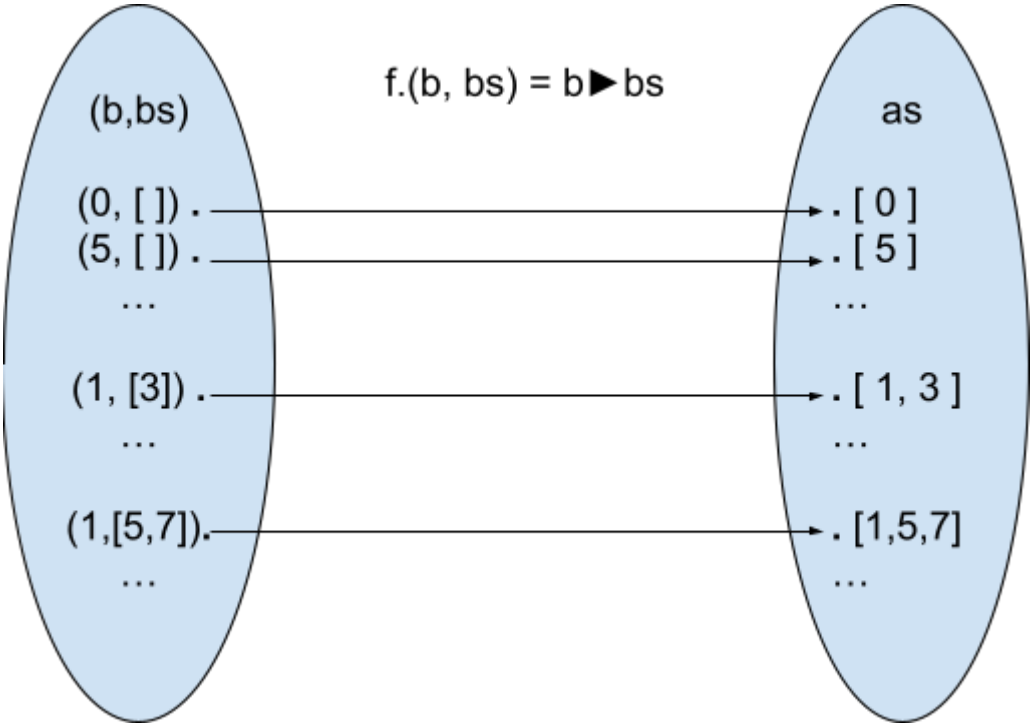
Este cambio se aplica a una variable cuantificada de tipo lista "as" y **se reemplaza por dos variables cuantificadas "b" y "bs"**, una para el primer elemento de la lista y otra para la lista de los elementos restantes. Para que la función sea sobreyectiva, el rango original

debe forzar que “as ≠ [ ]” (**“as” no es la lista vacía**), ya que no es posible llegar a ese elemento con f.

Ejemplo:

⟨ Max as : as ≠ [ ] ∧ #as ≤ 3 : #as ⟩

Acá el rango claramente excluye la lista vacía. En la siguiente figura podemos ver que la función es biyectiva (los conjuntos son infinitos!):



⟨ Max as : as ≠ [ ] ∧ #as ≤ 3 : #as ⟩
= { cambio de variable con f.(b, bs) = b ► bs }
⟨ Max b, bs : b►bs ≠ [ ] ∧ #(b►bs) ≤ 3 : #(b►bs) ⟩
= { **(hacemos más pasos por diversión:)** propiedad de listas y lógica }
⟨ Max b, bs : #(b►bs) ≤ 3 : #(b►bs) ⟩
= { def. de # }
⟨ Max b, bs : #bs + 1 ≤ 3 : #bs + 1 ⟩
= { distrib. de + con Max }
⟨ Max b, bs : #bs + 1 ≤ 3 : #bs ⟩+ 1

Ejercicios

Ejercicio 14a:
⟨ ∑ i : | i | < 5 : i div 2 ⟩ con f.j = 2 * j

¿Qué debe cumplir f? debe ser biyectiva considerando a la imagen de la función

como el conjunto definido por el rango de la cuantificación.
En este caso el rango es: {-4, -3, -2, -1, 0, 1, 2, 3, 4}

¿es f inyectiva acá? Si
¿es f sobreyectiva acá? No, porque no tengo forma de llegar al -3, -1 ni a ningún impar.

No se puede aplicar cambio de variable.

¿qué valores puede tomar j? estamos hablando del **dominio de f**. Este conjunto se determina tomando la imagen y volviendo hacia atrás con f. O sea, encontrar aquellos j tales que f.j ∈ {**-4**, -3, **-2**, -1, **0**, 1, **2**, 3, **4**}. En este caso el conjunto es {-2, -1, 0, 1, 2}.

Ejercicio: Aplicarlo como si se pudiera y verificar que **NO** da lo mismo usando la lectura operacional de ambas versiones.

Ejercicio 14b:
⟨ ∑ i : par.i ∧ | i | < 5 : i div 2 ⟩ con f.j = 2 * j

En este caso el rango es: {-4, -2, 0, 2, 4} (o sea la imagen de f)

¿cuál es el dominio de f? es el conjunto {-2, -1, 0, 1, 2}.

¿es f inyectiva acá? Si.
¿es f sobreyectiva acá? Sí, porque cubro todos los elementos de la imagen: {-4, -2, 0, 2, 4}

Luego, **sí se puede aplicar cambio de variable.**

Ejercicio: Aplicarlo y verificar que da lo mismo usando la lectura operacional de ambas versiones.

T2: Eliminación de variable

⟨ ⊕ i , j : i = C ∧ R.i.j : T.i.j ⟩= ⟨ ⊕ j : R.C.j : T.C.j ⟩

Tenemos dos variables cuantificadas pero una de ellas tiene un valor fijo C. Luego esa variable no hace falta, podemos eliminarla y reemplazar toda ocurrencia (apariciones en R y T) de i por su valor C.

Se diferencia del rango unitario en que igual queda el cuantificador porque tengo más variables cuantificadas.

Demostración: Partimos del lado izquierdo y llegamos al lado derecho.
⟨ ⊕ i , j : i = C ∧ R.i.j : T.i.j ⟩
= { anidado }
⟨ ⊕ i : i = C : ⟨ ⊕ j : R.i.j : T.i.j ⟩ ⟩
= { rango unitario }

⟨ ⊕ j : R.C.j : T.C.j ⟩

T3: Rango unitario y condición

⟨ ⊕ i : i = C ∧ P.i : T.i ⟩= (P.C → T.C
[] ¬ P.C → e
)

donde e es el elemento neutro de ⊕.

Tengo una variable cuantificada i, y un predicado que me dice que i tiene un valor fijo pero que además debe satisfacer una condición adicional. Si la condición no vale, tengo rango vacío, si vale, tengo rango unitario.

Demostración: Partimos del lado izquierdo, llegamos al derecho:
⟨ ⊕ i : i = C ∧ P.i : T.i ⟩
= { Leibniz 2: i = C ∧ P.i es lo mismo que i = C ∧ P.C }
⟨ ⊕ i : i = C ∧ **P.C** : T.i ⟩

- (abrimos un análisis por casos)
- Primer caso: **P.C** (esto quiere decir “P.C es verdadero”)
  = { sustitución }
  ⟨ ⊕ i : i = C ∧ True : T.i ⟩
  = { lógica }
  ⟨ ⊕ i : i = C : T.i ⟩
  = { rango unitario }
  T.C
- Segundo caso: **¬ P.C**
  = { sust. }
  ⟨ ⊕ i : i = C ∧ False: T.i ⟩
  = { lógica }
  ⟨ ⊕ i : False : T.i ⟩
  = { rango vacío, siendo e el elemento neutro de ⊕ }
  e

T11: Leibniz 2

Esto es una propiedad de la lógica:

e = f ∧ E(z := e) ≡ e = f ∧ E(z := f)

Primero recordemos: X(a := b) que decir agarrar la expresión “X” y reemplazar toda ocurrencia de “a” por “b”.

Esto quiere decir que si tengo una conjunción entre una igualdad y una segunda expresión, en la segunda expresión puedo hacer reemplazo de iguales por iguales según indica la igualdad.

Ejemplo: $i = 37 \land \mathbf{x \bmod i = 2} \equiv i = 37 \land \mathbf{x \bmod 37 = 2}$

- "e" es i
- "f" es 37
- "E(z := e)" es x mod i = 2
    - Para que esto suceda, E debe ser (x mod z = 2).

Conteo

A8: Definición de conteo

Muchas veces vamos a querer escribir expresiones cuantificadas para contar cuántas veces sucede algo.

Ejemplo: ¿Cuántos números divisibles por 7 hay entre 0 y 100?

$\langle \sum i : 0 \leq i \leq 100 \land i \bmod 7 = 0 : 1 \rangle$

Lectura operacional: El rango es $i \in \{ 0, 7, 14, \dots , 70, 77, \dots . \}$
(uno por cada divisible por 7 entre 0 y 100). El resultado es:
1 + 1 + 1 + ….. + 1 + 1 + …. (tantas veces como elementos tenga en el rango)
Luego, el resultado es el que yo esperaba.
Fin del ejemplo

Como vamos a hacer esto muy seguido, nos vamos a inventar una notación para escribir este tipo de cuantificaciones (sumas de 1's), y la vamos a llamar **conteo**:

$$\langle N i : R.i : T.i \rangle = \langle \sum i : R.i \land T.i : 1 \rangle$$

(N mayúsculas es la notación para el cuantificador)

Importante: El conteo no es una expresión cuantificada de primer nivel (como las que venimos viendo) sino sólo una notación para ahorrar espacio. Por lo tanto, no valen necesariamente las reglas y axiomas que venimos viendo.

Ejemplo usando conteo:
$\langle N i : 0 \leq i \leq 100 : i \bmod 7 = 0 \rangle$

Observación: A diferencia de los cuantificadores de primer nivel, en el conteo el término siempre es un booleano y el resultado siempre es un número (o sea el tipo del término y el tipo del resultado son distintos).

La **lectura operacional** vista para los cuantificadores comunes ya no vale para el cuantificador de conteo (estaríamos sumando booleanos). Lo que hay que hacer es primero pasar a sumatoria usando def. de conteo.

Digesto

El conteo tiene su propio digesto: conteo.pdf

Reglas adicionales para el conteo

Recordemos la definición de conteo:
⟨ N i : R.i : T.i ⟩ = ⟨ ∑ i : R.i ∧ T.i : 1 ⟩

Para pasar cosas del rango al término:

⟨ N i : R.i : T.i ⟩ = ⟨ N i : R.i ∧ T.i : True ⟩

⟨ N i : R.i : T.i ⟩ = ⟨ N i : T.i : R.i ⟩

(ejercicio: demostrar)
(otro ejercicio: verificar con ejemplos)

Ejercicio 15b

⟨ N i : i - n = 1 : par.i ⟩
= { definición de conteo }
⟨ ∑ i : i - n = 1 ∧ par.i : 1 ⟩
= { aritmética }
⟨ ∑ i : i = 1 + n ∧ par.i : 1 ⟩
= { rango unitario y condición }
(par.(1 + n) → 1
[] ¬ par.(1 + n) → 0
)

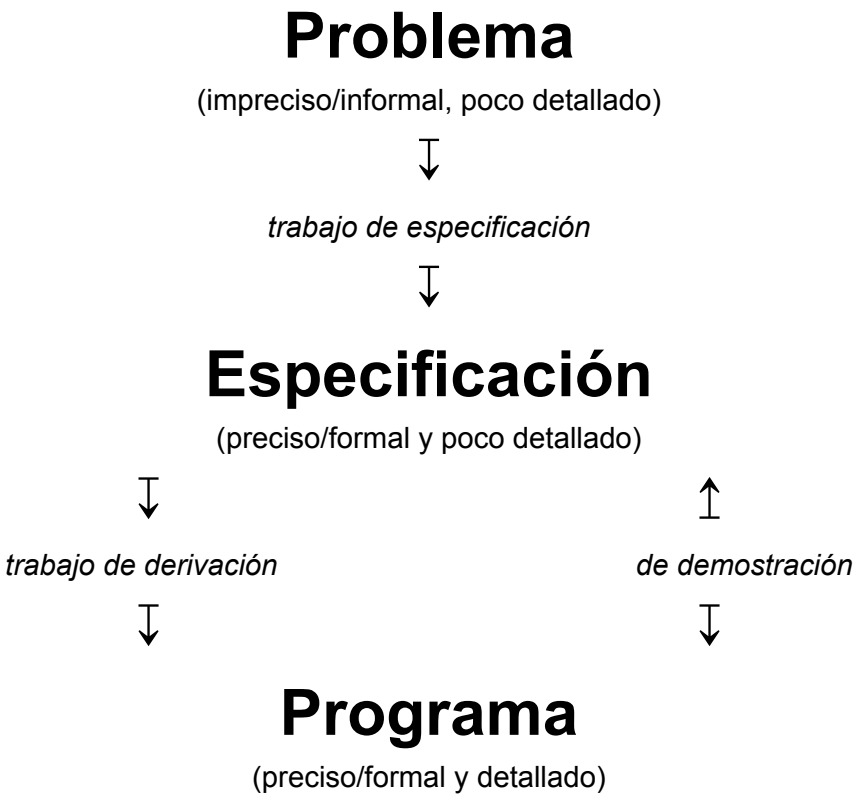
Ejemplo: si n=4, queda: 0 porque (n + 1) es impar. si n=37, queda 1 porque (n + 1) es par.

Ejercicios

practico1.pdf

Programación funcional

Introducción



Problemas

Lo que uno quiere solucionar, expresado de manera informal e imprecisa en lenguaje natural.

Ejemplo 1: “dado un número quiero saber si es impar”

Ejemplo 2: “dada una lista de números, quiero obtener su promedio”

Ejemplo 3: “dada una lista de números, quiero saber si todos sus elementos son iguales a un valor dado” (ejercicio 2b del práctico 2)

Especificaciones

Una especificación formaliza el problema a resolver de manera formal pero no detallada (o sea, expresa el qué pero no el cómo).

En el caso de programación funcional, decidimos resolver el problema a través de una función.

Luego, una especificación tendrá los siguientes componentes:

1. **Nombre para la función que resuelve el problema**
2. **Tipo de la función**
3. **Predicado que indica qué debe satisfacer la función (para que se pueda considerar que resuelve el problema)**

Ejemplos

Ejemplo 1: "dado un número quiero saber si es impar"

- nombre de la función: esImpar (esto es camel case, snake case es "es_impar")
- tipo de la función: Int → Bool
- **predicado:**

$$esImpar.n = (n \bmod 2 = 1)$$

Ejemplo 2: "dada una lista de números, quiero obtener su promedio"

- nombre de la función: promedio
- tipo de la función: [Int] → Float
- **predicado:**

$$promedio.xs = \langle \sum\ i : 0 \le i < \#xs : xs!i \rangle\ /\ \ \#xs$$

Observaciones:

- promedio.[ ] no está definido.
- Especificación precisa: Me distingue claramente cuáles funciones son solución.
- Poco detallada: Hay que sumar los elementos de la lista, pero a las especificación no le importa cómo se hace esto en la compu.

Ejemplo 3: "dada una lista de números, quiero saber si todos son iguales a un valor dado"

- nombre de la función: usamos "iga", como en el práctico
- tipo de la función: Int → [Int] → Bool
- predicado:

Recurso útil para especificar: hacer un ejemplo y usar "lectura operacional" pero hacia atrás. Supongamos que xs = [2, 4, -7] , e = 4, ¿qué predicado me interesa calcular?

$$(xs\ !\ 0 = e)\ \wedge\ (xs\ !\ 1 = e)\ \wedge\ (xs\ !\ 2 = e)$$

Esto me da una idea de que el término de lo que quiero cuantificar tiene la forma:

$$(xs\ !\ \mathbf{i} = e)$$

con i tomando valores { 0 , 1 , 2 }.

$$iga.e.xs = \langle\ \forall\ i : 0 \le i < \#xs :\ xs\ !\ i = e\ \rangle$$

Observación: En clase se propuso usar ∈ , pero no existe esto para listas (ver digesto de listas: listas.pdf). En introalg, se vio una ∈_L que sí funciona para listas, pero para poder usarlo tendríamos que especificarlo y derivarlo primero, así que no nos interesa complicarnos con eso. Recomendación para algoritmos 1: **Nunca usen ∈_L**.

Desarrollo sobre ∈_L: "sumar los elementos de una lista"

⟨ ∑ x : x ∈_L xs : x ⟩

¿es correcto esto? ¿realmente me suma todos los elementos de la lista?
Queda como ejercicio. Piensenlo. Y comparar con:

⟨ ∑ i : 0 ≤ i < #xs : xs ! i ⟩

Observación:: Todos los ejemplos vistos hasta ahora tienen la forma f.x = E (la función aplicada a sus parámetros es igual a algo). **No siempre una especificación es de esta forma**.

Ejemplo 4: "dados dos números, quiero obtener un factor común de ellos (que no sea el 1)".
Ejemplo: Si me dan el 44 y el 16, un factor común posible es 4, otro es 2.
Especifiquemos:

- nombre de la función: factorComun
- tipo: Int → Int → Int
- predicado de especificación:

  n mod x = 0 ∧ m mod x = 0 ∧ x ≠ 1

¿Está bien esta especificación? Está mal porque **no menciona a la función que está siendo especificada.** Una especificación siempre debe hablar de lo que tiene que satisfacer la función que va a resolver mi problema. Además acá se menciona una "x" que no se sabe qué es.

La arreglamos así, usando una **definición local**:

n mod x = 0 ∧ m mod x = 0 ∧ x ≠ 1
donde x = factorComun.n.m

O también sin usar definiciones locales:

n mod (factorComun.n.m) = 0 ∧
m mod (factorComun.n.m) = 0 ∧
(factorComun.n.m) ≠ 1

Una especificación me indica todo lo que es **necesario y suficiente** para que la función devuelva una solución correcta a mi problema.

¿Porqué una especificación es precisa y al mismo tiempo poco detallada?
Precisa: de todo el mundo de soluciones posibles (o sea todas las funciones de tipo: Int → Int → Int), la **especificación me distingue claramente cuáles resuelven el problema y cuales no**.
Poco detallada: Igual, puede haber más de una función que me resuelva el problema, y la especificación no tiene preferencia por ninguna de ellas (para el ejemplo que vimos con n=44 y m=16, podria dar 2 o 4).

Observación última: No necesariamente un problema tiene una única solución.

Observaciones

1. Usualmente llamamos especificación directamente al predicado de especificación (el nombre y el tipo de la función se deducen de ver el predicado).
2. El predicado de especificación **debe mencionar** a la función que está siendo especificada.
3. **No necesariamente debo usar expresiones cuantificadas.** Éstas son sólo una herramienta que puedo usar o no usar de acuerdo a mis necesidades.
4. Si uso expresiones cuantificadas, no necesariamente es lo que está afuera de todo en el predicado (en el ejemplo dividimos por #xs afuera). E incluso puedo tener varias expresiones cuantificadas operadas entre sí.
5. **No siempre** una especificación es de la forma: f.x = E (la función aplicada a sus parámetros es igual a algo).

Predicados

¿Qué predicados son válidos en una especificación? Básicamente todas las cosas formales que podemos tomar de la matemática, la lógica de predicados, la lógica de primer orden, y la cuantificación general. También usamos todos los tipos conocidos y sus funciones asociadas (ver Introducción a los algoritmos):

- Booleanos (∧ , ∧ , ¬ , etc.)
- Integers (+ , - , div , mod , max , min , etc.)
- Listas (! , # , ++ , etc)
- etc.

Programas

Los programas son **expresiones ejecutables** (yo suelo decir **"programables"**), es decir que tienen reglas asociadas que les permiten ser reducidas a otras expresiones (con suerte, que me permiten llegar a un valor final). En los programas, también podemos escribir definiciones. Las definiciones de las funciones crean nuevas reglas.

- Expresiones:
    - Constantes: 0, 42, -10, 3.14159, ∞, -∞, True, False, [ ] , [9], (3, True), etc.
    - Variables: n, x, xs, xss, etc.
    - Operaciones básicas: +, *, div, mod, max, min, ►, ∧, ∨, ¬, <, >, ≤, ≥. =, etc.
    - LLamadas a funciones
    - Análisis por casos
    - Definiciones locales
    - **No son expresiones válidas las expresiones cuantificadas**.
- Definiciones:

- Introducen nuevas funciones con sus respectivos nombres, tipos, y reglas de reducción.
- En las reglas de reducción se indica cómo se sustituye una aplicación de una función por una expresión válida en el lenguaje de programación.
- En las definiciones se puede usar pattern-matching (ver intro a los algoritmos).

En el contexto de la resolución de problemas, cada problema será resuelto con una **definición para la función** que fue especificada. La expresión que resuelve el problema es simplemente una llamada a la función con los parámetros apropiados.

Ejemplos

Ejemplo 1:
La especificación era:

$$esImpar.n = (n \bmod 2 = 1)$$

El programa se puede obtener directamente de la especificación ya que la especificación contiene todas operaciones permitidas en el lenguaje de programación (son ejecutables o "programables").

La **definición** de la función es directamente:

$$esImpar.n \doteq (n \bmod 2 = 1)$$

Aclaración importante: El símbolo "$\doteq$" quiere decir que estamos ante una definición. En una definición, **el lado izquierdo es algo nuevo que no existía antes** (que no estaba definido antes) y que está siendo definido ahora. No hace falta demostrar una definición, ya que la definición está estableciendo esa identidad. **Una definición no es un predicado.**
Se diferencia de la igualdad ("=") en que en la igualdad, ambos lados están previamente definidos, y lo que se está haciendo es **afirmar** que ambas cosas son iguales. Esta afirmación puede ser cierta o no (se puede demostrar o refutar). **Una igualdad es un predicado.**

En el ejemplo, la función "esImpar" no estaba definida previamente, pero sí estaba "especificada", esto quiere decir, estaba enunciado el predicado que la función debería satisfacer una vez que fuera definida.

Ejemplo 2:
La especificación era:

$$promedio.xs = \langle \sum\ i : 0 \le i < \#xs : xs!i \rangle\ /\ \ \#xs$$

En este caso, el programa no sale directamente de la especificación, ya que tenemos una expresión cuantificada (esto es algo **no programable**).

¿Cómo obtenemos el programa que me resuelve el problema?

1. **De la galera:** Como somos expertos programadores funcionales, sabemos escribir directamente la función que resuelve este problema.

2. **Derivando:** Como no somos expertos programadores funcionales, pero sí somos muy metódicos, vamos a aplicar estrategias conocidas que me permiten obtener el programa a partir de la especificación.

De la galera: (lo hago yo)

promedio :: [Int] → Float
promedio.xs ≐ sum.xs / #xs

sum.[ ] ≐ 0
sum.(x ►xs) ≐ x + sum.xs

¿Cómo sabemos que este programa satisface la especificación?
¿Con ejemplos? No, porque se nos puede escapar algún caso en el que no valga. Igual **es buena idea probar ejemplos**.

Debemos **demostrar** que la función satisface su especificación. Para eso usaremos todos los recursos matemáticos conocidos, incluyendo el principio de inducción de ser necesario.

Ejemplo 3:
La especificación era:

$$iga.e.xs = \langle\ \forall\ i : 0 \le i < \#xs :\ xs\ !\ i = e\ \rangle$$

Un programa posible es (sacado de la galera):

iga : Int -> [Int] -> Bool
iga.e.[ ] ≐ True
iga.e.(x ►xs) ≐ (x = e) ∧ iga.e.xs

Recordar: Siempre usar puntitos "." al aplicar parámetros a funciones.

Demostración

Dados una especificación y un programa, podemos demostrar que el programa satisface la especificación.

Ejemplo 2: Queremos demostrar que la función definida como:

promedio :: [Int] → Float
promedio.xs ≐ sum.xs / #xs

sum.[ ] ≐ 0
sum.(x ►xs) ≐ x + sum.xs

Satisface el siguiente predicado: **Para toda** lista posible xs ≠ [ ] .

$$promedio.xs = \langle\ \sum\ i : 0 \le i < \#xs : xs!i\ \rangle\ /\ \ \#xs$$

Como tengo que demostrar un “para todo” sobre listas, puedo usar inducción sobre listas. Pero en este caso el **caso base** es con listas de un elemento (xs es de la forma [x] , lista de un solo elemento x).

Caso base: Supongamos que xs = [x] . Agarremos todo y tratemos de llegar a True (que es lo más seguro ya que podemos aplicar propiedades de ambos lados).

promedio.[x] = ⟨ ∑ i : 0 ≤ i < #[x] : [x]!i ⟩ / #[x]
= { def. de # }
promedio.[x] = ⟨ ∑ i : 0 ≤ i < 1 : [x] ! i ⟩ / 1
= { lógica }
promedio.[x] = ⟨ ∑ i : i = 0 : [x] ! i ⟩ / 1
= { rango unitario }
promedio.[x] = [x] ! 0 / 1
= { definición de ! }
promedio.[x] = x / 1
= { aritmética }
promedio.[x] = x
= { def. de promedio }
sum.[x] / #[x] = x
= { def. # y aritmética }
sum.[x] = x
= { hago explícito el patrón }
sum.(x ► [ ]) = x
= { def. de sum }
x + sum.[ ] = x
= { def. de sum }
x + 0 = x
= { lógica }
True

Paso inductivo: Queda como ejercicio.

Derivación

Es el proceso **creativo** pero metódico que permite obtener un programa a partir de una especificación.

Como el lenguaje de los programas es un subconjunto del lenguaje de las especificaciones

Ejemplo 3 (práctico 2 ejercicio 2b): Recordemos la especificación:

$$iga.e.xs = \langle\ \forall\ i : 0 \le i < \#xs :\ xs\ !\ i = e\ \rangle$$

¿Cómo obtenemos un programa a partir de esta especificación?
Observamos que tenemos una expresión cuantificada. Esto quiere decir que tenemos una cantidad indeterminada de términos operados entre sí (una conjunción de cosas, cuya cantidad de términos depende de los parámetros de entrada). ¿Cuántos términos tiene?

Tiene #xs términos. Para resolver eso con mi lenguaje de programación, el único recurso que tengo es el de definir una **función recursiva**. Luego, vamos a necesitar aplicar inducción para obtener esta definición recursiva. La función se va a definir de la siguiente forma:

Caso base:

iga.e.[ ] ≐ ???

Caso recursivo/Paso inductivo:

iga.e.(x►xs) ≐ ????

(seguramente acá va a aparecer la llamad recursiva "iga.e.xs")

Intentemos resolver todas las incógnitas que tenemos **usando siempre la especificación**.

Caso base:

iga.e.[ ]
= { especificación de iga }
⟨ ∀ i : 0 ≤ i < #[ ] : [ ] ! i = e ⟩
= { def # }
⟨ ∀ i : 0 ≤ i < 0 : [ ] ! i = e ⟩
= { lógica }
⟨ ∀ i : False : [ ] ! i = e ⟩
= { rango vacío }
True

Paso inductivo:

En la derivación, voy a partir de "iga.e.(x►xs)" y quiero llegar a algo que sea ejecutable / programable. En el medio, seguramente voy a necesitar hacer que aparezca la llamada recursiva "iga.e.xs". Para eso necesito plantear mi Hipótesis Inductiva:

H.I. : iga.e.xs = ⟨ ∀ i : 0 ≤ i < #xs : xs ! i = e ⟩

Vamos a eso:

iga.e.(x►xs)
= { especificación }
⟨ ∀ i : 0 ≤ i < #(x►xs) : (x►xs) ! i = e ⟩
= { def # }
⟨ ∀ i : 0 ≤ i < #xs + 1 : (x►xs) ! i = e ⟩
= { 0 ≤ i < #xs + 1 es lo mismo que (i = #xs) ∨ (0 ≤ i < #xs)
0 ≤ i < #xs + 1 también es lo mismo que (i = 0) ∨ (1 ≤ i < #xs+1)
Acá puedo aplicar cualquiera de las dos, pero sólo una me sirve para llegar a la H.I.
Probemos la primera. (**vamos a ver que no anda**) }
⟨ ∀ i : 0 ≤ i < #xs ∨ i = #xs : (x►xs) ! i = e ⟩
= { partición de rango }
⟨ ∀ i : 0 ≤ i < #xs : (x►xs) ! i = e ⟩ ∧
⟨ ∀ i : i = #xs : (x►xs) ! i = e ⟩
= { ¿en qué terminó me conviene concentrarme? Siempre primero me conviene ver si puedo llegar a la H.I., porque si no se puede, todos estos pasos no sirven.
Me conviene concentrarme en el 1er término.
¿podemos aplicar alguna regla que me lleve a la H.I.? me falta que el término quede bien, podria intentar hacer cambio de variable i → i + 1 . }
⟨ ∀ i : 0 ≤ i + 1 < #xs : (x►xs) ! (i + 1) = e ⟩ ∧

〈 ∀ i : i = #xs : (x►xs) ! i = e 〉
= { def. ! }
〈 ∀ i : 0 ≤ i + 1 < #xs : xs ! i = e 〉 ∧
〈 ∀ i : i = #xs : (x►xs) ! i = e 〉
= { Ahora se me arregló el término para la H.I., pero se rompió el rango.
NO HAY FORMA DE LLEGAR A LA H.I. **NO ERA POR ACÁ**.}
= { APLICAMOS LA OTRA ESTRATEGIA:
0 ≤ i < #xs + 1 es lo mismo que (i = 0) ∨ (1 ≤ i < #xs+1) }
〈 ∀ i : (i = 0) ∨ (1 ≤ i < #xs+1) : (x►xs) ! i = e 〉
= { partición de rango }
〈 ∀ i : i = 0 : (x►xs) ! i = e 〉 ∧
〈 ∀ i : 1 ≤ i < #xs+1 : (x►xs) ! i = e 〉
= { probemos de nuevo cambio de variable i → i + 1}
〈 ∀ i : i = 0 : (x►xs) ! i = e 〉 ∧
〈 ∀ i : 1 ≤ i + 1 < #xs+1 : (x►xs) ! (i + 1) = e 〉
= { algebra (resto 1 en los tres miembros de la desigualdad) }
〈 ∀ i : i = 0 : (x►xs) ! i = e 〉 ∧
〈 ∀ i : 0 ≤ i < #xs : (x►xs) ! (i + 1) = e 〉
= { def. ! }
〈 ∀ i : i = 0 : (x►xs) ! i = e 〉 ∧
〈 ∀ i : 0 ≤ i < #xs : xs ! i = e 〉
= { llegamos a la Hipótesis Inductiva !! }
〈 ∀ i : i = 0 : (x►xs) ! i = e 〉 ∧ iga.e.xs
= { rango unitario }
(x►xs) ! 0 = e ∧ iga.e.xs
= { acá ya es un programa correcto pero igual puedo simplificar aplicando def ! }
x = e ∧ iga.e.xs

Listo. Resultado final:

iga : Int -> [Int] -> Bool
iga.e.[ ] ≐ True
iga.e.(x ►xs) ≐ (x = e) ∧ iga.e.xs

Testing / Ejecución

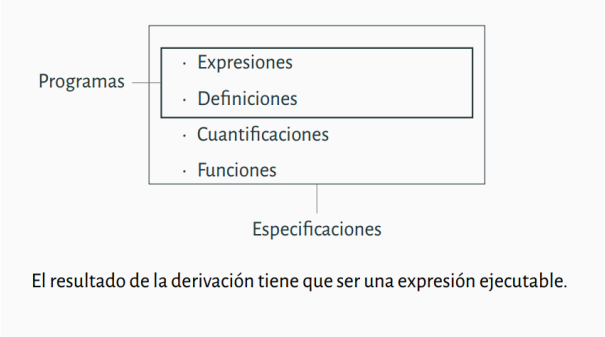
(reducción, cálculo, etc.)

Ejemplo (ejercicio 3 del practico 2): Vamos a **testear** la función ahora a ver si funciona como esperamos.
Probemos con xs = [5, 5, -7] , e = 5. Vamos a ejecutar el programa (o sea reducirlo a un valor canónico).

iga.5.[5, 5, -7] ↠ (5 = 5) ∧ iga.5.[5, -7]
↠ (5 = 5) ∧ ((5 = 5) ∧ iga.5.[-7])
↠ (5 = 5) ∧ ((5 = 5) ∧ ((-7 = 5) ∧ iga.5.[ ]))

↠ (5 = 5) ∧ ((5 = 5) ∧ ((-7 = 5) ∧ True))
↠ (5 = 5) ∧ ((5 = 5) ∧ (False ∧ True))
….
↠ False

Resumen: El lenguaje de programación funcional



Expresiones: Es lo que efectivamente se ejecuta.

1. Constantes y variables.
2. Operadores
3. Llamadas a funciones
4. Análisis por casos
5. Definiciones locales

Definiciones: Para definir funciones que pueden ser usadas (y re-usadas) en expresiones (llamadas).

- Reglas de reducción (de la forma f.x ≐ E)
- Pattern matching

Tipos:

1. Tipos básicos: Bool, Nat, Int, Char, Num o Float.
2. Listas
3. Tuplas
4. Funciones

Modularización

Ejemplo: Promedio de una lista

Ejemplo: “Dada una lista de números, quiero obtener su promedio.”

La especificación era:

$$promedio : [Num] \to Num$$
$$promedio.xs = \langle \sum\ i : 0 \le i < \#xs : xs!i \rangle\ /\ \ \#xs$$

Derivación: Queremos obtener un programa a partir de la especificación.
¿Es mi especificación ya directamente ejecutable (programable)?
No porque contiene una expresión cuantificada que no es ejecutable/programable.
Sin embargo, todo el resto (la división y el denominador) sí es programable. Luego podemos inventar una función nueva para la parte que no es programable y derivarla aparte.
Esta función me suma los elementos de la lista así que la llamaremos sum.
Especificamos la nueva función:

$$sum : [Num] \to Num$$
$$sum.xs = \langle \sum\ i : 0 \le i < \#xs : xs!i \rangle$$

Separamos la derivación en dos partes. Primero la derivación de promedio, después la de sum.

Derivación de promedio: Sale derecho así:

promedio.xs
= { especificación }
$\langle \sum\ i : 0 \le i < \#xs : xs!i \rangle\ /\ \ \#xs$
= { **introducimos modularización con función sum.xs =** $\langle \sum\ i : 0 \le i < \#xs : xs!i \rangle$ }
sum.xs / #xs

Listo la derivación de promedio. Resultado parcial:

promedio.xs ≐ sum.xs / #xs

Derivación de sum:
Especificación: $sum.xs = \langle \sum\ i : 0 \le i < \#xs : xs!i \rangle$

Acá tenemos que hacer inducción en xs. Tendremos dos casos:

- Caso base: sum.[ ] ≐ ???
- Paso inductivo: sum.(x►xs) ≐ ???? (mencionando a sum.xs)

Ejercicio: Derivar!

Resultado final:

promedio.xs ≐ sum.xs / #xs

sum.[ ] ≐ 0
sum.(x ►xs) ≐ x + sum.xs

Testing / verificación: Probar la función promedio con una lista de ejemplo: xs = [7, 4, 6, 2]. **Ejercicio!!**

Ejemplo: Suma de potencias

Otro ejemplo (ejercicio 4a del práctico 2):

Especificación:
sum_pot : Num → Nat → Num
sum_pot.x.n = ⟨ ∑ i : 0 ≤ i < n : x^i ⟩

¿qué hace sum_pot? Computa la suma de potencias de un número.
Ejemplo: x = 3.33 , n = 4.
sum_pot.x.n = $3.33^0 + 3.33^1 + 3.33^2 + 3.33^3$
Vamos a suponer que x ≠ 0 ya que si no, mi función no va a estar definida.

Derivación: Necesitamos hacer inducción en n. (en x no tiene sentido ya que no existe la inducción sobre los reales). Mi programa resultado va a tener la forma:

Caso base: sum_pot.x.0 ≐ ???
Caso recursivo/paso inductivo: sum_pot.x.(n+1) ≐ ??? (en términos de sum_pot.x.n)

Vamos a eso.
Caso base:
sum_pot.x.0
= { especificación }
⟨ ∑ i : 0 ≤ i < 0 : x^i ⟩
= { lógica y rango vacío }
0

Paso inductivo: Hipótesis Inductiva: sum_pot.x.n = ⟨ ∑ i : 0 ≤ i < n : x^i ⟩

sum_pot.x.(n + 1)
= { especificación }
⟨ ∑ i : **0 ≤ i < n + 1** : x^i ⟩
= { tenemos dos opciones de lógica, probemos primero: 0 ≤ i < n ∨ i = n }
⟨ ∑ i : 0 ≤ i < n ∨ i = n : x^i ⟩
= { partición de rango ya q son disjuntas las dos partes }
⟨ ∑ i : 0 ≤ i < n : x^i ⟩ + ⟨ ∑ i : i = n : x^i ⟩
= { Hipótesis Inductiva }
sum_pot.x.n + ⟨ ∑ i : i = n : x^i ⟩
= { Rango unitario }
sum_pot.x.n + x^n
= { x^n no es programable (así definimos el lenguaje), así que **introducimos una modularización "exp" especificada por exp.x.n = x^n**}
sum_pot.x.n + exp.x.n

Ya llegamos a algo programable. Terminamos de derivar sum_pot (falta exp).

Resultado parcial:

sum_pot.x.0 ≐ 0
sum_pot.x.(n + 1) ≐ sum_pot.x.n + exp.x.n

Derivación de exp: Especificación: exp.x.n = x^n

¿Cómo derivamos? Esto es como un cuantificador, sale por inducción en n.

Caso base:
  exp.x.0
= { especificación }
  x^0
= { arit }
  1

Paso inductivo: Hipótesis Inductiva: exp.x.n = x^n
  exp.x.(n + 1)
= { especificación }
  x^{n+1}
= { prop. de la exponenciación }
  $x^n * x$
= { H.I. }
  exp.x.n * x

¡¡Listo!! Resultado final:

sum_pot.x.0 ≐ 0
sum_pot.x.(n + 1) ≐ sum_pot.x.n + exp.x.n

exp.x.0 ≐ 1
exp.x.(n + 1) ≐ exp.x.n * x

Resumen: ¿Qué hicimos?

En el medio de una derivación, apareció una expresión no programable (por ejemplo una expresión cuantificada) (y que no es la H.I. si es que estoy en el medio de una inducción). Lo que hacemos al modularizar es inventar una nueva función cuya especificación me indica que calcula esa expresión. Luego, puedo usar esa función en mi derivación para reemplazar la expresión no programable por una llamada a la función.
Para terminar el programa, debemos obtener aparte un programa para la función que modularicé (haciendo otra derivación).

Usualmente, la función modularizada lo que hace es resolver un **problema accesorio** del problema original que estoy resolviendo.
(por ejemplo, para el promedio de una lista, la suma de la lista,
 otro ejemplo, para la suma de potencias, las potencias mismas)

NUNCA VAMOS a aplicar por anticipado la estrategia de modularización. Solamente vamos a aplicar modularización en el momento en el que surja la necesidad en el medio de una derivación.

APARTADO: Derivación de sum_pot usando una estrategia diferente (sale sin modularización):

Caso base:

sum_pot.x.0
= {... }
0

Paso inductivo: Hipótesis Inductiva: $sum\_pot.x.n = \langle \sum i : 0 \le i < n : x^i \rangle$

sum_pot.x.(n + 1)
= { especificación }
$\langle \sum i : \mathbf{0 \le i < n + 1} : x^i \rangle$
= { tenemos dos opciones de lógica, probemos ahora **la otra opción**: $i = 0 \lor 1 \le i < n + 1$ }
$\langle \sum i : i = 0 \lor 1 \le i < n + 1 : x^i \rangle$
= { partición de rango }
$\langle \sum i : i = 0 : x^i \rangle + \langle \sum i : 1 \le i < n + 1 : x^i \rangle$
= { en la 2da, cambio de variable $i \rightarrow i + 1$ }
$\langle \sum i : i = 0 : x^i \rangle + \langle \sum i : 1 \le i + 1 < n + 1 : x^{i+1} \rangle$
= { aritmética }
$\langle \sum i : i = 0 : x^i \rangle + \langle \sum i : 0 \le i < n : x^{i+1} \rangle$
= { prop. de exponenciación }
$\langle \sum i : i = 0 : x^i \rangle + \langle \sum i : 0 \le i < n : x^i * x \rangle$
= { distributividad }
$\langle \sum i : i = 0 : x^i \rangle + \langle \sum i : 0 \le i < n : x^i \rangle * x$
= { H.I. }
$\langle \sum i : i = 0 : x^i \rangle +$ sum_pot.x.n * x
= { rango unitario }
$x^0 +$ sum_pot.x.n * x
= { aritmética }
1 + sum_pot.x.n * x

¡¡Listo!! Resultado final:

sum_pot.x.0 $\doteq$ 0
sum_pot.x.(n + 1) $\doteq$ 1 + sum_pot.x.n * x

No hizo falta hacer modularización!!

