LENGUAJES Y COMPILADORES

DANIEL FRIDLENDER Y HÉCTOR GRAMAGLIA

1. Semántica de un lenguaje: primera aproximación

El establecer el significado de las frases de un lenguaje de programación es un problema de múltiples aristas en tanto puede tener variados objetivos, que van desde la comprensión humana hasta la necesidad de que una máquina los pueda interpretar o traducir a una secuencia de instrucciones ejecutables. Del significado trata un manual de usuario, en tanto provee una descripción intuitiva de una acción o una denotación, y también un intérprete, un compilador, o una herramienta teórica destinada a desentrañar principios básicos de diseño. Todas constituyen vertientes de significación que responden a distintos intereses y usos de los lenguajes de programación.

Como una primera aproximación, podemos destacar cuatro maneras diferentes de dar significado a los lenguajes de programación. Las primeras tres de ellas ponen énfasis en el sentido dinámico finito, en tanto la cuarta se refiere a un significado ideal en un universo formado por objetos matemáticos.

- *informal, intuitiva*: explica el funcionamiento de los programas a través de frases comprensibles en el lenguaje natural (ejemplo: manuales, documentación tipo javadoc)
- *axiomática*: explica el sentido dinámico de manera implícita, estableciendo en el marco de una lógica qué propiedades son asignables a una determinada frase del lenguaje, estableciendo así una manera de razonar sobre programas (ejemplo: $\{P/v \leftarrow e\}\ v := e\ \{P\}$)
- *operacional*: explica el sentido dinámico de manera explícita, diciendo de que manera se ejecuta un programa (ejemplo: intérprete)
- *denotacional*: asigna a cada programa un significado estático en un universo semántico especialmente definido para representar los fenómenos que el lenguaje describe (este universo podría ser el universo de las frases de otro lenguaje, por ejemplo un compilador)

En este curso ocupa un lugar destacado el último enfoque, y en menor medida el anteúltimo. A partir del desarrollo de la Teoría de Dominios la semántica denotacional adquiere un relevancia especial, no sólo por tratarse de objetos matemáticos perfectamente definidos en el contexto de una teoría particular, sino además porque comienza a ser utilizada como

Las siguientes personas señalaron algunos errores: Javier Mansilla, Álvaro Roy Schachner.

la definición del lenguaje y luego, si se proponen otras semánticas (operacional, axiomática), se las demuestra correctas con respecto a dicha definición. La semántica operacional también ocupará un lugar importante por ser una manera formal y la vez intuitiva de aproximarse por primera vez al significado de un lenguaje, poniéndo énfasis en el auténtico sentido de sus construcciones.

Para atribuir un significado denotacional al lenguaje debemos primero fijar nuestro *dominio semántico*, o sea el universo de objetos matemáticos que constituirán los significado de las frases. Luego debemos definir una función que asigna a cada frase un significado. Al utilizar el concepto matemático de función nos aseguramos que el significado de cada frase será único.

Pero para el caso de los lenguajes de programación que nos interesan, tanto la definición del dominio semántico como la definición de la función semántica requieren el uso de herramientas matemáticas no triviales. Antes de abordar el estudio de los lenguajes propiamente dichos, vamos a abordar el estudio de estas herramientas. En las secciones siguientes trataremos:

- Herramientas para dar la sintaxis del lenguaje, es decir una determinación del conjunto de frases que serán consideradas prográmas válidos (sección 2).
- El problema de la buena definición del dominio semántico y la función semántica (sección 3).
- El manejo de variables, la ligadura y los problemas de captura (secciones 4, y 5).
- Herramientas matemáticas para caracterizar los objetos definidos mediante recursión (próximo apunte).

2. Gramáticas abstractas: la definición del lenguaje

La herramienta por excelencia para la definición de lenguajes formales es la noción de gramática. Para nuestros propósitos la noción de gramática convencional no es del todo adecuada, porque los lenguajes que estudiaremos sufren del problema de la *ambiguedad*. Consideremos por ejemplo el siguiente lenguaje de expresiones aritméticas:

$$\begin{array}{rcl} \langle intexp \rangle & ::= & 0 \mid 1 \mid 2 \mid ... \\ & & -\langle intexp \rangle \mid \\ & & \langle intexp \rangle + \langle intexp \rangle \mid \\ & & \langle intexp \rangle * \langle intexp \rangle \end{array}$$

Observemos ejemplos de frases generadas por esta gramática:

142
-15
-15+3
2+3+4
2*-4

La gramátíca es ambigua en el siguiente sentido: algunas frases admiten diferentes maneras de generarse. Tal es el caso de 2+3+4:

(1) $\langle intexp\rangle$
$\rightarrow \langle intexp\rangle + \langle intexp\rangle$
$\rightarrow \langle intexp\rangle + 4$
$\rightarrow \langle intexp\rangle + \langle intexp\rangle + 4$
$\rightarrow \langle intexp\rangle + 3 + 4$
$\rightarrow 2 + 3 + 4$

(2) $\langle intexp\rangle$
$\rightarrow \langle intexp\rangle + \langle intexp\rangle$
$\rightarrow 2 + \langle intexp\rangle$
$\rightarrow 2 + \langle intexp\rangle + \langle intexp\rangle$
$\rightarrow 2 + 3 + \langle intexp\rangle$
$\rightarrow 2 + 3 + 4$

(1) se corresponde intuitivamente con asociar a izquierda, es decir $(2 + 3) + 4$, mientras que (2) con asociar a derecha, es decir $2 + (3 + 4)$. Lo cierto es que ninguna de estas dos frases puede generarse, porque los paréntesis no están entre los símbolos terminales de la gramática.