Acabo de obtener dos programas distintos que calculan exactamente lo mismo (o sea resuelven el mismo problema) de dos maneras distintas.

Testing / verificación:
Ejemplo: x = 3.33 , n = 4.
Me debería dar: sum_pot.x.n = $3.33^0 + 3.33^1 + 3.33^2 + 3.33^3$

sum_pot.(3.33).4 = sum_pot.(3.33).(3+1) // luego aplica el patrón n+1 con n=3.
→ 1 + sum_pot.(3.33).3 * (3.33)
→ 1 + (1 + sum_pot.(3.33).2 * (3.33)) * (3.33)
→ 1 + (1 + (1 + sum_pot.(3.33).1 * (3.33)) * (3.33)) * (3.33)
→ 1 + (1 + (1 + (1 + sum_pot.(3.33).0 * (3.33)) * (3.33)) * (3.33)) * (3.33) // acá apliqué el caso base
→ 1 + (1 + (1 + (1 + 0 * (3.33)) * (3.33)) * (3.33)) * (3.33)
→ 1 + (1 + (1 + (1 + 0) * (3.33)) * (3.33)) * (3.33)
→ 1 + (1 + (1 + 1 * (3.33)) * (3.33)) * (3.33)
→ 1 + (1 + (1 + 3.33) * (3.33)) * (3.33)
= 1 + (1 + 3.33 + 3.33^2) * (3.33) // por distributividad
= 1 + 3.33 + 3.33^2 + 3.33^3 // por distributividad

(el resultado es 52.344937)

Inducción

Usamos el inducción a la hora de derivar programas que calculan expresiones cuantificadas (o “esconden” una cuantificación, como la exponenciación o el factorial). La especificación de la función debe ser exactamente una expresión cuantificada, sin ninguna operación extra por fuera de la cuantificación. El resultado será una función recursiva. En una ejecución, la recursión se encargará de recorrer todos los términos representados por la expresión cuantificada.

Esquemas de inducción

Hay muchas formas de derivar usando inducción. Pero siempre cada forma está asociada a un teorema que me dice que efectivamente la inducción aplicada vale.

Supongamos que tenemos una función f sobre los números naturales y su predicado de especificación P, un predicado sobre la función y los naturales.

Ejemplo: El factorial de un número: La función es fac, y el predicado de especificación es:

P.n ≡ fac.n = n!

Lo que queremos obtener es una definición para la función “fac” tal que valga P para todo n:

$\forall$ P.n (o sea: P.0 $\wedge$ P.1 $\wedge$ P.2 $\wedge$ P.3 $\wedge$ …….)

¿Qué nos dice el teorema de inducción? Lo siguiente:

$\forall$ n : P.n (o sea: P.0 $\wedge$ P.1 $\wedge$ P.2 $\wedge$ P.3 $\wedge$ …)
$\equiv$
P.0 $\wedge$ ($\forall$ n : P.n $\Rightarrow$ P.(n+1)) (o sea: P.0 $\wedge$ (P.0 $\Rightarrow$ P.1) $\wedge$ (P.1 $\Rightarrow$ P.2) $\wedge$ (P.2 $\Rightarrow$ P.3) $\wedge$ …)

En esta equivalencia, veamos:
¿Vale la ida $\Rightarrow$? Sí, muy fácilmente porque todo es True, P.0, P.n y P.(n+1).
¿Vale la vuelta $\Leftarrow$? Es un poquito más complicado.
La hipótesis es:
P.0 $\wedge$ ($\forall$ n : P.n $\Rightarrow$ P.(n+1))
Queremos demostrar:
$\forall$ n : P.n (o sea: P.0 $\wedge$ P.1 $\wedge$ P.2 $\wedge$ P.3 $\wedge$ …)
¿Vale P.0? Sí, muy fácilmente porque es una de mis hipótesis.
¿Vale P.1? Sí, porque tengo hipótesis P.0 $\Rightarrow$ P.1, y además sé que vale P.0.
¿Vale P.2? Sí, porque tengo hipótesis P.1 $\Rightarrow$ P.2, y además acabo de ver que vale P.1
Y así vale para siempre: P.3, P.4, ….

La idea de la inducción es que hay uno o más casos base (cosas que valen de entrada) y otros casos en los que cuando una cosa vale, hacen valer una cosa nueva:

~~P.0~~ $\wedge$ ~~P.1~~ $\wedge$ ~~P.2~~ $\wedge$ ~~P.3~~ $\wedge$ …

Ahora, esquemas de este tipo hay muchos más, siempre y cuando garanticen que el predicado vale para todo n.

<u>Otro esquema:</u>

P.0 $\wedge$ P.1 $\wedge$ ($\forall$ n : P.n $\wedge$ P.(n+1) $\Rightarrow$ P.(n+2))

Este es el que se usa para fibonacci:
fib.0 $\doteq$ 0
fib.1 $\doteq$ 1
fib.(n+2) $\doteq$ fib.(n+1) + fib.n)

¿Vale este esquema? Verifiquemos:

~~P.0~~ $\wedge$ ~~P.1~~ $\wedge$ ~~P.2~~ $\wedge$ ~~P.3~~ $\wedge$ ~~P.4~~ $\wedge$ …

<u>Otro esquema:</u>

P.0 $\wedge$ P.1 $\wedge$ ($\forall$ n : P.n $\Rightarrow$ P.(n+2))

¿Vale este esquema? Verifiquemos:

~~P.0~~ ∧ ~~P.1~~ ∧ ~~P.2~~ ∧ ~~P.3~~ ∧ ~~P.4~~ ∧ …

Otro esquema:

P.0 ∧ (∀ n : P.n ⇒ P.(n+2))

¿Vale este esquema? No!! Verifiquemos:

~~P.0~~ ∧ P.1 ∧ ~~P.2~~ ∧ P.3 ∧ ~~P.4~~ ∧ …

A la hora de hacer derivaciones, debemos usar un esquema inductivo que nos garantice que la especificación vale para todos los elementos del dominio de mi función.
La definición de la función va a tener un **pattern matching** que es acorde al esquema inductivo usado.

Inducciones con naturales

Inducción básica:
f.0 ≐ ???
f.(n+1) ≐ ??? (en términos de f)

fac.0 ≐ 1
fac.(n+1) ≐ (n+1) * fac.n

Inducción a lo fibonacci:
fib.0 ≐ 0
fib.1 ≐ 1
fib.(n+2) ≐ fib.(n+1) + fib.n

Otras inducciones:
f.1 ≐ 1
f.(n+2) ≐ (n+2) * f.n

(función sólo definida para los números impares)
(es una función que me acabo de inventar y no tengo idea qué calcula)
(ah, es como el factorial pero sólo de impares: f.9 = 9 * 7 * 5 * 3 * 1).

Inducciones con listas

Inducción básica:
f.[ ] ≐ ???
f.(x ► xs) ≐ ??? (en función de f.xs)

(ejemplos: iga, sum)

Inducción con dos o más elementos:
f.[ ] ≐ ???
f.[x] ≐ ???
f.(x ►y►xs) ≐ ??? (en función de f.(y►ys), o también f.ys)
(sirve para las funciones iguales, mínimo y creciente)

Inducciones combinadas: Sub-inducción

A veces vamos a tener más de un parámetro sobre el cual se puede aplicar inducción.
La estrategia general que vamos a tomar es hacer inducción en uno solo de ellos.
Posiblemente, en la derivación del caso base o del paso inductivo, surja la necesidad de hacer inducción también en el otro

Dos naturales:
Tenemos una función f : Nat → Nat → Nat.
Hacemos inducción en el primer parámetro:

- Caso base: f.0.m ≐ ???? (salió bien la derivación)
- Paso inductivo: f.(n+1).m ≐ ??? (acá nos trabamos al derivar y vemos que necesitamos hacer inducción también en m). Lo llamamos **sub-inducción**.
    - Caso base para m: f.(n+1).0 ≐ ????
    - Paso inductivo para m: f.(n+1).(m+1) ≐ ??? (en términos de f.n.m).

La función queda definida así:
f.0.m ≐ ????
f.(n+1).0 ≐ ???
f.(n+1).(m+1) ≐ ??? (en términos de f.n.m)

Dos listas: También puede haber necesidad de hacer subinducción.

Ejemplo: (práctico 2 ejercicio 5d) prod : [Num] → [Num] → Num, que calcula el producto entre pares de elementos en iguales posiciones de las listas y suma estos resultados (producto punto). Si las listas tienen distinto tamaño se opera hasta la última posición de las más chica.

prod.xs.ys = ⟨ ∑ i : 0 ≤ i < #xs min #ys : (xs ! i) * (ys ! i) ⟩

Un posible resultado de derivar sería:

prod.[ ].ys ≐ 0
prod.(x ►xs).[ ] ≐ 0
prod.(x ►xs).(y ►ys) ≐ x * y + prod.xs.ys

Un natural y una lista: Se ve mucho en las funciones de listas estándar.

Ejemplos: Indexación:
(x►xs) ! 0 ≐ x
(x►xs) ! (n+1) ≐ xs ! n

(acá hay inducción en n)

Tirar: "xs ↓ n = tirar los primeros n elementos de xs"
[ ] ↓ n ≐ [ ]
(x ►xs) ↓ 0 ≐ (x ►xs)
(x ►xs) ↓ (n+1) ≐ xs ↓ n

(acá hay inducción en los dos parámetros, primero en la lista, y luego **solo en el caso inductivo** hacemos también subinducción en el número.)

¿Cómo sería si hacemos primero inducción en el número?
xs ↓ 0 ≐ xs
[ ] ↓ (n+1) ≐ [ ]
(x ►xs) ↓ (n+1) ≐ xs ↓ n

Esta definición es equivalente a la anterior pero usa un esquema inductivo distinto. Primero inducción en el número y después, dentro del caso inductivo, sub-inducción en la lista.

APARTADO: FUNCIONES CON NOTACION INFIJA, PREFIJA Y SUFIJA

Notación infija: Una función cuya llamada se pone al medio de los dos parámetros (como un operador): x * y , xs ! 6 , etc.
Notación prefija: Una función cuya llamada se pone antes de los parámetros: tirar.6.xs, tail.xs, iga.7.xs, #xs , etc.
Notación sufija: la función al final. Ejemplo: n!

Generalización

Ejemplo: Función psum

Práctico 3 - Ejercicio 1a:
Tenemos esta especificación:
psum : [Num] → Bool
psum.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.(xs↑i) ≥ 0 ⟩

¿Qué calcula esta función? En palabras:
Algo que no es: “Me dice si la suma de toda la lista es ≥ 0”.
Si fuera esto, sería psum.xs = sum.xs ≥ 0
Una posible: “La suma de los n primeros elementos siempre es ≥ 0 para todo n posible.”
Otra: “Todos los segmentos iniciales de la lista suman ≥ 0.”

Definición: Un **segmento inicial (o prefijo)** de una lista xs es una lista que está “al principio de xs”. Formalmente, es una lista ys tal que: xs = ys ++ as (para alguna otra lista as).
Ejemplo: Los segmentos iniciales de la lista xs = [4, -3, 5, -7, 10] son:
[ ], [4] , [4, -3], [4, -3, 5], [4, -3, 5, -7], [4, -3, 5, 7, 10].
Observaciones:

- ¿Cuántos segmentos iniciales tiene una lista xs? #xs + 1
- Tanto la lista vacía [ ] como la lista entera xs son siempre considerados segmentos iniciales.

Ejemplo: xs = [4, -3, 5, -7, 10].
Lectura operacional para psum:

1. El rango es: i ∈ { 0, 1, 2, 3, 4, 5 } (porque es con ≤)
2. Los términos son:
   sum.([4, -3, 5, -7, 10]↑0) ≥ 0 ∧
   sum.([4, -3, 5, -7, 10]↑1) ≥ 0 ∧
   sum.([4, -3, 5, -7, 10]↑2) ≥ 0 ∧
   sum.([4, -3, 5, -7, 10]↑3) ≥ 0 ∧
   sum.([4, -3, 5, -7, 10]↑4) ≥ 0 ∧
   sum.([4, -3, 5, -7, 10]↑5) ≥ 0 ∧
   O sea, resolviendo ↑ :
   sum.([ ]) ≥ 0 ∧
   sum.([4]) ≥ 0 ∧
   sum.([4, -3]) ≥ 0 ∧
   sum.([4, -3, 5]) ≥ 0 ∧
   sum.([4, -3, 5, -7]) ≥ 0 ∧
   sum.([4, -3, 5, -7, 10]) ≥ 0 ∧
   Resolviendo las sumas:
   0 ≥ 0 ∧ 4 ≥ 0 ∧ 1 ≥ 0 ∧ 6 ≥ 0 ∧ -1 ≥ 0 ∧ 9 ≥ 0
   O sea:
   False

Observación: En la especificación de psum, la variable cuantificada “i” indica cuántos elementos tiene el segmento inicial.

Derivación: Intentaremos hacerlo por inducción en xs.
Si resulta, el programa tendrá la forma:

psum.[ ] ≐ ???
psum.(x►xs) ≐ ???

Caso base:

psum.[ ]
= { especificación }
⟨ ∀ i : 0 ≤ i ≤ #[ ] : sum.([ ]↑i) ≥ 0 ⟩
= { def. # }
⟨ ∀ i : 0 ≤ i ≤ 0 : sum.([ ]↑i) ≥ 0 ⟩
= { lógica }
⟨ ∀ i : i = 0 : sum.([ ]↑i) ≥ 0 ⟩
= { rango unitario }
sum.([ ]↑0) ≥ 0
= { def ↑ y de sum }
0 ≥ 0
= { lógica }
True

Paso inductivo: Hipótesis Inductiva: psum.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.(xs↑i) ≥ 0 ⟩

psum.(x►xs)
= { especificación }
⟨ ∀ i : 0 ≤ i ≤ #(x►xs) : sum.((x►xs)↑i) ≥ 0 ⟩
= { def de # }
⟨ ∀ **i : 0 ≤ i ≤ #xs + 1 : sum.((x►xs)↑i) ≥ 0** ⟩
= { lógica }
⟨ ∀ i : i = 0 ∨ 1 ≤ i ≤ #xs + 1 : sum.((x►xs)↑i) ≥ 0 ⟩
= { partición de rango
(observación: estamos separando el caso del segmento inicial vacío (i = 0) de los segmentos iniciales no vacíos (1 ≤ i ≤ #xs + 1)
}
⟨ ∀ i : i = 0 : sum.((x►xs)↑i) ≥ 0 ⟩ ∧
⟨ ∀ i : 1 ≤ i ≤ #xs + 1 : sum.((x►xs)↑i) ≥ 0 ⟩
= { **Acá la prioridad es llegar a la H.I. así que nos concentramos en la segunda parte.**
Cambio de variable i → i + 1. }
⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 1 ≤ i + 1 ≤ #xs + 1 : sum.((x►xs)↑(i+1)) ≥ 0 ⟩
= { arit: resto 1 }
⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.((x►xs)↑(i+1)) ≥ 0 ⟩
= { def de ↑ }
⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.(x ► (xs↑i)) ≥ 0 ⟩
= { def de sum }
⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 0 ≤ i ≤ #xs : x + sum.((xs↑i)) ≥ 0 ⟩

Recordemos la H.I.: psum.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.(xs↑i) ≥ 0 ⟩.
No hay forma de llegar a la H.I., no hay nada que podamos hacer para deshacernos de ese cacho “x + “ que molesta.
No podemos seguir. Peeeeeeeeeeeero: Si mi especificación original hubiera sido ligeramente diferente, acá capaz sí podría funcionar la H.I.
Podría **especificar** por ejemplo así:

gpsum.n.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : n + sum.(xs↑i) ≥ 0 ⟩

¿Qué acabo de hacer acá? Acabo de especificar una función nueva, parecida a la psum pero que tiene un parámetro adicional “n” (que además aparece sumado en un lugar que va a ser conveniente para la derivación).

¿Cómo se relaciona gpsum con psum?

- psum.xs **es caso particular** de gpsum.n.xs cuando n = 0. Más formalmente:
  psum.xs = gpsum.0.xs.
- gpsum **generaliza** a psum

Si yo tuviera un programa para gpsum ya resuelto, podría derivar psum y obtener un programa de la siguiente manera:

Nueva derivación de psum:

```
  psum.xs
= { especificación psum }
  ⟨ ∀ i : 0 ≤ i ≤ #xs :  sum.(xs↑i) ≥ 0 ⟩
= { aritmética }
  ⟨ ∀ i : 0 ≤ i ≤ #xs :  0  +  sum.(xs↑i) ≥ 0 ⟩
= { especificación gpsum }
  gpsum.0.xs
```

Resultado: psum.xs ≐ gpsum.0.xs

Esto quiere decir que la derivación por inducción que habíamos intentado antes **se descarta**:

~~psum.[ ] ≐ ???~~
~~psum.(x►xs) ≐ ???~~

Sólo me sirvió para darme cuenta de que debo generalizar.
(por eso conviene empezar por el paso inductivo, para ahorrar tiempo)

¿Podemos derivar un programa para gpsum? Intentemos.

Derivación de gpsum: Especificación:

gpsum.n.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : n + sum.(xs↑i) ≥ 0 ⟩

Parecido a lo que hicimos con psum. Intentamos con inducción en xs.
El programa tendrá la forma:

```
  gpsum.n.[ ] ≐ ???
  gpsum.n.(x►xs) ≐ ???
```

Caso base:

```
  gpsum.n.[ ]
= { especificación }
  ⟨ ∀ i : 0 ≤ i ≤ #[ ] : n + sum.([ ]↑i) ≥ 0 ⟩
= { def. # }
  ⟨ ∀ i : 0 ≤ i ≤ 0 : n + sum.([ ]↑i) ≥ 0 ⟩
= { lógica }
  ⟨ ∀ i : i = 0 : n + sum.([ ]↑i) ≥ 0 ⟩
```

= { rango unitario }
  n + sum.([ ]↑0) ≥ 0
= { def ↑ y de sum }
  n + 0 ≥ 0
= { arit }
  n ≥ 0
¡Listo este caso!

Paso inductivo: H.I.: ∀ E : gpsum.E.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : E + sum.(xs↑i) ≥ 0 ⟩

gpsum.n.(x►xs)
= { especificación }
  ⟨ ∀ i : 0 ≤ i ≤ #(x►xs) : n + sum.((x►xs)↑i) ≥ 0 ⟩
= { def de # }
  ⟨ ∀ i : 0 ≤ i ≤ #xs + 1 : n + sum.((x►xs)↑i) ≥ 0 ⟩
= { lógica }
  ⟨ ∀ i : i = 0 ∨ 1 ≤ i ≤ #xs + 1 : n + sum.((x►xs)↑i) ≥ 0 ⟩
= { partición de rango
    (observación: de nuevo estamos separando el caso del segmento inicial vacío (i = 0) de los segmentos iniciales no vacíos (1 ≤ i ≤ #xs + 1)
  }
  ⟨ ∀ i : i = 0 : n + sum.((x►xs)↑i) ≥ 0 ⟩ ∧
  ⟨ ∀ i : 1 ≤ i ≤ #xs + 1 : n + sum.((x►xs)↑i) ≥ 0 ⟩
= { **Acá la prioridad es llegar a la H.I. así que nos concentramos en la segunda parte.**
    Cambio de variable i → i + 1 }
  ⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 1 ≤ i + 1 ≤ #xs + 1 : n + sum.((x►xs)↑(i+1)) ≥ 0 ⟩
= { arit: resto 1 }
  ⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 0 ≤ i ≤ #xs : n + sum.((x►xs)↑(i+1)) ≥ 0 ⟩
= { def de ↑ }
  ⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 0 ≤ i ≤ #xs : n + sum.(x ► (xs↑i)) ≥ 0 ⟩
= { def de sum }
  ⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 0 ≤ i ≤ #xs : n + (x + sum.((xs↑i))) ≥ 0 ⟩
= { **¿Podemos llegar a la H.I.? Aplicamos asociatividad.** }
  ⟨ ∀ … ⟩ ∧ ⟨ ∀ i : 0 ≤ i ≤ #xs : (n+x) + sum.((xs↑i)) ≥ 0 ⟩
= { Ahora sí H.I. y volvemos a mirar la primera parte. }
  ⟨ ∀ i : i = 0 : n + sum.((x►xs)↑i) ≥ 0 ⟩ ∧ gpsum.(n+x).xs
= { Rango unitario y más pasos como el caso base. }
  n ≥ 0 ∧ gpsum.(n+x).xs

¡¡Listo!! Ya tenemos un programa para gpsum.

Resultado final de **todo el ejercicio**:

psum.xs ≐ gpsum.0.xs

gpsum.n.[ ] ≐ n ≥ 0
gpsum.n.(x►xs) ≐ n ≥ 0 ∧ gpsum.(n+x).xs

¿Cómo funciona este programa? Es raro porque si bien la especificación usa sum y ↑ (tomar), en la definición no se usan para nada.

Verificación / testing: xs = [4, -3, 5, -7, 10].

Recordemos el programa:
psum.xs ≐ gpsum.0.xs
gpsum.n.[ ] ≐ n ≥ 0
gpsum.n.(x►xs) ≐ n ≥ 0 ∧ gpsum.(n+x).xs

psum.[4, -3, 5, -7, 10] ↠ gpsum.0.[4, -3, 5, -7, 10]
↠ 0 ≥ 0 ∧ gpsum.(0+4).[-3, 5, -7, 10]
↠ 0 ≥ 0 ∧ (4 ≥ 0 ∧ gpsum.(4+(-3)).[5, -7, 10])
↠ 0 ≥ 0 ∧ (4 ≥ 0 ∧ (4+(-3) ≥ 0 ∧ gpsum.(4+(-3)+5).[-7, 10]))
↠ … (ejercicio: completar!)
↠ 0 ≥ 0 ∧ 4 ≥ 0 ∧ 4+(-3) ≥ 0 ∧ 4+(-3)+5 ≥ 0 ∧
4+(-3)+5+(-7) ≥ 0 ∧ 4+(-3)+5+(-7)+10 ≥ 0
↠ …
↠ False

Vemos que en el primer parámetro de gpsum se va “guardando” la suma de los elementos que ya pasaron (o mejor dicho de los segmentos iniciales). Este parámetro se suele llamar **acumulador**.

Resumen: ¿Qué hicimos?

En una derivación por inducción, en el paso inductivo nos trabamos y no hubo forma de llegar a la H.I. Pero notamos que si definimos una función más general con un parámetro adicional, la H.I. sí se podría aplicar y la derivación saldría bien. Entonces, **especificamos y derivamos (por inducción) esta nueva función**, y **a la función original la definimos directamente en una línea** como el caso particular de la función más general.

Observación: El intento de definición por inducción que hicimos para la función original se descarta. Sólo sirvió para darnos cuenta de que debo generalizar.
En el ejercicio, hicimos este intento de definición pero fracasó:

psum.[ ] ≐ ???
psum.(x►xs) ≐ ???

Como al final el problema se resolvió con generalización, la definición sale en una línea (*one-liner*):

psum.xs ≐ gpsum.0.xs

APARTADO: OTRA SOLUCIÓN PARA PSUM

Una solución distinta sale de generalizar:

gpsum2.n.xs ≐ ⟨ ∀ i : 0 ≤ i ≤ n : sum.(xs↑i) ≥ 0 ⟩

Se debe derivar gpsum2 por inducción en n (**no en xs**).
El resultado de la derivación es:

psum.xs ≐ gpsum2.(#xs).xs
gpsum2.0.xs ≐ True
gpsum2.(n+1).xs ≐ sum.(xs↑(n+1)) ≥ 0 ∧ gpsum2.n.xs

Esta definición es más parecida a la especificación, pero es más ineficiente que la solución con gpsum, ya que debe calcular explícitamente sum y ↑ muchas veces.

Más ejemplos

Generalización con dos parámetros nuevos

https://docs.google.com/document/d/1cOvl0hd8SyacGPHtvQxON4FpRo3kQW6z1vq2aQb7Q14/edit

h.xs = ⟨ ∃ as, bs : xs = as ++ bs : sum.as = #as + 1 ⟩

Salteamos cosas y vamos derecho al paso inductivo:

h.(x ▶xs)
= { especificación }
⟨ ∃ as, bs : x▶xs = as ++ bs : sum.as = #as + 1 ⟩
= { 3ro excluído }
⟨ ∃ as, bs : x▶xs = as ++ bs ∧ (as = [ ] ∨ as ≠ [ ]) : sum.as = #as + 1 ⟩
= { distrib. y part. rango }
⟨ ∃ as, bs : x▶xs = as ++ bs ∧ as = [ ] : sum.as = #as + 1 ⟩ ∨
⟨ ∃ as, bs : x▶xs = as ++ bs ∧ as ≠ [ ] : sum.as = #as + 1 ⟩
= { cambio var. as → a ▶as}
⟨ ∃ as, bs : x▶xs = as ++ bs ∧ as = [ ] : sum.as = #as + 1 ⟩ ∨
⟨ ∃ a, as, bs : x▶xs = (a ▶as) ++ bs ∧ a▶as ≠ [ ] : sum.(a▶as) = #(a▶as) + 1 ⟩
= { varias propiedades de listas }
⟨ ∃ as, bs : x▶xs = as ++ bs ∧ as = [ ] : sum.as = #as + 1 ⟩ ∨
⟨ ∃ a, as, bs : x = a ∧ xs = as ++ bs : sum.(a▶as) = #(a▶as) + 1 ⟩
= { eliminación de variable a }
⟨ ∃ as, bs : x▶xs = as ++ bs ∧ as = [ ] : sum.as = #as + 1 ⟩ ∨
⟨ ∃ as, bs : xs = as ++ bs : sum.(x▶as) = #(x▶as) + 1 ⟩

= { def sum y def # }
⟨ ∃ as, bs : x►xs = as ++ bs ∧ as = [ ] : sum.as = #as + 1 ⟩ ∨
⟨ ∃ as, bs : xs = as ++ bs : x + sum.as = (#as + 1) + 1 ⟩
= { arit. }
⟨ ∃ as, bs : x►xs = as ++ bs ∧ as = [ ] : sum.as = #as + 1 ⟩ ∨
⟨ ∃ as, bs : xs = as ++ bs : x + sum.as = #as + 1 + 1 ⟩

Acá me trabo. No hay forma de llegar a la H.I. Debo generalizar.

Opción 1: Metemos dos variables

genh.xs.n.m = ⟨ ∃ as, bs : xs = as ++ bs : n + sum.as = #as + 1 + m ⟩

genh generaliza a h ya que: genh.xs.0.0 = h.xs

Opción 2: Si paso restando el 1 para la izquierda y lo junto con el x me queda:

⟨ ∃ as, bs : xs = as ++ bs : **(x - 1)** + sum.as = #as + 1 ⟩

Así que puedo generalizar así:

genh.xs.n = ⟨ ∃ as, bs : xs = as ++ bs : n + sum.as = #as + 1 ⟩

genh generaliza a h ya que: genh.xs.0 = h.xs

Segmentos de lista

Definiciones

Segmento / sublista: Un segmento de una lista es otra lista que está "contenida" en la lista original. Formalmente, una lista ys es *segmento* (también *sublista*) de una lista xs si y sólo si:

existen as, bs tales que xs = as ++ ys ++ bs.

(o sea, puedo obtener xs agarrando ys y concatenandole una lista a la izquierda y otra a la derecha)

¿Es xs segmento de xs? Sí. En este caso as = [ ], bs = [ ].
¿Es [ ] segmento de xs? Sí. En este caso as = xs, bs = [ ] o también as = [ ], bs = xs, o también muchas otras posibilidades…
¿Cuántos segmentos tiene una lista xs?
Propuesta: #xs + 1 ?? No.
Ejemplo: xs = [-3, 2, -7].

Segmentos: [ ] , [-3] , [2] , [-7], [-3, 2] , [2, -7] , [-3, 2 , -7].
En este caso son **7 segmentos únicos.**
Ojo porque hay dos formas de contar: Si xs = [-3, 2, -3], tengo el segmento [-3] dos veces. Contar segmentos únicos no se puede en general porque puede haber segmentos repetidos. Lo que vamos a contar es todas formas posibles de dividir la lista xs en una concatenación as ++ ys ++ bs:

xs = [-3, 2, -7] = as ++ ys ++ bs

as	ys	bs
[ ]	[ ]	[-3, 2, -7]
[ ]	[-3]	[2, -7]
[ ]	[-3 , 2]	[- 7]
[ ]	[-3 , 2 , -7]	[ ]
[-3]	[ ]	[2 , -7]
[-3]	[2]	[-7]
[-3]	[2, -7]	[ ]
[-3, 2]	[ ]	[-7]
[-3, 2]	[-7]	[ ]
[-3 , 2 , 7]	[ ]	[ ]

En total obtuvimos 10 formas distintas de dividir en 3 partes una lista de 3 elementos.

Observación: Si nos fijamos en la columna para ys obtenemos:
[ ] , [-3] , [-3 , 2] , [-3 , 2 , -7] , [ ] , [2] , [2, -7] , [ ] , [-7] , [ ].

¿Cuántas veces aparece la lista [ ] como segmento de xs? Aparece #xs + 1 veces.

Ejercicio: Resolver usando conteo una fórmula general para la cantidad de combinaciones posibles (respuesta: (#xs + 1) * (#xs + 2) / 2).

Segmento inicial / prefijo: Un segmento inicial de una lista es un segmento que además está al principio de la lista. Formalmente, una lista ys es *segmento inicial* (también *prefijo*) de una lista xs si y sólo si:

existe as tal que xs = ys ++ as.

¿Es xs segmento inicial de xs? Sí, con as = [ ]
¿Es [ ] segmento inicial de xs? Sí, con as = xs

¿Cuántos segmentos iniciales tiene una lista xs? #xs + 1.

Segmento final / sufijo: Un segmento final de una lista es un segmento que además está al final de la lista. Formalmente, una lista ys es *segmento final* (también *sufijo*) de una lista xs si y sólo si:

existe as tal que xs = as ++ ys.

¿Es xs segmento final de xs? Sí, con as = [ ]
¿Es [ ] segmento final de xs? Sí, con as = xs
¿Cuántos segmentos finales tiene una lista xs? #xs + 1.

Derivaciones con segmentos iniciales

Habiendo visto estas definiciones ya estamos listos para usar estos conceptos para especificar y derivar problemas con segmentos de lista.

Práctico 3 - Ejercicio 1a:

psum : [Num] → Bool
psum.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.(xs↑i) ≥ 0 ⟩

¿Qué calcula esta función? En palabras:
"Todos los **segmentos iniciales** de la lista suman ≥ 0."

Esta especificación usa la función tomar, pero podemos reescribirla usando la definición que conocemos para segmentos iniciales:

psum : [Num] → Bool
psum.xs = ⟨ ∀ as : "as es segmento inicial de xs" : sum.as ≥ 0 ⟩

Refinando:

psum : [Num] → Bool
psum.xs = ⟨ ∀ as :⟨ ∃ bs : : xs = as ++ bs ⟩ : sum.as ≥ 0 ⟩

O equivalentemente, y mucho más simple:

psum : [Num] → Bool
psum.xs = ⟨ ∀ as, bs : xs = as ++ bs : sum.as ≥ 0 ⟩

¿Estamos seguros de que esta especificación es equivalente a la original?

Verifiquemos con un ejemplo: xs = [4, -3, 5, -7, 10].
Lectura operacional para psum:

1. El rango es: (as, bs) ∈ { ([ ] , [4, -3, 5, -7, 10]),
   ([4] , [-3, 5, -7, 10]) ,
   ([4, -3] , [5, -7, 10]),
   ([4, -3, 5] , [-7, 10]) ,
   ([4, -3, 5, -7] , [10]) ,
   ([4, -3, 5, -7, 10] , [ ]) ,
   }
2. Aplicamos el término estas 6 veces:
   sum.[ ] ≥ 0 ∧
   sum.[4] ≥ 0 ∧
   sum.[4, -3] ≥ 0 ∧
   sum.[4, -3, 5] ≥ 0 ∧
   sum.[4, -3, 5, -7] ≥ 0 ∧
   sum.[4, -3, 5, -7, -10] ≥ 0

El resto queda como ejercicio pero ya se ve que es igual a la especificación original.

Recordemos la especificación:
psum.xs = ⟨ ∀ as, bs : xs = as ++ bs : sum.as ≥ 0 ⟩

Vamos a derivar por inducción en xs. Hacemos primero el paso inductivo para ver si podemos llegar a la H.I.

Paso inductivo: La H.I. es psum.xs = ⟨ ∀ as, bs : xs = as ++ bs : sum.as ≥ 0 ⟩

psum.(x►xs)
= { especificación }
⟨ ∀ as, bs : x ►xs = as ++ bs : sum.as ≥ 0 ⟩
= { **NUEVA ESTRATEGIA: Acá vamos a preguntar por "as". ¿Es vacía o no?**
Sale en varios pasos: 1er paso: lógica }
⟨ ∀ as, bs : x ►xs = as ++ bs ∧ True : sum.as ≥ 0 ⟩
= { lógica (3ro excluido) }
⟨ ∀ as, bs : x ►xs = as ++ bs ∧ (as = [ ] ∨ as ≠ [ ]) : sum.as ≥ 0 ⟩
= { distributividad }
⟨ ∀ as, bs : (x ►xs = as ++ bs ∧ as = [ ]) ∨ (x ►xs = as ++ bs ∧ as ≠ [ ]) : sum.as ≥ 0 ⟩
= { partición de rango
(observación: de nuevo estamos separando el caso del segmento inicial vacío (as = [ ]) de los segmentos iniciales no vacíos (as ≠ [ ]))
}
⟨ ∀ as, bs : x ►xs = as ++ bs ∧ as = [ ] : sum.as ≥ 0 ⟩
∧ ⟨ ∀ as, bs : x ►xs = as ++ bs ∧ as ≠ [ ] : sum.as ≥ 0 ⟩
= { **Acá la prioridad es llegar a la H.I. así que nos concentramos en la segunda parte.**
Como as es no vacía (as ≠ [ ]) tiene la forma c ►cs.
Podemos hacer el siguiente cambio de variable: as ← c ► cs.
Mejor reusamos el nombre "as": cambio de variable as ← a ► as. }
⟨ ∀ … ⟩ ∧ ⟨ ∀ a, as, bs : x ►xs = (a ► as) ++ bs ∧ a ► as ≠ [ ] : sum.(a►as) ≥ 0 ⟩
= { prop. listas }

⟨ ∀ … ⟩ ∧ ⟨ ∀ a, as, bs : x ►xs = (a ► as) ++ bs ∧ True : sum.(a►as) ≥ 0 ⟩
= { lógica }
⟨ ∀ … ⟩ ∧ ⟨ ∀ a, as, bs : x ►xs = (a ► as) ++ bs : sum.(a►as) ≥ 0 ⟩
= { prop. listas (def. ++) }
⟨ ∀ … ⟩ ∧ ⟨ ∀ a, as, bs : x ►xs = a ► (as ++ bs) : sum.(a►as) ≥ 0 ⟩
= { prop. listas }
⟨ ∀ … ⟩ ∧ ⟨ ∀ a, as, bs : x = a ∧ xs = as ++ bs : sum.(a►as) ≥ 0 ⟩
= { eliminación de variable }
⟨ ∀ … ⟩ ∧ ⟨ ∀ as, bs : xs = as ++ bs : sum.(x►as) ≥ 0 ⟩
= { def. de sum }
⟨ ∀ … ⟩ ∧ ⟨ ∀ as, bs : xs = as ++ bs : x + sum.as ≥ 0 ⟩

Acá de nuevo nos pasa que no hay nada que podamos hacer para llegar a la Hipótesis Inductiva. **Debemos generalizar:**

gpsum.n.xs = ⟨ ∀ as, bs : xs = as ++ bs : n + sum.as ≥ 0 ⟩

gpsum generaliza a psum ya que gpsum.0.xs = psum.xs (ejercicio: demostrar! igual sale en un paso). Luego, podemos definir (o sea, este es el programa para psum):

psum.xs ≐ gpsum.0.xs

(y descartamos el intento de derivar por inducción psum)

Falta derivar gpsum. Lo haremos por inducción en xs.

Paso inductivo: H.I.: para todo E : gpsum.E.xs = ⟨ ∀ as, bs : xs = as ++ bs : E + sum.as ≥ 0 ⟩

gpsum.n.(x►xs)
= { especificación }
⟨ ∀ as, bs : x ►xs = as ++ bs : n + sum.as ≥ 0 ⟩
= { **mismos pasos que hicimos antes solo que con el "n +" agregado** }
⟨ ∀ as, bs : x ►xs = as ++ bs ∧ as = [ ] : n + sum.as ≥ 0 ⟩
∧ ⟨ ∀ as, bs : xs = as ++ bs : n + (x + sum.as) ≥ 0 ⟩
= { asociatividad }
⟨ ∀ as, bs : x ►xs = as ++ bs ∧ as = [ ] : n + sum.as ≥ 0 ⟩
∧ ⟨ ∀ as, bs : xs = as ++ bs : (n + x) + sum.as ≥ 0 ⟩
= { H.I. con E = n+x }
⟨ ∀ as, bs : x ►xs = as ++ bs ∧ as = [ ] : n + sum.as ≥ 0 ⟩
∧ gpsum.(n + x). xs
= { eliminación de variable as }
⟨ ∀ bs : x ►xs = [ ] ++ bs : n + sum.[ ] ≥ 0 ⟩
∧ gpsum.(n + x). xs
= { def. ++ }
⟨ ∀ bs : x ►xs = bs : n + sum.[ ] ≥ 0 ⟩
∧ gpsum.(n + x). xs
= { **rango unitario!!** (siempre es más seguro aplicar rango unitario que término constante) }

$$
\begin{array}{ll}
 & n + \underline{sum.[\,]} \geq 0 \quad \wedge \quad gpsum.(n + x).\, xs \\
= & \{ \text{ def. sum } \} \\
 & n + 0 \geq 0 \quad \wedge \quad gpsum.(n + x).\, xs \\
= & \{ \text{ arit. } \} \\
 & n \geq 0 \quad \wedge \quad gpsum.(n + x).\, xs
\end{array}
$$

Ahora sí terminamos el paso inductivo.

Caso base:

$$
\begin{array}{ll}
 & gpsum.n.[\,] \\
= & \{ \text{ especificación } \} \\
 & \langle\, \forall\ as, bs : \underline{[\,] = as \mathbin{++} bs} :\ n + sum.as \geq 0 \,\rangle \\
= & \{ \text{ prop. listas } \} \\
 & \langle\, \forall\ as, bs :\ [\,] = as \ \wedge\ [\,] = bs :\ n + sum.as \geq 0 \,\rangle \\
= & \{ \text{ eliminación de variable as } \} \\
 & \langle\, \forall\ bs :\ [\,] = bs :\ n + sum.[\,] \geq 0 \,\rangle \\
= & \{ \text{ rango unitario } \} \\
 & n + sum.[\,] \geq 0 \\
= & \{ \ldots \} \\
 & n \geq 0
\end{array}
$$

Listo!! Resultado final:

$$
\begin{array}{l}
psum.xs \doteq gpsum.0.xs \\
gpsum.n.[\,] \doteq n \geq 0 \\
gpsum.n.(x \blacktriangleright xs) \doteq n \geq 0 \quad \wedge \quad gpsum.(n + x).\, xs
\end{array}
$$

Obtuvimos exactamente el mismo programa que habíamos obtenido para la otra especificación.

Ejercicio: Repasar la verificación (el testing).

Derivaciones con segmentos finales

La salteamos pero la estrategia es igual a la de segmentos iniciales.

Derivaciones con segmentos arbitrarios

Veremos que salen siempre con una modularización, donde la función modularizada resuelve el mismo problema pero para segmentos iniciales.

Ejemplo: Segmento de suma máxima: “Dada una lista xs, considerar todos los segmentos posibles y obtener la suma de aquel que tiene suma máxima”.

Especificación:

sumax.xs = ⟨ Max bs : “bs es segmento de xs” : sum.bs ⟩

Refinando:

sumax.xs = ⟨ Max bs : ⟨ ∃ as, cs : : xs = as ++ bs ++ cs ⟩ : sum.bs ⟩

O más simple:

sumax.xs = ⟨ Max as, bs, cs : xs = as ++ bs ++ cs : sum.bs ⟩

Me quedo con esta especificación.

Ejemplo: xs = **[-3, 2, -7]**.

1. Lectura operacional: (as, bs, cs) ∈ { ([ ], [ ], [-3, 2, 7]) ….. } (y así como en la tabla que hicimos antes, son 10 tuplas).
2. Aplicamos el término a los bs: [ ] , [-3] , [-3 , 2] , [-3 , 2 , -7] , [ ] , [2] , [2, -7] , [ ] , [-7] , [ ].
   sum.[ ] max
   sum.[-3] max
   … max
   sum.[ ]
3. Al final me queda: 2. (ejercicio: verificar)

Derivación de sumax:

Recordemos la especificación:

sumax.xs = ⟨ Max as, bs, cs : xs = as ++ bs ++ cs : sum.bs ⟩

Hacemos inducción en xs.

Paso inductivo: H.I.: sumax.xs = ⟨ Max as, bs, cs : xs = as ++ bs ++ cs : sum.bs ⟩

sumax.(x►xs)
= { especificación }
⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs : sum.bs ⟩
= { aplicamos la estrategia: preguntamos por as (3ro excluido) }
⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ (as = [ ] ∨ as ≠ [ ]): sum.bs ⟩
= { distributividad y partición de rango }
⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as = [ ]: sum.bs ⟩max
⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as ≠ [ ]: sum.bs ⟩
= { seguimos con la estrategia: cambio de variable: as ← a ►as }
⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as = [ ]: sum.bs ⟩max
⟨ Max a, as, bs, cs : x►xs = (a►as) ++ bs ++ cs ∧ (a►as) ≠ [ ]: sum.bs ⟩

= { prop listas }
  ⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as = [ ]: sum.bs ⟩max
  ⟨ Max a, as, bs, cs : x►xs = (a►as) ++ bs ++ cs : sum.bs ⟩
= { más prop listas }
  ⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as = [ ]: sum.bs ⟩max
  ⟨ Max a, as, bs, cs : x►xs = a►(as ++ bs ++ cs) : sum.bs ⟩
= { más prop listas }
  ⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as = [ ]: sum.bs ⟩max
  ⟨ Max a, as, bs, cs : x = a ∧ xs = as ++ bs ++ cs : sum.bs ⟩
= { eliminación de variable a }
  ⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as = [ ]: sum.bs ⟩max
  ⟨ Max as, bs, cs : xs = as ++ bs ++ cs : sum.bs ⟩
= { Hipótesis Inductiva! no hace falta generalizar }
  ⟨ Max as, bs, cs : x►xs = as ++ bs ++ cs ∧ as = [ ]: sum.bs ⟩max
    sumax.xs
= { eliminación de variable as }
  ⟨ Max bs, cs : x►xs = [ ] ++ bs ++ cs : sum.bs ⟩max sumax.xs
= { prop. listas }
  ⟨ Max bs, cs : x►xs = bs ++ cs : sum.bs ⟩max sumax.xs
= { **no es rango unitario ¿qué es?**
      posible estrategia: continuar con bs = [ ] ó bs ≠ [ ] (ejercicio: probar!).
      igual no tiene mucho sentido porque no tengo H.I. a la que quiero llegar.
    La expresión cuantificada que queda molestando es exactamente el mismo problema original pero esta vez sólo sobre segmentos iniciales.
    Es un problema accesorio al problema original (o un subproblema).
    ¡¡Debemos modularizar!! Primero especificamos una nueva función sumaxp:
      sumaxp.xs = ⟨ Max bs, cs : xs = bs ++ cs : sum.bs ⟩
    Ahora llamemos a esta función en lo que estamos derivando.
  }
    sumaxp.(x►xs) max sumax.xs

Listo el paso inductivo de sumax!! **Falta el caso base (ejercicio)**. El resultado parcial es:

sumax.[ ] ≐ ???
sumax.(x►xs) ≐ sumaxp.(x►xs) max sumax.xs

Falta derivar sumaxp. Ejercicio.
(sale parecido a psum pero en este caso no hace falta generalizar gracias a que las suma distribuye con Max).

Resultado final:

sumax.[ ] ≐ 0
sumax.(x►xs) ≐ sumaxp.(x►xs) max sumax.xs

sumaxp.[ ] ≐ 0
sumaxp.(x►xs) ≐ 0 max (x + sumaxp.xs)

Testing: TODO

Otro ejemplo: TODO: ssum (como psum pero para segmentos en general)

Rangos con pares de elementos de lista

Ver también:
https://wiki.cs.famaf.unc.edu.ar/lib/exe/fetch.php?media=algo1:2017-2:consejos_funcional.pdf

Problema: Dada una lista, considerar todas las formas posibles de tomar dos elementos, y contar cuántas veces éstos son iguales:

f : [Num] -> Nat
f.xs =⟨N i, j: 0 ≤ i < j < #xs: xs!i = xs!j ⟩

Ejemplo: xs = [2, 1, 3, 2, 3, 2].

(i, j) ∈ { (0, 1), (0, 2), **(0, 3)**, (0, 4), **(0, 5)**,
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 3), **(2, 4)**, (2, 5),
(3, 4), **(3, 5)**,
(4, 5) }

Resultado: 4 (coinciden en las posiciones: (0, 3), (0, 5), (2, 4), (3, 5)).

Más gráficamente:

i \ j	0	1	2	3	4	5
0		x	x	**x**	x	**x**
1			x	x	x	x
2				x	**x**	x
3					x	**x**
4						x

Derivación: Por inducción en xs.

f.[ ] ≐ ???
f.(x►xs) ≐ ???

Paso inductivo:
f.(x►xs)
= { esp. f }

⟨N i, j : 0 ≤ i < j < #(x►xs) : (x►xs)!i = (x►xs)!j ⟩
= { def # }
⟨N i, j : 0 ≤ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩
= { lógica } (MAL)
⟨N i, j : **i = 0 ∨ 1 ≤ i < j < #xs + 1** : (x►xs)!i = (x►xs)!j ⟩ (MAL)
(este paso está mal, la parte "i = 0" no dice nada sobre j, debemos proceder con mayor cuidado)
= { reescribo desigualdad para separar 0 ≤ i }
⟨N i, j : 0 ≤ i ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩
= { ahora sí lógica sobre 0 ≤ i }
⟨N i, j : (i = 0 ∨ 1 ≤ i) ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩
= { distributiva }
⟨N i, j : (i = 0 ∧ i < j < #xs + 1) ∨ (1 ≤ i ∧ i < j < #xs + 1) : (x►xs)!i = (x►xs)!j ⟩
= { partición de rango }
⟨N i, j : i = 0 ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩+
⟨N i, j : 1 ≤ i ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩
= { lógica: vuelvo a juntar desigualdades }
⟨N i, j : i = 0 ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩+
⟨N i, j : 1 ≤ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩
= { dos cambios de variable i ← i + 1 , j ← j + 1 }
⟨N i, j : i = 0 ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩+
⟨N i, j : 1 ≤ i + 1 < j + 1 < #xs + 1 : (x►xs)!(i+1) = (x►xs)!(j+1) ⟩
= { aritmética en el rango, def. ! en el término }
⟨N i, j : i = 0 ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩+
⟨N i, j : 0 ≤ i < j < #xs : xs!i = xs!j ⟩
= { H.I. }
⟨N i, j : i = 0 ∧ i < j < #xs + 1 : (x►xs)!i = (x►xs)!j ⟩+
f.xs
= { eliminación de variable i }
⟨N j : 0 < j < #xs + 1 : (x►xs)!0 = (x►xs)!j ⟩+ f.xs
= { (acá ya podríamos modularizar, pero mejor hacer unos pasos para simplificar primero)
lógica: 0 < j es lo mismo que 1 ≤ j }
⟨N j : 1 ≤ j < #xs + 1 : (x►xs)!0 = (x►xs)!j ⟩+ f.xs
= { cambio de variable j ← j + 1 }
⟨N j : 1 ≤ j + 1 < #xs + 1 : (x►xs)!0 = (x►xs)!(j+1) ⟩+ f.xs
= { aritmética en el rango, def. ! en el término }
⟨N j: 0 ≤ j < #xs : x = xs!j ⟩+ f.xs
= { modularización con nueva función:
m : Num -> [Num] -> Nat
m.e.xs =⟨N j: 0 ≤ j < #xs: e = xs!j ⟩
¿Qué hace esta función? Dice cuántas veces aparece "e" en la lista "xs".
}
m.x.xs + f.xs

Resultado final:

f.[ ] ≐ 0
f.(x►xs) ≐ m.x.xs + f.xs

m.e.[ ] ≐ 0
m.e.(x►xs) ≐ (x = e → 1 + m.e.xs
[] x ≠ e → m.e.xs
)

Testing: TODO

Rangos con análisis por casos

TODO

Resumen de técnicas de derivación

Supongamos que tenemos una especificación :

$$f.x = E$$

Queremos obtener un programa, esto es, una definición para la función f.

1. Si toda la expresión E es programable, la definición es directamente E:

$$f.x \doteq E$$

2. Si E tiene algunas partes programables y otras no, las partes no programables se **modularizan** con funciones nuevas. En E se reemplazan todas las partes no programables por llamadas a las funciones, para obtener E' que es completamente programable:

$$f.x \doteq E'$$

Después hay que obtener definiciones para todas las funciones modularizadas.

3. Si E es una expresión cuantificada, hará falta una recursión para calcularla, y por lo tanto hay que hacer **inducción** sobre alguno de los parámetros.
    3.1. Elegir parámetro
    3.2. Elegir esquema inductivo
    3.3. Derivar: priorizar el caso inductivo, y dentro de éste, lograr aplicar la hipótesis inductiva. Si no se puede, es posible que haya que **generalizar** (ver punto siguiente).Si se puede, es posible que hayan otras partes extra que sean no programables, en cuyo caso hay que **modularizar**.

4. Si se intentó derivar por inducción y no se pudo, hay que **generalizar**. Se especifica una nueva función gf de manera parecida a f pero con un parámetro adicional "y":

$$gf.x.y = E'$$

donde E' es parecida a E pero aparece el nuevo parámetro "y".Además, gf generaliza a f o, lo que es lo mismo decir, f es caso particular de gf. El caso particular se da cuando "y" tiene un valor específico constante "e". Luego, se puede definir f directamente en una sola línea:

$$f.x \doteq gf.x.e$$

Después hay que obtener una definición recursiva para gf, cosa que debería ser posible si se hizo bien la generalización.

Temas complementarios

Técnicas de autocorrección

Señales de que hay algo mal en una especificación:

- Hay errores de tipo:
    - En una expresión cuantificada, el tipo del término no se corresponde con el tipo del operador.
      Ejemplo: ⟨∑ i : 0 ≤ i < #xs : xs!i mod 2 = 0 ⟩
      (MAL: el término debe ser un número)

Señales de que hay algo mal en un programa:

- Variables cuantificadas que aparecen en el programa resultado.
  Ejemplo:
  gpsum.n.(x►xs) ≐ n ≥ 0 ∧ gpsum.(n * a).xs (MAL: “a” no existe!)
- Parámetros que faltan: variables que aparecen en el lado derecho de una definición pero que no están entre los parámetros.
  Ejemplo:
  m.(x►xs) ≐ e = x ∧ m.xs (MAL: falta parámetro e)
- Parámetros que colisionan: dos parámetros o más comparten el mismo nombre.
  Ejemplo:
  m.x.(x►xs) ≐ …. (MAL: está dos veces x como parámetro)

Testing!!

Traducción a Haskell

Recursos:

- https://wiki.cs.famaf.unc.edu.ar/lib/exe/fetch.php?media=introalg:guia-ghc.pdf
- Introduction to GHCi

A partir de Haskell 2010, Haskell no soporta más *pattern matchings* de la forma “n+k” (“n+1”, “n+2”, etc.).[2]

fac.0 ≐ 1
fac.(n+1) ≐ (n+1) * fac.n

se debe traducir a:

```
fac 0 = 1
fac n = n * fac (n-1)
```

–

fib.0 ≐ 0
fib.1 ≐ 1
fib.(n+2) ≐ fib.n + fib.(n+1)

se traduce a:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

Tabla de correspondencia de funciones y operadores:

En el teórico/práctico	En Haskell	¿Qué es?
≐	=	definición de función
=	==	comparación de igualdad
▶	:	constructor de listas
!	!!	indexación
xs ↑ n	take n xs	tomar
xs ↓ n	drop n xs	tirar

[2] Se puede activar este pattern matching usando el pragma `NPlusKPatterns`. Ver:
https://en.wikipedia.org/wiki/Haskell#Haskell_2010
https://stackoverflow.com/questions/3748592/what-are-nk-patterns-and-why-are-they-banned-from-haskell-2010

Problemas Exponenciales

Ejemplo: Problemas con sublistas de listas (salteando elementos).

Dada una lista xs y un número n: ¿Existe una sublista cuya suma da n ?

Especificación: Usaremos una lista de ceros y unos para especificar todas las formas posibles de seleccionar elementos de la lista:

sublistaN.xs.n = ⟨ ∃ as : binaria.as ∧ #as = #xs : ⟨∑ i : 0 ≤ i < #xs : xs!i * as!i ⟩= n ⟩

Donde: binaria.xs =⟨ ∀ i : 0 ≤ i < #xs : xs!i = 0 ∨ xs!i = 1 ⟩
O mejor:
binaria.[ ] = True
binaria.(x►xs) = (x = 0 ∨ x = 1) ∧ binaria.xs

Ejemplo: xs = [-2, 1, 10], n = 8

as ∈ { [0, 0, 0] , // suma 0
[0, 0, 1] , // suma 10
[0, 1, 0] , // suma 1
[0, 1, 1] , // suma 11
[1, 0, 0] , // suma -2
[1, 0, 1] , // ESTE DA SUMA 8!
[1, 1, 0] , // suma -1
[1, 1, 1] } // suma 9

Resultado: True

Derivación: Por inducción en xs.

Caso base: sublistaN.[ ].n = (n = 0)

Paso inductivo:

sublistaN.(x ► xs).n
= { esp. }
⟨ ∃ as : binaria.as ∧ #as = #(x►xs) : ⟨∑ i : 0 ≤ i < #(x►xs) : (x►xs)!i * as!i ⟩= n ⟩
= { cambio var as → a ►as }
⟨ ∃ a, as : binaria.(a ►as) ∧ #(a ►as) = #(x►xs) :
⟨∑ i : 0 ≤ i < #(x►xs) : (x►xs)!i * (a ►as)!i ⟩= n ⟩
= { def. # y aritmética }
⟨ ∃ a, as : binaria.(a ►as) ∧ #as = #xs :
⟨∑ i : 0 ≤ i < #(x►xs) : (x►xs)!i * (a ►as)!i ⟩= n ⟩
= { def. binaria }
⟨ ∃ a, as : (a = 0 ∨ a = 1) ∧ binaria.as ∧ #as = #xs : … ⟩
= { distrib. y partición de rango }

⟨ ∃ a, as : a = 0 ∧ … : … ⟩
  ∨⟨ ∃ a, as : a = 1 ∧ … : … ⟩
= { eliminación de a en ambos cuantificadores }
⟨ ∃ as : binaria.as ∧ #as = #xs : ⟨∑ i : 0 ≤ i < #(x►xs) : (x►xs)!i * (0 ►as)!i ⟩= n ⟩
  ∨⟨ ∃ as : binaria.as ∧ #as = #xs :⟨∑ i : 0 ≤ i < #(x►xs) : (x►xs)!i * (1 ►as)!i ⟩= n ⟩
= { def #, partición de rango y rango unitario }
⟨ ∃ as : binaria.as ∧ #as = #xs :
                (x►xs)!0 * (0 ►as)!0 +⟨∑ i : 1 ≤ i < #xs+1 : (x►xs)!i * (0 ►as)!i ⟩= n ⟩
  ∨ …
= { def ! y neutro * }
⟨ ∃ as : binaria.as ∧ #as = #xs : ⟨∑ i : 1 ≤ i < #xs+1 : (x►xs)!i * (0 ►as)!i ⟩= n ⟩
  ∨ …
= { cambio var. i → i + 1 }
⟨ ∃ as : binaria.as ∧ #as=#xs:⟨∑ i : 1 ≤ i+1 < #xs+1 : (x►xs)!(i+1) * (0 ►as)!(i+1) ⟩= n ⟩
  ∨ …
= { aritmética y def. ! }
⟨ ∃ as : binaria.as ∧ #as=#xs:⟨∑ i : 0 ≤ < #xs : xs!i * as!i ⟩= n ⟩
  ∨ …
= { H.I. y retomamos la otra parte }
  sublistaN.xs.n
  ∨⟨ ∃ as : binaria.as ∧ #as = #xs :⟨∑ i : 0 ≤ i < #(x►xs) : (x►xs)!i * (1 ►as)!i ⟩= n ⟩
= { mismos pasos: partición, rango unitario, cambio de variable, etc. }
  sublistaN.xs.n
  ∨⟨ ∃ as : binaria.as ∧ #as = #xs : x * 1 +⟨∑ i : 0 ≤ i < #xs : xs!i * as!i ⟩= n ⟩
= { aritmética: paso restando x }
  sublistaN.xs.n ∨⟨ ∃ as : binaria.as ∧ #as = #xs :⟨∑ i : 0 ≤ i < #xs : xs!i * as!i ⟩= n - x ⟩
= { H.I. de nuevo! }
  sublistaN.xs.n ∨ sublistaN.xs.(n - x)

Resultado final:
  sublistaN.[ ].n ≐ n = 0
  sublistaN.(x ► xs).n ≐ sublistaN.xs.n ∨ sublistaN.xs.(n - x)

Testing: xs = [-2, 1, 10], n = 8

sublistaN.[-2, 1, 10].8
  → sublistaN.[1, 10].8 ∨ sublistaN.[1, 10].10
  → (sublistaN.[10].8 ∨ sublistaN.[10].7) ∨ sublistaN.[1, 10].10
  → (sublistaN.[10].8 ∨ sublistaN.[10].7) ∨ (sublistaN.[10].10 ∨ sublistaN.[10].9)
  → (sublistaN.[ ].8 ∨ sublistaN.[ ].(-2)) ∨ sublistaN.[10].7
                                ∨ sublistaN.[10].10 ∨ sublistaN.[10].9
  → …
  → sublistaN.[ ].8 ∨
    sublistaN.[ ].(-2) ∨
    sublistaN.[ ].7 ∨
    sublistaN.[ ].(-3) ∨
    sublistaN.[ ].10 ∨
    sublistaN.[ ].0 ∨

```
    sublistaN.[ ].9     ∨
    sublistaN.[ ].(-1) ∨
  → 8 = 0    ∨   // [0, 0, 0]  <--  EN “n = 0” , EL n significa “lo que le falta a la suma para dar 8”
    -2 = 0   ∨   // [0, 0, 1]  <--  acá es -2 porque sumó 10, se pasó en dos
    7 = 0    ∨   // [0, 1, 0]  <--  acá es 7 porque sumó 1, le faltan 7
    -3 = 0   ∨   // [0, 1, 1]  <--  y así…
    10 = 0   ∨   // [1, 0, 0]
    0 = 0    ∨   // [1, 0, 1]  <--  acá sumó 8, le faltan 0, da True!!
    9 = 0    ∨   // [1, 1, 0]
    -1 = 0 ∨     // [1, 1, 1]
  → True
```

Técnica de Tuplas

Cómo pasar Fibonacci de exponencial (lenta) a lineal (rápida):

fib.0 ≐ 0
fib.1 ≐ 1
fib.(n+2) ≐ fib.n + fib.(n+1)

Problema: este programa es lentísimo, para calcular fib.n se requiere aprox. 2^n pasos de reducción (exponencial).

Ejemplo: fib.6 → fib.4 + fib.5
→ fib.4 + (fib.3 + fib.4)

(TODO: hacer dibujo del árbol de llamadas)

Observación: Hay muchas llamadas redundantes.

Especificación:

tfib : Nat -> (Nat, Nat)
tfib.n = (fib.n, fib.(n+1))

Fibonacci se puede definir en términos de tfib así:
fib.n ≐ fst.(tfib.n)

Derivación: Por inducción en n.

Caso base: tfib.0 ≐ (0, 1) (ejercicio)

Paso inductivo: La H.I. es “tfib.n = (fib.n, fib.(n+1))”.
tfib.(n+1)
= { esp. tfib }

```
  (fib.(n+1), fib.(n+2))
= { def. fib }
  (fib.(n+1), fib.n + fib.(n+1))
= { definición local }
  (b, a + b)
          |[ a = fib.n
             b = fib.(n+1) ]|
= { armo tupla }
  (b, a + b)
          |[ (a, b) = (fib.n, fib.(n+1)) ]|
= { H.I. }
  (b, a + b)
          |[ (a, b) = tfib.n]|
```

Resultado final:

```
fib.n ≐ fst.(tfib.n)

tfib.0       ≐ (0, 1)
tfib.(n+1) ≐ (b, a + b)
                    |[ (a, b) = tfib.n]|
```

Recursión final

Ver capítulos 15 y 20 del libro Cálculo de Programas.

Ejercicios

practico2.pdf

Programación Imperativa

Introducción

Antes: Modelo computacional de la programación funcional

En programación funcional un programa es una expresión, y la ejecución de un programa es la reducción de esa expresión a una forma normal o canónica (a un valor).

Ejemplo: Definimos la función sum:

```
sum.[ ] ≐ 0
```

sum.(x►xs) ≐ x + sum.xs

Un programa posible es la expresión:
sum.([1,2] ++ [3,4])

Una ejecución de este programa:
sum.([1,2] ++ [3,4]) ↠ sum.(1► [2] ++ [3,4]))
↠ sum.(1► [2] ++ [3,4]))
↠ 1 + sum.([2] ++ [3,4]))
….
↠ 10

Este modelo computacional es una abstracción que dista mucho del verdadero funcionamiento de una computadora (y su modelo computacional subyacente).

Ahora: Modelo computacional de la programación imperativa

En programación imperativa un programa es una “receta” o una “serie de pasos” y la ejecución de un programa parte de un **estado** inicial y lo va transformando hasta llegar a un **estado** final.

DEFINICIÓN: Estado: es una asignación de valores a un conjunto fijo y predeterminado de variables. En un programa, las variables que componen el estado (con su nombre y tipo) se definen al principio en lo que se llama la “declaración de constantes y variables”.

Programa de ejemplo:

```
      Var x, y : Int       ← declaración de variables

ℓ1    x := 3 ;              ← sentencias del programa (la “receta”)
ℓ2    y := x + 3
```

Secuenciación: Usamos “;” para secuenciar una sentencia atrás de otra.

Asignación: Un tipo de sentencia que modifica determinados valores del estado.

Variables que componen el estado: “x” e “y” de tipo Int.

Ejemplo de estado inicial: σ_0: x ↦ 4, y ↦ 8

(observación: **siempre** ponemos valores en los estados, nunca expresiones)
(el estado me indica qué contiene la memoria de la computadora en un momento determinado)

Ejemplo de ejecución del programa a partir de este estado inicial:

línea	estado	observaciones
	σ_0: x ↦ 4, y ↦ 8	estado inicial
ℓ_1 x := 3	σ_1: x ↦ 3, y ↦ 8	
ℓ_2 y := x + 3	σ_2: x ↦ 3, y ↦ 6	estado final

Siempre el estado final de la ejecución de un programa depende tanto del programa como del estado inicial.

Un programa imperativo es un **“transformador de estados”**.

Sintaxis vs. Semántica

Siempre que hablemos de lenguajes de programación tendremos estos dos aspectos:

Sintaxis: ¿Cómo se escriben los programas? Un programa es un texto (una secuencia de letras). La sintaxis de un lenguaje me dice qué textos son programas válidos.

Semántica: ¿Qué significan los programas? ¿Qué hacen? Un programa se puede ejecutar, y esa ejecución tiene un efecto sobre un “mundo semántico”. En el caso de la programación imperativa, el mundo semántico es el estado. En el caso de la programación funcional, el mundo semántico es el de las expresiones.

El lenguaje de programación Imperativa

Asignación

Otro ejemplo:

```
y := (x + 3) * y   // en una asignación puedo usar expresiones
```

Otro ejemplo:

```
n := n + 1        // en una asignación puedo usar
                  // la misma variable que estoy asignando
```

Ejemplo: con estado inicial n → 7 , el estado final es n → 8.

Otro ejemplo (asignación “múltiple”):

```
x , y :=  3 , x + 3     // asigno dos variables al mismo tiempo
                        // (x a “3”, y a “x + 3”)
```

Ejemplo: con estado inicial x ↦ **4**, y ↦ 8, el estado final es x ↦ **3**, y ↦ 7. Es 7 y no 6 ya que en la semántica de la asignación siempre usamos para todas las expresiones los valores en el estado anterior. La asignación es atómica e indivisible (no hay “estados intermedios”).

Observación: C no tiene asignación múltiple (hay otros lenguajes que sí, como Python).

Verificación en la consola de Python:

```
>>> x = 4                  ← acá definimos el estado inicial
>>> y = 8                  ← acá definimos el estado inicial
>>>
>>> x, y = 3, x + 3    ← este es el programa (asignación múltiple)
>>>
>>> x                        ← muestro el resultado
3
>>> y
7
```

Condicional (if) y skip

Un if es una sentencia que me permite ejecutar distintas sub-sentencias dependiendo del cumplimiento de una o más condiciones.

Ejemplo:

```
Var x : Int

if   x ≥ 0 →              ← las condiciones se llaman “guardas”
  skip                   ← skip es una sentencia que no hace nada (no modifica el estado)
□   x < 0 →
  x := -x
fi
```

¿Qué hace este programa? Este programa calcula ~~el módulo de x~~ un estado final en el x vale el “valor absoluto” o “módulo” del valor de x en el estado inicial.
Observación importante: Nunca el resultado de un programa es un valor (como un número), si no un estado final.

Ejemplo: con estado inicial x → 33, el estado final es x → 33.
Ejemplo: con estado inicial x → -11, el estado final es x → 11.

Observación: El “if” es una sentencia compuesta ya que tiene sentencias adentro.

Repetición / ciclo (do)

```
ℓ₁ do x ≠ 0 →              // el “do” tiene una guarda
ℓ₂     x := x - 1          // y tiene una sentencia adentro llamadada “cuerpo”
```

ℓ_3 **od**

Semántica: “Mientras valga que x ≠ 0, ejecutar x := x - 1.”

Ejemplo: con estado inicial x ↦ 2

línea	estado/guardas	observaciones
	σ_0: x ↦ 2	estado inicial
ℓ_1	σ_0: x ↦ 2, x ≠ 0 ≡ True	evalúo guarda
ℓ_2 x := x - 1	σ_1: x ↦ 1	cuerpo del ciclo
ℓ_1	σ_1: x ↦ 1, x ≠ 0 ≡ True	evalúo guarda
ℓ_2 x := x - 1	σ_2: x ↦ 0	cuerpo del ciclo
ℓ_1	σ_2: x ↦ 0, x ≠ 0 ≡ False	evalúo guarda
ℓ_3	σ_2: x ↦ 0	estado final

Semántica más detallada:

1. a partir del estado inicial, chequeo si vale o no vale la guarda.
2. si la guarda vale,
    a. ejecuto el cuerpo del ciclo.
    b. a partir del estado que resulta de ejecutar el cuerpo del ciclo, chequeo nuevamente si vale o no la guarda, y **vuelvo al paso 2**.
3. si la guarda no vale, el ciclo termina y el estado queda como estaba.

Observaciones:

- puede pasar que la guarda no valga al principio de todo y el cuerpo del ciclo no se ejecute nunca (el estado no cambia).
- puede pasar que la guarda valga para siempre y entre en un ciclo infinito. un programa que hace esto es un programa que **“no termina”**.

Cosas que NO tiene el lenguaje de programación

Los lenguajes imperativos reales tienen muchos otros tipos de sentencias. Uno de los ejemplos más antiguos es la **sentencia “GO TO”** (o “GOTO”). Otros ejemplos más vigentes son las **sentencias break y continue** presentes en muchos lenguajes incluyendo C y Python.

Este tipo de sentencias no nos interesan y no serán parte de nuestro lenguaje de programación teórico por dos grandes razones:

1. No son necesarias, ya que es bien sabido que:

   Todo problema que puede ser resuelto por programación, puede ser resuelto usando solamente asignaciones (:=), condicionales (if) y ciclos (do)
   (Teorema de Böhm–Jacopini)

2. Agregan complejidad lógica y dificultad para entender y razonar sobre los programas. Sobran publicaciones respaldando esto, siendo la siguiente carta de Dijkstra una de las más famosas:

   Dijkstra, Edsger W. (March 1968). *"Letters to the editor: Go to statement considered harmful" (PDF)*. Communications of the ACM. 11 (3): 147–148. doi:10.1145/362929.362947. S2CID 17469809.
   https://www.cs.utexas.edu/~EWD/ewd02xx/EWD215.PDF

Ejemplo: contar múltiplos de 6

Vamos a hacer un programa que usa todo lo que acabamos de ver.

Dado un número entero N > 0, queremos contar cuántos **números entre 0 y N** son múltiplos de 6.

Idea del algoritmo: Con un ciclo, recorrer los números desde 0 hasta N (o al revés, desde N hasta 0). Para cada valor, fijarme **si** es múltiplo de 6 o no. Si sí lo es, sumamos 1 al resultado.

Programa final:

```
Const N : Int ;
Var numero_actual, resultado : Int ;

numero_actual , resultado := 0 , 0 ;  // esta asignación se llama “inicialización”
do numero_actual ≤ N →
       if  numero_actual mod  6 = 0  →
              resultado := resultado + 1 ;        // FUNDAMENTAL: ASIGNACIÓN ES :=
                                                  // USAR “;” PARA SECUENCIAR
              numero_actual := numero_actual + 1
       □ numero_actual mod  6 ≠ 0 →
              numero_actual := numero_actual + 1
       fi
od
```

Importante: Indentación!! Organizar y tabular el código para que pueda leerse correctamente.

¿Es correcto este programa? ¿Hace lo que esperamos?
Si N = 12, **debería dar** un estado final con resultado=3 (el 0, el 6 y el 12).
Sí es correcto.

Ejercicio: verificar haciendo la ejecución con un ejemplo concreto de estado inicial (valores para **N**, numero_actual y resultado).

Otro programa posible:

```
Const N : Int ;
Var numero_actual, resultado : Int ;

numero_actual , resultado := 0 , 0 ;  // esta asignación se llama “inicialización”
do numero_actual ≤ N →
       if  numero_actual mod  6 = 0  →
              resultado := resultado + 1
       □ numero_actual mod  6 ≠ 0 →
              skip
       fi    ;     // <- secuenciación entre el if y la asignación que sigue
       numero_actual := numero_actual + 1
od
```

Otro programa posible (recorriendo hacia atrás):

```
Const N : Int ;
Var numero_actual, resultado : Int ;

numero_actual , resultado := N , 0 ;
do numero_actual ≥ 0 →
       if  numero_actual mod  6 = 0  →
              resultado := resultado + 1
       □ numero_actual mod  6 ≠ 0 →
              skip
       fi    ;     // <- secuenciación entre el if y la asignación que sigue
       numero_actual := numero_actual - 1
od
```

Otro programa posible (usando conocimientos matemáticos básicos):

```
Const N : Int ;
Var numero_actual, resultado : Int ;

numero_actual , resultado := 0 , 0 ;
do numero_actual ≤ N →
       resultado := resultado + 1  ;
       numero_actual := numero_actual  + 6
od
```

Otro programa (usando conocimientos matemáticos básicos):

Sale con una sola asignación:

```
Const N : Int ;
```

```
Var resultado : Int ;

resultado := N div 6 + 1
```

Otro ejemplo: factorial

Dado un número N > 0 queremos calcular su factorial:

resultado = 1 * 2 * 3 * … * (N-1) * N

Es evidente que no es posible hacer todas las multiplicaciones en una sola asignación ya que no existe una expresión válida en el lenguaje para hacer eso.

Por lo tanto es necesario usar un ciclo **do para ir incorporando de a uno cada número que se desea multiplicar**. El resultado comenzará valiendo 1 y luego en cada iteración del ciclo será modificado con una nueva multiplicación:

- 1er paso: resultado = 1
- 2do paso: resultado = 1 * 2
- 3er paso: resultado = 1 * 2 * 3
  …
- N-1-ésimo paso: resultado = 1 * 2 * 3 * … * (N-1)
- N-ésimo paso: resultado = 1 * 2 * 3 * … * (N-1) * N

Para ello necesitaremos una nueva variable que recorra todos los números que hay que multiplicar. La llamaremos numero_actual, y tomará todos los valores a multiplicar desde 2 hasta N.

El programa va a comenzar con una asignación múltiple que da los valores iniciales para las variables resultado y numero_actual:

```
resultado, numero_actual := 1, 2
```

Luego sigue el ciclo. En cada paso del ciclo se debe multiplicar un nuevo término al resultado, mientras que el número actual debe avanzar al siguiente:

```
do numero_actual ≤ N →
        resultado, numero_actual := resultado * numero_actual, numero_actual + 1
od
```

Resultado final:

```
Const N : Int ;
Var numero_actual, resultado : Int ;

resultado, numero_actual := 1, 1 ;    // inicialización
```

```
do numero_actual ≤ N →
        resultado, numero_actual := resultado * numero_actual, numero_actual + 1
od
```

Arreglos

Es el único tipo de dato que me permite tener una colección de valores de algún tipo (en el lenguaje imperativo del teórico/práctico). Como de costumbre, la semántica de la programación imperativa se acerca más a la verdadera arquitectura de las computadoras, y los arreglos son el tipo de colecciones más cercano al verdadero funcionamiento de la memoria de una computadora.

El **arreglo** define una colección de valores **ordenada** y de **tamaño fijo**.

Se puede ver como una lista de programación funcional pero de **largo fijo**. No existe el concepto de agregar o quitar elementos en un arreglo (tampoco de concatenar arreglos). También se puede ver como una tupla pero con todas las componentes son del mismo tipo.

En **Algoritmos I** usaremos los arreglos sólo para leer su información y no para modificarlos, escribirlos o calcular resultados contenidos en ellos.

Declaración (tipo)

Sintaxis: `Array[N, M) of <tipoX>`

Semántica: Tengo un arreglo de (M - N) elementos de tipo <tipoX>, cuyos índices son N, N+1, N+2, …. , M-1.

Observaciones:

- <tipoX> puede ser cualquier tipo válido, incluso otro arreglo (e.g. para definir matrices o arreglos multidimensionales).
- los arreglos son siempre “cerrado/abierto” (o sea “[“ / “)”).
- por lo general usaremos letras mayúsculas para variables que contienen arreglos.
- por lo general usaremos N = 0.

Ejemplos:

```
Const A : Array[0 , 4) of Int ;
      // este arreglo tiene elementos A.0, A.1, A.2 y A.3

Var MiArreglo : Array[11, 15) of Bool;

Const N, M : Nat ;
Var Matriz : Array[0, N) of Array[0, M) of Int;
     // tenemos una matriz de N x M números enteros.
```

Consulta / acceso (expresión)

Si yo tengo un arreglo, me interesa acceder a los valores para usarlos en una expresión (por ejemplo para usar la expresión en una asignación o en una guarda de un if o un do).

Sintaxis: Si mi arreglo se llama A , y tengo una expresión E que es tipo entero, la sintaxis es:

```
A.E
```

Semántica:

1. Calcular el valor entero asociado a la expresión E. Sea n este valor.
2. “A.E” me devuelve el valor alojado en la n-ésima posición del arreglo A. (este valor devuelto es del tipo <tipoX>)

Observaciones:

- si E resulta en un valor n que no es una posición válida del arreglo, el acceso da error (el programa se rompe).

Ejemplo: Si tengo:

```
Const A : Array[0 , 4) of Int ;
Var x : Int ;
```

Puedo hacer:

```
x := x + A.3 * 2
```

Otro ejemplo: Si tengo una matriz de 3x3:

```
Var Matriz : Array[0, 3) of Array[0, 3) of Int;
Var x : Int ;
```

Puedo hacer la suma de la diagonal así:

```
x := (A.0).0 + (A.1).1 + (A.2).2
```

Asignación para arreglos (sentencia)

Con la sentencia de asignación para arreglos puedo modificar un valor en una posición.

Sintaxis:

```
A.E := F
```

donde E es una expresión de tipo Nat, y F es una expresión de tipo <tipoX> a donde <tipoX> es el tipo de los elementos del arreglo.

Semántica:

1. Calcular los valores asociados a E y a F.

2. Modificar el estado, cambiando el valor asociado a la posición E por el resultado de F.

Observaciones:

- esta asignación es una asignación distinta a la normal y no se pueden mezclar.
- **NO LO VAMOS A USAR EN ALGORITMOS I.**

Ejemplo: Sumar elementos de un arreglo

Supongamos que tengo N > 0 y un arreglo de N elementos de tipo entero. Quiero calcular la suma de todos los elementos.

Idea del programa: Con un ciclo, usamos una variable para recorrer las posiciones del arreglo, desde la primera hasta la última, y en otra variable vamos calculando la suma.

```
Const N : Int, A : Array[0, N) of Int;
Var pos, res : Int;

res, pos := 0, 0 ;
do pos < N →
      res := res + A.pos ;
      pos := pos + 1
od
```

Equivalentemente:

```
Const N : Int, A : Array[0, N) of Int; <- declaración del arreglo
Var pos, res : Int;

res, pos := 0, 0 ;
do pos < N →
      res, pos := res + A.pos, pos + 1
                           <- consulto el valor en una posición del
                              arreglo
od
```

Uno que no anda:

```
Const N : Int, A : Array[0, N) of Int;
Var pos, res : Int;
res, pos := 0, 0 ;
do pos < N →
      pos := pos + 1
      res := res + A.pos ;
od
```

Este no anda porque se saltea el primer elemento y además da error porque intenta acceder a la posición “N” que no es una posición válida.

Ejercicio: Verificar los tres programas con un ejemplo de un arreglo en concreto.

Testing / Ejecución

Lenguaje imperativo completo

Sintaxis de un programa. Dos secciones principales:

1. Declaración de constantes y variables
2. Sentencia del programa

Semántica de un programa:

- Estado: variables y constantes definidas por las declaraciones.
- Ejecución: Dado un estado inicial, ejecutar la sentencia del programa, modificando el estado hasta llegar a un estado final.

Diferencias con funcional: En imperativo (del teórico) no hay funciones y **no hay un valor resultado**. El **resultado es el estado final entero**.

Diferencias con C: En imperativo (del teórico) no hay funciones ni procedimientos. Sólo estamos tomando de C lo que se llama programación “in the small” (en pequeña escala): Los algoritmos pequeños, sin la superestructura de los programas que podemos definir en C. Tampoco vamos a hacer input/output.

Declaración de constantes y variables

Especificador: `Var` o `Const`. (para avisar si declaro variable o constante)

Nombres (identificadores): Palabras con letras y números (pero empezando con una letra).

Tipos: `Nat, Int, Real, Bool, Char`, arreglos. **No hay listas**.

Sintaxis:

```
Var/Const   nombre1, nombre2, ... , nombren  :  Tipo;
...
Var/Const   nombre1, nombre2, ... , nombren  :  Tipo;
```

Ejemplo:

```
Var x, y : Int;
Var resultado : Bool;
```

Otro ejemplo:

```
Const divisor, dividendo : Int;
Var cociente, resto : Int;
```

Observación: El valor de las constantes no estará determinado en los programas (no lo escribiremos como parte de la sintaxis), si no que podrá ser cualquier valor (en principio). Luego veremos cómo se podrá restringir los valores posibles. (por ejemplo: me interesa que el divisor no sea cero).

Otro ejemplo (con variables de varios tipos en la misma línea):

```
Var x : Int, b : Bool;
```

Sentencias

Dos tipos:

- Elementales: sentencias básicas que no necesitan de otras sentencias.
- Compuestas: sentencias que se arman a partir de otras sentencias (o sea, sentencias que tienen adentro otras sentencias).

Skip

Es una sentencia elemental que no hace nada.

Sintaxis:

`skip`

Semántica: No modifica el estado.

Asignación (:=)

Sentencia elemental que se usa para asignar valores a variables.

Sintaxis:

`v1, v2, …, vn := E1, E2, …, En`

donde `v1, v2, …, vn` son variables declaradas del programa, y `E1, E2, ..., En` son expresiones válidas en lenguaje y del tipo que se corresponde con las variables.

Semántica:

1. Se calculan valores-resultado para todas las expresiones E1, E2, …, En. (O sea, se hace usando el estado actual, no el nuevo estado.)
2. Se modifica el estado, asignando a cada variable vi el resultado de calcular la expresión Ei.