Pregunta: ¿con cuáles de las otras frases mencionadas más arriba ocurre lo mismo?

El problema podría resolverse agregando paréntesis y desambiguando la gramática. Por ejemplo, cambiando la gramática por la siguiente:

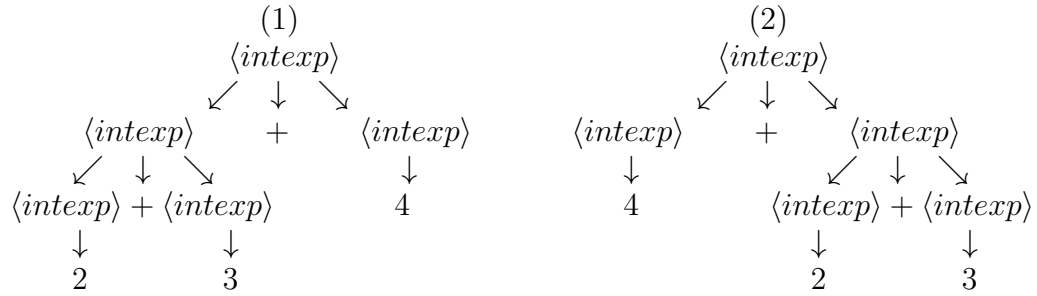
$$\begin{array}{lcl} \langle intexp\rangle & ::= & \langle intexp\rangle + \langle termexp\rangle | \langle termexp\rangle \\ \langle termexp\rangle & ::= & \langle groundexp\rangle | - \langle termexp\rangle \\ \langle groundexp\rangle & ::= & 0|1|2|....|(\langle intexp\rangle) \end{array}$$

donde queda claro que el + asocia a izquierda y que el menos tiene mayor precedencia. Esta gramática dice cómo se escriben concretamente las frases del lenguaje. Podríamos llamarla **gramática concreta**, y a las frases que genera, **frases concretas**. Podríamos decir que define la **sintaxis concreta** del lenguaje.

La gramática concreta resulta más complicada que la que dimos anteriormente. Nos obligaría a fijar detalles de la sintaxis del lenguaje que son irrelevantes para nuestros propósitos, y que de hecho cada lenguaje de programación lo resuelve de una manera distinta, constituyendo una molestia a la hora de concentrarse en lo sustancial del lenguaje (que en la gramática anterior era evidente): que el mini-lenguaje tiene constantes no negativas, un operador unario (-) y un operador binario (+). Por esta razón, se prefiere la gramática que se dió en primer lugar, a la que llamaremos **gramática abstracta**, en parte porque no expresa detalles de cómo se escriben las expresiones (asociatividades, precedencias, paréntesis) sino que expresa qué construcciones tiene el lenguaje, cuál es la estructura de las frases que hay, cuáles son las subfrases.

Pero debemos dejar sentado, justamente por el problema de ambigüedad, que la gramática no pretende describir la notación exacta o concreta, sólo la estructura de las frases, dice cuáles son las construcciones que hay en el lenguaje. No interesa cómo se escriben las frases sino qué frases hay. Diremos que es una *gramática abstracta*, que describe la *sintaxis abstracta* y determina *frases abstractas*. Trabajar a este nivel de abstracción es muy conveniente ya que nos permite desentendernos de detalles propios de la notación concreta que no tienen interés cuando se trata de dar significado a las frases.

En el ejemplo anterior, la estructura abstracta diferente se pone de manifiesto al representar cada derivación a través de un *árbol sintáctico*:



El libro de Reynolds (a partir de ahora, "el libro") hace un tratamiento detallado del significado preciso de gramática abstracta. Es muy interesante y lectura recomendada para entender esto con precisión (no es imprescindible para comprender la materia).

Una vez aceptado que trabajaremos con gramáticas abstractas, surge el problema de la notación que utilizaremos en este texto (y en la clase) para referirnos a una producción particular de tal gramática (por ejemplo la (1)). En las pocas ocasiones en que nos resulte necesario precisar notación específica resolveremos los problemas de ambigüedad mencionados estableciendo precedencias, stopping symbols, y utilizando los paréntesis que sean necesarios. Aunque resulte reiterativo aclaramos que tales símbolos no serán parte de la gramática en cuestión, sino sólo convenciones de notación que nos permitirán entender de qué producción de la gramática abstracta estamos hablando.