Expresiones

Las expresiones válidas en programación imperativa son parecidas a las expresiones en funcional, pero con algunas diferencias:

- valores constantes
- variables y constantes declaradas
- operaciones básicas (+, -, *, /, div, mod, max, min, $\wedge$, $\vee$, $\neg$, =, $\neq$, $\leq$, <, >, etc.)
- accesos a elementos de arreglos
- **NO: llamadas a funciones, cuantificadores ni cosas que esconden cuantificadores (factorial, exponenciación, etc.)**
- **NO: análisis por casos, definiciones locales**
- **NO: expresiones de tipo lista**

Secuenciación (;)

Primera sentencia compuesta que vemos. La secuenciación me permite ejecutar una sentencia y después otra.

Sintaxis:

```
S1 ; S2
```

donde S1 y S2 son dos sentencias.

Semántica:

1. Ejecutar S1 a partir del estado inicial.
2. Ejecutar S2 a partir del estado que resulta del paso 1. (el estado queda como lo deja la ejecución de S2).

Observaciones:

- La secuenciación es **asociativa**:

Es lo mismo (semánticamente)

```
(S1 ; S2) ; S3
```

que

```
S1 ; (S2 ; S3)
```

Luego, podremos escribir directamente:

```
S1 ; S2 ; S3
```

- Un programa es en realidad siempre **una sola sentencia**. Cuando hablamos de varias sentencias en realidad éstas forman una sola sentencia a través de la secuenciación.

Condicional (if)

Sentencia compuesta. El condicional me permite ejecutar diferentes sentencias dependiendo de una condición booleana.

Sintaxis:

```
if B1 → S1
□ B2 → S2
...
□ Bn → Sn
fi
```

donde B1, B2, …, Bn (llamadas **guardas**) son expresiones de tipo booleano, y S1, S2, …, Sn son sentencias.

Semántica:

1. Se evalúan todas las guardas B1, …, Bn usando el estado actual. (se obtienen valores booleanos).
2. Se elige alguna de las guardas que da True. Si ninguna da True, el programa **termina con error** (“se rompe”). Supongamos que elegimos la k-ésima guarda (Bk = True).
3. Ejecutar esa rama del if: Sk. (el estado final es el que resulta de esta ejecución).

Observación: **No determinismo:** Puede haber varias guardas que den True. En ese caso, la semántica dicta que se puede ejecutar cualquiera de esas (pero siempre una y sólo una). La semántica no me dice cuál va a ser (no necesariamente es la primera). El programador no elige.

Repetición, ciclo o bucle (do)

Sentencia compuesta. Me permite repetir la ejecución de una sentencia mientras se cumpla una condición.

Sintaxis:

do B → S od

donde B es una expresión booleana (**guarda**), y S es una sentencia (**cuerpo del ciclo**).

Semántica:

1. Se evalúa la guarda B en el estado actual.
2. Si da False, termina la ejecución.
   Si da True, modifica el estado ejecutando S, y luego vuelve al punto 1.

Anotaciones de programa

Recordemos este programa: “Dado un número entero N > 0, queremos contar cuántos **números entre 0 y N** son múltiplos de 6.”

Solución:

```
    Const N : Int ;
    Var numero_actual, resultado : Int ;

ℓ1  numero_actual, resultado := 0, 0 ;
ℓ2  do numero_actual ≤ N →
          < - prestamos atención a los estados posibles en este punto
ℓ3      if numero_actual mod 6 = 0 →
ℓ4          resultado := resultado + 1 ;
ℓ5      □ numero_actual mod 6 ≠ 0 →
ℓ6          skip
ℓ7      fi ;
ℓ8      numero_actual := numero_actual + 1
ℓ9  od
```

Supongamos el siguiente estado inicial:

σ_0: N ↦ 7, numero_actual ↦ -11, resultado ↦ 77

En la línea 2, voy a tener una secuencia de estados:

σ_1: N ↦ 7, numero_actual ↦ 0, resultado ↦ 0
σ_2: N ↦ 7, numero_actual ↦ 1, resultado ↦ 1
σ_3: N ↦ 7, numero_actual ↦ 2, resultado ↦ 1
σ_4: N ↦ 7, numero_actual ↦ 3, resultado ↦ 1
σ_5: N ↦ 7, numero_actual ↦ 4, resultado ↦ 1
σ_6: N ↦ 7, numero_actual ↦ 5, resultado ↦ 1
σ_7: N ↦ 7, numero_actual ↦ 6, resultado ↦ 1
// todavia no se hizo el chequeo para el 6
σ_8: N ↦ 7, numero_actual ↦ 7, resultado ↦ 2
σ_9: N ↦ 7, numero_actual ↦ 8, resultado ↦ 2
// aca la guarda se hace False, el programa termina
…

Para otros estados iniciales posibles, voy a tener otras secuencias posibles de estados en ese punto (la línea 2).

En general, en toda línea de mi programa voy a tener un conjunto de estados posibles en ese punto. Para describir propiedades que cumplen los estados posibles en un punto de mi programa puedo usar **predicados.**

Ejemplo: En la línea 2 puedo escribir los siguientes predicados:

- True
- numero_actual ≥ 0
- numero_actual ≤ N+1
- numero_actual ≤ N+10000
- resultado ≥ 0

Incluso podemos poner conjunciones de los anteriores:

- $0 \leq numero\_actual \leq N+1$
- $0 \leq numero\_actual \leq N+1 \ \wedge \ 0 \leq resultado \leq N+1$

Incluso podemos usar expresiones cuantificadas para decir cosas más interesantes:

- $resultado = \langle \ N \ \ i \ : 0 \leq i \leq N \ : i \bmod 6 = 0 \ \rangle$

¿es cierto esto en la línea 2? ¿vale en todo estado posible en la linea 2?
(segun este predicado, resultado debe valer 5 que sería en realidad el resultado final)
NO VALE SIEMPRE en la línea 2.
Lo corregimos un poco:

- $resultado = \langle \ N \ \ i \ : 0 \leq i < numero\_actual \ : i \bmod 6 = 0 \ \rangle$

(estoy diciendo que en la línea 2 el estado es tal que en resultado he calculado un **"resultado parcial"**, que en este caso sería contar los múltiplos de 6 pero sólo hasta numero_actual-1).

Reescribamos el programa pero con algunas anotaciones:

```
  Const N : Int ;
  Var numero_actual, resultado : Int ;
          {  N > 0  }
          // PRECONDICIÓN: esta anotación de programa me habla del estado inicial
ℓ1 numero_actual, resultado := 0, 0 ;
          {  numero_actual = 0 ∧ resultado = 0 }
ℓ2 do numero_actual ≤ N →
          { resultado = ⟨ N  i  : 0 ≤ i < numero_actual  : i mod 6 = 0 ⟩ }
ℓ3      if numero_actual mod 6 = 0 →
ℓ4          resultado := resultado + 1 ;
ℓ5      □ numero_actual mod 6 ≠ 0 →
ℓ6          skip
ℓ7      fi ;
ℓ8      numero_actual := numero_actual + 1
ℓ9 od
   { numero_actual = N + 1 ∧  resultado = ⟨ N  i  : 0 ≤ i ≤ N  : i mod 6 = 0 ⟩  }
     // POSTCONDICIÓN: esta anotación de programa me habla del estado final
```

DEFINICIÓN: Una **anotación de programa** es un predicado que se puede introducir en un punto de un programa. Este predicado afirma algo que es satisfecho por todo estado posible en ese punto del programa.

Observaciones:

- Las anotaciones de programa **no son** parte de los programas.
- Las vamos a insertar en medio de los programas usando llaves "{" , "}".
- Todo estado posible en ese punto del programa debe satisfacer el predicado, pero también el predicado puede admitir más estados que nunca suceden en ese punto.
- **Corolario:** El predicado "True" es siempre una anotación de programa correcta. (aunque es poco informativa)

Precondición y postcondición

DEFINICIÓN: Llamamos **precondición** a la anotación de programa que usamos para describir el estado inicial, o sea la que ubicamos en el punto inicial del programa.

DEFINICIÓN: Llamamos **postcondición** a la anotación de programa que usamos para describir el estado final, o sea la que ubicamos en el punto final del programa.

Observaciones:

- La precondición me dice cosas que puedo asumir acerca de la información de entrada de mi problema.
- La postcondición me sirve para expresar qué debe calcular el programa, o sea, qué problema está siendo resuelto.
- La precondición y las postcondición me sirven para **especificar**: Puedo traducir un enunciado informal en una especificación.

Especificaciones

DEFINICIÓN: En programación imperativa, una **especificación** es una descripción formal del problema que se quiere resolver a través de una precondición y una postcondición. Debe incluirse también la declaración de las constantes y variables que definen el problema.

Notación para especificar:

Const …
Var …
{ P } ← precondición (predicado explícito)
S ← programa incógnita (letra S)
{ Q } ← postcondición (predicado explícito)

Ejemplos

Ejemplo 1: Para el problema ya visto: “Dado un número entero N > 0, queremos contar cuántos **números entre 0 y N** son múltiplos de 6.”

Una especificación posible es:

```
Const N : Int
Var resultado : Int

{ N > 0 }
  S
{ resultado = ⟨ N i : 0 ≤ i ≤ N : i mod 6 = 0 ⟩ }
```

(observación: no se declara numero_actual, porque no es parte de la especificación si no de la solución.)

Ejemplo 2: Sumar los elementos de un arreglo: “Dados N > 0 y un arreglo de N elementos de tipo entero, **calcular la suma de los elementos**.”

Especificación:

```
Const N : Int, A : array [0, N) of Int
Var resultado : Int
{ P: N > 0 }
  S
{ Q:  resultado = ⟨ ∑ i : 0 ≤ i < N : A.i  ⟩}
```

Ejemplo 3 (práctico 5, ejercicio 1): Especificar! Ejercicio…

Ejemplo 4: Dado N ≥ 0, calcular el factorial de N.
Especifiquemos:

```
Const N : Int
Var factorial : Int

{ P: N ≥ 0 }
  S
{ Q: factorial = ⟨ ∏ i : 1 ≤ i ≤ N : i ⟩}
```

Equivalentemente, ya que podemos usar cualquier cosa de la matemática:

```
Const N : Int
Var factorial : Int

{ P: N ≥ 0 }
  S
{ Q: factorial =  N ! }
```

(no hace falta usar expresiones cuantificadas)

Ejemplo 5: “Escriba un programa que sume, por un lado, los valores pares y, por otro, los valores múltiplos de 3 de los números entre N y M (inclusives).”

Especificamos:

```
Const N, M : Int
Var suma_mult2, suma_mult3 : Int

{ P:  N ≤ M }
  S
{ Q: suma_mult2 = ⟨∑ i : N ≤ i ≤ M ∧ i mod 2 = 0 : i ⟩∧
     suma_mult3 = ⟨∑ i : N ≤ i ≤ M ∧ i mod 3 = 0 : i ⟩   }
```

Ejemplo 6: Algoritmo de la división: “Dados dos números x e y, con x ≥ 0 e y > 0, calcular cociente y resto de la división entera de x por y.”

Especificamos:

```
Const x, y : Int
Var q, r : Int

{ P: x ≥ 0  ∧  y > 0 }
  S
{ Q: q = x div y  ∧  q = x mod y  }
```

Equivalentemente, usando el teorema de la división entera:

```
Const x, y : Int
Var q, r : Int

{ P: x ≥ 0  ∧  y > 0 }
  S
{ Q: x = q * y + r  ∧  0 ≤ r < y }
```

Ternas de Hoare

Introducción

¿Para qué sirve una especificación? Sirve para describir lo que el programa **tiene** que hacer si quiere resolver el problema especificado.

```
Const ...
Var ...
{ P }    ← precondición (predicado explícito)
  S      ← programa incógnita (letra S)
{ Q }    ← postcondición (predicado explícito)
```

Para que el programa S resuelva el problema especificado, debe suceder lo siguiente: Que ***cada vez que se ejecute el programa comenzando de un estado inicial que satisface la precondición P, la ejecución debe terminar en un estado final que satisface la postcondición Q.***

Ejemplo: Supongamos que tenemos dos variables dadas x e y de tipo entero, quiero intercambiar sus valores.

Especificación:

```
Var x, y : Int
```

```
{ P: x = A  ∧  y = B }   // fijo los valores de x e y en el estado inicial.
                         // A y B son variables de especificación

  S                      // en el programa, A y B no existen

{ Q: x = B ∧ y = A }
```

Variables de especificación

DEFINICIÓN: son variables que usamos sólo en las anotaciones de los programas para fijar valores en determinados puntos. No son parte del programa, no se declaran (ni como Const ni como Var) y no se usan en el programa.

¿Qué sentencia **S** soluciona el problema especificado?

```
x, y := A, B    // NO: justamente dijimos no se pueden usar A y B.

x, y := y, x   // ANDA!
```

También se soluciona introduciendo una tercer variable `aux : Int`:

```
aux := x ;
x := y ;
y := aux    // ANDA!
```

Ambas sentencias funcionan, esto quiere decir que valen las especificaciones si yo reemplazo S por las sentencias correctas. Haciendo el reemplazo me quedan:

```
Var x, y : Int ;

 { P: x = A  ∧  y = B }
   x, y := y , x
 { Q: x = B ∧ y = A }

Var x, y, aux : Int ;

 { P: x = A  ∧  y = B }
   aux := x ;
   x := y ;
   y := aux
 { Q: x = B ∧ y = A }
```

En general, si tenemos dos predicados P y Q **dados**, y tenemos una sentencia S **dada**, podemos preguntarnos si vale o no vale lo siguiente:

Cada vez que empiezo de un estado inicial que cumple P,
la ejecución de S termina en un estado final que satisface Q.

Esta pregunta tiene respuesta SI o NO (True o False), o sea es un **predicado** que puede valer o no.

A este predicado lo llamaremos **Terna de Hoare** y lo escribiremos de la siguiente manera:

```
{ P }
  S
{ Q }
```

o todo junto:

```
{ P } S { Q }
```

(se llama “Terna” porque tiene tres componentes: la pre, la sentencia y la post)

Definición

DEFINICIÓN: Dados dos predicados P y Q y una sentencia S, decimos

```
{ P }  S  { Q }
```

cuando sucede que

“para todo estado inicial que satisface P,
la ejecución de S termina en un estado final que satisface Q”

Llamamos **Terna de Hoare** a este predicado “{P} S {Q}”.
(se llama “Terna” porque tiene tres componentes: la pre, la sentencia y la post)

Observaciones

Observación 1: Siempre vale la terna

```
{ P } S { True }
```

, siempre y cuando S termine (no dé error ni pueda entrar en ciclo infinito).

Observación 2:

```
{ False } S { Q }
```

¿Vale? **SI.** Afirma que siempre que empiece en un estado que satisface **False** termine en un estado que satisface Q. Si no valiera, debería ser capaz de dar un contraejemplo: un **estado inicial que satisface False** tal que el estado final no satisface Q. No existe tal cosa, así que la terna vale. (es una especie de rango vacío)

Observación 3:

```
{ P } S { False }
```

¿Vale? Depende… si P es False, **vale**. Si P no es False, debe haber algún estado inicial posible y por lo tanto la terna **no vale**.

Observación 4:

```
{ True } S { Q }
```

¿Vale? Depende, debe suceder que para cualquier estado inicial, ejecutar S me deja en un estado final que satisface Q.

Ejemplos

Ejemplo 1: "Swap" (intercambio de valores):

```
{ x = A  ∧  y = B }
  x, y := y , x
{ x = B ∧ y = A }
```

(A y B son variables de especificación)

¿Vale esta terna?
Si empezamos en un estado que satisface P, ¿al ejecutar S terminamos en un estado que satisface Q? ¡Si! Siempre.

Ejemplo 2: ¿Vale esta terna?

```
{ P: x = A  ∧  y = B }
  aux := x ;
  x := y ;
  y := aux
{ Q: x = B ∧ y = A }
```

También.

Ejemplo 3: ¿Vale esta terna?

```
{ P: x = A  ∧  y = B }
x := y ;
y := x
{ Q: x = B ∧ y = A }
```

No vale. Luego, debe haber al menos un contraejemplo: Algún estado inicial que satisface P, tal que luego de ejecutar S el estado final no satisface Q.
Contraejemplo: σ_0: x → 17 , y → 20 (vale P con A = 17, B = 20)
σ_1: x → 20 , y → 20 (no vale Q con A = 17, B = 20)

Usaremos las Ternas de Hoare para afirmar cosas que satisfacen los programas / sentencias.

Ejemplo 4 (práctico 5, ejercicio 4a):

```
{ x > 0 }   x := x * x   { True }
```

¿Vale? Sí vale, ya que siempre terminamos en un estado final que satisface **True**.

Ejemplo 5 (práctico 5, ejercicio 4b):

```
{ x ≠ 100 } x := x * x { x ≥ 0 }
```

¿Vale? Sí, ya que x es el cuadrado de un número, que siempre es ≥ 0.

```
{ x = 100 } x := x * x { x ≥ 0 }
```

¿Vale? También, en realidad no importa la precondición, la terna siempre vale.

Ejemplo 6: ¿Vale esta terna?

```
Var  r, x, y : Int

{ True }
if x ≥ y →
    r := x
[] x ≤ y →
    r := y
fi
{ r = x max y }
```

Observación: No determinismo: puede suceder que ambas guardas sean verdaderas pero esto no representa ningún problema (repasar la semántica del if). Si ambas son verdaderas, se ejecuta sólo una de las dos (no sabemos cual).

Ejemplo 7: Cálculo del máximo mal especificado:

```
Var r, x, y : Int

{ P: True }
S
{ Q: r = x max y }
```

¿Cuál es el programa más simple que podemos dar que satisface esta especificación?

¿Vale la siguiente terna?:

```
Var r, x, y : Int

{ True }
r, x, y := 0, 0, 0
```

```
{ r = x max y }
```

¡Vale!

¿Es correcta esta especificación para el cálculo del máximo entre x e y? No, porque admite programas que no reflejan el problema que yo quiero solucionar.

La especificación funciona como **<u>contrato</u>** entre la persona que quiere resolver el problema y la persona que lo va a solucionar.

Especificación correcta para max:

```
Var  r, x, y : Int

{ x = A ∧ y = B }
  S
{ r = A max B }
```

(usando variables de especificación A y B)

Derivación vs demostración

<u>Derivación:</u> Dada una especificación (con su correspondiente precondición P y postcondición Q), derivar es obtener un programa S (o sea, una sentencia S) tal que vale la Terna de Hoare:

$$\{ P \}\ S\ \{ Q \}$$

<u>Demostración:</u> Dada una especificación (con pre P y post Q), y un programa S, demostrar que el programa satisface la especificación, o equivalentemente, demostrar que vale la Terna de Hoare:

$$\{ P \}\ S\ \{ Q \}$$

En lo que resta de la materia veremos cómo hacer esto.

Precondición más débil

¿Para cuáles precondiciones vale la siguiente terna?

```
{ P }
  x := x * x
{ x ≥ 9 }
```

<Propongamos predicados P para los que vale esta terna:

- **True**? No vale: Si el estado inicial es x→ 1, se rompe la terna.
- x = 1000? Sí. Es un caso muy particular.
- x ≥ 20? Sí, y abarca más casos que el predicado anterior.
- x ≥ 3? Sí, y abarca más casos todavía.
- x = -10? Sí, otro caso **particular**
- **False**? Sí vale (no hay contraejemplo para refutarlo). Es el caso más particular posible (conjunto vacío de estados iniciales posibles).
- x ≤ -3 ? Sí vale, y abarca todos los x negativos para los que vale.
- **<u>El más general de todos (juntando dos anteriores):</u>**
    - x ≤ -3 ∨ x ≥ 3
    - equivalentemente, | x | ≥ 3
    - equivalentemente, x^2 ≥ 9

Siempre que tenemos un programa S y una postcondición Q, nos interesa preguntarnos cuál es la precondición más general (o abarcativa) P tal que vale la Terna de Hoare:

$$\{ P \}\ S\ \{ Q \}$$

Llamamos a P **<u>precondición más débil (weakest precondition)</u>** de S y Q, y la denotamos:

$$wp.S.Q \quad \text{(esto es P)}$$

Fortaleza y debilidad de predicados

Dados dos predicados P y Q, decimos que P es **<u>más débil</u>** que Q, o que Q es **<u>más fuerte</u>** que P, si y sólo sí:

$$Q \Rightarrow P$$

(ejemplo: P ≡ x ≥ 3,
Q ≡ x = 10.
Q es mas fuerte que P, P es más débil que Q, ya que claramente
x = 10 ⇒ x ≥ 3)

<u>Intuiciones:</u> Un predicado es más débil si exige menos cosas (o sea, admite más posibilidades, o un conjunto más grande de estados posibles), y es más fuerte y exige más cosas (un conjunto más chico de estados posibles).

Dibujo de el conjunto de estados que satisfacen un predicado. Si Q es mas fuerte que P, o sea Q ⇒ P, tengo lo siguiente:

Observaciones:

- El predicado más débil de todos es True (y define el conjunto universal de estados posibles).
- El predicado más fuerte de todos es False (y define el conjunto vacío de estados posibles).
- No siempre dos predicados están relacionados por debilidad/fortaleza.
    - Ejemplo: P ≡ x = 10, Q ≡ x = -3 (son disjuntos)
    - Ejemplo: P ≡ x ≥ 3, Q ≡ x ≤ -3 (son disjuntos)
    - Ejemplo de dos que no son disjuntos: P ≡ x > 3, Q ≡ x < 5, ya que el estado x→1 satisface el 2do pero no el 1ro, y el estado x→7 satisface el 1ro pero no el 2do. (igual no son disjuntos ya que ambos contienen al “x→4”)
    - Otro: P ≡ x ≤ 3, Q ≡ x ≥ -3

Weakest Precondition (WP)

DEFINICIÓN: Dado un programa S (o sea, una sentencia) y un predicado Q (la postcondición), la **weakest precondition** (precondición más débil) es el predicado más débil P tal que vale

$$\{ P \} S \{ Q \}$$

Denotamos a P como “wp.S.Q”.

Relación entre la WP y la Terna de Hoare

Observacion 1: ¿Vale la siguiente terna?

$$\{ wp.S.Q \} \; S \; \{ Q \}$$

Sí vale, ya que por definición la wp es tal que cumple la terna.

Observacion 2: Si buscamos otra precondición P' tal que vale la terna:

$$\{ P' \} \; S \; \{ Q \}$$

¿Qué podemos decir de la relación entre la wp y P'?
Podemos decir que **P' es más fuerte que la wp:**

$$P' \Rightarrow wp.S.Q$$

ya que por definición la wp es la más débil de las que cumplen la terna.

Resumiendo:

$$\mathbf{\{ P' \} \; S \; \{ Q \} \quad implica \quad P' \Rightarrow wp.S.Q}$$

Observación 3: Si tenemos un predicado **P' más fuerte que la wp**:

$$P' \Rightarrow wp.S.Q \; \mathbf{(*)}$$

¿Qué podemos decir de la siguiente terna?

$$\{ P' \} \; S \; \{ Q \}$$

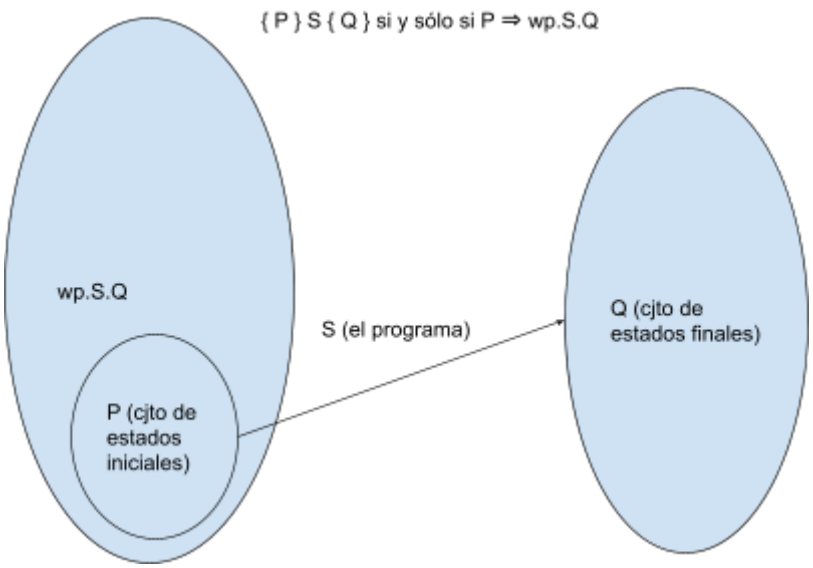
La terna vale, ya que si un estado inicial satisface P' entonces satisface la wp (por **(*)**). Por otro lado, la wp satisface la terna (observación 1), luego ejecutar S nos deja en un estado final que satisface Q.

Conclusión: A partir de las observaciones 2 y 3 podemos concluir que:

$$\{ P' \} \; S \; \{ Q \} \text{ sí y sólo si } P' \Rightarrow wp.S.Q$$

Reescribiendo usando P en lugar de P':

$$\mathbf{\{ P \} \; S \; \{ Q \} \text{ sí y sólo si } P \Rightarrow wp.S.Q}$$



Intuición: La wp.S.Q es la precondición más débil, luego incluye a cualquier estado inicial válido (o sea, estado inicial tal que al ejecutar S termino en un estado final que satisface Q). Luego cualquier otra precondición válida posible P debe ser más fuerte que la wp (o sea, implicarla: P ⇒ wp.S.Q)

Ejemplo anterior:
{ P }
x := x * x ← S
{ x ≥ 9 } ← Q

Acá ya vimos que la wp es |x| ≥ 3. O sea:

|x| ≥ 3 ≡ wp.S.Q ≡ wp.(x := x * x).(x ≥ 9)

Otro ejemplo:
{ P }
x := x * x
{ x ≥ 9 ∧ x mod 2 = 0 }

¿Cuál es la wp.S.Q?

- Es |x| ≥ 3 pero además x debe ser par: |x| ≥ 3 ∧ x mod 2 = 0

Otro ejemplo:
{ P }
y, x := y + y, x + y
{ x = 7 ∧ y = 10 }

¿Cuál es la wp.S.Q? Es x = 2 ∧ y = 5. Podemos chequearlo haciendo la ejecución.

Otro ejemplo:

{ P }
  x := x + 2
{ x ≥ 7 }

La wp.S.Q es x ≥ 5, ya que para que x sea ≥ 7 después de haberle asignado x + 2, originalmente x tendría que haber sido ≥ 5.

¿Hay alguna forma sistemática de calcular la precondición más débil? Sí para la mayoría de los tipos de sentencias que tenemos:

- skip
- asignación (:=)
- secuenciación (;)
- condicional (if)

Para el ciclo (do) no vamos a tener una forma de calcular la wp (usaremos otras técnicas).

WP de la asignación

Supongamos que tenemos la siguiente asignación S:

$$v1, v2, \ldots, vn := E1, E2, \ldots, En$$

y una postcondición Q.

Luego, la weakest precondition para S y Q es el siguiente predicado:

$$wp.S.Q \equiv Q(v1 \leftarrow E1, v2 \leftarrow E2, ..., vn \leftarrow En)$$

Esto es, tomar la postcondición, y reemplazar cada variable asignada por la expresión que lleva asignada.

Probemos con un ejemplo:

{ P }
  x := x + 2 ← S
{ x ≥ 7 } ← Q

Luego,

  wp.(x := x + 2).(x ≥ 7)
≡ { definición de wp para la asignación }
  (x ≥ 7)(x ← x + 2)
≡ { hago la sustitución }
  x + 2 ≥ 7
≡ { arit. }
  x ≥ 5

o sea que parece que anda.

Otro:
{ P }
  y, x := y + y, x + y
{ x = 7 ∧ y = 10 }

  wp.(y, x := y + y, x + y).(x = 7 ∧ y = 10)
≡ { def. wp para := }
  (x = 7 ∧ y = 10)(y ← y + y, x ← x + y)
≡ { aplico sust. }
  x + y = 7 ∧ y + y = 10
(hasta acá alcanza pero podemos simplificar un poco:)
≡ { despejamos la y }
  x + y = 7 ∧ y = 5
≡ { leibniz }
  x + 5 = 7 ∧ y = 5
≡ { despejamos la x }
  x = 2 ∧ y = 5

¡Anda!

Ejercicio: Calcular la wp análiticamente (o sea, usando la definición) para todos los ejemplos que vimos a ojo.

WP del skip

Supongamos que tengo el programa skip y una poscondición Q:

{ ??? }
 skip
{ Q }

¿Cuál es la wp.skip.Q?
Veamos predicados que como pre, hacen valer la terna:

- True? Seguro que no porque al ejecutar skip (o sea, no hacer nada), no puedo garantizar que vale Q.
- False? Si, lo hemos visto varias veces
- Q? Sí, porque al ejecutar skip (o sea, no hacer nada), llego a un estado que satisface Q.
- Algo más fuerte que Q? O sea, un Q' tal que Q' ⇒ Q? Vale, ya que Q' sería un subconjunto de Q.
- Algo más débil que Q? No necesariamente, me estaría acercando a True.,

De todas estas reflexiones podemos ver que: con Q vale la terna, y no podemos encontrar nada más débil que Q, así que Q es la wp.S.Q:

wp.skip.Q ≡ Q

Derivación de programas imperativos

Tengo la terna:

{ P } ← **precondición dada**
S ??? ← **el programa que tengo que encontrar**
{ Q } ← **postcondición dada**

Proponer y demostrar. Refinar y encontrar incógnitas.

Dos grandes formas de derivar/demostrar con Ternas:

- directamente con un predicado asociado a la Terna
- usando la wp y demostrando el predicado: “P ⇒ wp.S.Q”.

Digesto

imperativo.pdf

Skip

Verificación con Terna de Hoare:

{ P } skip { Q }
≡ (“vale sí y sólo si …”)
P ≡ Q (podría ser, pero es demasiado fuerte)
Q ⇒ P (Q es mas fuerte que P = Q incluido en P = hay elementos de P que no estan en Q
empiezo de un estado que satisface P, no hago nada,
el estado satisface Q? NO!)
Ejemplo: Q ≡ x = 100, P ≡ x ≥ 3.
¿Vale esta terna? { P: x ≥ 3 } skip { Q: x = 100 } NO!
P ⇒ Q (empezamos de un estado que satisface P, como P está incluido en Q (o P ⇒ Q)
ese estado necesariamente satisface Q sin necesidad de hacer nada).
Ejemplo: P ≡ x = 100, Q ≡ x ≥ 3.

Precondición más débil:

wp.skip.Q ≡ Q

(la wp es el conjunto que está incluído en Q y que es lo más grande posible, por lo tanto es el mismo Q.)

Observación: Acá da lo mismo usar la verificación con la terna que con la wp.

El skip en derivaciones

Tengo la terna:
{ P } ← **precondición dada**
S ← **el programa que tengo que encontrar**
{ Q } ← **postcondición dada**

¿Cuál es el programa más simple que podría existir? Es skip.
¿Puede ser que S sea skip?
¿Qué predicado debe valer para que valga la terna?
Debe valer: P ⇒ Q. **Es lo primero que uno debe pensar a la hora de derivar.**

Ejemplo: Supongamos que tenemos las siguientes constantes/variables:
Const N : Int, A : Array[0, N) of Int;
Var pos, res : Int;

{ P: res = 0 ∧ pos = 0 }
S
{ Q: res =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩}

¿Puede S ser skip? Sí, ya que P ⇒ Q.
Hagamos la demostración. Supongamos P (hipótesis) y partiendo de Q lleguemos a True.

Q
≡ { def. Q }
res =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩
≡ { hip. P }
0 =⟨ ∑ i : 0 ≤ i < 0 : A.i ⟩
≡ { rango vacío }
0 = 0
≡ { lógica }
True

Otro ejemplo:
{ P: res =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ ∧ pos = N }
S
{ Q: res =⟨ ∑ i : 0 ≤ i < N : A.i ⟩} // res tiene la suma de todos los elementos del arreglo

¿Puede S ser skip? ¿Qué debe valer para que valga la terna? P ⇒ Q. ¿Vale? Sí vale!
Supongamos P y veamos Q:
Q
≡ { def. Q }
res =⟨ ∑ i : 0 ≤ i < N : A.i ⟩
≡ { hip. pos = N }
res =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩

≡ { esta es la otra parte de la hip }
  True

¿Qué pasa con Q ⇒ P? ¿Vale? Sí así fuera, tendría P ≡ Q. No vale!! Q no dice nada de pos, podría valer cualquier cosa. **Ejemplo:** N = 3, A = [43, 65, -5], res = 103, pos = 5. (vale Q pero no vale P).

Asignación

Sea S la sentencia:

$$v1, v2, \ldots, vn := E1, E2, \ldots, En$$

Verificación con Terna de Hoare:

{ P }
v1, v2, …, vn := E1, E2, …, En
{ Q }
≡
P ⇒ Q(v1 ← E1, v2 ← E2, ..., vn ← En)

Precondición más débil:

wp.(v1, v2, …, vn := E1, E2, …, En).Q ≡ Q(v1 ← E1, v2 ← E2, ..., vn ← En)

Esto es, tomar la postcondición, y reemplazar cada variable asignada por la expresión que lleva asignada.

Observación: Acá da lo mismo usar la verificación con la terna que con la wp.

Ejemplo:

{ P }
y, x := y + y, x + y
{ x = 7 ∧ y = 10 }

Calculemos la wp:

wp.(y, x := y + y, x + y).(x = 7 ∧ y = 10)
≡ { def. wp para := }
(x = 7 ∧ y = 10)(y ← y + y , x ← x + y)
≡ { aplicamos el reemplazo } // a estos dos pasos muchas veces vamos a hacerlos juntos
(x + y) = 7 ∧ (y + y) = 10
≡ { despejo y de la 2da ecuación }
(x + y) = 7 ∧ y = 5
≡ { leibniz (reemplazo iguales por iguales }
(x + 5) = 7 ∧ y = 5
≡ { despejo x }
x = 2 ∧ y = 5

Otro ejemplo: Supongamos que tenemos las siguientes constantes/variables:

Const N : Int, A : Array[0, N) of Int;
Var pos, res : Int;

Veamos esta terna:

{ ??? }
res, pos := 0, 0
{ res =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ }

¿Cual es la wp? ¿Sale a ojo? Sí, no importa el estado inicial, la terna vale siempre, por lo tanto la wp es True. Cálculo:

wp.(res, pos := 0, 0).(res =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩)
≡ { def wp para :=, y aplico la sustitución }
0 =⟨ ∑ i : 0 ≤ i < 0 : A.i ⟩
≡ { rango vacío y lógica }
True

La asignación en derivaciones

Tengo la terna:
{ P } ← precondición dada
S ← el programa que tengo que encontrar
{ Q } ← postcondición dada

Ya sabemos que S no es skip (o sea, P no implica Q).
¿Puede ser que S sea una asignación? Podemos probar tomando todas las variables del programa y buscando asignarlas a "incógnitas":
v1, v2, … , vn := I1, I2, …, In

Luego, intentar demostrar la terna (o sea, demostrar **P ⇒ wp.S.Q**), y en el medio de la demostración, **elegir** expresiones posibles para las incógnitas de manera estratégica (o sea, que me sirvan esas decisiones para llegar a **True**).

Ejemplo:

{ P: True }
S
{ Q: res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ }

¿Puede S ser skip? No, P no implica Q.
¿Puede S ser una asignación? Probemos con esta:
res, pos := E, F (incógnitas E y F)

Tratemos de demostrar la terna, eligiendo convenientemente valores para E y F que me permitan llegar a True. O sea, demostrar P ⇒ wp.(res, pos := E, F).Q. Supongamos P (realmente no tengo hipótesis que me sirvan) y veamos la wp:

Libertades: puedo elegir que E y F sean lo que se me cante (siempre que sean expresiones programables).
Objetivo: llegar a True

wp.(res, pos := E, F).Q
≡ { def. wp }
E = ⟨ ∑ i : 0 ≤ i < F: A.i ⟩
≡ { me conviene forzar un rango vacío, o sea **elegir F=0** }
E = ⟨ ∑ i : 0 ≤ i < 0 : A.i ⟩
≡ { ahora ya tengo rango vacío }
E = 0
≡ { está claro que **debo elegir E = 0** }
0 = 0
≡ { lógica }
True

Luego, acabamos de derivar/demostrar el siguiente programa:

{ P: True }
res, pos := 0, 0
{ Q: res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ }

Otro ejemplo: A veces en las asignaciones vamos a saber algunas de las expresiones a priori (ya sea porque nos las dicen o porque lo sabemos por intuición/creatividad).

En este ejemplo ya sabemos la asignación para pos: Quiero sí o sí incrementar pos en 1.

{ P: res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ ∧ 0 ≤ pos < N }
res, pos := E, pos + 1
{ Q: res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ }

¿Qué pasaría si no estuviera obligado a incrementar pos en 1? Podría directamente usar skip ya que la pre ⇒ la post:

{ res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ ∧ 0 ≤ pos < N }
skip
{ res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ }

Obviamente si esto fuera el cuerpo de un ciclo, no serviría ya que no terminaría nunca.

Sólo nos falta encontrar la incógnita E en el programa. Hagamos la demostración y elijamos E convenientemente.
Supongamos la hipótesis P: res =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ ∧ **0 ≤ pos < N**
Y veamos la wp:
wp.(res, pos := E, pos + 1).(res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩)
≡ { def. wp para := }
E =⟨ ∑ i : 0 ≤ i < pos + 1: A.i ⟩ (esto suma: A.0 + A.1 + … + A.(pos-1) + A.pos)
(de esta suma, lo subrayado es “res”)
(hagamos aparecer res:)
≡ { lógica }

E =⟨ ∑ i : 0 ≤ i < pos ∨ i = pos : A.i ⟩
≡ { part. rango }
E =⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ + ⟨ ∑ i : i = pos : A.i ⟩
≡ { hipótesis }
E = res + ⟨ ∑ i : i = pos : A.i ⟩
≡ { rango unitario }
E = res + A.pos
≡ { elijo **E = res + A.pos** }
res + A.pos = res + A.pos
≡ { lógica }
True

Lo logramos! Acabamos de derivar/demostrar el siguiente programa:

{ P: res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ ∧ 0 ≤ pos < N }
res, pos := res + A.pos, pos + 1
{ Q: res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ }

(que si se fijan, es el cuerpo del ciclo del programa que suma un arreglo)

Observación: Fijarse que la pre y la post tiene una parte en común muy importante:
res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩
Que “no cambia” en el sentido de que vale **antes y después** de la asignación. Esta asignación, recordemos, es el cuerpo del ciclo de un programa más grande, y este predicado se denomina **invariante** del ciclo.

Pequeño repaso: Quiero encontrar E tal que vale la siguiente Terna:
{ P }
res, pos := E, pos + 1
{ Q }
O sea, equivalentemente, que vale:
P ⇒ wp.(res, pos := E, pos + 1).Q
O sea, equivalentemente, que si yo tomo P como hipótesis, vale:
wp.(res, pos := E, pos + 1).Q
Luego, si intento hacer la demostración, y en algún momento encuentro alguna expresión que vaya ser E que me sirva para llegar a True, **ya estaría**.
Y pude hacerlo. ya que si elijo que E sea “res + A.pos”, puedo llegar a True.

Condicional (if)

Verificación con Terna de Hoare:

{ P }
if $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
…
[] $B_n \rightarrow S_n$

fi
{ Q }

≡

i) Sí o sí debe valer al menos una de las guardas:

$\mathbf{P \Rightarrow B_1 \vee B_2 \vee \ \ldots \ \vee B_n}$

ii) Todas las ramas son válidas: Para todo i,

para la rama i, **si ejecuto S_i**, quedo en un estado final que satisface Q.

¿Qué puedo asumir que vale en el estado inicial? Vale P, pero además vale B_i, ya que si no, no ejecutaría esta rama.

Escrito con Ternas, sería:

$\{ P \wedge B_i \} \ S_i \ \{ Q \}$

Dijimos que era para todos los i, escribimos:

$\mathbf{\{ P \wedge B_1 \} \ S_1 \ \{ Q \}}$
$\mathbf{\wedge \ \{ P \wedge B_2 \} \ S_2 \ \{ Q \}}$
$\mathbf{\wedge \ \ldots.}$
$\mathbf{\wedge \{ P \wedge B_n \} \ S_n \ \{ Q \}}$

Resumiendo:

{ P }
if $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
…
[] $B_n \rightarrow S_n$
fi
{ Q }

≡

$\mathbf{(P \Rightarrow B_1 \vee B_2 \vee \ \ldots \ \vee B_n)}$
$\mathbf{\wedge \{ P \wedge B_1 \} \ S_1 \ \{ Q \}}$
$\mathbf{\wedge \ \{ P \wedge B_2 \} \ S_2 \ \{ Q \}}$
$\mathbf{\wedge \ \ldots.}$
$\mathbf{\wedge \{ P \wedge B_n \} \ S_n \ \{ Q \}}$

Ejemplo:

{ P }
if $x > 0 \rightarrow$ “hago algo”
[] $x < 0 \rightarrow$ “hago otra cosa”
fi
{ Q }

Acá, si P es True esta terna no vale ya que puede suceder que todas las guardas sean falsas (si en el estado $x \rightarrow 0$).

Si P es $x \neq 0$ tengo garantizado que sí o sí una de las guardas es verdadera.

—

Precondición más débil: Ejercicio verla uds en el digesto (no preocuparse por esto igual).

El condicional en derivaciones

Tengo la terna:
{ P } ← **precondición dada**
S ← **el programa que tengo que encontrar**
{ Q } ← **postcondición dada**

¿Puede ser que S sea un condicional?
Supongamos que ya sabemos que S no es skip, y vamos a probar con una asignación, planteando las incógnitas.
Si al derivar la asignación, surge la necesidad de un análisis por casos, entonces lo que estoy derivando en realidad es un **if** cuyas ramas son asignaciones distintas.

Ejemplo: Máximo entre dos números, suponiendo que no puedo usar el operador max en mi programa.

Var x, y, res : Int;
{ P: x = X ∧ y = Y } (X e Y variables de especificación)
S
{ Q: res = X max Y }

S claramente no es skip, así que voy a probar con una asignación. Planteo la asignación:
res, x, y := E, F, G
¿Hace falta realmente? No, la postcondición sólo habla de res, así que sólo debo preocuparme por asignar a res:
res := E (E incógnita)

Ahora, derivemos: O sea, busquemos E tal que vale la terna, o sea, que vale
P ⇒ wp.S.Q.
Suponemos P como hipótesis: x = X ∧ y = Y
Vemos la wp:

wp.(res := E).(res = X max Y)
≡ { def. wp para la asignación }
E = X max Y
≡ { ¿puedo elegir X max Y? no, porque X e Y no existen en el programa.
Resolvemos eso usando la hipótesis: x = X ∧ y = Y }
E = x max y
≡ { ¿puedo elegir x max y? no, porque según el enunciado no puedo usar "max"
puedo resolver esto haciendo un **análisis por casos** para resolver el max.
}

- Caso 1: **x ≥ y**: (1er guarda del if)
  ≡ { propiedad de max, si x ≥ y, x max y = x }
  E = x
  ≡ { **elijo que E sea "x"** }

True

- Caso 2: **x ≤ y** (también sirve x < y): (2da guarda del if)
  ≡ { prop. max }
  E = y
  ≡ { **elijo que E sea "y"** }
  True

Acabo de intentar derivar una asignación, pero llegué a un análisis por casos, con dos elecciones distintas para E. Esto se traduce a un if, y el programa (con su terna) queda de la siguiente manera:

Var x, y, res : Int;
{ P: x = X ∧ y = Y } (X e Y variables de especificación)
if x ≥ y → res := x
[] x ≤ y → res := y
fi
{ Q: res = X max Y }

Observaciones:

- Las guardas de los casos deben ser expresiones válidas en lenguaje de programación.
- Los casos deben cubrir todas las posibilidades (siempre suponiendo la hipótesis P).

Otro ejemplo:
Sacado del problema de contar cuántos números divisibles por 6 tenemos entre 0 y un número N dado.

Const N : Int ;
Var numero_actual, resultado : Int ;

{ P: resultado = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩ ∧ numero_actual ≥ 0 }
resultado, numero_actual := E, numero_actual + 1
{ Q: resultado = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩ }

Derivemos: Encontremos E tal que vale esta terna, o sea que vale P ⇒ wp.S.Q.
Supongamos P:
resultado = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩ ∧ numero_actual ≥ 0
Y veamos la wp:

wp.(resultado, numero_actual := **E**, numero_actual + 1).Q
≡ { def wp. para := }
E = ⟨ N i : 0 ≤ i < **numero_actual + 1** : i mod 6 = 0 ⟩
≡ { ¿puedo elegir E = ⟨ N i : 0 ≤ i < numero_actual +1 : i mod 6 = 0 ⟩ ?
no, no es programable.
lógica y partición de rango de conteo,
podemos ya que el rango es no vacío, sabiendo que **numero_actual ≥ 0**.
}

E = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩+
⟨ N i : i = numero_actual : i mod 6 = 0 ⟩
≡ { hipótesis: esta parte ya esta calculada en la variable resultado }
E = resultado + ⟨ N i : i = numero_actual : i mod 6 = 0 ⟩
≡ { rango unitario para el conteo }
E = resultado + (numero_actual mod 6 = 0 → 1
[] numero_actual mod 6 ≠ 0 → 0
)
≡ { ¿puedo elegir E = resultado + (numero_actual mod 6 = 0 → 1
[] numero_actual mod 6 ≠ 0 → 0
) ? No puedo, **porque no es una expresión válida en mi lenguaje de programación el análisis por casos**.
para deshacerme de la expresión con análisis por casos, hago en análisis por casos en la derivación }

- Caso 1: **numero_actual mod 6 = 0** (es programable)
  ≡ { resuelvo la expresión por casos }
  E = resultado + 1
  ≡ { elijo **E = resultado + 1** }
  True
- Caso 2: **numero_actual mod 6 ≠ 0** (es programable)
  ≡ { resuelvo la expresión por casos }
  E = resultado + 0
  ≡ { arit. }
  E = resultado
  ≡ { elijo **E = resultado** }
  True

Luego, acabamos de derivar el siguiente programa (con su terna):

{ P: resultado = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩ ∧ numero_actual ≥ 0 }
if numero_actual mod 6 = 0 →
resultado, numero_actual := resultado + 1, numero_actual + 1
[] numero_actual mod 6 ≠ 0 →
resultado, numero_actual := resultado, numero_actual + 1
fi
{ Q: resultado = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩ }

O, equivalentemente,

{ P: resultado = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩ ∧ numero_actual ≥ 0 }
if numero_actual mod 6 = 0 →
resultado, numero_actual := resultado + 1, numero_actual + 1
[] numero_actual mod 6 ≠ 0 →
numero_actual := numero_actual + 1
fi
{ Q: resultado = ⟨ N i : 0 ≤ i < numero_actual : i mod 6 = 0 ⟩ }

Observación: Al derivar, encontré que la incógnita E es la misma variable que estaba asignando, esto es como no hacer nada así que se puede omitir esa asignación.

Observación: Este es sólo un escenario posible en el que pueden aparecer IF en derivaciones. En otras situaciones, tendremos que proponer IF's usando más creatividad.

Secuenciación (;)

Verificación con Terna de Hoare:
Tengo dos programas S y T secuenciados:

$\{ P \}\ S\ ;\ T\ \{ Q \}$

La correctitud de esta terna, va a depender de la correctitud de ternas planteadas sobre los dos programas: Si yo encuentro un predicado R tal que valen estas dos ternas:

$\{ P \}\ \ S\ \ \{ R \}\ \wedge$
$\{ R \}\ \ T\ \ \{ Q \}$

$\{ P \}$
S ;
T
$\{ Q \}$
≡
Existe un predicado R tal que:
$\{ P \}\ \ S\ \ \{ R \}\ \ \wedge$
$\{ R \}\ \ T\ \ \{ Q \}$

(R hace de postcondición para la primer terna, y de precondición para la segunda)
(R es un predicado intermedio que habla de los estados posibles luego de ejecutar S y antes de ejecutar T.)

Precondición más débil:

$$wp.(S\ ;\ T).Q \equiv wp.S.(wp.T.Q)0$$

Ejemplo (práctico 6, ej 1e):

{ P: ??? }
a, x := x, y ; // S ;
y := a // T
{ Q: x = B ∧ y = A}

Cuál es la wp.(a, x := x, y ; y := a).Q ???? Veamos:

wp.(a, x := x, y ; y := a).Q
≡ { def. wp para la secuenciación: wp.(S ; T).Q ≡ wp.S.(wp.T.Q) }
wp.(a, x := x, y).(wp.(y := a).Q)

≡ { primero aplicamos def. de wp para := en la wp de adentro }
  wp.(a, x := x, y).(x = B ∧ a = A)
≡ { def. wp para := de nuevo }
  y = B ∧ x = A

Esta es la wp, así que la terna queda:

  { P: y = B ∧ x = A }
    a, x := x, y ;
    y := a
  { Q: x = B ∧ y = A}

¿Qué pasa si yo quiero demostrar esta Terna usando la "**Verificación con Terna de Hoare".** Debo encontrar un predicado R tal que valgan estas dos ternas:

  { P: y = B ∧ x = A }
    a, x := x, y ;
  { R }
y
  { R } // si pongo wp.(y :=a).Q me aseguro que vale esta terna
    y := a
  { Q: x = B ∧ y = A }

¿Qué puede ser R? Si yo hago que R sea wp.(y := a).(x = B ∧ y = A), ya me aseguro la 2da terna por definición **de wp**. Sólo me quedaría demostrar la primer terna:

  { P: y = B ∧ x = A }
    a, x := x, y ;
  { R: wp.(y:=a).Q }

Y si nos fijamos bien, demostrar esta terna es demostrar:
    P ⇒ wp.(a, x := x, y).R
o sea
    P ⇒ wp.(a, x := x, y).(wp.(y:=a).Q)

Entonces, se justifica que esta dos wp anidadas sean de hecho la wp de la secuenciación.

La secuenciación en derivaciones

Tengo la terna:
  { P } ← precondición dada
    S ← el programa que tengo que encontrar
  { Q } ← postcondición dada

¿Puede ser que S sea una secuenciación?

Muchas veces lo vamos a ver por creatividad.

Ejemplo: Promedio de un arreglo de N elementos.

{ P: N > 0 }
  S
{ Q: promedio = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ / N }

¿Cómo sería el programa que resuelve este problema?
Primero hay que sumar todos los elementos del arreglo, **después** dividir por N. Esto no es otra cosa que una secuenciación de dos programas, con un predicado intermedio:

Const N : Int, A : array[0,N) of Float;
Var promedio, suma : Float;
{ P: N > 0 }
  S1 ; // acá calculo la suma
{ R: suma = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ }
  S2
{ Q: promedio = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ / N }

Acabo de refinar mi programa S, convirtiéndolo en una secuenciación, y ahora tengo que hacer dos derivaciones: la de S1 (con su terna), y la de S2 (con su terna).
S1 va a ser un ciclo (ya lo hicimos), ya veremos cómo se deriva/demuestra.
S2 va a ser el siguiente programa: promedio := suma / N **(ejercicio: derivar esto!)**

Observación: Parecido a modularización!! Primero identificamos un problema accesorio (la suma), necesario para resolver el problema principal (el promedio).

Ciclo / Repetición (do)

Dado un número entero N > 0, queremos contar cuántos **números entre 0 y N** son múltiplos de 6.

Programa:

```
Const N : Int ;
Var numero_actual, resultado : Int ;
{ N ≥ 0 }
numero_actual , resultado := 0 , 0 ;
{ P: numero_actual = 0 ∧ resultado = 0 }   <-- precondición del do
do numero_actual ≤ N →
    if  numero_actual mod 6 = 0  →
        numero_actual, resultado := numero_actual + 1, resultado + 1 ;
    □ numero_actual mod 6 ≠ 0 →
        numero_actual := numero_actual + 1
    fi
od
{ Q: resultado = ⟨ N  i  : 0 ≤ i ≤ N  : i mod 6 = 0 ⟩  }   <-- postcondición del do
```

Tenemos entonces una terna de la forma:

{ P }
do B → // guarda
S // cuerpo del ciclo
od
{ Q }

donde S es el cuerpo del ciclo (un if en este caso).

¿Cómo demostramos esta terna? Tenemos que lograr conectar lógicamente la precondición con la postcondición. Sin embargo, no sabemos cuántas veces vamos a iterar el ciclo. Supongamos que iteramos n veces. Ejecutar el ciclo es equivalente a ejecutar la siguiente secuenciación:

S ; S ; S ; S ; …. ; S (n veces)

O sea que queremos demostrar una terna:

{ P }
S ; S ; S ; S ; …. ; S (n veces)
{ Q }

Recordando la terna de la secuenciación, ¿qué necesito para poder demostrar esto?
Necesitamos **predicados intermedios** que valen entre todas las secuenciaciones.
Nos conviene tener un único predicado intermedio para facilitar las pruebas. Llamemos por ahora R a ese predicado:

{ P }
skip
{ R }
S ;
{ R }
S ;
{ R }
…. ;
S (n veces)
{ R }
skip
{ Q }

Algo importante que nos faltó considerar acá es la guarda. ¿En qué momentos vale y en qué momentos no vale?

{ P }
skip
{ R ∧ B } // si voy a ejecutar S, es que la guarda dio True
S ;
{ R ∧ B } // aca tambien
S ;
{ R ∧ B } // aca tambien
…. ;
S (n veces)
{ R ∧ ¬ B } // acá la guarda no vale, el ciclo termina
skip

{ Q }

Al plantearlo de esta manera, me sirve ya que las demostraciones que tengo que hacer se reducen a las siguientes:

{ P } **skip** { R } // en este estado final puede valer B ó ¬B
{ R ∧ B } S { R } // en este estado final puede valer B ó ¬B
{ R ∧ ¬ B } **skip** { Q }

El predicado R se llama "**invariante de ciclo"** así que por lo general vamos a denotarlo "I" o "INV":
{ P } **skip** { INV } // "el invariante vale al **principio** del ciclo"
{ INV ∧ B } S { INV } // "el cuerpo del ciclo preserva el invariante"
{ INV ∧ ¬ B } **skip** { Q } // "el invariante implica la **postcondición** al terminar el ciclo"

¿Porqué se llama "invariante"? Porque es un predicado que se preserva a lo largo de todo el ciclo, y eso me permite conectar lógicamente la precondición con la postcondición.

Reescribamos las ternas skip usando su definición:
i) P ⇒ INV // "el invariante vale al **principio** del ciclo"
ii) { INV ∧ B } S { INV } // "el cuerpo del ciclo preserva el invariante"
iii) INV ∧ ¬ B ⇒ Q // "el invariante implica la **postcondición** al terminar el ciclo"

Me falta además garantizar la terminación del ciclo:
iv) "el ciclo termina" (ya lo veremos más adelante)

Resumiendo:

Verificación con Terna de Hoare:

{ P }
do B →
  S
od
{ Q }

≡

Existe invariante INV tal que:
i) P ⇒ INV
ii) { INV ∧ B } S { INV }
iii) INV ∧ ¬ B ⇒ Q
iv) “el ciclo termina”

Precondición más débil: No la vemos.

Intuición del invariante: El invariante expresa el “resultado intermedio” o “resultado parcial” que se está calculando en el ciclo, es decir, la parte del problema que llevamos resuelta hasta ahora. Al terminar el ciclo, el “resultado intermedio” se convierte en el “resultado final” (gracias al requisito iii).

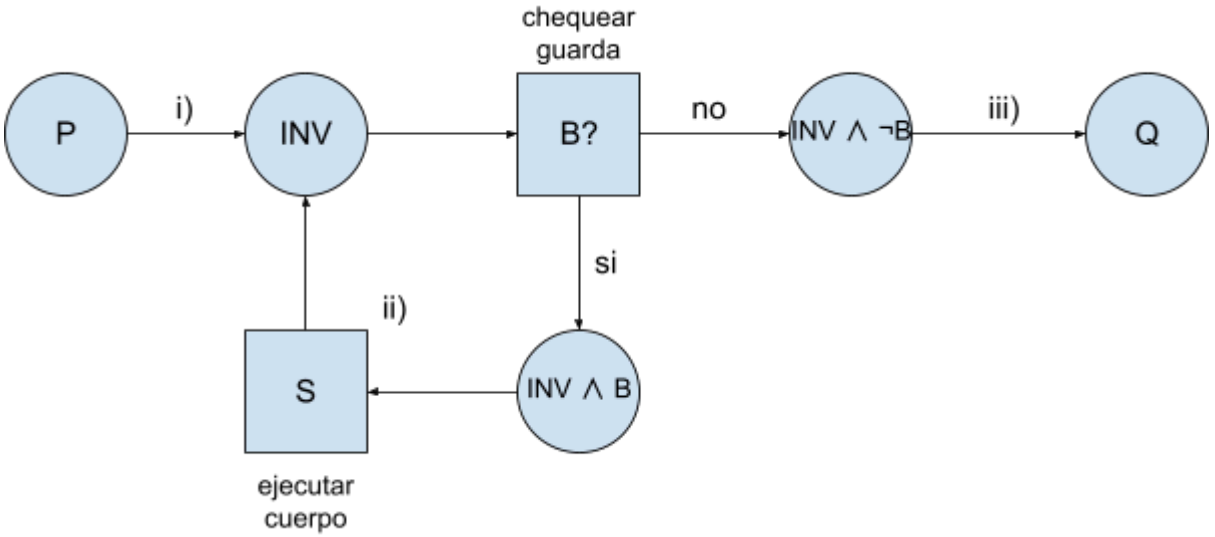
Figura: Este gráfico ilustra la ejecución completa de un ciclo.

{ P }
do B →
  { INV ∧ B }
  S
  { INV }
od
{ Q }

i) P ⇒ INV

ii) { INV ∧ B } S { INV }

iii) INV ∧ ¬ B ⇒ Q



Ejemplo: Dado un número entero N > 0, queremos contar cuántos **números entre 0 y N** son múltiplos de 6.

Programa:

```
Const N : Int ;
Var numero_actual, resultado : Int ;
{ N ≥ 0 }
numero_actual , resultado := 0 , 0 ;
{ P: numero_actual = 0 ∧ resultado = 0 }   <-- precondición del do
  skip ;   // por la condición i) "el invariante vale al principio"
{ INV }
do numero_actual ≤ N →
    {  INV   ∧    (numero_actual ≤ N)    }
    { resultado = ???? }
    if  numero_actual mod 6 = 0  →
        numero_actual, resultado := numero_actual + 1, resultado + 1 ;
    □ numero_actual mod 6 ≠ 0 →
        numero_actual := numero_actual + 1
    fi
    { INV }  // por la condición ii) "el invariante se preserva en el cuerpo"
od
{ INV ∧ ¬ B}
  skip    // por la condición iii) "el invariante implica la post al terminar"
{ Q: resultado = ⟨ N  i  : 0 ≤ i ≤ N  : i mod 6 = 0 ⟩  }
// postcondición del do (y también de todo el programa)
```

B ≡ numero_actual ≤ N
P ≡ numero_actual = 0 ∧ resultado = 0
Q ≡ resultado =⟨ N i : 0 ≤ i ≤ N : i mod 6 = 0 ⟩

Debo encontrar el invariante INV tal que vale lo siguiente:
i) $P \Rightarrow INV$ (“el invariante vale al principio”)
ii) $\{ INV \wedge B \} S \{ INV \}$ (“el cuerpo del ciclo preserva el invariante”)
iii) $INV \wedge \neg B \Rightarrow Q$ (“el invariante garantiza la postcondición al terminar el ciclo”)

¿Qué puede ser INV?

- Puede ser True?
  - Vale i)? Sí.
  - Vale ii)? Sí (siempre que S termine)
  - **Vale iii)? $True \wedge \neg B \Rightarrow Q$?**
    No! Falta muchísima información para poder garantizar Q.

Necesito sí o sí que el INV me diga algo sobre la variable resultado para que **valga iii).**

Apartado: Apreciaciones generales sobre los requisitos para el invariante: Tanto los requisitos i) y ii) son fáciles de satisfacer con un INV débil (incluso siempre valen con True):

i) $P \Rightarrow INV$ (a más débil INV, más fácil de que valga)
ii) $\{ INV \wedge B \} S \{ INV \}$ (lo mismo)

Pero el **requisito iii) me exige cierta fortaleza en el invariante** (un invariante **informativo**)

iii) $INV \wedge \neg B \Rightarrow Q$

(al aparecer del lado izquierdo del $\Rightarrow$ debe tener la fuerza suficiente para implicar a la postcondición Q)

El invariante sí o sí me debe decir algo sobre el resultado.
¿Qué podemos decir?

- **<u>resultado ≥ 0</u>** vale, pero no alcanza para implicar Q
- **<u>resultado ≤ N+1</u>** vale pero tampoco alcanza.
- resultado = **<u>“el resultado intermedio que tenemos calculado hasta el momento”</u>**
  = “hasta ahora sabemos la cantidad de divisibles por 6 entre 0 y numero_actual - 1 (en esta iteracion veremos qué pasa con numero_actual)”
  $= \langle N\ i : 0 \leq i \leq numero\_actual - 1 : i \bmod 6 = 0 \rangle$

Probemos con este invariante:

$INV \equiv resultado = \langle N\ i : 0 \leq i \leq numero\_actual - 1 : i \bmod 6 = 0 \rangle$

Este invariante me representa el “resultado parcial” que yo estoy calculando.
<u>Pregunta:</u> ¿Cuándo este “resultado parcial” se convierte en un “resultado final” (caracterizado Q)? Cuando el ciclo termina, o sea, cuando la guarda se hace falsa (iii):

¿Vale?:

$INV \wedge \neg B \Rightarrow Q$

Veamos:

resultado = ⟨ N i : 0 ≤ i ≤ **numero_actual - 1**: i mod 6 = 0 ⟩ // INV
∧ numero_actual > N // ¬ B
⇒ resultado =⟨ N i : 0 ≤ i ≤ N : i mod 6 = 0 ⟩ // Q

¿me alcanzan las hipótesis para garantizar Q? Tendría que pasar que
numero_actual - 1 = N, o sea que
numero_actual = N + 1. ¿sabemos eso? casi, sabemos numero_actual > N, no vendría bien saber también que **numero_actual ≤ N + 1**.

¿Podremos agregalo al invariante? Nuevo invariante:
INV ≡ resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩
∧ numero_actual ≤ N + 1

Ahora sí vamos a tener que INV ∧ ¬ B ⇒ Q, ya que:
resultado = ⟨ N i : 0 ≤ i ≤ **numero_actual - 1**: i mod 6 = 0 ⟩ // INV
∧ numero_actual ≤ N + 1 // INV
∧ numero_actual > N // ¬ B
⇒ resultado =⟨ N i : 0 ≤ i ≤ N : i mod 6 = 0 ⟩ // Q

Fijarse que por 3ro excluido, numero_actual debe ser sí o sí N + 1.
Luego hemos encontrado un INV tal que vale el requisito iii).
Falta demostrar los requisitos i) y ii):

i) P ⇒ INV
o sea
numero_actual = 0 ∧ resultado = 0
⇒
resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩
∧ numero_actual ≤ N + 1

¿vale o no vale? Sí vale.

ii) { INV ∧ B } S { INV }

o sea:

{ resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩
∧ numero_actual ≤ N + 1
∧ numero_actual ≤ N
}

```
if  numero_actual mod 6 = 0  →
    numero_actual, resultado := numero_actual + 1, resultado + 1 ;
 numero_actual mod 6 ≠ 0 →
    numero_actual := numero_actual + 1
fi
```

{ resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩
∧ numero_actual ≤ N + 1
}

¿Vale esta terna? Casi vale, en realidad al intentar demostrarla vamos a ver que necesitamos agregar al invariante una cosita más: “0 ≤ numero_actual”:

$$\text{INV} \equiv \text{resultado} = \langle N\ i : 0 \le i \le \text{numero\_actual} - 1 : i \bmod 6 = 0 \rangle \\ \wedge\ 0 \le \text{numero\_actual} \le N + 1$$

(a esta demostración/derivación ya la hicimos la clase pasada al ver el if).

Conclusión: Este invariante me caracteriza el “resultado parcial” ya que me dice que la variable **resultado** guarda el conteo de los múltiplos de 6 entre 0 y numero_actual - 1. Además me dice que numero_actual está entre 0 y N+1.

- Al inicio, numero_actual está en 0. (y resultado = 0, por lo que vale INV).
- En el cuerpo del ciclo, numero_actual se incrementa en 1.
- Al terminar el ciclo, numero_actual vale N+1.
  Entonces, el INV me dice que la var. resultado guarda el conteo desde 0 y N, o sea que vale la postcondición.

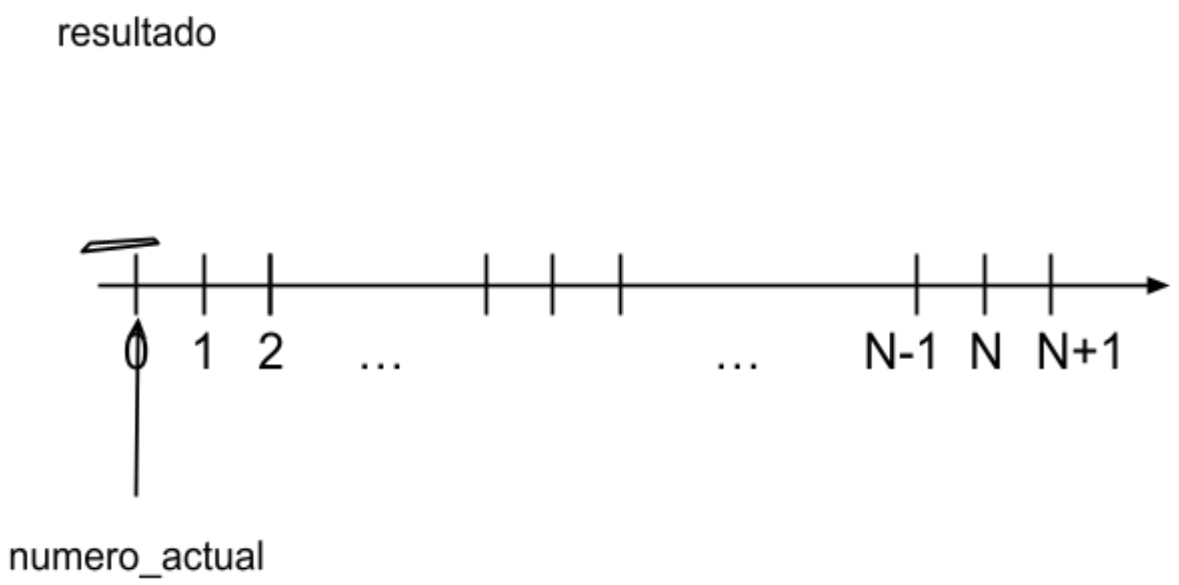
==========

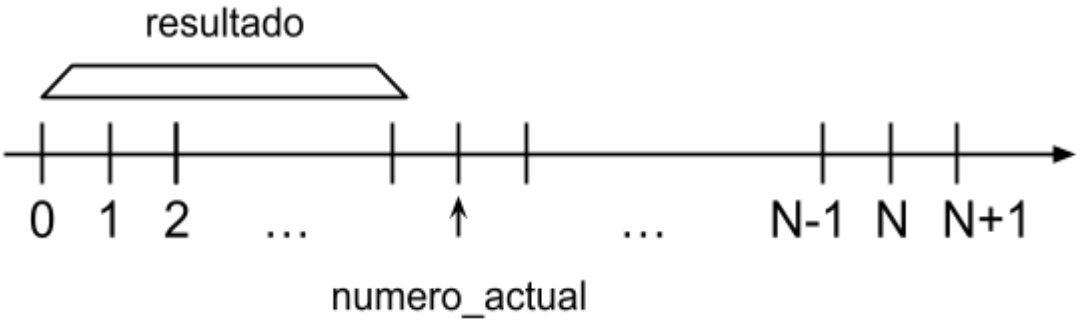
¿Qué puede ser INV? Proponemos:

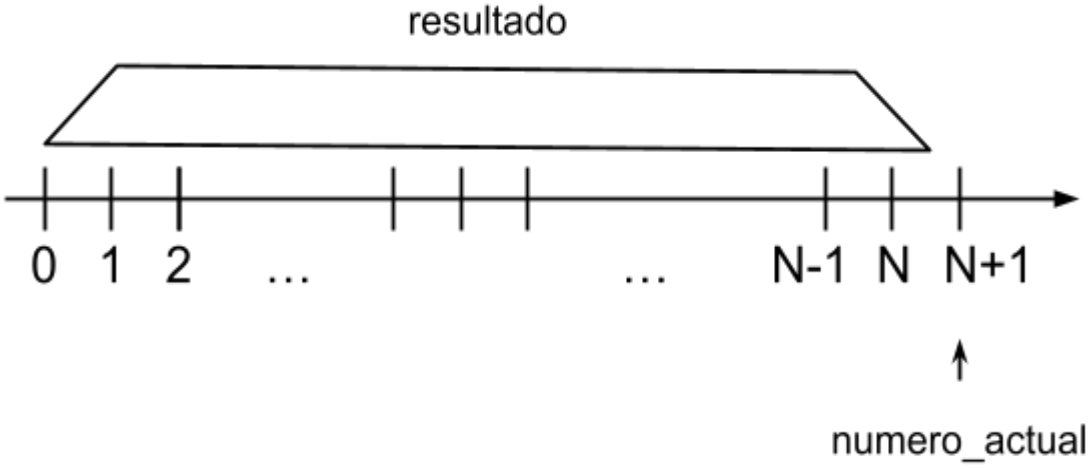
$$\text{INV} \equiv \text{resultado} = \langle N\ i : 0 \le i \le \text{numero\_actual} - 1 : i \bmod 6 = 0 \rangle \\ \wedge\ 0 \le \text{numero\_actual} \le N + 1$$

Este invariante me caracteriza el “resultado parcial” ya que me dice que la variable **resultado** guarda el conteo de los múltiplos de 6 entre 0 y numero_actual - 1. Además me dice que numero_actual está entre 0 y N+1.

- Al inicio, numero_actual está en 0. (y resultado = 0, por lo que vale INV).
- En el cuerpo del ciclo, numero_actual se incrementa en 1.
- Al terminar el ciclo, numero_actual vale N+1.
  Entonces, el INV me dice que la var. resultado guarda el conteo desde 0 y N, o sea que vale la postcondición.


resultado
0 1 2 … … N-1 N N+1
numero_actual


resultado
0 1 2 … … N-1 N N+1
numero_actual


resultado
0 1 2 … … N-1 N N+1
numero_actual

==========

Demostración:

B ≡ numero_actual ≤ N
P ≡ numero_actual = 0 ∧ resultado = 0
Q ≡ resultado =⟨ N i : 0 ≤ i ≤ N : i mod 6 = 0 ⟩

INV ≡ resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩
∧ 0 ≤ numero_actual ≤ N + 1

i) El invariante vale al principio: P ⇒ INV

Suponemos P (numero_actual = 0 ∧ resultado = 0) como hipótesis y veamos INV:
INV
≡ { def. INV }
resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩ ∧ 0 ≤ numero_actual ≤ N + 1
≡ { hip }
0 = ⟨ N i : 0 ≤ i ≤ 0 - 1: i mod 6 = 0 ⟩ ∧ 0 ≤ 0 ≤ N + 1
≡ { rango vacío del conteo }
0 = 0 ∧ 0 ≤ 0 ≤ N + 1
≡ { lógicas varias }
0 ≤ N + 1
≡ { es True porque N ≥ 0 en todo el programa, es una **hipótesis global** }
(**hipótesis global:** todo lo que se diga en la precondición del programa entero sobre las constantes podemos asumir que vale siempre).
True

ii) { INV ∧ B } S { INV }
donde S es el cuerpo del ciclo. O sea:

```
{ INV ∧ B }
  if  numero_actual mod 6 = 0  →
      numero_actual, resultado := numero_actual + 1, resultado + 1 ;
  □ numero_actual mod 6 ≠ 0 →
      numero_actual := numero_actual + 1
  fi
{ INV }
```

¿Cómo demostramos esto?
Vemos en el digesto para el if dos opciones: Usar directo la terna de hoare o traducir a wp. Es casi lo mismo (mejor usar directo la terna).

Observación: A esto ya lo hicimos en una clase anterior al derivar una terna para la asignación

```
numero_actual, resultado := numero_actual + 1, E ;
```

y llegamos a un análisis por casos igual al que tenemos ahora.

Les queda como ejercicio.

iii) INV ∧ ¬ B ⇒ Q

Suponemos INV ∧ ¬ B como hipótesis:
¬ B ≡ numero_actual > N
INV ≡ resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩
∧ 0 ≤ numero_actual ≤ N + 1

Si supongo estas hipotesis, entonces yo se que:
1. numero_actual > N ≡ True
2. resultado = ⟨ N i : 0 ≤ i ≤ numero_actual - 1: i mod 6 = 0 ⟩ ≡ True
3. 0 ≤ numero_actual ≤ N + 1 ≡ True

Veamos Q:
Q
≡ {def. Q }
resultado =⟨ N i : 0 ≤ i ≤ N : i mod 6 = 0 ⟩
≡ { por hip, sabemos que numero_actual > N y que numero_actual ≤ N + 1, luego
numero_actual = N+1, luego N = numero_actual - 1 }
resultado =⟨ N i : 0 ≤ i ≤ numero_actual - 1 : i mod 6 = 0 ⟩
≡ { por hip en INV }
True

Observación: Fijarse que acá usamos todas las hip. salvo 0 ≤ numero_actual (esta hace falta para demostrar ii).

Ejemplo: Suma de los elementos de un arreglo.

```
Const N : Int, A : Array[0, N) of Int;
Var pos, res : Int;
{ P: N ≥ 0 }
res, pos := 0, 0 ;
{ R: res = 0 ∧ pos = 0 }
do pos < N →
    { hasta acá ya sumé todas las posiciones desde 0 hasta pos no inclusive }
    res, pos := res + A.pos, pos + 1
od
{ Q: res = ⟨∑ i : 0 ≤ i < N : A.i ⟩}
```

Invariante: Recordemos que expresa el “resultado intermedio”:

INV ≡ res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ ∧ 0 ≤ pos ≤ N

Lo sacamos “de la galera”. Ahora vamos a demostrar que es correcto:

i) R ⇒ INV

Suponemos R como hip: res = 0 ∧ pos = 0.
Vemos INV:
INV
≡ { ????? (ejercicio) }
True

ii) El invariante se preserva en el cuerpo del ciclo:
{ INV ∧ B } S { INV }
donde S es el cuerpo del ciclo: res, pos := res + A.pos, pos + 1

Demostración: Usamos wp: Probamos que INV ∧ B ⇒ wp.S.INV.

Suponemos INV ∧ B como hipótesis:
INV ≡ res = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩
∧ **0 ≤ pos ≤ N**
res = A.0 + …. + A.(pos-1)
B ≡ **pos < N**

Veamos la wp:
wp.(res, pos := res + A.pos, pos + 1).INV
≡ { def. wp para := }
res + A.pos = ⟨ ∑ i : 0 ≤ i < pos + 1 : A.i ⟩ ∧ 0 ≤ pos + 1 ≤ N
/// **A.0 + …. + A.(pos-1) + A.pos**
≡ { lógica y partición de rango }
res + A.pos = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ + ⟨ ∑ i : i = pos : A.i ⟩ ∧ 0 ≤ pos + 1 ≤ N
≡ { rango unitario }
res + A.pos = ⟨ ∑ i : 0 ≤ i < pos : A.i ⟩ + A.pos ∧ 0 ≤ pos + 1 ≤ N
≡ { hip. me dice que esa sumatoria es res }
res + A.pos = res + A.pos ∧ 0 ≤ pos + 1 ≤ N
≡ { reflexividad }
True ∧ 0 ≤ pos + 1 ≤ N
≡ { neutro, y lógica }
0 ≤ pos + 1 ∧ pos + 1 ≤ N
≡ { 0 ≤ pos + 1 vale por hip pos ≥ 0. }
pos + 1 ≤ N
≡ { pos + 1 ≤ N vale por hip B: pos < N }
True

Observación: en este punto usamos todas las hipótesis salvo pos ≤ N (se va a usar en iii).

iii) Al terminar, el invariante garantiza la post: INV ∧ ¬ B ⇒ Q
Suponemos INV ∧ ¬ B como hipótesis y partimos de Q para llegar a True.
(ejercicio.)

La repetición en derivaciones

Tengo la terna:

{ P } ← **precondición dada**
S ← **el programa que tengo que encontrar**
{ Q } ← **postcondición dada**

¿Puede ser que S sea un ciclo, o deba contener uno? Podemos saber que S debe tener un ciclo si vemos que en la postcondición el estado tiene que tener calculada una expresión cuantificada con una cantidad indeterminada de términos que **no estaba** ya previamente calculada en la precondición.

Observación: una reflexión parecida hacíamos en funcional al decidir si necesitamos inducción / recursión.

Ejemplo: Acá sí hace falta un ciclo:
{ P: N ≥ 0 }
S
{ Q: res = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ }

Ejemplo: Acá no, porque en la precondición ya tengo resuelta gran parte de la sumatoria:
{ P: N > 0 ∧ res = ⟨ ∑ i : 0 ≤ i < N-1 : A.i ⟩ } // aca tengo sumado todo menos el A.(N-1)
S
{ Q: res = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ } // aca tengo sumado todo

Acá alcanza con que S sea el siguiente programa:
res := res + A.(N-1)

Otro:
{ P: N > 1 ∧ res = ⟨ ∑ i : 0 ≤ i < N-2 : A.i ⟩ } // aca tengo sumado todo menos el A.(N-2)
S
{ Q: res = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ } // aca tengo sumado todo

S es res := res + A.(N-2) + A.(N-1)

Otro:
{ P: N > 1 ∧ M ≤ N ∧ res = ⟨ ∑ i : 0 ≤ i < N - M : A.i ⟩ }
// aca tengo sumado todo menos A.(N-M) + … + A.(N-1), falta sumar M términos
S
{ Q: res = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ } // aca tengo sumado todo

Sí hace falta un ciclo.

Derivación de ciclos: Técnicas para encontrar invariantes

Idea general

Tengo la terna:

{ P } ← **precondición dada**
S ← **el programa que tengo que encontrar**
{ Q } ← **postcondición dada**

Una vez que nos damos cuenta de que necesitamos un ciclo, debemos proponerlo y proponer su invariante (y la guarda). Para ello usaremos **técnicas para encontrar invariantes**. La técnica que use me va a dar un invariante INV y una guarda B (ya veremos cómo) tal que se garantiza el **requisito iii: INV ∧ ¬ B ⇒ Q**.

Una vez que tengo el invariante y la guarda, tengo que proponer cómo va a quedar mi programa S.
Siempre lo vamos a proponer de la siguiente manera:
S va a ser una secuenciación:

1) una inicialización S1
2) ciclo de la forma: **do** B → S2 **od**.

O sea, **S** sería **S1 ; do B → S2 od** .

O sea tendremos el siguiente programa anotado:

```
{ P }
 S1   ;   // inicialización
{ INV }
 do B →
       { INV ∧ B }
       S2                  // cuerpo del ciclo
       { INV }
 od
{ Q }
```

Ahora tengo planteadas dos ternas de Hoare nuevas (dos subproblemas nuevos a resolver):

1) Inicialización (garantiza requisito i):
   { P }
   S1
   { INV }
2) Cuerpo del ciclo (garantiza requisito ii):
   { INV ∧ B }
   S2
   { INV }

S1 y S2 deben ser derivados usando las técnicas estándar de derivación.

Observación: S2 seguro no va a ser skip porque si no tengo un ciclo que no termina.

1ra técnica: Tomar términos de una conjunción

Ejemplo: Algoritmo de la división: “Dados dos números x e y, con x ≥ 0 e y > 0, calcular cociente y resto de la división entera de x por y.” (No se puede usar div ni mod ni división real)

Const X, Y : Int ;
Var q, r : Int ;
{ P: $X \geq 0 \ \wedge \ Y > 0$ }
S
{ Q: $X = q * Y + r \ \wedge \ 0 \leq r \ \wedge \ r < Y$ }
$Q_1 \ \wedge \ Q_2 \ \wedge \ Q_3$

Como todas las técnicas, esta se deriva del requisito iii:

iii) $INV \ \wedge \ \neg B \ \Rightarrow \ Q$

Ahora si tengo que Q es una conjunción de varias cosas:
$Q_1 \ \wedge \ Q_2 \ \wedge \ Q_3$

Puedo elegir que una parte sea el INV y que otra parte sea ¬ B. En ese caso tendría garantizado el requisito iii).

¿Qué me conviene elegir? Esta elección se hace por “creatividad”.

- Q_1 no puede ser ¬ B ni a palos, va seguro al invariante.
- Si elijo que Q_2 sea ¬ B, me queda B ≡ r < 0. Voy a estar “ciclando” si mi resto es negativo. Raro.
- Si elijo que Q_3 sea ¬ B, me queda B ≡ r ≥ Y. Voy a estar “ciclando” si r ≥ Y.

Capaz! Suena mejor que Q_2. Probemos a ver si anda.

Elegimos:
INV ≡ $Q_1 \wedge Q_2$ ≡ $X = q * Y + r \ \wedge \ 0 \leq r$
B ≡ $\neg Q_3$ ≡ $r \geq Y$

¿Vale el requisito iii)? $INV \ \wedge \ \neg B \ \Rightarrow \ Q$
Obvio que vale, de hecho son equivalentes:

$$(Q_1 \wedge Q_2) \wedge \neg (\neg Q_3) \Rightarrow Q_1 \wedge Q_2 \wedge Q_3$$

Planteamos/refinamos el programa S:

Const X, Y : Int ;
Var q, r : Int ;
{ P: $X \geq 0 \ \wedge \ Y > 0$ }

S_1 ; **// inicialización**
{ INV }
do r ≥ Y →
{ INV ∧ B }
S_2 **// cuerpo del ciclo**
{ INV }
od
{ Q: X = q * Y + r ∧ 0 ≤ r ∧ r < Y }

¿Qué falta para terminar de resolver el problema?

1) Inicialización: Encontrar S_1 tal que vale la terna:

{ P: X ≥ 0 ∧ Y > 0 }
S_1 ; **// incialización**
{ INV: X = q * Y + r ∧ 0 ≤ r }

Skip seguro que no es. Asignación? Puede ser, la planteamos con incógnitas:

q, r := E, F

Derivamos la asignación: Usando wp, tengo que ver que P ⇒ wp.S_1.INV.
Supongamos P y veamos la wp:

wp.S_1.INV
≡ { def. wp para := }
X = E * Y + X ∧ 0 ≤ F
≡ { elijo E = 1, F = X - Y, en ese caso vale la primera parte pero no vale 0 ≤ F
elijo E = 0, F = X }
X = 0 * Y + X ∧ 0 ≤ X
≡ { arit. }
X = X ∧ 0 ≤ X
≡ { lógica e **hip.** }
True

Listo!! S1 es

q, r := 0, X.

2) Cuerpo del ciclo: Encontrar S_2 tal que vale la terna:

{ INV ∧ B : X = q * Y + r ∧ 0 ≤ r ∧ **r ≥ Y** }
S_2 // cuerpo del ciclo
{ INV : X = q * Y + r ∧ 0 ≤ r }

¿Puede S_2 ser skip? **Nunca**.
¿Puede S_2 ser una asignación? Probemos:

q, r := **E** , **F**

¿Podemos conocer de antemano E o F ?
Queremos que el ciclo termine, la guarda dice **r ≥ Y**, necesito que r se achique.
Replanteamos:

q, r := **E** , r - **F**

(ya prevemos que F es Y, pero tratemos de verlo en la derivación)

Derivamos: Suponemos la precondición:

INV ∧ B : $X = q * Y + r \ \wedge \ 0 \leq r \ \wedge \ r \geq Y$

Y vemos la wp:

$wp.S_2.INV$

≡ { def. wp para := }

$X = \underline{\mathbf{E}} * Y + (r - \underline{\mathbf{F}}) \ \wedge \ \underline{0 \leq r - \mathbf{F}}$

≡ { arit. }

$X = \underline{\mathbf{E}} * Y + (r - \underline{\mathbf{F}}) \ \wedge \ \underline{\mathbf{F} \leq r}$

≡ { por la hip. r ≥ Y, pruebo eligiendo **F = Y**}

$X = \underline{\mathbf{E}} * Y + (r - Y) \ \wedge \ Y \leq r$

≡ { uso la hip. }

$X = \underline{\mathbf{E}} * Y + (r - Y)$

≡ { despejo E }

$E = (X - r + Y) \ / \ Y$ (por acá no es porque no se puede usar la division)

≡ { reemplazo X por hip. }

$q * Y + r = \underline{\mathbf{E}} * Y + (r - Y)$

≡ { despejo E de acá. paso 1 }

$q * Y + r - (r - Y) = \underline{\mathbf{E}} * Y$

≡ { despejo E de acá. paso 2 }

$q * Y + Y = \underline{\mathbf{E}} * Y$

≡ { despejo E de acá. paso 3 }

$(q + 1) * Y = \underline{\mathbf{E}} * Y$

≡ { despejo E de acá. paso 4 }

$q + 1 = \underline{\mathbf{E}}$

≡ { elijo **E = q + 1** }

True

Listo!! S_2 es q, r := q + 1 , r - Y

Resultado final:

Const X, Y : Int ;
Var q, r : Int ;
{ P: $X \geq 0 \ \wedge \ Y > 0$ }
q, r := 0 , X ;
do $r \geq Y \rightarrow$
q, r := q + 1 , r - Y
od
{ Q: $X = q * Y + r \ \wedge \ 0 \leq r \ \wedge \ r < Y$ }

Testing: Ejecutar a mano un ejemplo: X = 19, Y = 5. (q = 3, r = 4.)

iteración	sentencia	q	r	guarda (r ≥ Y)
inicialización	q, r := 0 , X	0	19	True

1	q, r := q + 1 , r - Y	1	14	True
2	q, r := q + 1 , r - Y	2	9	True
3	q, r := q + 1 , r - Y	3	4	False

Luego, al dividir 19 por 5, el cociente es 3 y el resto es 4.

Otro ejemplo: Búsqueda lineal (p. 274 del libro)

2da técnica: Reemplazo de constantes por variables

La técnica más usada por lejos.

Ejemplo: Suma de los elementos de un arreglo:

Const N : Int, A : Array[0, N) of Int;
Var res : Int;
{ P: N ≥ 0 }
S
{ Q: res = ⟨∑ i : 0 ≤ i < N : A.i ⟩}

Recordemos que necesitamos un ciclo ya que tenemos que calcular una expresión cuantificada con una cantidad indeterminada de términos:

res = A.0 + A.1 + … + A.(N-1)

Acá nos gustaría tener control de la cantidad de términos calculados. Para ello lo que haremos es crear una nueva variable y reemplazar alguna de las constantes que determina la cantidad de términos por esta variable nueva.

En ejemplo:
Q ≡ res = ⟨∑ i : 0 ≤ i < **N** : A.i ⟩

Probemos reemplazando la constante N por una nueva variable pos. Haciendo este reemplazo, el invariante que proponemos es:
INV ≡ res = ⟨∑ i : 0 ≤ i < N : A.i ⟩

Además, queremos que INV esté bien definido, y para eso necesitamos fortalecerlo con un predicado que restrinja los valores posibles para pos:
INV ≡ res = ⟨∑ i : 0 ≤ i < pos : A.i ⟩ ∧ 0 ≤ pos ≤ N

(este es el invariante que habíamos dado ya)

(Obs: si pos > N, la sumatoria no estaría bien definida porque excede los límites del arreglo)

¡Falta la guarda B! Queremos que sea tal que vale el requisito iii):

INV $\wedge \neg B \Rightarrow Q$

$res = \langle \sum i : 0 \leq i < pos : A.i \rangle \wedge 0 \leq pos \leq N \wedge \neg B \Rightarrow res = \langle \sum i : 0 \leq i < \underline{\mathbf{N}} : A.i \rangle$

1 2 3

Q es casi la hip (1), sólo me falta que pos = N.

Entonces si elijo $\neg B \equiv$ pos = N, ya estaría.

También funciona $\neg B \equiv$ **$\underline{pos \geq N}$**, ya que también tengo la hip. **$pos \leq N$** (y luego pos = N)

Luego sirven ambas guardas:

- $B \equiv pos \neq N$
- **$B \equiv pos < N$**

Elegimos cualquiera. Por ejemplo la segunda: pos < N.

Planteamos/refinamos el programa S:

Const N : Int, A : Array[0, N) of Int;
Var res, pos : Int;
{ P: $N \geq 0$ }
  S_1 ; // inicialización
{ INV }
  do pos < N →
    { INV $\wedge$ B }
    S_2 // cuerpo del ciclo
    { INV }
  od
{ Q: $res = \langle \sum i : 0 \leq i < N: A.i \rangle$}

Falta encontrar S_1 y S_2.

Inicialización: Encontrar S_1 tal que vale la terna:

{ P: $N \geq 0$ }
  S_1 ; // inicialización
{ INV: $res = \langle \sum i : 0 \leq i < pos : A.i \rangle \wedge 0 \leq pos \leq N$ }

Probamos con una asignación: res, pos := E, F.

Derivamos: Supongamos la precondición P: $N \geq 0$, y veamos la wp:

wp.S_1.INV
≡ { def. wp para := }
$E = \langle \sum i : 0 \leq i < F : A.i \rangle \wedge 0 \leq F \leq N$
≡ { elijo **$F = 0$** , **$E = 0$** }
$0 = \langle \sum i : 0 \leq i < 0 : A.i \rangle \wedge 0 \leq 0 \leq N$
≡ { pasos varios … }
True

Cuerpo del ciclo: Encontrar S_2 tal que vale la terna:

{ INV $\wedge$ B : res = $\langle \sum$ i : 0 ≤ i < pos : A.i $\rangle$ $\wedge$ 0 ≤ pos ≤ N $\wedge$ pos < N }
S_2 // cuerpo del ciclo
{ INV : res = $\langle \sum$ i : 0 ≤ i < pos : A.i $\rangle$ $\wedge$ 0 ≤ pos ≤ N }

¿Qué es S_2? Probamos con una asignación:

res, pos := E, F

¿Puedo conocer de antemano alguna de las incógnitas? Está claro que F = pos + 1.
Replanteo:

res, pos := E, pos + 1

Derivación: Queda como ejercicio descubrir que me conviene elegir E = res + A.pos.

Resultado final:

```
Const N : Int, A : Array[0, N) of Int;
Var res, pos : Int;
{ P: N ≥ 0 }
  res, pos := 0, 0 ;  // inicialización
  do pos < N →
        res, pos := res + A.pos, pos + 1  // cuerpo del ciclo
  od
{ Q: res = ⟨∑ i : 0 ≤ i < N : A.i ⟩}
```

Ejercicio: Hacer testing de este programa.

Otro ejemplo: Exponenciación: Dados X > 0, e Y ≥ 0, calcular X^Y.

```
Const X, Y : Int ;
Var res : Int ;
{ P : X > 0 ∧ Y ≥ 0 }
  S
{ Q:  res = X^Y }
```

Acá en la postcondición tengo una multiplicación de una cantidad indeterminada de términos:

res = X * X * … * X Y veces

Claramente necesitamos un ciclo. Cómo aplico reemplazo de constante por variable?
Reemplazamos Y por una variable nueva n.
Planteamos el invariable fortalecido:

INV ≡ res = X^n $\wedge$ 0 ≤ n ≤ Y

$B \equiv n \neq Y$

De esta manera vale el requisito iii): $INV \wedge \neg B \Rightarrow Q$

Planteamos el programa como va quedando:
Const X, Y : Int ;
Var res , n : Int ;
{ P : $X > 0 \wedge Y \geq 0$ }
  S1 ;
  { INV }
  do $n \neq Y \rightarrow$
        { INV $\wedge$ B }
        S2
        { INV }
  od
{ Q: $res = X^Y$ }

Falta derivar S_1 (inicialización) y S_2 (cuerpo del ciclo).

Inicialización:
{ P : $X > 0 \wedge Y \geq 0$ }
  S1 ;
{ INV: $res = X^n \wedge 0 \leq n \leq Y$ }

¿Qué es S1? res, n := 1, 0 (**ejercicio: derivar/demostrar)**

Cuerpo del ciclo:
        { INV $\wedge$ B: $res = X^n \wedge 0 \leq n \leq Y \wedge n \neq Y$ }
        S2
        { INV: $res = X^n \wedge 0 \leq n \leq Y$ }

¿Qué es S2? Se puede derivar partiendo de: res, n := E, n+1
Resultado: res, n := res * X, n+1

Resultado final:
Const X, Y : Int ;
Var res , n : Int ;
{ P : $X > 0 \wedge Y \geq 0$ }
  res, n := 1, 0 ;
  do $n \neq Y \rightarrow$
        res, n := res * X, n + 1
  od
{ Q: $res = X^Y$ }

Ejercicio: Hacer TESTING!!

Ejemplo: Factorial de un número N: Dado N ≥ 0 quiero calcular el factorial de N.

```
Const N : Int;
Var res : Int;
{ P: N ≥ 0 }
  S
{ Q:  res = N!  }
```

¿Nos saldrá este programa a ojo? Queremos calcular:

1 * 2 * 3 * 4 * …. * (N-1) * N

La idea es hacer un ciclo que recorra los números desde 1 hasta el N (en una variable n), e ir multiplicando en res esos números.

```
Const N : Int;
Var res , n : Int;
{ P: N ≥ 0 }
  res , n  :=  1 , 1  ;
  do n ≤ N →
       res , n := res * n , n + 1
  od
{ Q:  res = N!  }
```

La otra forma a ojo, recorriendo al revés:

```
Const N : Int;
Var res , n : Int;
{ P: N ≥ 0 }
  res , n  :=  1 , N  ;
  do n ≥ 1 →
       res , n := res * n , n - 1
  od
{ Q:  res = N!  }
```

Ahora probemos derivando:

Usando la técnica de reemplazo de constante por variable, creamos una nueva variable n : Int, y proponemos:

INV ≡ res = n! ∧ 0 ≤ n ≤ N

B ≡ n ≠ N (también andaría n < N)

Replanteamos el programa:

```
Const N : Int;
Var res : Int;
{ P: N ≥ 0 }
  S1 ;
{ INV }
  do n ≠ N →
     { INV ∧ B }
```

S2
{ INV }
od
{ Q: res = N! }

Inicialización:
{ P: N ≥ 0 }
S1 ;
{ INV: res = n ! ∧ 0 ≤ E ≤ N }

S1 debe ser de la forma: res, n := E , F
(F no puede ser 1 porque no puedo saber 1 ≤ N)
Elijo **F = 0, E = 1** y sale todo bien (ejercicio: verificarlo con la wp)

Cuerpo del ciclo:
{ INV ∧ B: **res = n ! ∧ 0 ≤ n ≤ N ∧ n ≠ N** }
S2
{ INV: res = n ! ∧ 0 ≤ n ≤ N }

S2 debe ser de la forma: res, n := E , n + 1.

Derivemos: Supongamos como hipótesis **INV ∧ B**, y veamos la wp:

wp.(res, n := E , n + 1).INV
≡ { def. wp para := }
E = **(n+1) !** ∧ 0 ≤ n+1 ≤ N
≡ { algebra (prop !) }
E = **n!** * (n + 1) ∧ 0 ≤ n+1 ≤ N
≡ { hip. INV } ← FUNDAMENTAL A LA HORA DE DERIVAR UN CUERPO DE CICLO
E = res * (n + 1) ∧ 0 ≤ n+1 ≤ N
≡ { elijo **E = res * (n+1)** }
res * (n+1) = res * (n + 1) ∧ 0 ≤ n+1 ≤ N
≡ { lógica }
0 ≤ n+1 ≤ N
≡ { 0 ≤ n+1 vale por hip. 0 ≤ n, n+1 ≤ N vale por hip n ≤ N ∧ n ≠ N }
True

Listo! Resultado final:
Const N : Int;
Var res : Int;
{ P: N ≥ 0 }
res, n := 1, 0 ;
do n ≠ N →
res, n := res * (n+1) , n + 1
od
{ Q: res = N! }

Es muy parecido pero no igual al que hicimos a ojo. Ambos andan bien igual.

Ejemplo: De nuevo suma de los elementos de un arreglo:

Const N : Int, A : Array[0, N) of Int;
Var res : Int;
{ P: N ≥ 0 }
  S
{ Q: **res = ⟨∑ i : 0 ≤ i < N : A.i ⟩**}

Esta vez derivamos distinto. Aplicamos cambio de constante por variable de la siguiente manera:

INV ≡ res = ⟨∑ i : n ≤ i < N : A.i ⟩ ∧ 0 ≤ n ≤ N
B ≡ **n ≠ 0**

(acá reemplazamos la constante 0 por la variable nueva n)

Luego vale el requisito iii): INV ∧ ¬ B ⇒ **Q**

Replanteamos el programa:

Const N : Int, A : Array[0, N) of Int;
Var res , n : Int;
{ P: N ≥ 0 }
  S1 ;
{ INV }
  do n ≠ 0 →
      { INV ∧ B }
      S2
      { INV }
  od
{ Q: res = ⟨∑ i : 0 ≤ i < N : A.i ⟩}

Inicialización:

{ P: N ≥ 0 }
  S1 ;
{ INV: res = ⟨∑ i : n ≤ i < N : A.i ⟩ ∧ 0 ≤ n ≤ N }

¿Qué es S1? De la forma: res, n := E , F.

Derivemos: Supongamos P como hip. y veamos la wp:

wp.(res, n := E , F).INV
≡ { def wp para := }
E = ⟨∑ i : F ≤ i < N : A.i ⟩ ∧ 0 ≤ F ≤ N
≡ { **elijo F = N** para forzar un rango vacío }
E = ⟨∑ i : N ≤ i < N : A.i ⟩ ∧ 0 ≤ N ≤ N

≡ { **elijo E = 0** y hago más pasos }
True

Cuerpo del ciclo:

{ INV ∧ B: **res =** ⟨∑ **i : n ≤ i < N : A.i** ⟩ ∧ 0 ≤ n ≤ N ∧ n ≠ N }
S2
{ INV: res = ⟨∑ i : n ≤ i < N : A.i ⟩ ∧ 0 ≤ n ≤ N }

Por intuición sabemos que S2 será de la forma: res, n := E, n - 1
Derivemos: Supongamos como hip. INV ∧ B y veamos la wp:

wp.(res, n := E, n - 1).INV
≡ { def wp para := }
E = ⟨∑ i : n - 1 ≤ i < N : A.i ⟩ ∧ 0 ≤ n-1 ≤ N
≡ { n-1 ≤ N vale ya que n ≤ N, 0 ≤ n-1 vale ya que por hip n ≥ 0 y además n ≠ 0 (o sea n ≥ 1). }
E = ⟨∑ i : n - 1 ≤ i < N : A.i ⟩
≡ { reescribimos rango por lógica }
Ayudita: n = 4, N = 8: tenemos: n - 1 ≤ i < N : i ∈ { **3**, **4, 5, 6, 7** }
queremos: **n ≤ i < N :** i ∈ { 4, 5, 6, 7}
podemos aplicar esto: n - 1 ≤ i < N ≡ i = n-1 ∨ **n ≤ i < N**
E = ⟨∑ i : i = n - 1 ∨ n ≤ i < N : A.i ⟩
≡ { part. rango }
E = ⟨∑ i : n ≤ i < N : A.i ⟩ + ⟨∑ i : i = n - 1 : A.i ⟩
≡ { hip. }
E = res + ⟨∑ i : i = n - 1 : A.i ⟩
≡ { rango unitario }
E = res + A.(n-1)
≡ { elijo **E = res + A.(n-1)** }
True

Listo! Resultado final:

```
Const N : Int, A : Array[0, N) of Int;
Var res , n : Int;
  res, n := 0, N ;
  do n ≠ 0 →
       res, n := res + A.(n-1) , n - 1
  od
{ Q: res = ⟨∑ i : 0 ≤ i < N : A.i ⟩}
```

Este programa recorre el arreglo desde el final hacia adelante.

Ejemplo: Promedio de los elementos de un arreglo:

Const N : Int, A : Array[0, N) of Int;

Var res : Float;
{ P: N > 0 }
  S
{ Q: res = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ / N }

¿Cómo derivamos acá?

Opción 1: Algo que ya hicimos: Plantear una secuenciación de dos programas: primero calcular la suma y después dividir por N:

Const N : Int, A : Array[0, N) of Int;
Var res, sum : Float, Int ;
{ P: N > 0 }
  S1 ; // esto es el programa que ya vimos (el que suma un arreglo)
{ R: sum = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ }
  S2 // esto es una asignación res := sum / N
{ Q: res = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ / N }

Opción 2: No planteamos la secuenciación y vamos derecho al ciclo usando reemplazo de constante por variable.

INV ≡ res = ⟨ ∑ i : 0 ≤ i < n : A.i ⟩ / N ∧ 0 ≤ n ≤ N
B ≡ n ≠ N

Observación importante: Al hacer cambio de constante por variable, sólo se hace el cambio en el punto que me permite controlar la cantidad de términos de mi cuantificación (o sea, el rango). En el ejemplo, la otra mención de la constante N no debe ser reemplazada.

Replanteamos el programa:
Const N : Int, A : Array[0, N) of Int;
Var res : Float;
{ P: N > 0 }
  S1 ; // res, n := 0, 0
{ INV: res = ⟨ ∑ i : 0 ≤ i < n : A.i ⟩ / N ∧ 0 ≤ n ≤ N }
  do n ≠ N →
    { INV ∧ B }
    S2 // res, n := res + A.n / N , n+1
    { INV }
  od
{ Q: res = ⟨ ∑ i : 0 ≤ i < N : A.i ⟩ / N }

Ejercicio: Derivar bien este programa.

Ciclos anidados

Ejemplo

Ejemplo (2025, practico 4, ejercicio 11): Veamos esta especificación:

```
Const N, A : array[0,N) of Int ;
Var res : Bool ;
{ P: N ≥ 0 }
  S
{ Q: res =⟨ ∀ i : 0 ≤ i < N : A.i =  i !   ⟩}
```

¿Qué calcula este programa? “el programa verifica si cada elemento del arreglo es el factorial de su posición” (esto se calcula en la variable booleana r).
··¿Sale a ojo este programa? Necesitamos seguro un ciclo para recorrer los elementos del arreglo. Y una variable “pos” para las posiciones del arreglo.

```
Const N, A : array[0,N) of Int ;
Var res : Bool , fac : Int ;
{ P: N ≥ 0 }
  res ,   pos  :=  True , 0 ;
  do   pos < N →     //  también sirve:   “pos < N  ∧  res” (pero ya lo veremos)
         // acá quiero ver si   A.pos = pos !
           res , pos := res    ∧    (A.pos = pos !) ,   pos + 1   // casi!! esto estaría bien si el
                                                                  // factorial fuera programable.
         // tengo que calcular acá el factorial de pos: pos !
         // usemos una variable nueva fac y planteamos la especificación del problema:
         T ;                              // este mini-programa T debe calcular el factorial de pos
         {  fac = pos ! }                 // postcondición para T
         res , pos  :=  res ∧ (A.pos = fac )   ,   pos + 1
  od
{ Q: res =⟨ ∀ i : 0 ≤ i < N : A.i =  i !   ⟩}
```

Como el factorial no es programable, necesitamos una variable extra fac, y calcular fac = pos ! dentro del ciclo, en un nuevo ciclo:

```
Const N : Int, A : array[0, N) of Int ;
Var res : Bool,
     pos, fac, n : Int ;
{ P: N ≥ 0 }
  res, pos := True , 0 ;
  do   pos < N   →
         // acá calcular fac = pos !
         fac, n := 1, 0 ;
         do n ≠ pos →
              fac,  n := fac * (n+1) , n + 1
```

od ;
{ fac = pos ! }
res, pos := res ∧ (A.pos = fac) , pos + 1
od
{ Q: res =⟨ ∀ i : 0 ≤ i < N : A.i = i ! ⟩ }

Listo el programa a ojo!

A ver qué sale derivando:
Const N, A : array[0,N) of Int ;
Var res : Bool ;
{ P: N ≥ 0 }
S
{ Q: res =⟨ ∀ i : 0 ≤ i < N : A.i = i ! ⟩}

Necesitamos un ciclo sí o sí. Aplicamos cambio de constante por variable. Reemplazamos la constante N por una nueva variable pos:

INV ≡ res =⟨ ∀ i : 0 ≤ i < pos : A.i = i ! ⟩ ∧ 0 ≤ pos ≤ N
B ≡ pos ≠ N

Luego, vale INV ∧ ¬ B ⇒ Q.

Replanteamos/refinamos:
Const N, A : array[0,N) of Int ;
Var res : Bool ;
{ P: N ≥ 0 }
S1 ;
{ INV }
do pos ≠ N →
{ INV ∧ B }
S2
{ INV }
od
{ Q: res =⟨ ∀ i : 0 ≤ i < N : A.i = i ! ⟩}

Inicialización: S1 es res, pos := True, 0. (ejercicio)

Cuerpo del ciclo: Tenemos la siguiente terna:
{ INV ∧ B: res =⟨ ∀ i : 0 ≤ i < pos : A.i = i ! ⟩ ∧ 0 ≤ pos ≤ N ∧ pos ≠ N }
S2
{ INV: res =⟨ ∀ i : 0 ≤ i < pos : A.i = i ! ⟩ ∧ 0 ≤ pos ≤ N }

Probemos con S2 de la forma: res, pos := E, pos + 1
Derivemos: Supongamos INV ∧ B y veamos la wp:

wp.(res, pos := E, pos + 1).INV
≡ { def. wp para := }
E =⟨ ∀ i : 0 ≤ i < pos + 1 : A.i = i ! ⟩ ∧ 0 ≤ pos + 1 ≤ N
≡ { esta parte vale por hip 0 ≤ pos ≤ N ∧ pos ≠ N (ya lo vimos varias veces) }
E =⟨ ∀ i : 0 ≤ i < pos + 1 : A.i = i ! ⟩
≡ { lógica y partición de rango }
E = ⟨ ∀ i : 0 ≤ i < pos : A.i = i ! ⟩∧⟨ ∀ i : i = pos : A.i = i ! ⟩
≡ { hip. }
E = res ∧⟨ ∀ i : i = pos : A.i = i ! ⟩
≡ { rango unitario }
E = res ∧ A.pos = pos !
≡ { si ! fuera programable, acá podría elegir E = res ∧ A.pos = pos ! }

Acá nos trabamos, porque “pos !” no es programable. Nos hubiera venido bien tener una hipótesis adicional que me diga que hay una variable (llamémosla “fac”) en la que está calculado “pos !”.
Una cosa así:
{ INV ∧ B ∧ fac = pos ! }
res , pos := E , pos + 1
{ INV }

Para esto, necesitaría tener un programa antes S3 cuya postcondición es
“INV ∧ B ∧ fac = pos ! “.

Antes teníamos:

{ INV ∧ B }
S2
{ INV }

Ahora lo replanteamos/refinamos de la siguiente manera:

{ INV ∧ B }
S3 ;
{ INV ∧ B ∧ fac = pos ! }
res, pos := E, pos + 1
{ INV }

(acá S2 es “S3 ; res, pos := E, pos + 1”, y no cambiamos ni pre ni post)

Ahora tenemos dos nuevos problemas:

Cierre del cuerpo del ciclo: res, pos := E, pos + 1
(son los mismos pasos que hicimos hasta trabarnos, pero nos destrabamos con la nueva hipótesis y terminamos eligiendo E = res ∧ (A.pos = fac))

Ciclo anidado: S3, con la terna:

{ P_2: INV ∧ B }
S3 ;
{ Q_2: INV ∧ B ∧ fac = pos ! }

¿Qué es S3? Necesito un ciclo para calcular pos ! = 1 * 2 * 3 * …. * (pos - 1) * pos.

Observaciones:

- Acá, INV ∧ B valen al principio y deben valer al final. No debo tocar en S3 ninguna de las variables involucradas en INV ∧ B (pos, res).
- O sea, a los ojos del programa S3, **pos y res pueden ser vistas como constantes.**

Para derivar este nuevo ciclo, aplicamos la técnica de cambio de constante por variable, cambiando la "constante" pos por una nueva variable n:

INV_2 ≡ INV ∧ B ∧ fac = n ! ∧ 0 ≤ n ≤ pos
B_2 ≡ n ≠ pos

Luego, vale el requisito iii): $INV_2 \wedge B_2 \Rightarrow Q_2$

Replanteamos/refinamos S3:

{ P_2: INV ∧ B }
S_4 ;
´{ INV_2 }
do n ≠ pos →
{ INV_2 ∧ B_2 }
S_5
{ INV_2 }
od
{ Q_2: INV ∧ B ∧ fac = pos ! }

Ahora tenemos dos nuevos subproblemas:

Inicialización 2: La terna es:

{ P_2: INV ∧ B }
S_4 ;
´{ INV_2 }

S_4 va a ser de la forma fac, n := E, F
(recordemos: no tocamos ni pos ni res.)

Ya se que **E = 1, y F = 0**. Demostremos en lugar de derivar:
Supongamos la pre P_2 y veamos la wp:

wp.(fac, n := 1, 0).INV_2
≡ { def. wp para := }

INV ∧ B ∧ 1 = 0 ! ∧ 0 ≤ 0 ≤ pos
≡ { INV ∧ B ya valen por hip. P_2 }
1 = 0 ! ∧ 0 ≤ 0 ≤ pos
≡ { 1 = 0 ! por arit. }
0 ≤ 0 ≤ pos
≡ { pos ≥ 0 por hip INV }
True

Cuerpo del ciclo 2: La terna es:

{ INV_2 ∧ B_2 }
S_5
{ INV_2 }

S_5 va a ser de la forma: fac, n := E, n + 1.

Derivemos: Supongamos **INV_2** ∧ B_2 y veamos la wp:

wp.(fac, n := E, n + 1).INV_2
≡ { def. wp para := }
INV ∧ B ∧ E = (n + 1) ! ∧ 0 ≤ n + 1 ≤ pos
≡ { vale por hip INV_2 }
E = (n + 1) ! ∧ 0 ≤ n + 1 ≤ pos
≡ { por hip. 0 ≤ n ≤ pos (parte de INV_2), n ≠ pos (B_2) }
E = (n + 1) !
≡ { prop ! }
E = **n !** * (n + 1)
≡ { hip. INV_2}
E = fac * (n + 1)
≡ { elijo **E = fac * (n + 1)** }
True

Listo el ciclo anidado, luego el cuerpo del ciclo S2 queda así:

{ INV ∧ B }
fac, n := 1, 0 ;
do n ≠ pos →
fac, n := fac * (n+1) , n + 1
od
res, pos := E, pos + 1
{ INV }

Y el programa completo queda así:

Const N, A : array[0,N) of Int ;
Var res : Bool, pos, fac, n : Int ;

```
{ P: N ≥ 0 }
  res, pos  := True, 0 ;
 do pos ≠ N →
       fac, n := 1, 0 ;
       do  n ≠ pos →
           fac, n := fac * (n+1) , n + 1
       od
        res, pos := r ∧ (A.pos = fac), pos + 1
 od
{ Q: res =⟨ ∀ i : 0 ≤ i < N : A.i =  i !   ⟩}
```

Observación: Quedó igual que el que hicimos a ojo!

Otro ejemplo

Ejemplo: Considere la siguiente especificación:

```
Const N : Int, A : array[0, N) of Int ;
Var r : Int ;
{ P: N ≥ 0 }
  S
{ Q: r =⟨ N i, j : 0 ≤ i < j < N : A.i = A.j ⟩}
```

¿Qué calcula este programa? Calcula la cantidad de veces que dos posiciones distintas del arreglo contienen el mismo valor.

Ejemplo con testing: N = 5. A = [1 , 2 , 1 , 1 , 2].
0 1 2 3 4
El resultado va a ser r = ????.
Podemos hacer la lectura operacional:

El rango es : (i , j) ∈ { (0, 1) , **(0, 2)** , **(0, 3)**, (0, 4)
(1, 2) , (1, 3), **(1, 4)**
(2, 3), (2, 4)
(3, 4) }
El resultado va a ser 4.

Derivación: Necesitamos un ciclo. Aplicamos reemplazo de constante por variable.
Reemplazamos la constante N por una nueva variable n:

INV ≡ r =⟨ N i, j : 0 ≤ i < j < n : A.i = A.j ⟩ ∧ 0 ≤ n ≤ N
B ≡ n ≠ N

Replanteamos el ciclo:
Const N : Int, A : array[0, N) of Int ;
Var r, n : Int ;
{ P: N ≥ 0 }
  S1 ;
{ INV }
  do n ≠ N →
      { INV ∧ B }
       S2
      { INV }
   od
{ Q: r =⟨ N i, j : 0 ≤ i < j < N : A.i = A.j ⟩}

Inicialización: S1 es r , n := 0 , 0 (ejercicio)

Cuerpo del ciclo: Probamos que S2 sea la asignación: r, n := E , n + 1.
 Derivemos: Supongamos INV ∧ B, y veamos la wp:

  wp.(r, n := E , n + 1).INV
≡ { def. wp para := }
  E =⟨ N i, j : 0 ≤ i < j < n + 1 : A.i = A.j ⟩ ∧ 0 ≤ n+1 ≤ N
≡ { por hip. (ya sabemos hacerlo) }
  E =⟨ N i, j : 0 ≤ i < j < n + 1 : A.i = A.j ⟩
≡ { lógica:
        0 ≤ i < j < n + 1
es lo mismo que :
        0 ≤ i < j   ∧   j < n + 1
que es lo mismo que :
        0 ≤ i < j   ∧   (j < n ∨ j = n)
y distribuimos:
        (0 ≤ i < j   ∧ j < n) ∨ (0 ≤ i < j   ∧ j = n)
que es lo mismo que:
        (0 ≤ i < j < n) ∨ (0 ≤ i < j   ∧ j = n)
 }
  E =⟨ N i, j : (0 ≤ i < j < n) ∨ (0 ≤ i < j ∧ j = n) : A.i = A.j ⟩
≡ { part. de rango }
  E =⟨ N i, j : 0 ≤ i < j < n   : A.i = A.j ⟩+⟨ N i, j : 0 ≤ i < j ∧ j = n : A.i = A.j ⟩
≡ { hip. }
  E = r +⟨ N i, j : 0 ≤ i < j ∧ j = n : A.i = A.j ⟩
≡ { elim. de variable j }
  E = r +⟨ N i : 0 ≤ i < n : A.i = A.n ⟩

Nos trabamos, no hay forma de elegir E programable, necesito una nueva hip y un ciclo anidado para calcularla. Replanteo S2 de la siguiente manera:
      { INV ∧ B }
       S3 ;
      { INV ∧ B ∧ r2 =⟨ N i : 0 ≤ i < n : A.i = A.n ⟩ }

r , n := r + r2 , n + 1
{ INV }

(r2 cuenta cuantos elementos anteriores son iguales a A.n)

Falta derivar el ciclo anidado S3:

{ P_2: INV $\wedge$ B }
S3 ;
{ Q_2: INV $\wedge$ B $\wedge$ r2 =⟨ N i : 0 ≤ i < n : A.i = A.n ⟩ }

Para S3, n y r son constantes. Esto es un ciclo, aplicamos cambio de constante por variable, reemplazando la "constante" n por la variable m:

$INV_2 \equiv$ INV $\wedge$ B $\wedge$ r2 =⟨ N i : 0 ≤ i < m : A.i = A.n ⟩ $\wedge$ 0 ≤ m ≤ n
$B_2 \equiv m \neq n$

Por supuesto vale el requisito iii).

Observación: solo hacer el reemplazo en el rango!!!!

Replanteamos/refinamos S3:

{ P_2 }
S4 ;
{ INV_2 }
do m ≠ n →
{ $INV_2 \wedge B_2$}
S5
{ INV_2 }
{ Q_2 }

Inicialización del ciclo anidado: S4 es r2, m := 0 , 0 (ejercicio: demostrar!!)

Cuerpo del ciclo anidado: S5 es de la forma: r2, m := E , m + 1.

Derivemos:
Al derivar me voy a encontrar con un análisis por casos:

- Si A.m = A.n: Debo elegir E = r2 + 1
- Si A.m ≠ A.n: Debo elegir E = r2.

Con esto estaría todo, y el programa final final quedaría:

```
Const N : Int, A : array[0, N) of Int ;
Var r, n , r2, m : Int ;
{ P: N ≥ 0 }
 r, n := 0 , 0 ;
 do n ≠ N →
     r2, m := 0 , 0;
     do m ≠ n →
         if A.m = A.n → r2, m := r2 + 1 , m + 1
         [] A.m ≠ A.n →m := m + 1
```

```
        fi
      od
      r , n :=  r + r2 , n + 1
  od
{ Q: r =⟨ N i, j : 0 ≤ i < j < N : A.i = A.j ⟩}
```

Ejercicio: Testear con el arreglo de ejemplo que vimos.

N = 5. A = [1 , 2 , 1 , 1 , 2].
0 1 2 3 4

Testeamos centrandonos en el if. Cada fila corresponde a una ejecución del if:

vez q paso por el if	n (va de 0 hasta N-1)	m (va de 0 hasta n-1)	r	r2	A.n = A.m?
1	1	0	0	0	False
2	2	0	0	0	True
3	2	1	0	1	False
4	3	0	1	0	
5	3	1			
6	3	2			
7	4	0			
8	…				

Ejercicio: completar!

Fortalecimiento de invariantes

Ejemplo

Ejemplo (2025, practico 4, ejercicio 11): Dado un arreglo a : array[0, N) of Num con N ≥ 0 determinar si sus elementos son iguales al factorial de la posición.

Const N, A : array[0,N) of Int ;
Var r : Bool ;
{ P: N ≥ 0 }
  S
{ Q: r =⟨ ∀ i : 0 ≤ i < N : A.i = i ! ⟩}

Recordemos el algoritmo con ciclos anidados:

Const N, A : array[0,N) of Int ;
Var r : Bool, pos, fac, n : Int ;
{ P: N ≥ 0 }
  r, pos := True, 0 ;
  do pos ≠ N →

```
        fac, n := 1, 0 ;
        do  n ≠ pos →
            fac, n := fac * (n+1) , n + 1
        od ;
         r, pos := r ∧ (A.pos = fac), pos + 1
  od
{ Q: r =⟨ ∀ i : 0 ≤ i < N : A.i =  i !  ⟩}
```

Ejemplo: Tengo un arreglo de 1000 elementos. En la ejecución, debo chequear que

A.998 = 998! = 1 * 2 * 3 … * 998

y también debo chequear en la iteración siguiente:

A.999 = 999! = 1 * 2 * 3 … * 998 * 999

¿Podemos hacer esto más eficiente? Sí, podemos recordar el factorial que habíamos calculado en la iteración anterior y sólo actualizarlo con el multiplicando que falta (999 o sea ”pos”).

Veamos si nos sale a ojo:

```
Const N, A : array[0,N) of Int ;
Var r : Bool, pos, fac, n : Int ;
{ P: N ≥ 0 }
   r, pos, fac  := True, 0, 1 ;
  do pos ≠ N →
~~        fac, n := 1, 0 ;~~
~~        do  n ≠ pos →~~
~~            fac, n := fac * (n+1) , n + 1~~
~~        od ;~~
         r, pos := r ∧ (A.pos = fac), pos + 1 ;
         fac := fac * pos
  od
{ Q: r =⟨ ∀ i : 0 ≤ i < N : A.i =  i !  ⟩}
```

Parece que salió. (Faltaría testear: **ejercicio!**)

Ahora probemos derivando: El proceso empieza igual que antes: reemplazo de constante por variable para obtener:

INV ≡ r =⟨ ∀ i : 0 ≤ i < pos : A.i = i ! ⟩ ∧ 0 ≤ pos ≤ N

B ≡ pos ≠ N

Garantizando el requisito iii): INV ∧ ¬ B ⇒ Q.

<u>Inicialización:</u> igual que antes obtenemos "r, pos := True, 0".

<u>Cuerpo del ciclo:</u> Los mismos pasos que antes:

{ INV ∧ B }
S_2
{ INV }

Probamos con S_2 siendo "r, pos := E, pos + 1" y derivamos:
Suponemos INV ∧ B y vemos la wp:

wp.(r, pos := E, pos + 1).INV
≡ { def wp para := }
E =⟨ ∀ i : 0 ≤ i < pos + 1 : A.i = i ! ⟩ ∧ 0 ≤ pos + 1 ≤ N
≡ { pasos varios usando varias hip. }
E = r ∧ (A.pos = pos !)

Acá nos trabamos. Necesitamos una hipótesis adicional. Que una nueva variable es igual a la parte que no es programable:

fac = pos !

- **<u>Antes:</u>** lo habíamos resuelto replanteando S_2 como una secuenciación de un ciclo con la asignación, y el ciclo me garantiza esta hipótesis que necesito.
- **<u>Ahora:</u>** Vamos a probar si funciona directamente agregar al invariante que ya tenía la nueva hipótesis que necesito:

<u>Vamos a probar fortaleciendo el invariante:</u>

INV' ≡ INV ∧ fac = pos !

Este invariante reemplaza al original por lo que debo replantear todo el programa de nuevo: (el replanteo incluye declarar la nueva variable fac y reemplazar todo INV por INV').

Const N, A : array[0,N) of Int ;
Var r : Bool, pos, fac ; Int ;
{ P: N ≥ 0 }
S1 ;
{ INV' }
<u>do</u> pos ≠ N →
{ INV' ∧ B }
S2
{ INV' }
<u>od</u>
{ Q: r =⟨ ∀ i : 0 ≤ i < N : A.i = i ! ⟩}

<u>Inicialización de nuevo:</u> La que tenía ya no vale porque la post. INV' es más fuerte que antes, debo además asignar algo a fac. Podemos plantear:

r, fac, pos := E, F, 0

y derivar (ejercicio!).
O directamente se puede elegir E = True, F = 1 y demostrar.

Cuerpo del ciclo de nuevo: Tenemos la siguiente terna:

{ INV' ∧ B }
S2
{ INV': **r =⟨ ∀ i : 0 ≤ i < pos : A.i = i ! ⟩ ∧ 0 ≤ pos ≤ N ∧ fac = pos !** }

Vamos a probar con S2 de la forma: r, fac, pos := E, F , pos + 1.
Supongamos la **hip. INV' ∧ B** y veamos la wp:

wp.(r, fac, pos := E, F , pos + 1).INV'
≡ { def. wp para := }
E =⟨ ∀ i : 0 ≤ i < pos + 1 : A.i = i ! ⟩ ∧ 0 ≤ pos + 1 ≤ N ∧ F = (pos + 1) !
≡ { mismos pasos que antes }
E = r ∧ (A.pos = pos !) ∧ F = (pos + 1)!
≡ { hip. nueva }
E = r ∧ (A.pos = fac) ∧ F = (pos + 1)!
≡ { elijo **E = r ∧ (A.pos = fac)** }
F = (pos + 1)!
≡ { prop. ! }
F = (pos + 1) * pos!
≡ { hip. nueva de nuevo }
F = (pos + 1) * fac
≡ { elijo **F = (pos + 1) * fac** }
True

Perfecto!! El programa final queda:

```
Const N, A : array[0,N) of Int ;
Var r : Bool, pos, fac, n : Int ;
{ P: N ≥ 0 }
  r, fac, pos := True, 1, 0 ;
 do pos ≠ N →
        r, fac, pos := r ∧ (A.pos = fac), fac * (pos + 1) ,  pos + 1 ;
 od
{ Q: r =⟨ ∀ i : 0 ≤ i < N : A.i = i ! ⟩}
```

Casi lo mismo pero mejor.

Análisis de complejidad

Tengo un arreglo de 1000 elementos.
¿Cuántas iteraciones hacía el algoritmo con ciclos anidados?

1 -> 1

2 -> 1 * 2
3 -> 1 * 2 * 3
…
1000 -> 1 * 2 * 3 * …. * 1000

Esto tiene forma de triángulo N * N / 2 = N^2 / 2 que es aprox. la cantidad de iteraciones. Es lo que llamamos complejidad cuadrática.

¿Cuántas iteraciones hace el algoritmo con fortalecimiento? Aproximadamente N. Es lo que llamamos complejidad lineal. Mucho mejor que cuadrática.

¿Qué pasó?

- Derivando el cuerpo del ciclo me trabo porque hay una parte no programable.
- Necesito una nueva variable y una hipótesis que me diga que la variable vale esa parte no programable.
- Dos opciones:
    - Replantear S2 para que tenga un ciclo anidado. **Funciona siempre.**
    - Fortalecer el invariante y replantear el ciclo principal. **Funciona a veces pero da un programa más eficiente.**

Ejemplo (fibonacci)

Const N : Int ;
Var r : Int ;
{ P: N ≥ 0 }
S
{ Q: r = fib.N }

donde la función fib está definida como:

fib.0 ≐ 0
fib.1 ≐ 1
fib.(n+2) ≐ fib.n + fib.(n+1)

Derivación: Necesitamos un ciclo (hay que calcular una función recursiva!). Aplicamos técnica de cambio de constante por variable. Creamos variable nueva n y decimos:

INV ≡ r = fib.n ∧ 0 ≤ n ≤ N
B ≡ n ≠ N

Luego vale requisito iii).

Replanteo el programa:

Const N : Int ;
Var r : Int ;
{ P: N ≥ 0 }
S1

{ INV }
  do n ≠ N →
     { INV ∧ B }
       S2
     { INV }
   od
{ Q: r = fib.N }

Inicialización: Será: r, n := 0, 0. (ejercicio: demostrar!)

Cuerpo del ciclo: S2 será de la forma: r, n := E, n+1.
Derivamos: Supongamos INV ∧ B y veamos la wp:
  wp.(r, n := E, n+1).INV
≡ { def. wp para := }
   E = fib.(n+1) ∧ 0 ≤ n+1 ≤ N
≡ { burocracia: vale por hip varias }
   E = fib.(n+1)

Acá nos trabamos de entrada ya que no podemos aplicar definición de fib (si n = 0, aplica el 2do caso base y si n > 0 aplica el caso recursivo).
Necesitamos una nueva hipótesis que nos diga que una nueva variable vale fib.(n+1).
Probamos fortaleciendo el invariante:

INV' ≡ INV ∧ r2 = fib.(n+1)
≡ **r = fib.n ∧ r2 = fib.(n+1) ∧ 0 ≤ n ≤ N**

Inicialización de nuevo: Ahora va a ser: r, r2, n := 0 , 1 , 0 (ejercicio: demostrar!)

Cuerpo del ciclo de nuevo: Ahora va a ser: r, r2, n := E , F , n+1.

Derivamos: Supongamos INV' ∧ B y veamos la wp:
  wp.(r, n := E, n+1).INV'
≡ { def. wp para := }
   E = fib.(n+1) ∧ F = fib.((n+1)+1) ∧ 0 ≤ n+1 ≤ N
≡ { burocracia }
   E = fib.(n+1) ∧ F = fib.((n+1)+1)
≡ { hip. nueva }
   E = r2 ∧ F = fib.((n+1)+1)
≡ { elijo **E = r2** }
   F = fib.((n+1)+1)
≡ { arit }
   F = fib.(n+2)
≡ { def. fib }
   F = fib.n + fib.(n+1)
≡ { hip. para r y r2 }
   F = r + r2
≡ { elijo **F = r + r2** }
   True

Listo! El programa queda:

Const N : Int ;
Var r, r2, n : Int ;
{ P: N ≥ 0 }
  r, r2, n := 0, 1, 0 ;
  do n ≠ N →
        r, r2, n := r2 , r + r2, n + 1
  od
{ Q: r = fib.N }

Esto anda. **Ejercicio:** testear!!

Terminación de ciclos: Función de cota

Tenemos la siguiente terna de Hoare:

$$\{ P \}\ \textbf{do}\ B \rightarrow S\ \textbf{od}\ \{ Q \}$$

Teníamos vistos tres requisitos que deben valer para que valga la terna:
Existe invariante INV:

i) **P** ⇒ INV
ii) { INV ∧ B } S { INV }
iii) INV ∧ ¬B ⇒ **Q**

Nos falta un requisito importante: La demostración de que el ciclo termina siempre.

Vamos a introducir el concepto de **función de cota**. Como el invariante, la función de cota es una cosa que inventamos para demostrar que el ciclo es correcto, pero **no es parte del programa ni del lenguaje de programación**.

La función de cota es una función que me calcula un número entero a partir de mi estado (o sea a partir del valor de las variables y constantes en un punto determinado de la ejecución). Llamaremos **t** : Estado → Int a la función de cota.

Para poder demostrar que el ciclo termina, vamos a tener que demostrar dos cosas acerca de la función de cota en relación al ciclo:

iv.a) Si estoy en el ciclo, la cota es ≥ 0.

$$INV \wedge B \Rightarrow t \geq 0$$

Equivalentemente, si la cota es < 0, entonces el ciclo termina (version contrarecíprocca).

$$INV \wedge t < 0 \Rightarrow \neg B$$

iv.b) La cota se achica en cada ejecución del cuerpo del ciclo.
Formalmente,

$$\{ INV \wedge B \wedge t = T \}$$ // fijo el valor la cota antes de ejecutar el cuerpo

(usando la variable de especificación T)
S // cuerpo del ciclo
{ INV ∧ t < T } // al terminar, la cota vale menos de lo que valía antes

Si lo logramos demostrar i) y ii), sabremos que **el ciclo termina siempre**.
¿Porque? No importa cuánto valga la cota, por ii), se va a achicar siempre que se ejecute el cuerpo del ciclo. Luego, sí o sí, en algún momento se va a hacer negativa, y por i), sabemos que si la cota es negativa el ciclo termina.

Ejemplo: Suma de los elementos de un arreglo:

Aplicamos cambio de constante por variable de la siguiente manera:
INV ≡ res = ⟨∑ i : 0 ≤ i < pos : A.i ⟩ ∧ 0 ≤ pos ≤ N
B ≡ pos ≠ N

```
Const N : Int, A : Array[0, N) of Int;
Var res, pos : Int;
{ P: N ≥ 0 }
  res, pos := 0, 0 ;  // inicialización
  { INV }
  do pos < N →
        { INV ∧ B }
        res, pos := res + A.pos, pos + 1
        { INV }
  od
{ Q: res = ⟨∑ i : 0 ≤ i < N : A.i ⟩}
```

Demostremos que este ciclo termina.

Deducción de la cota

La función de cota t debe cumplir:

iv.b) La cota decrece:
{ INV ∧ B ∧ t = T }
res, pos := res + A.pos , pos + 1
{ INV ∧ t < T }
Acá, sabemos que pos crece en el cuerpo del ciclo, luego la cota podría ser de la forma:
t = "algo" - pos

iv.a) INV ∧ t < 0 ⇒ ¬ B,
o sea:
(res = ⟨∑ i : 0 ≤ i < pos : A.i ⟩ ∧ 0 ≤ pos ≤ N) ∧ **t < 0 ⇒ pos ≥ N**

t < 0
≡ “algo” - pos < 0
≡ “algo” < pos
≡ “algo” + 1 ≤ pos
O sea, debo elegir “algo” tal que **“algo” + 1 ≤ pos** ⇒ pos ≥ N (o sea **N ≤ pos**)
Si elijo que “algo” + 1 sea N, ya estaría, o sea “algo” = N - 1.

En fin, luego de toda esta deducción, la función de cota que me sirve es:

t = (N - 1) - pos

Otra cota que también sirve:

t = N - pos

(sale a partir de ver iv.a como INV ∧ B ⇒ t ≥ 0).

Demostración formal de la cota

Quedémosnos con la cota **t = N - pos**.
Faltaría hacer la demostración **formal de todo**.
Según el digesto, debemos demostrar:

iv.a) INV ∧ B ⇒ t ≥ 0

Demostración: Suponemos INV ∧ B y vemos:
t ≥ 0
≡ { def. t }
N - pos ≥ 0
≡ { arit. }
N ≥ pos
≡ { hip. del INV }
True

iv.b) { INV ∧ B ∧ t = T } res, pos := res + A.n, pos + 1 { t < T }

Demostración: Suponemos como hip “INV ∧ B ∧ **t = T**” y vemos la wp:
wp.(res, pos := res + A.n, pos + 1).(t < T)
≡ { def. t }
wp.(res, pos := res + A.n, pos + 1).(N - pos < T)
≡ { def. wp }
N - (pos + 1) < T
≡ { arit. }
N - pos - 1 < T
≡ { por hip, T = t = N - pos }
N - pos - 1 < N - pos
≡ { arit. }
-1 < 0
≡ { logica }
True

Observaciones:

- Siempre que recorramos un arreglo de izquierda a derecha, la cota va a tener esta misma forma.
- Demostrar los requisitos de la cota formalmente es un poco burocrático para lo obvios que son.
- Muchas veces para la cota admitiremos explicaciones con palabras en lugar de demostraciones formales.

Ejemplo: Suma de los elementos de un arreglo de otra forma:

```
Const N : Int, A : Array[0, N) of Int;
Var res , n : Int;
  res, n := 0, N ;
  do n ≠ 0 →
       res, n :=  res + A.(n-1) , n - 1
  od
{ Q: res = ⟨∑ i : 0 ≤ i < N : A.i ⟩}
```

Aplicamos cambio de constante por variable de la siguiente manera:

INV ≡ res = ⟨∑ i : n ≤ i < N : A.i ⟩ ∧ 0 ≤ n ≤ N
B ≡ n ≠ 0

La función de cota es:

- t = N - n NO ANDA porque crece en el cuerpo del ciclo.
- t = n ???

Recordemos requisitos:
iv.a) Si estoy en el ciclo, t ≥ 0. t = n funciona!!! (ver INV: 0 ≤ n)
iv.b) t se achica con el cuerpo del ciclo. obvio que se achica porque se asigna n := n - 1.
CON ALGO ASÍ ALCANZA PARA PARCIAL/RECUPERATORIO.

Conclusión: t = n funciona.

Pregunta: ¿Hay otra función de cota que vale?
¿Vale la funcion de cota t = n + 344565? Si! Valen tanto iv.a) como iv.b).

Observación:

- La función de cota no es única.
- En particular, si tengo una función de cota t correcta, también valen como función de cota todas las funciones t' = t + C (a donde C > 0 es una constante cualquiera).
-

Ejemplo: Algoritmo de la división

```
Const X, Y : Int ;
Var q, r : Int ;
{ P:  X ≥ 0  ∧  Y > 0 }
  q, r := 0 , X ;
```

do r ≥ Y →
q, r := q + 1 , r - Y
od
{ Q: X = q * Y + r ∧ 0 ≤ r ∧ r < Y }

INV ≡ X = q * Y + r ∧ 0 ≤ r

La función de cota es: t = r ???

iv.a) Si estoy en el ciclo, r ≥ 0? Sí, por INV.
iv.b) r se achica en el cuerpo del ciclo? Sí, ya que asignamos r := r - Y con Y > 0.

Luego, **t = r** es una función de cota correcta y el ciclo termina.

Observación:

- Mirar especialmente la guarda y el cuerpo del ciclo para determinar la cota. (el INV también ayuda)
- Otra cota para el algoritmo de la división podría ser de la forma t = "algo" - q (ya que q crece).

Ejemplo (practico 6, ejercicio 2f):

{ True }
r := N ;
do r ≠ 0 →
if r < 0 → r := r + 1 // r es negativo, por ejemplo de -10 pasa a -9.
[] r > 0 → r := r - 1 // r es positivo, por ejemplo de 13 pasa a 12.
fi
od
{ r = 0 }

Este programa arranca con r en N y va hacia el 0, luego en el cuerpo del ciclo el valor absoluto de r (o sea su distancia al 0) se achica siempre.

¿Cuál es el invariante?
INV ≡ ???

¿Cuál es la función de cota?
t = N **- | r |** **// | r | se achica siempre,**
// entonces me conviene sumarlo no restarlo.

Mejor probemos:
t = | r |

iv.a) INV ∧ B ⇒ | r | ≥ 0 ??? Vale siempre por def de valor absoluto.

iv.b) Se achica? Si, ya lo pensamos. Ejercicio: demostrar formalmente
(hay una terna de un **if**).

Ejercicio: Hacer testing pero además calculando el valor de la función de cota en cada iteración del ciclo.

Problemas de bordes

Más material acá (2021-12-01 Consulta)

Y acá (2021-12-14 Consulta)

Ejercicio con segmentos iniciales

Enunciado: Parecido a la función psum pero con arreglos en lugar de listas:
"Dado un arreglo con N enteros, decidir si todos los segmentos iniciales del arreglo suman ≥ 0."

Especificación:

Const N : Int, A : array[0, N) of Int;
Var res : Bool;
{ P : N ≥ 0 }
  S
{ Q : res = ⟨∀ i : 0 ≤ i ≤ N : psum.i ≥ 0 ⟩ }
  |[psum.i = ⟨∑ j : 0 ≤ j < i : A.j ⟩]|

Derivación:
Necesitamos un ciclo. Usando la técnica de reemplazo de constante por variable proponemos:

INV ≡ res =⟨ ∀ i : 0 ≤ i ≤ pos : psum.i ≥ 0 ⟩ ∧ 0 ≤ pos ≤ N
B ≡ pos < N

donde pos es una nueva variable de tipo Int.
De esta manera tenemos asegurado INV ∧ ¬B ⇒ Q.

El programa será de la forma:
Const N, A : array[0,N) of Int ;
Var res : Bool, pos : Int ;
{ P: N ≥ 0 }
  S1 ;
{ INV }
  do pos ≠ N →

{ INV ∧ B }
S2
{ INV }
od
{ Q }

Cuerpo del ciclo (S2): Proponemos una asignación de la forma res, pos := E, pos+1.

Suponemos INV ∧ B como hipótesis y vemos la wp:
wp.(res, pos := E, pos+1).INV
≡ { Def. wp }
E =⟨ ∀ i : 0 ≤ i ≤ pos+1 : psum.i ⟩ ∧ 0 ≤ pos+1 ≤ N
≡ { hipótesis 0 ≤ pos y pos < N }
E =⟨ ∀ i : 0 ≤ i ≤ pos+1 : psum.i ⟩
≡ { partición de rango y rango unitario }
E =⟨ ∀ i : 0 ≤ i ≤ pos : psum.i ⟩ ∧ (psum.(pos+1) ≥ 0)
≡ { hipótesis }
E = res ∧ (psum.(pos+1) ≥ 0)
≡ { def. psum }
E = res ∧ (⟨∑ j : 0 ≤ j < pos+1 : A.j ⟩ ≥ 0)
≡ { part. de rango y rango unitario }
E = res ∧ (⟨∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos ≥ 0)

Nos trabamos! No podemos elegir una expresión “programable” para E por la sumatoria.

Fortalecemos el invariante:
INV’ ≡ INV ∧ sum = ⟨∑ j : 0 ≤ j < pos : A.j ⟩

Cuerpo del ciclo de nuevo (S2): Proponemos una asignación de la forma
res, sum, pos := E, F, pos+1.

Suponemos INV’ ∧ B como hipótesis y vemos la wp:
wp.(res, sum, pos := E, F, pos+1).INV’
≡ { Def. wp y mismos pasos ya hechos }
E = res ∧ (⟨∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos ≥ 0) ∧ F = ⟨∑ j : 0 ≤ j < pos+1 : A.j ⟩
≡ { part. de rango y rango unitario }
E = res ∧ (⟨∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos ≥ 0) ∧ F = ⟨∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos
≡ { hipótesis nueva dos veces }
E = res ∧ (sum + A.pos ≥ 0) ∧ F = sum + A.pos
≡ { elijo convenientemente }
True

Inicialización: Ejercicio. Resultado: res, sum, pos := True, 0, 0.

Resultado final:
Const N : Int, A : array[0, N) of Int;
Var r : Int;
{P : N ≥ 0}

res, sum, pos := True, 0, 0 **;**
do pos < N →
res, sum, pos := res ∧ (sum + A.pos ≥ 0), sum + A.pos, pos + 1
od
{ Q : res = ⟨∀ i : 0 ≤ i ≤ N : psum.i ≥ 0 ⟩ }

¿Qué pasa si fortalecemos mal?

Volvemos atrás:

- Cuerpo del ciclo (S2): Proponemos una asignación de la forma res, pos := E, pos+1.

Suponemos INV ∧ B como hipótesis y vemos la wp:
wp.(res, pos := E, pos+1).INV
≡ { … }
E = res ∧ (⟨∑ j : 0 ≤ j < pos+1 : A.j ⟩ ≥ 0)

Fortalecemos así (**mala idea**):
INV' ≡ INV ∧ sum = ⟨∑ j : 0 ≤ j < pos+1 : A.j ⟩

Sabemos que 0 ≤ pos ≤ N, así que cuando pos = N la sumatoria "sum" queda:
A.0 + A.1 + … + A.(N-1) + A.N
¡Mal! Se va de los límites del arreglo.

Si derivamos el cuerpo del ciclo de nuevo:

- Cuerpo del ciclo de nuevo (S2): Proponemos una asignación de la forma res, sum, pos := E, F, pos+1.

Suponemos INV' ∧ B como hipótesis y vemos la wp:
wp.(res, sum, pos := E, F, pos+1).INV'
≡ { Def. wp y mismos pasos ya hechos }
E = res ∧ (⟨∑ j : 0 ≤ j < pos+1 : A.j ⟩ ≥ 0) ∧ F = ⟨∑ j : 0 ≤ j < pos+2 : A.j ⟩
≡ { part. de rango y rango unitario }
E = res ∧ (⟨∑ j : 0 ≤ j < pos : A.j ⟩ + A.(pos+1) ≥ 0) ∧
F = ⟨∑ j : 0 ≤ j < pos+1 : A.j ⟩ + A.(pos+1)
≡ { hipótesis nueva dos veces }
E = res ∧ (sum + A.(pos+1) ≥ 0) ∧ F = sum + A.(pos+1)
≡ { elijo convenientemente }
True

Resultado:
do pos < N →
res, sum, pos := res ∧ (sum + A.(pos+1) ≥ 0), sum + A.(pos+1), pos + 1
od

¡Mal! En la última iteración del ciclo pos = N - 1 y A.(pos+1) = A.N se va de los límites del arreglo.

Ejercicio con segmentos arbitrarios

Segmento de suma máxima

Especificación:

Const N : Int, A : array[0, N) of Int;
Var r : Int;
{P : N ≥ 0}
  S
{Q : r = ⟨Max p, q : **0 ≤ p ≤ q ≤ N** : sum.p.q ⟩
        |[sum.p.q = ⟨∑ i : **p ≤ i < q** : A.i ⟩]| }

¿Qué calcula este programa? Considera todos los segmentos posibles del arreglo (incluso los segmentos vacíos, porque puede ser p = q). p y q funcionan como índices de comienzo y fin del segmento: p cerrado y q abierto. Ejemplo: si (p, q) = (6, 9) estamos hablando de la suma: A.6 + A.7 + A.8. De todos los segmentos se calcula la suma y se toma el máximo.

Derivación: VER EN LIBRO CÁLCULO DE PROGRAMAS.

Resultado final:

Const N : Int, A : array[0, N) of Int;
Var r : Int;
{P : N ≥ 0}
  r, r2, n := 0, 0, 0 **;**
  do n ≠ N →
        r, r2, n := r max (r2 + A.n) max 0 , (r2 + A.n) max 0 , n+1 ;
  od
{Q : r = ⟨Max p, q : **0 ≤ p ≤ q ≤ N** : sum.p.q ⟩
        |[sum.p.q = ⟨∑ i : **p ≤ i < q** : A.i ⟩]| }

Ejercicio: Testear que efectivamente esto está calculando el segmento de suma máxima.

Función de cota: t = N - n
(la misma que tenemos siempre que recorremos un arreglo de izquierda a derecha)

Segmento de suma máxima sin Segmentos vacíos

En esta versión más simple del problema de “Segmento de suma máxima” no consideraremos segmentos vacíos. Observar en la especificación que dice p < q en lugar de p ≤ q.

Especificación:

Const N : Int, A : array[0, N) of Int;
Var res : Int;
{P : N ≥ 0}
  S
{Q : res = ⟨Max p, q : 0 ≤ **p < q** ≤ N : sum.p.q ⟩
        |[sum.p.q = ⟨∑ i : p ≤ i < q : A.i ⟩]| }

Derivación:
Necesitamos un ciclo. Usando la técnica de reemplazo de constante por variable proponemos:

INV ≡ res = ⟨Max p, q : 0 ≤ p < q ≤ pos : sum.p.q ⟩ ∧ 0 ≤ pos ≤ N
B ≡ pos < N

donde pos es una nueva variable de tipo Int.
De esta manera tenemos asegurado INV ∧ ¬B ⇒ Q.

El programa será de la forma:
Const N, A : array[0,N) of Int ;
Var res : Int, pos : Int ;
{ P: N ≥ 0 }
 S1 ;
{ INV }
 do pos < N →
      { INV ∧ B }
        S2
      { INV }
 od
{ Q }

Cuerpo del ciclo (S2): Proponemos una asignación de la forma:

res, pos := E, pos + 1

Suponemos INV ∧ B como hipótesis y vemos la wp:
 wp.(res, pos := E, pos+1).INV
≡ { def. wp }
 E = ⟨Max p, q : 0 ≤ p < q ≤ pos + 1 : sum.p.q ⟩ ∧ 0 ≤ pos + 1 ≤ N
≡ { por hipótesis 0 ≤ pos ≤ N y pos < N }
 E = ⟨Max p, q : 0 ≤ p < q ≤ pos + 1 : sum.p.q ⟩
≡ { lógica }
 E = ⟨Max p, q : 0 ≤ p < q ∧ (q ≤ pos ∨ q = pos + 1) : sum.p.q ⟩
≡ { lógica: distributividad }
 E = ⟨Max p, q : (0 ≤ p < q ≤ pos) ∨ (0 ≤ p < q ∧ q = pos + 1) : sum.p.q ⟩
≡ { partición de rango }
 E = ⟨Max p, q : 0 ≤ p < q ≤ pos : sum.p.q ⟩ max
      ⟨Max p, q : 0 ≤ p < q ∧ q = pos + 1 : sum.p.q ⟩
≡ { hipótesis }
 E = res max ⟨Max p, q : 0 ≤ p < q ∧ q = pos + 1 : sum.p.q ⟩
≡ { eliminación de variable q }
 E = res max ⟨Max p : 0 ≤ p < pos + 1 : sum.p.(pos + 1) ⟩

≡ { **no fortalecer todavía!!** este Max considera todos los segmentos no vacíos terminados en la posición p inclusive (A.p) , por lo que **tenemos un problema de borde**.
Faltan más pasos: lógica
}
E = res max ⟨Max p : 0 ≤ p < pos ∨ p = pos : sum.p.(pos + 1) ⟩
≡ { partición de rango y rango unitario }
E = res max ⟨Max p : 0 ≤ p < pos : sum.p.(pos + 1) ⟩ max sum.pos.(pos + 1)
≡ { def. sum }
E = res max ⟨Max p : 0 ≤ p < pos : sum.p.(pos + 1) ⟩ max ⟨∑ i : pos ≤ i < pos + 1: A.i ⟩
≡ { rango unitario: i = pos }
E = res max ⟨Max p : 0 ≤ p < pos : sum.p.(pos + 1) ⟩ max A.pos
≡ { def. sum }
E = res max ⟨Max p : 0 ≤ p < pos : ⟨∑ i : p ≤ i < pos + 1 : A.i ⟩⟩ max A.pos
≡ { lógica }
E = res max ⟨Max p : 0 ≤ p < pos : ⟨∑ i : p ≤ i < pos ∨ i = pos : A.i ⟩⟩ max A.pos
≡ { partición de rango y rango unitario }
E = res max ⟨Max p : 0 ≤ p < pos : ⟨∑ i : p ≤ i < pos : A.i ⟩+ A.pos ⟩ max A.pos
≡ { distributividad de + con Max, y deshacer definición de sum }
E = res max (⟨Max p : 0 ≤ p < pos : sum.p.pos ⟩ + A.pos) max A.pos

Acá nos detenemos. Este Max ya no incluye al elemento A.pos, ya que considera todos los segmentos no vacíos terminados en la posición pos-1:

A.0 + A.1 + … + A.(pos-2) + A.(pos-1)
A.1 + … + A.(pos-2) + A.(pos-1)
…
A.(pos-2) + A.(pos-1)
A.(pos-1)

Por lo tanto podemos fortalecer sin problemas.
INV' ≡ INV ∧ aux = ⟨Max p : 0 ≤ p < pos : sum.p.pos ⟩

Cuerpo del ciclo de nuevo (S2): Proponemos una asignación de la forma
res, aux, pos := E, F, pos + 1.

Suponemos INV' ∧ B como hipótesis y vemos la wp:
wp.(res, aux, pos := E, F, pos + 1).INV'
≡ { def. wp }
E = ⟨Max p, q : 0 ≤ p < q ≤ pos + 1 : sum.p.q ⟩ ∧ 0 ≤ pos + 1 ≤ N ∧
F = ⟨Max p : 0 ≤ p < pos + 1 : sum.p.(pos + 1) ⟩
≡ { mismos pasos que hicimos antes }
E = res max (⟨Max p : 0 ≤ p < pos : sum.p.pos ⟩ + A.pos) max A.pos ∧
F = ⟨Max p : 0 ≤ p < pos + 1 : sum.p.(pos + 1) ⟩
≡ { **hipótesis nueva** }
E = res max (aux + A.pos) max A.pos ∧
F = ⟨Max p : 0 ≤ p < pos + 1 : sum.p.(pos + 1) ⟩
≡ { elegimos **E = res max (aux + A.pos) max A.pos** }
F = ⟨Max p : 0 ≤ p < pos + 1 : sum.p.(pos + 1) ⟩

≡ { a estos pasos ya los hicimos también, lo hago rápido: }
  F = ⟨Max p : 0 ≤ p < pos : sum.p.(pos + 1) ⟩ max A.pos
≡ { separación de término en sum }
  F = ⟨Max p : 0 ≤ p < pos : sum.p.pos + A.pos ⟩ max A.pos
≡ { distributividad }
  F = (⟨Max p : 0 ≤ p < pos : sum.p.pos ⟩ + A.pos) max A.pos
≡ { **hipótesis nueva** }
  F = (aux + A.pos) max A.pos
≡ { elijo **F = (aux + A.pos) max A.pos** }
  True

Inicialización (S1): Queda como ejercicio. Obtenemos:

res, aux, pos := -infinito, -infinito, 0

Resultado final:

```
Const N : Int, A : array[0, N) of Int;
Var res, aux, pos : Int;
{P : N ≥ 0}
  res, aux, pos := -infinito, -infinito, 0 ;
  do n < N →
       res, aux, pos := res max ( aux + A.pos  ) max A.pos,
                        (aux + A.pos) max A.pos,
                        pos + 1
  od
{Q : res = ⟨Max p, q : 0 ≤ p < q ≤ N : sum.p.q ⟩
       |[sum.p.q = ⟨∑ i : p ≤ i < q : A.i ⟩]|   }
```

Ejercicio: Testear que efectivamente esto está calculando el segmento de suma máxima.

Función de cota: t = N - n
(la misma que tenemos siempre que recorremos un arreglo de izquierda a derecha)

Especificaciones con segmentos de arreglo

¿Cómo escribimos cuantificadores sobre segmentos de arreglo?

Solemos usar dos variables cuantificadas p, q para representar el segmento:

A.p, A.(p+1), … , A.(q-1)

Es decir, el segmento va desde la posición “p” hasta la posición “q - 1”.
(**observar que no incluye la posición q**, es cerrado-abierto: “[p, q)”).

La cuantificación sería de la siguiente forma:

$$\langle \oplus\ p\ ,\ q\ :\ R.p.q \wedge \ldots : T.p.q \rangle$$

donde R es un predicado que indica el tipo de segmento:

0 ≤ p ≤ q ≤ N: segmentos arbitrarios (o sea, todos los segmentos posibles)
0 ≤ p < q ≤ N: segmentos no vacíos
0 < p ≤ q ≤ N: segmentos no iniciales (no prefijos)
0 ≤ p ≤ q < N: segmentos no finales (no sufijos)
0 < p ≤ q < N: segmentos interiores (no iniciales ni finales)
0 < p < q < N: segmentos interiores no vacíos

p = 0 ∧ 0 ≤ q ≤ N: segmentos iniciales (se puede eliminar p, queda: **0 ≤ q ≤ N**)
0 ≤ p ≤ N ∧ q = N: segmentos finales (se puede eliminar q, queda: **0 ≤ p ≤ N**)

TODO: PARTICIONES DE RANGO CON SEGMENTOS Y CON PARES DE ELEMENTOS

Temas complementarios

Terminación anticipada de ciclos

Volvemos al ejercicio de esta sección:

El programa era:
Const N : Int, A : array[0, N) of Int;
Var r : Int;
{P : N ≥ 0}
  res, sum, pos := True, 0, 0 **;**
  do pos < N →
      res, sum, pos := res ∧ (sum + A.pos ≥ 0), sum + A.pos, pos + 1
  od
{ Q : res = ⟨∀ i : 0 ≤ i ≤ N : psum.i ≥ 0 ⟩ }

Notamos que al primer False podemos terminar sin necesidad de recorrer el resto del arreglo. Nueva guarda:

$$\mathbf{B' \equiv\ pos < N \wedge res}$$

(sólo quiero continuar en el ciclo si res es True)

¿Qué debo demostrar al introducir esta modificación?
Lugares donde aparece la guarda:

- Requisito 2:
  Ya habiamos demostrado:
    { INV ∧ B } S1 { INV } (donde S1 es el cuerpo del ciclo)
  ¿Vale con B'?:
    { INV ∧ B' } S1 { INV }
  Respuesta: Sí, si vale una terna, también vale cuando fortalecemos su precondición.

El conjunto de estados iniciales admitidos por INV ∧ B' está incluido en el conjunto admitido por INV ∧ B.

- Requisito 3:
  Ya habíamos demostrado:
  INV ∧ ¬B ⇒ Q
  ¿Vale con B'?:
  INV ∧ ¬B' ⇒ Q
  ¡No necesariamente!
  Veamos:
  INV ∧ ¬B' ⇒ Q
  ≡ INV ∧ ¬(B ∧ res) ⇒ Q
  ≡ INV ∧ (¬B ∨ ¬res) ⇒ Q
  ≡ (INV ∧ ¬B) ∨ (INV ∧ ¬res) ⇒ Q
  ≡ (INV ∧ ¬B ⇒ Q) ∧ (INV ∧ ¬res ⇒ Q)

La primera "INV ∧ ¬B ⇒ Q" ya la habíamos demostrado.
Falta demostrar la segunda: INV ∧ ¬res ⇒ Q.

Solamente hace falta demostrar:

INV ∧ ¬B' ⇒ Q.

Ahora, sabemos que:

INV ∧ ¬B' ⇒ Q ≡ (INV ∧ ¬B ⇒ Q) ∧ (INV ∧ ¬res ⇒ Q)

(**ejercicio: demostrar**)
Y también sabemos que vale "INV ∧ ¬B ⇒ Q".
Luego sólo hace falta demostrar:

INV ∧ ¬res ⇒ Q

Demostración: Queremos demostrar **INV ∧ ¬res ⇒ Q**.
Hipótesis 1: INV ≡ res = ⟨∀ i : 0 ≤ i ≤ n : psum.i ≥ 0 ⟩ ∧ 0 ≤ n ≤ N ∧ …
Hipótesis 2: ¬res

Suponemos verdaderas estas dos hipótesis. Y vemos Q:

Q
≡ { def. Q }
res = ⟨∀ i : 0 ≤ i ≤ N : psum.i ≥ 0 ⟩
≡ { hip. 2 }
False = ⟨∀ i : 0 ≤ i ≤ N : psum.i ≥ 0 ⟩
≡ { lógica (ver abajo explicación), sabiendo que 0 ≤ n ≤ N (hip. 1) }
False = ⟨∀ i : 0 ≤ i ≤ n ∨ n < i ≤ N : psum.i ≥ 0 ⟩
≡ { part. rango }
False = ⟨∀ i : 0 ≤ i ≤ n : psum.i ≥ 0 ⟩ ∧ ⟨∀ i : n < i ≤ N : psum.i ≥ 0 ⟩
≡ { hip. 1 }
False = res ∧ ⟨∀ i : n < i ≤ N : psum.i ≥ 0 ⟩
≡ { hip. 2 de nuevo }
False = False ∧ ⟨∀ i : n < i ≤ N : psum.i ≥ 0 ⟩
≡ { **lógica: absorbente de** ∧ }
False = False
≡ { lógica }

True

$i \in \{ 0 , 1 , 2, \ldots\ldots\ldots\ldots\ldots\ldots , N \}$ $(0 \le i \le N)$
$i \in \{ 0 , 1 , 2, \ldots. n \}$ $(0 \le i \le n)$
$i \in \{ n+1, n+2, \ldots., N \}$ $(n < i \le N)$ ó $(n+1 \le i \le N)$

El programa finalmente queda:

```
Const N : Int, A : array[0, N) of Int;
Var r : Int;
{P : N ≥ 0}
  res, sum, pos := True, 0, 0 ;
  do pos < N ∧ res →
        res, sum, pos :=  res ∧ ( sum + A.pos ≥ 0 ), sum + A.pos, pos + 1
  od
{ Q : res = ⟨∀ i : 0 ≤ i ≤ N : psum.i ≥ 0 ⟩ }
```

Traducción a C

Ver Apéndice B del libro Cálculo de Programas.

Ejercicios

practico3.pdf

practico4.pdf

Resoluciones de ejercicios

Ejercicio de final con varias cosas interesantes

- Problema con pares de elementos de arreglo
- Problema de “borde”
- Análisis por casos en el cuerpo (if)
- Dos fortalecimientos

Consideremos la siguiente especificación:

```
Const N: Int, A : array[0, N) of Int;
Var r : Int;
{ P: N ≥ 0 }
  S
{ Q : r = ⟨ ∑ i , j : 0 ≤ i < j < N ∧  A.i * A.j > 0  : A.i * A.j ⟩}
```

Este programa calcula la suma de todas formas posibles de multiplicar dos elementos distintos del arreglo, siempre que ese producto sea > 0.

Ejemplo (testing): Para el arreglo A = [3, -2, 1, 0, -2], el resultado es:

3 * 1 + (-2) * (-2) = 7

¿Programa a ojo? Se podría hacer con ciclos anidados, el ciclo de afuera para fijar un elemento, el ciclo de adentro para fijar el otro elemento, y un if para ver si el producto es > 0.

Derivación: Necesitamos un ciclo. Aplicamos cambio de constante por variable:

INV ≡ $r = \langle \sum i, j : 0 \le i < j < n \land A.i * A.j > 0 : A.i * A.j \rangle \land 0 \le n \le N$

B ≡ $n \neq N$

La cota va a ser t = N - n (ya sabemos que vamos a recorrer el arreglo de izq. a der.).

El programa será de la forma:

Const N: Int, A : array[0, N) of Int;
Var r : Int;
{ P: N ≥ 0 }
S1
{ INV }
do n ≠ N →
{ INV ∧ B }
S2
{ INV }
od
{ Q : $r = \langle \sum i, j : 0 \le i < j < N \land A.i * A.j > 0 : A.i * A.j \rangle$ }

Cuerpo del ciclo: Será de la forma: r, n := E, n + 1.
Sup. INV ∧ B y veamos la wp:

wp.(r, n := E, n + 1).INV
≡ { def. wp }
$E = \langle \sum i, j : 0 \le i < j < n+1 \land A.i * A.j > 0 : A.i * A.j \rangle \land 0 \le n+1 \le N$
≡ { hip. 0 ≤ n ≤ N y n ≠ N }
$E = \langle \sum i, j : 0 \le i < j < n+1 \land A.i * A.j > 0 : A.i * A.j \rangle$
≡ { reescribimos: $0 \le i < j < n+1 \equiv 0 \le i < j \land (j < n \lor j = n)$
distribuimos y partimos rango }
$E = \langle \sum i, j : 0 \le i < j \land j < n \land A.i * A.j > 0 : A.i * A.j \rangle +$
$\langle \sum i, j : 0 \le i < j \land j = n \land A.i * A.j > 0 : A.i * A.j \rangle$
≡ { hip. para r }
$E = r + \langle \sum i, j : 0 \le i < j \land j = n \land A.i * A.j > 0 : A.i * A.j \rangle$
≡ { eliminación j = n }
$E = r + \langle \sum i : 0 \le i < n \land A.i * \mathbf{A.n} > 0 : A.i * \mathbf{A.n} \rangle$

// aca si quisiera podría ya plantear ciclo anidado, y obtener el mismo programa que
// había pensado a ojo.
// pero quiero fortalecer

// no puedo fortalecer como está, porque para calcular esto no puedo **reusar**
// lo que se calculó en la iteración anterior.
// me doy cuenta porque esta cuenta depende de **A.n**, y es un elemento que no
// habíamos visto nunca antes en la iteraciones anteriores.
// para poder fortalecer, debo poder **deshacerme de las menciones a A.n**.

≡ { distributividad }

$E = r + \langle \sum\ i : 0 \le i < n\ \wedge\ \mathbf{A.i * A.n > 0}\ : A.i \rangle * A.n$

≡ { para poder deshacerme de este A.n, puedo recurrir a la **regla de los signos**:

$a * b > 0\ \equiv\ (a > 0 \wedge b > 0)\ \vee\ (a < 0 \wedge b < 0)$

}

$E = r + \langle \sum\ i : 0 \le i < n\ \wedge\ (\ (A.i > 0 \wedge \mathbf{A.n > 0})\ \vee\ (A.i < 0 \wedge \mathbf{A.n < 0})\ ) : A.i \rangle * A.n$

≡ { "**A.n > 0**" y "**A.n < 0"** no dependen de i, tienen un valor de verdad fijo. Podemos preguntar por su valor de verdad con un **if**!!! }

- Caso 1: **A.n > 0**: (<-- debe ser programable ya que es la guarda del if)

  ≡ { sustituyo los valores de verdad }

  $E = r + \langle \sum\ i : 0 \le i < n\ \wedge\ \underline{(\ (A.i > 0 \wedge True)\ \vee\ (A.i < 0 \wedge False)\ )}\ : A.i \rangle * A.n$

  ≡ { simplifico por lógica }

  $E = r + \underline{\langle \sum\ i : 0 \le i < n\ \wedge\ A.i > 0 : A.i \rangle} * A.n$

  Ahora sí me trabo pero puedo fortalecer: $s = \langle \sum\ i : 0 \le i < n\ \wedge\ A.i > 0 : A.i \rangle$

  ¿Qué es s? s es la sumatoria de los elementos positivos vistos hasta el momento.

- Caso 2: **A.n < 0**: (<-- debe ser programable ya que es la guarda del if)

  ≡ { sustituyo los valores de verdad }

  $E = r + \langle \sum\ i : 0 \le i < n\ \wedge\ \underline{(\ (A.i > 0 \wedge False\ )\ \vee\ (A.i < 0 \wedge True)\ )}\ : A.i \rangle * A.n$

  ≡ { simplifico por lógica }

  $E = r + \langle \sum\ i : 0 \le i < n\ \wedge\ A.i < 0 : A.i \rangle * A.n$

  Ahora también me trabo pero puedo fortalecer:

  $t = \langle \sum\ i : 0 \le i < n\ \wedge\ A.i < 0 : A.i \rangle$

  ¿Qué es t? t es la sumatoria de los elementos negativos vistos hasta el momento.

- Caso 3: **A.n = 0**: (<-- debo cubrir todas las posibilidades, tengo tres guardas)

  ≡ { sustituyo los valores de verdad }

  $E = r + \langle \sum\ i : 0 \le i < n\ \wedge\ \underline{(\ (A.i > 0 \wedge False\ )\ \vee\ (A.i < 0 \wedge False\ )\ )}\ : A.i \rangle * A.n$

  ≡ { simplifico por lógica }

  $E = r + \langle \sum\ i : 0 \le i < n\ \wedge\ False : A.i \rangle * A.n$

  ≡ { rango vacío }

  $E = r + 0 * A.n$

  ≡ { elijo **E = r** }

  True

Nuevo invariante:

$INV' \equiv INV \wedge\ s = \langle \sum\ i : 0 \le i < n\ \wedge\ A.i > 0 : A.i \rangle \wedge\ t = \langle \sum\ i : 0 \le i < n\ \wedge\ A.i < 0 : A.i \rangle$

Cuerpo del ciclo de nuevo:

Sup. INV' $\wedge$ B y vemos la wp:

wp.(r, s, t, n := E, F, G, n+1).INV'
≡ { def. wp para := }
E = ⟨ ∑ i , j : 0 ≤ i < j < n+1 ∧ A.i * A.j > 0 : A.i * A.j ⟩ ∧ 0 ≤ n+1 ≤ N
∧ F = ⟨ ∑ i : 0 ≤ i < n+1 ∧ A.i > 0 : A.i ⟩ ∧ G = ⟨ ∑ i : 0 ≤ i < n+1 ∧ A.i < 0 : A.i ⟩
≡ { partición de rango e hipotesis }
E = ⟨ ∑ i , j : 0 ≤ i < j < n+1 ∧ A.i * A.j > 0 : A.i * A.j ⟩ ∧ 0 ≤ n+1 ≤ N
∧ F = s +⟨ ∑ i : i = n ∧ A.**i** > 0 : A.i ⟩ ∧ G = t +⟨ ∑ i : i = n ∧ A.**i** < 0 : A.i ⟩
≡ { Leibniz: reemplazo i por n ya que i = n, pero eliminar i }
E = ⟨ ∑ i , j : 0 ≤ i < j < n+1 ∧ A.i * A.j > 0 : A.i * A.j ⟩ ∧ 0 ≤ n+1 ≤ N
∧ F = s +⟨ ∑ i : i = n ∧ A.n > 0 : A.i ⟩ ∧ G = t +⟨ ∑ i : i = n ∧ A.n < 0 : A.i ⟩
≡ { ahora sí, repito los pasos que hice antes hasta el analisis por casos }
E = r + ⟨ ∑ i : 0 ≤ i < n ∧ ((A.i > 0 ∧ **A.n > 0**) ∨ (A.i < 0 ∧ **A.n < 0**)) : A.i ⟩ * A.n
∧ F = s +⟨ ∑ i : i = n ∧ **A.n > 0** : A.i ⟩ ∧ G = t +⟨ ∑ i : i = n ∧ **A.n < 0** : A.i ⟩

- Caso 1: A.n > 0:
  ≡ { sustituyo valores de verdad }
  E = r + ⟨ ∑ i : 0 ≤ i < n ∧ A.i > 0 : A.i ⟩ * A.n ∧
  F = s +⟨ ∑ i : i = n ∧ True : A.i ⟩ ∧ G = t +⟨ ∑ i : i = n ∧ False : A.i ⟩
  ≡ { hip para s }
  E = r + s * A.n ∧
  F = s +⟨ ∑ i : i = n ∧ True : A.i ⟩ ∧ G = t +⟨ ∑ i : i = n ∧ False : A.i ⟩
  ≡ { el 1ro es rango unitario, y el 2do es rango vacio }
  E = r + s * A.n ∧ F = s + A.n ∧ G = t
  ≡ { **elijo convenientemente** }
  True

- Caso 2: A.n < 0:
  ≡ { pasos similares al caso anterior }
  E = r + t * A.n ∧ F = s ∧ G = t + A.n
  ≡ { **elijo convenientemente** }
  True

- Caso 3: A.n = 0:
  ≡ { sustituyo valores de verdad }

E = r + ⟨ ∑ i : 0 ≤ i < n ∧ ((A.i > 0 ∧ **False**) ∨ (A.i < 0 ∧ False)) : A.i ⟩ * A.n
∧ F = s +⟨ ∑ i : i = n ∧ **False** : A.i ⟩ ∧ G = t +⟨ ∑ i : i = n ∧ False : A.i ⟩
≡ { todos rangos vacíos }
E = r ∧ F = s ∧ G = t
≡ { **elijo convenientemente** }
True

Listo el cuerpo del ciclo!!! Es un if con asignaciones adentro.

Inicialización: r, s, t, n := 0, 0, 0, 0 (todos rangos vacíos de sumatorias)

Resultado final:
Const N: Int, A : array[0, N) of Int;
Var r : Int;

{ P: N ≥ 0 }
 r, s, t, n := 0, 0, 0, 0 ;
 do n ≠ N →
  if A.n > 0 → r, s, ~~t,~~ n := r + s * A.n , s + A.n , ~~t~~, n+1
  [] A.n < 0 → r, ~~s,~~ t, n := r + t * A.n , ~~s~~, t + A.n , n+1
  [] A.n = 0 → ~~r, s, t,~~ n := ~~r, s, t,~~ n+1
  if
 od
{ Q : r = ⟨ ∑ i , j : 0 ≤ i < j < N ∧ A.i * A.j > 0 : A.i * A.j ⟩}

Conclusión: A ojo nos salió con dos ciclos anidados, pero gracias a la derivación pudimos encontrar una forma de hacer fortalecimiento que salga mucho más eficiente con un solo ciclo.

Ejercicio: Testear con el arreglo de ejemplo que vimos, chequear que da 7.

Referencias

Cálculo de programas
Javier Blanco, Silvina Smith, Damián Barsotti
ISBN 978-950-33-0642-0. Año 2009.

Hoare, C. A. R. (October 1969). "An axiomatic basis for computer programming". Communications of the ACM. 12 (10): 576–580.

Edsger Dijkstra: The Humble Programmer (PDF file, 473kB)
http://cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF

Hoare 1996, "How Did Software Get So Reliable Without Proof?"
https://gwern.net/doc/math/1996-hoare.pdf

Dijkstra, Edsger W. (March 1968). "Letters to the editor: Go to statement considered harmful" (PDF). Communications of the ACM. 11 (3): 147–148. doi:10.1145/362929.362947. S2CID 17469809.
https://www.cs.utexas.edu/~EWD/ewd02xx/EWD215.PDF

Apéndice

Ejercicios

practico1.pdf

practico2.pdf

practico3.pdf

practico4.pdf

Digestos

Digesto de Funciones de Listas y Propiedades

listas.pdf

Digesto de Cuantificadores y Cálculo Proposicional

digesto.pdf

Cuantificador de Conteo N

conteo.pdf

Digesto para la Programación Imperativa

imperativo.pdf

Videos

Dictado completo del teórico 2022:

https://www.youtube.com/playlist?list=PLSddYh_b7LP8a0mdrpx1-3M9S9Grnz1pQ

Contenido extra I: Clases prácticas y consultas

Pizarras 2021

Grabaciones 2021

Funcional / Cuantificación general (2021)

2021-08-26 Práctico (práctico 1)

2021-08-31 Práctico - Sala 2 (práctico 1, práctico 2: sum_cuad, iga)

2021-09-02 Práctico - Sala 2 (práctico 2: cuantos, busca, creciente)

2021-09-09 Práctico (práctico 2: prod_suf, cos, minimo)

2021-09-14 Práctico (sum_ant, esCuad)

2021-09-16 Práctico (segmentos de lista)

2021-09-23 Práctico (segmentos de lista)

2021-09-28 Práctico

Imperativo (2021)

2021-10-12 Práctico

2021-10-14 Práctico

2021-10-19 Práctico

2021-10-21 Práctico

2021-10-26 Práctico

2021-10-28 Práctico

2021-11-02 Práctico

2021-11-04 Práctico

2021-11-09 Práctico

2021-11-11 Práctico

Clases de consulta (2021)

2021-12-01 Consulta

2021-12-14 Consulta

Clases de consulta (2023)

CONSULTA ALGORITMOS 1 - 4/12/2023

CONSULTA ALGORITMOS 1 - 19/12/2023

CONSULTA ALGORITMOS 1 - 22/2/2024

Clases de consulta (2024)

CONSULTA ALGORITMOS 1 - 16/12/2024 (Recording)

Clases de consulta (2025)

CONSULTA ALGORITMOS 1 - 17/09/2025 (Recording)

CONSULTA ALGORITMOS 1 - 28/10/2025 (Recording)

CONSULTA ALGORITMOS 1 - 10/11/2025 (Recording)

Exámenes parciales (2016 – 2019)

https://wiki.cs.famaf.unc.edu.ar/doku.php?id=algo1:2019-2#examenes_anos_anteriores

- 2016 (2do cuat.):
  - 1er parcial
  - 2do parcial
  - Recuperatorio 1er parcial (con solución ej. 3)

        - Recuperatorio 2do parcial (con solución ej. 3)
    - 2017 (2do cuat.):
        - 1er parcial
        - 2do parcial
        - Recuperatorio 1er parcial
        - Recuperatorio 2do parcial
    - 2018 (2do cuat.):
        - 1er parcial
        - 2do parcial
        - Recuperatorio 1er parcial
        - Recuperatorio 2do parcial
    - 2019 (2do cuat.):
        - 1er parcial
        - 2do parcial: tema A, tema B
        - Recuperatorio 1er parcial
        - Recuperatorio 2do parcial

Solución 1er parcial turno tarde (2024)

EJERCICIO 1a.
Hago inducción en xs. Espero encontrar una definición de la forma:

f.[] ≐ ???
f.(x►xs) ≐ ???

Empiezo con el caso inductivo por las dudas haya que generalizar.

Caso inductivo:
  f.(x►xs)
= { especificación }
  ⟨ ∑ i : 0 ≤ i < #(x►xs) : (#(x►xs) - i) * (x►xs)!i ⟩
= { def. # y lógica }
  ⟨ ∑ i : i = 0 ∨ 1 ≤ i < #xs + 1 : (#xs + 1 - i) * (x►xs)!i ⟩
= { part. de rango y rango unitario }
  (#xs + 1 - 0) * (x►xs)!0 +⟨ ∑ i : 1 ≤ i < #xs + 1 : (#xs + 1 - i) * (x►xs)!i ⟩
= { def. ! y aritmética }
  (#xs + 1) * x +⟨ ∑ i : 1 ≤ i < #xs + 1 : (#xs + 1 - i) * (x►xs)!i ⟩
= { cambio de variable i --> i + 1 }
  (#xs + 1) * x +⟨ ∑ i : 1 ≤ i + 1 < #xs + 1 : (#xs + 1 - (i + 1)) * (x►xs)!(i + 1) ⟩
= { artimética en rango }
  (#xs + 1) * x +⟨ ∑ i : 0 ≤ i < #xs : (#xs + 1 - (i + 1)) * (x►xs)!(i + 1) ⟩
= { artimética y def ! en término }
  (#xs + 1) * x +⟨ ∑ i : 0 ≤ i < #xs : (#xs - i) * xs!i ⟩
= { H.I. }
  (#xs + 1) * x + f.xs

Pude llegar a la H.I. sin problemas!

Caso base:

f.[]
= { especificación }
⟨ ∑ i : 0 ≤ i < #[] : (#[] - i) * []!i ⟩
= { def. # }
⟨ ∑ i : 0 ≤ i < 0 : (#[] - i) * []!i ⟩
= { lógica y rango vacío }
0

Resultado final:
f.[] ≐ 0
f.(x►xs) ≐ (#xs + 1) * x + f.xs

EJERCICIO 1b.
f.[1,2,0,4,8] --> (#[2,0,4,8] + 1) * 1 + f.[2,0,4,8]
--> 5 * 1 + f.[2,0,4,8]
--> 5 * 1 + (#[0,4,8] + 1) * 2 + f.[0,4,8]
--> 5 * 1 + 4 * 2 + f.[0,4,8]
--> 5 * 1 + 4 * 2 + (#[4,8] + 1) * 0 + f.[4,8]
--> 5 * 1 + 4 * 2 + 3 * 0 + f.[4,8]
--> 5 * 1 + 4 * 2 + 3 * 0 + (#[8] + 1) * 4 + f.[8]
--> 5 * 1 + 4 * 2 + 3 * 0 + 2 * 4 + f.[8]
--> 5 * 1 + 4 * 2 + 3 * 0 + 2 * 4 + (#[] + 1) * 8 + f.[]
--> 5 * 1 + 4 * 2 + 3 * 0 + 2 * 4 + 1 * 8 + f.[]
--> 5 * 1 + 4 * 2 + 3 * 0 + 2 * 4 + 1 * 8 + 0
--> 5 + 8 + 0 + 8 + 8 + 0
--> 29

EJERCICIO 2.
Hago inducción en xs. Espero encontrar una definición de la forma:

P.[] ≐ ???
P.(x►xs) ≐ ???

Empiezo con el caso inductivo por las dudas haya que generalizar.

Caso inductivo:
P.(x►xs)
= { especificación }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) : sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { lógica (neutro ∧) }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ True : sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { lógica (3ro excluído) }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ (as = [] ∨ as ≠ []) : sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { lógica (distributividad) }
⟨ ∃ as, b, bs : (x►xs = as ++ (b►bs) ∧ as = []) ∨ (x►xs = as ++ (b►bs) ∧ as ≠ []) : sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { part. rango }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ as = [] : sum.as ≤ b ∧ sum.bs ≤ b ⟩

  ∨⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ as ≠ [] : sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { 2do cuantificador: cambio de variable as --> a►as }
  ... ∨ ⟨ ∃ a, as, b, bs : x►xs = (a►as) ++ (b►bs) ∧ a►as ≠ [] : sum.(a►as) ≤ b ∧ sum.bs ≤ b ⟩
= { def. ++ y propiedades de listas }
  ... ∨ ⟨ ∃ a, as, b, bs : x = a ∧ xs = as ++ (b►bs) : sum.(a►as) ≤ b ∧ sum.bs ≤ b ⟩
= { elim. variable a }
  ... ∨ ⟨ ∃ as, b, bs : xs = as ++ (b►bs) : sum.(x►as) ≤ b ∧ sum.bs ≤ b ⟩
= { def. sum }
  ... ∨ ⟨ ∃ as, b, bs : xs = as ++ (b►bs) : x + sum.as ≤ b ∧ sum.bs ≤ b ⟩

Me trabo. No puedo llegar a la H.I. Se cancela la derivación por inducción!
Debo generalizar. Especifico:

gP.n.xs = ⟨ ∃ as, b, bs : xs = as ++ (b►bs) : n + sum.as ≤ b ∧ sum.bs ≤ b ⟩

Ahora, gP generaliza a P, por lo que P se puede derivar de la siguiente manera:

  P.xs
= { especificación de P }
  ⟨ ∃ as, b, bs : xs = as ++ (b►bs) : sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { aritmética }
  ⟨ ∃ as, b, bs : xs = as ++ (b►bs) : 0 + sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { especificación de gP }
  gP.0.xs

Luego, podemos definir:

P.xs ≐ gP.0.xs

Falta derivar gP. Lo hago por inducción en xs.

Caso base:
  gP.[]
= { especificación }
  ⟨ ∃ as, b, bs : [] = as ++ (b►bs) : n + sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { prop. de listas }
  ⟨ ∃ as, b, bs : [] = as ∧ False : n + sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { lógica y RANGO VACIO! }
   False

Caso inductivo:
  gP.(x►xs)
= { mismos pasos que con P pero todo con el "n +" }
  ... ∨ ⟨ ∃ as, b, bs : xs = as ++ (b►bs) : n + x + sum.as ≤ b ∧ sum.bs ≤ b ⟩
= { H.I. }
  ... ∨ gP.(n+x).xs
= { retomo 1er cuantificador pero con el "n +" }

⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ as = [] : sum.as ≤ b ∧ sum.bs ≤ b ⟩
∨ gP.(n+x).xs
= { elim. variable as }
⟨ ∃ b, bs : x►xs = [] ++ (b►bs) : sum.[] ≤ b ∧ sum.bs ≤ b ⟩
∨ gP.(n+x).xs
= { prop. listas }
⟨ ∃ b, bs : x = b ∧ xs = bs : sum.[] ≤ b ∧ sum.bs ≤ b ⟩
∨ gP.(n+x).xs
= { rango unitario }
(sum.[] ≤ x ∧ sum.xs ≤ x) ∨ gP.(n+x).xs
= { def. sum }
(0 ≤ x ∧ sum.xs ≤ x) ∨ gP.(n+x).xs

Resultado final:
P.xs ≐ gP.0.xs

gP.[] ≐ False
gP.(x►xs) ≐ (0 ≤ x ∧ sum.xs ≤ x) ∨ gP.(n+x).xs

Solución 2do parcial turno tarde (2024)

Const N : Int, A : array [0, N) of Int;
Var res : Bool;
{ P: N ≥ 0 }
S
{ Q: res = ⟨ ∀ i : 0 ≤ i ≤ N : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩ }

a) A = [-2, 5, 7, 3]. N = 4
Rango: i ∈ {0, 1, 2, 3, 4}
Términos:
⟨ ∑ j : 0 ≤ j < 0 : A.j ⟩ < 2^0
∧ ⟨ ∑ j : 0 ≤ j < 1 : A.j ⟩ < 2^1
∧ ⟨ ∑ j : 0 ≤ j < 2 : A.j ⟩ < 2^2
∧ ⟨ ∑ j : 0 ≤ j < 3 : A.j ⟩ < 2^3
∧ ⟨ ∑ j : 0 ≤ j < 4 : A.j ⟩ < 2^4
≡ { resuelvo sumatorias y potencias }
0 < 1
∧ A.0 < 2
∧ A.0 + A.1 < 4
∧ A.0 + A.1 + A.2 < 8
∧ A.0 + A.1 + A.2 + A.3 < 16
≡ { resuelvo sumas }
0 < 1
∧ -2 < 2
∧ 3 < 4
∧ 10 < 8
∧ 13 < 16

≡ { resuelvo < }
  True
 ∧ True
 ∧ True
 ∧ False
 ∧ True
≡ { resuelvo ∧ }
 False

b) **Derivación:** Necesitamos un ciclo. Usamos la técnica de reemplazo de constante por variable. Obtenemos:

INV ≡ res = ⟨ ∀ i : 0 ≤ i ≤ pos : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩ ∧ 0 ≤ pos ≤ N
B ≡ pos < N

El programa tendrá la siguiente estructura:

Const N : Int, A : array [0, N) of Int;
Var res : Bool, pos : Int;
{ P }
 S_0
{ INV }
 do n < N →
  { INV ∧ B }
   S_1
  { INV }
 od
{ Q }

Cuerpo del ciclo: Probamos con S_1 de la forma **res, pos := E, pos + 1**. Suponemos INV ∧ B como hipótesis y vemos la wp:
 wp.(res, pos := E, pos + 1).INV
≡ { def. wp }
 E = ⟨ ∀ i : 0 ≤ i ≤ pos + 1 : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩ ∧ 0 ≤ pos + 1 ≤ N
≡ { vale por hip. 0 ≤ pos ≤ N y B (pos < N) }
 E = ⟨ ∀ i : 0 ≤ i ≤ pos + 1 : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩
≡ { lógica }
 E = ⟨ ∀ i : 0 ≤ i ≤ pos ∨ i = pos + 1 : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩
≡ { partición de rango y rango unitario }
 E = ⟨ ∀ i : 0 ≤ i ≤ pos : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩ ∧ ⟨ ∑ j : 0 ≤ j < pos + 1 : A.j ⟩ < $2^{(pos+1)}$
≡ { hipótesis }
 E = res ∧ ⟨ ∑ j : 0 ≤ j < pos + 1 : A.j ⟩ < $2^{(pos+1)}$
≡ { lógica, part. de rango y rango unitario }
 E = res ∧ ⟨ ∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos < $2^{(pos+1)}$
≡ { álgebra }
 E = res ∧ ⟨ ∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos < $2^{pos} * 2$

Acá nos trabamos pero podemos fortalecer:

INV' ≡ INV ∧ sum = ⟨ ∑ j : 0 ≤ j < pos : A.j ⟩ ∧ pow = 2^{pos}

Cuerpo del ciclo de nuevo: Ahora S_1 será de la forma **res, sum, pow, pos := E, F, G, pos + 1**. Suponemos INV' ∧ B como hipótesis y vemos la wp:

wp.(res, sum, pow, pos := E, F, G, pos + 1).INV'
≡ { def. wp }
E =⟨ ∀ … ⟩∧ 0 ≤ pos + 1 ≤ N ∧ F = ⟨ ∑ j : 0 ≤ j < pos + 1 : A.j ⟩ ∧ G = $2^{pos + 1}$
≡ { mismos pasos }
E = res ∧ ⟨ ∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos < 2^{pos} * 2 ∧ F = … ∧ G = …
≡ { hipótesis nuevas }
E = res ∧ sum + A.pos < pow * 2 ∧ F = … ∧ G = …
≡ { **elijo E = res ∧ sum + A.pos < pow * 2** }
F = ⟨ ∑ j : 0 ≤ j < pos + 1 : A.j ⟩ ∧ G = $2^{pos + 1}$
≡ { en F: lógica, part. de rango y rango unitario. en G: álgebra }
F = ⟨ ∑ j : 0 ≤ j < pos : A.j ⟩ + A.pos ∧ G = 2^{pos} * 2
≡ { hipótesis nuevas }
F = sum + A.pos ∧ G = pow * 2
≡ { elijo **F = sum + A.pos** y **G = pow * 2** }
True

Inicalización: S_0 será de la forma **res, sum, pow, pos := E, F, G, H**. Suponemos P como hipótesis y vemos la wp:

wp.(res, sum, pow, pos := E, F, G, H).INV'
≡ { def. wp }
E = ⟨ ∀ i : 0 ≤ i ≤ H : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩
∧ 0 ≤ H ≤ N ∧ F = ⟨ ∑ j : 0 ≤ j < H : A.j ⟩ ∧ G = 2^H
≡ { **elijo H = 0** }
E = ⟨ ∀ i : 0 ≤ i ≤ 0 : ⟨ ∑ j : 0 ≤ j < i : A.j ⟩ < 2^i ⟩
∧ 0 ≤ 0 ≤ N ∧ F = ⟨ ∑ j : 0 ≤ j < 0 : A.j ⟩ ∧ G = 2^0
≡ { rango unitario (i = 0) }
E = ⟨ ∑ j : 0 ≤ j < 0 : A.j ⟩ < 2^0
∧ 0 ≤ 0 ≤ N ∧ F = ⟨ ∑ j : 0 ≤ j < 0 : A.j ⟩ ∧ G = 2^0
≡ { rangos vacíos y álgebra }
E = 0 < 1 ∧ 0 ≤ 0 ≤ N ∧ F = 0 ∧ G = 1
≡ { elijo **E = True** , **F = 0** y **G = 1** }
0 ≤ 0 ≤ N
≡ { hipótesis }
True

Cota: La cota es t = N - pos.

1. Vale N - pos ≥ 0 ya que el invariante dice pos ≤ N.
2. Decrece con el cuerpo del ciclo ya que pos se incrementa en 1 y N es constante.

Resultado final:

Const N : Int, A : array [0, N) of Int;
Var res : Bool, pos, sum, pow : Int;

{ P }
  res, sum, pow, pos := True, 0, 1, 0;
  do pos < N →
      res, sum, pow, pos := res ∧ (sum + A.pos < pow * 2), sum + A.pos, pow * 2, pos + 1
  od
{ Q }

Variante: Otro fortalecimiento posible:

INV' ≡ INV ∧ sum = ⟨ ∑ j : 0 ≤ j < pos : A.j ⟩ ∧ pow = $2^{pos + 1}$

El resultado final en este caso es (los cambios en negrita y subrayado):

Const N : Int, A : array [0, N) of Int;
Var res : Bool, pos, sum, pow : Int;
{ P }
  res, sum, pow, pos := True, 0, **2**, 0;
  do pos < N →
      res, sum, pow, pos := res ∧ (sum + A.pos < **pow**), sum + A.pos, pow * 2, pos + 1
  od
{ Q }

c) Fortalecemos la guarda de la siguiente manera:

B' ≡ B ∧ res
   ≡ pos < N ∧ res

Debemos demostrar que INV' ∧ ¬ res ⇒ Q. Suponemos INV' ∧ ¬ res como hipótesis y vemos Q:
  res = ⟨ ∀ i : 0 ≤ i ≤ N : … ⟩
≡ { hip. ¬ res }
  False = ⟨ ∀ i : 0 ≤ i ≤ N : … ⟩
≡ { lógica }
  False = ⟨ ∀ i : 0 ≤ i ≤ pos ∨ pos < i ≤ N : … ⟩
≡ { part. de rango }
  False = ⟨ ∀ i : 0 ≤ i ≤ pos : … ⟩ ∧ ⟨ ∀ i : pos < i ≤ N : … ⟩
≡ { hipótesis INV }
  False = res ∧ ⟨ ∀ i : pos < i ≤ N : … ⟩
≡ { hip. ¬ res }
  False = False ∧ ⟨ ∀ i : pos < i ≤ N : … ⟩
≡ { absorbente y más lógica }
  True

Solución examen final 17/12/2024

Ejercicio 1:

p.xs = ⟨ ∃ as, b, bs : xs = as ++ (b► bs) : b = ⟨∑ i : 0 ≤ i < #bs ∧ (bs!i) mod 2 = 1 : bs!i ⟩ ⟩

a) Por inducción en xs.

Paso inductivo:

p.(x►xs)
= { esp. }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) : … ⟩
= { 3ro excluído }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ (as = [ ] ∨ as ≠ [ ]) : … ⟩
= { distrib. y part. de rango }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ as = [ ] : … ⟩ ∨
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ as ≠ [ ] : … ⟩
= { en el segundo cuantificador: cambio de variable as → a►as }
… ∨ ⟨ ∃ a, as, b, bs : x►xs = (a►as) ++ (b►bs) ∧ a►as ≠ [ ] :
b = ⟨∑ i : 0 ≤ i < #bs ∧ (bs!i) mod 2 = 1 : bs!i ⟩ ⟩
= { propiedades de listas }
… ∨ ⟨ ∃ a, as, b, bs : x = a ∧ xs = as ++ (b►bs) :
b = ⟨∑ i : 0 ≤ i < #bs ∧ (bs!i) mod 2 = 1 : bs!i ⟩ ⟩
= { eliminación de variable a }
… ∨ ⟨ ∃ as, b, bs : xs = as ++ (b►bs) : b = ⟨∑ i : 0 ≤ i < #bs ∧ (bs!i) mod 2 = 1 : bs!i ⟩ ⟩
= { Hipótesis Inductiva }
… ∨ p.xs
= { volvemos al primer cuantificador: }
⟨ ∃ as, b, bs : x►xs = as ++ (b►bs) ∧ as = [ ] : … ⟩ ∨ p.xs
= { eliminación de variable as }
⟨ ∃ b, bs : x►xs = [ ] ++ (b►bs) : b = ⟨∑ i : 0 ≤ i < #bs ∧ (bs!i) mod 2 = 1 : bs!i ⟩ ⟩ ∨ p.xs
= { propiedades de listas }
⟨ ∃ b, bs : x = b ∧ xs = bs : b = ⟨∑ i : 0 ≤ i < #bs ∧ (bs!i) mod 2 = 1 : bs!i ⟩ ⟩ ∨ p.xs
= { eliminación de variable (x) + rango unitario (xs) }
x = **⟨∑ i : 0 ≤ i < #xs ∧ (xs!i) mod 2 = 1 : xs!i ⟩ ⟩** ∨ p.xs
= { modularización!! introduzco nueva función especificada por:
sumImpar.xs = ⟨∑ i : 0 ≤ i < #xs ∧ (xs!i) mod 2 = 1 : xs!i ⟩
}
x = sumImpar.xs ∨ p.xs

Caso base:

p.[ ]
= { esp. }
⟨ ∃ as, b, bs : [ ] = as ++ (b► bs) : … ⟩
= { prop. listas }
⟨ ∃ as, b, bs : False : … ⟩
= { rango vacío }
False

Resultado parcial:

p.[ ] ≐ False
p.(x►xs) ≐ x = sumImpar.xs ∨ p.xs

Derivación de sumImpar: SE LAS DEBO

Resultado final:
p.[ ] ≐ False
p.(x►xs) ≐ x = sumImpar.xs ∨ p.xs

sumImpar.[ ] ≐ 0
sumImpar.(x►xs) ≐ (x mod 2 = 0 → sumImpar.xs
[] x mod 2 = 1 → x + sumImpar.xs
)

b) Con la especificación:

p.[4, 1, 0, 3] = (4 = 1 + 3) ∨ (1 = 3) ∨ (0 = 3) ∨ (3 = 0)
= True ∨ False ∨ False ∨ False
= True

p.[0] = (0 = 0)
= True

p.[2, 1] = (2 = 1) ∨ (1 = 0)
= False

Ejercicio 2:

Const N : Int, A : array[0, N) of Int;
Var r : Int;
{ P : N ≥ 0 }
S
{ Q : r =⟨ ∃ i : 0 ≤ i < N ∧ i mod 2 = 0 : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩ }

a) A = [7, -3, 0, 5, 2]

i ∈ {0, 2, 4}:
- i = 0: A.0 = 0 ≡ 7 = 0 ≡ False
- i = 2: A.2 = A.1 ≡ 0 = -3 ≡ False
- i = 3: A.4 = A.1 + A.3 ≡ 2 = (-3) + 5 ≡ True

Resultado final: False ∨ False ∨ True ≡ **True**.

b) Necesitamos un ciclo. Por reemplazo de constante por variable:

INV ≡ r =⟨ ∃ i : 0 ≤ i < pos ∧ i mod 2 = 0 : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
∧ 0 ≤ pos ≤ N

B ≡ pos ≠ N

Luego, INV ∧ ¬ B ⇒ Q.

Cuerpo del ciclo: Suponemos INV ∧ B y vemos la wp:

wp.(r, pos := E, pos + 1).INV
≡ { def. wp }
E =⟨ ∃ i : 0 ≤ i < pos + 1 ∧ i mod 2 = 0 : … ⟩
∧ 0 ≤ pos + 1 ≤ N
≡ { vale por hip. 0 ≤ pos + 1 ≤ N y pos ≠ N }
E =⟨ ∃ i : 0 ≤ i < pos + 1 ∧ i mod 2 = 0 : … ⟩
≡ { lógica }
E =⟨ ∃ i : (0 ≤ i < pos ∨ i = pos) ∧ i mod 2 = 0 : … ⟩
≡ { distrib. y part. de rango }
E =⟨ ∃ i : 0 ≤ i < pos ∧ i mod 2 = 0 : … ⟩
∨⟨ ∃ i : i = pos ∧ i mod 2 = 0 : … ⟩
≡ { Hipótesis }
E = r ∨⟨ ∃ i : i = pos ∧ i mod 2 = 0 : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
≡ { Leibniz }
E = r ∨⟨ ∃ i : i = pos ∧ pos mod 2 = 0 : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩

- Caso 1: pos mod 2 = 0
  ≡ { reemplazo }
  E = r ∨⟨ ∃ i : i = pos ∧ True : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
  ≡ { rango unitario }
  E = r ∨ A.pos = ⟨∑ j : 0 ≤ j < pos ∧ j mod 2 = 1 : A.j ⟩
  Acá me trabo! Debo fortalecer.

- Caso 2: pos mod 2 ≠ 0 (**o lo que es lo mismo: pos mod 2 = 1**)
  ≡ { reemplazo }
  E = r ∨⟨ ∃ i : i = pos ∧ False : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
  ≡ { lógica y rango vacío }
  E = r ∨ False
  ≡ { elijo E = r }
  True

Fortalecimiento:
INV' ≡ INV ∧ sum = ⟨∑ j : 0 ≤ j < pos ∧ j mod 2 = 1 : A.j ⟩

Cuerpo del ciclo de nuevo: Suponemos INV´ ∧ B y vemos la wp:

wp.(r, sum, pos := E, F, pos + 1).INV´
≡ { def. wp }
E =⟨ ∃ i : 0 ≤ i < pos + 1 ∧ i mod 2 = 0 : … ⟩
∧ 0 ≤ pos + 1 ≤ N
∧ F = ⟨∑ j : 0 ≤ j < pos + 1 ∧ j mod 2 = 1 : A.j ⟩
≡ { mismos pasos que antes hasta Leibniz }
E = r ∨⟨ ∃ i : i = pos ∧ pos mod 2 = 0 : … ⟩
∧ F = ⟨∑ j : 0 ≤ j < pos + 1 ∧ j mod 2 = 1 : A.j ⟩
≡ { lógica y partición de rango }

E = r ∨ ⟨ ∃ i : i = pos ∧ pos mod 2 = 0 : … ⟩
∧ F = ⟨∑ j : 0 ≤ j < pos ∧ j mod 2 = 1 : A.j ⟩ + ⟨∑ j : j = pos ∧ j mod 2 = 1 : A.j ⟩
≡ { hipótesis }
E = r ∨ ⟨ ∃ i : i = pos ∧ pos mod 2 = 0 : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
∧ F = sum + ⟨∑ j : j = pos ∧ j mod 2 = 1 : A.j ⟩
≡ { Leibniz }
E = r ∨ ⟨ ∃ i : i = pos ∧ pos mod 2 = 0 : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
∧ F = sum + ⟨∑ j : j = pos ∧ pos mod 2 = 1 : A.j ⟩

- Caso 1: pos mod 2 = 0:
  ≡ { “pos mod 2 = 0” es True y “pos mod 2 = 1” es False}
  E = r ∨ ⟨ ∃ i : i = pos ∧ True : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
  ∧ F = sum + ⟨∑ j : j = pos ∧ False : A.j ⟩
  ≡ { rango unitario en el 1ro, rango vacío en el 2do }
  E = r ∨ A.pos = ⟨∑ j : 0 ≤ j < pos ∧ j mod 2 = 1 : A.j ⟩
  ∧ F = sum + 0
  ≡ { hipótesis, neutro + }
  E = r ∨ A.pos = sum
  ∧ F = sum
  ≡ { elijo **E = r ∨ A.pos = sum** y **F = sum** }
  True
- Caso 2: pos mod 2 ≠ 0:
  ≡ { “pos mod 2 = 0” es False y “pos mod 2 = 1” es True }
  E = r ∨ ⟨ ∃ i : i = pos ∧ False : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩
  ∧ F = sum + ⟨∑ j : j = pos ∧ True : A.j ⟩
  ≡ { rango vacío en el 1ro, rango unitario en el 2do }
  E = r ∨ False
  ∧ F = sum + A.pos
  ≡ { elijo **E = r** y **F = sum + A.pos** }
  True

Inicialización: SE LAS DEBO. DA:

r, sum, pos := False, 0, 0

Resultado final:

```
Const N : Int, A : array[0, N) of Int;
Var r : Int;
{ P : N ≥ 0 }
  r, sum, pos := False, 0, 0;
  do n ≠ N →
    if pos mod 2 = 0→
          r, sum, pos := r ∨ (A.pos = sum), sum, pos + 1
    [] pos mod 2 = 1→
          r, sum, pos := r, sum + A.pos, pos + 1
    fi
  od
{ Q : r =⟨ ∃ i : 0 ≤ i < N ∧ i mod 2 = 0 : A.i = ⟨∑ j : 0 ≤ j < i ∧ j mod 2 = 1 : A.j ⟩ ⟩ }
```

c) Nueva guarda: B’ ≡ B ∧ ¬ r.

Demostración: Probar que INV ∧ ¬(¬r) ⇒ Q. SE LAS DEBO.

Calendario sugerido

1ra clase: 2022-08-23
Preliminares
Cuantificación General

2da clase: 2022-08-23
Cuantificación General
- Repaso: Sintaxis
- Repaso: Lectura operacional
- Ejemplos
- A1: Rango vacío
- A2: Rango unitario
- A3: Partición de rango

3ra clase: 2022-08-30
Cuantificación General
- Repaso
- A4: Regla del término
- A5: Término constante
- A6: Distributividad

4ta clase: 2022-09-01
Cuantificación General
- Repaso
- A7: Anidado
- T1: Cambio de variable
- T2: Eliminación de variable
- T3: Rango unitario y condición
- T11: Leibniz 2
- A8: Conteo

5ta clase: 2022-09-06
Cuantificación General
- Repaso
- Reglas adicionales para el conteo
- Ejercicio 15b

Programación funcional
- Introducción
  - … hasta Demostración

6ta clase: 2022-09-08

Programación funcional
- Repaso: Introducción
  - Derivación
  - Testing
  - El lenguaje de programación funcional

7ma clase: 2022-09-13
Programación funcional
- Repaso
- Derivación
- Modularización

8va clase: 2022-09-15
Programación funcional
- Repaso: Modularización
- Esquemas de inducción
- Generalización

9na clase: 2022-09-22
Programación funcional
- Repaso: Generalización
- Segmentos de lista

10ma clase: 1ER PARCIAL

11va clase: 2022-09-29
Programación imperativa
- Introducción
  - … hasta "Ejemplo: contar múltiplos de 6"

12va clase: 2022-10-04
Programación imperativa
- Repaso
- Arreglos
- Lenguaje completo
- Anotaciones de programa
- Especificación

13va clase: 2022-10-06
Programación imperativa
- Repaso
- Especificación
- Ternas de Hoare

14va clase: 2022-10-11
Programación imperativa
- Repaso
- Precondición más débil (weakest precondition)

15va clase: 2022-10-18
Programación imperativa
  Repaso
  Derivación de programas imperativos
    Skip
    Asignación

16va clase: 2022-10-20
Programación imperativa
  Repaso
  Derivación de programas imperativos
    Condicional (if)
    Secuenciación (;)

17va clase: 2022-10-22
Programación imperativa
  Repaso
  Derivación de programas imperativos
    Ciclo / Repetición (do)

18va clase: 2022-10-27
Programación imperativa
  Repaso
  Derivación de ciclos: Técnicas para encontrar invariantes
    Idea general

19va clase: 2022-11-01
Programación imperativa
  Repaso
  Derivación de ciclos: Técnicas para encontrar invariantes
    1ra técnica: Tomar términos de una conjunción
    2da técnica: Reemplazo de constantes por variables
                  … hasta 2do ejemplo

20va clase: 2022-11-03
Programación imperativa
  Repaso
  Derivación de ciclos: Técnicas para encontrar invariantes
    2da técnica: Reemplazo de constantes por variables
      Ejemplos

21va clase: 2022-11-08
Programación imperativa
  Repaso
  Derivación de ciclos: Técnicas para encontrar invariantes
    Ciclos anidados
      Ejemplos

22va clase: 2022-11-10
Programación imperativa
- Repaso
- Derivación de ciclos: Técnicas para encontrar invariantes
  - Fortalecimiento de invariantes
    - Ejemplos

23va clase: 2DO PARCIAL

24va clase:
Programación imperativa
- Repaso
- Terminación de ciclos: Función de cota
- Problemas de bordes