3. Función semántica

Una vez establecido el universo de frases abstractas que constituyen un lenguaje, dar una semántica denotacional implica definir una función que a cada frase abstracta le asigna una denotación en un dominio determinado. Cuando hablamos de lenguajes de programación no triviales, la definición de la función semántica presenta ciertas dificultades. Es necesario recurrir a herramientas matemáticas más complejas no sólo para la construcción de los dominios semánticos, sino además para la definición misma de la función. El objetivo es garantizar ciertas propiedades tanto de la semántica como de la definición de la misma. Aunque es difícil explicar ahora la importancia de "garantizar propiedades", en términos generales podemos decir que tienen que ver con

- que la semántica denotacional sea una herramienta que aporte claridad conceptual, y no que sea un instrumento de traducir algo poco comprensible en otra cosa incomprensible,
- que la semántica denotacional sea un referente con la cuál comparar otras posibles semánticas,
- qué se puedan utilizar resultados típicos de la teoría de lenguajes para estudiar las características del lenguaje en cuestión.

La herramienta matemática adecuada para definir la función semántica en los lenguajes que vamos a estudiar es la recursión. La misma ya fue utilizada en Introducción a la Lógica y la Computación para definir funciones sobre las proposiciones. El lector recordará que

disponíamos de un teorema de "buena definición", es decir, un teorema que garantizaba que si las definiciones recursivas sobre las proposiciones tenían un formato determinado, entonces eran buenas. De la misma manera utilizaremos recursión para definir funciones sobre los programas (en particular la semántica), fijando criterios estrictos sobre la manera en que tales definiciones deben hacerse.

Volvamos al lenguaje simple de expresiones enteras:

$$\begin{array}{rcl} \langle intexp \rangle & ::= & 0 \mid 1 \mid 2 \mid ... \\ & & -\langle intexp \rangle \mid \\ & & \langle intexp \rangle + \langle intexp \rangle \mid \\ & & \langle intexp \rangle * \langle intexp \rangle \end{array}$$

Una vez fijado $\mathbf{Z}$ como dominio semántico, definimos

$$[\![\_]\!] : \langle intexp \rangle \to \mathbf{Z} \qquad (*)$$

a través de las siguientes ecuaciones, llamadas *ecuaciones semánticas*:

$$\begin{array}{rcl} [\![0]\!] & = & 0 \\ [\![1]\!] & = & 1 \\ & \vdots & \\ [\![-e]\!] & = & -[\![e]\!] \\ [\![e + e']\!] & = & [\![e]\!] + [\![e']\!] \\ [\![e * e']\!] & = & [\![e]\!] * [\![e']\!] \end{array}$$

No hay duda de que la definición provee un mecanismo para calcular el significado de cada frase, por ejemplo:

$$\begin{array}{rcl} [\![-(5+2)]\!] & = & -[\![5+2]\!] \\ & = & -7 \\ [\![-(5+2)+10]\!] & = & [\![-(5+2)]\!] + [\![10]\!] \\ & = & -7 + 10 \\ & = & 3 \end{array}$$

Lo primero que observamos en esta secuencia de ecuaciones (que aspiramos a que realmente defina un función del tipo (*)), es que los mismos símbolos aparecen con dos sentidos distintos. El 0 de la izquierda es una frase del lenguaje de las expresiones enteras, y el de la derecha es el objeto cero perteneciente a los números enteros. Lo mismo ocurre con el $1, .., +, *, -$. Utilizamos esta aparente circularidad para no multiplicar los símbolos, lo que resultaría una molestia, debido a que tranquilamente desde el contexto podemos saber la naturaleza del objeto en cuestión. No hay circularidad en la definición ya que en un caso se trata de un operador del lenguaje y en otro de un objeto o función del dominio semántico. Esta aparente circularidad tiene un nombre: *metacircularidad*.

Otro aspecto a remarcar es la presencia de los símbolos e y e'. Cada una de las tres últimas ecuaciones representa, en realidad, infinitas ecuaciones. Por ejemplo, $[\![-e]\!] = -[\![e]\!]$ establece una propiedad que vale cualquiera sea la expresión e. Pero e NO es una expresión, sólo es un objeto que representa cualquier expresión del lenguaje. Es una variable. Pero NO es

una variable del lenguaje (revisemos la gramática para comprobar que no hay variables en el lenguaje, sólo constantes y operadores). Es una variable del *metalenguaje* que utilizamos para escribir una ecuación en vez de infinitas ecuaciones, una para cada expresión. A estas variables del metalenguaje se las llama *metavariables*.

La primer ecuación, en cambio, fue escrita sólo para la frase 0, y los puntos suspensivos expresan que hay una ecuación como esa para cada número natural. Podríamos escribir $[\![n]\!] = n$ para cada n. Pero acá estaríamos abusando de la notación, ya que el primer n es una metavariable para una frase y el segundo es una metavariable para un número natural. La metavariable n no representa en ambos lugares exactamente lo mismo. Esto suele resolverse escribiendo $[\![\lfloor n \rfloor]\!] = n$ para cada n, donde $\lfloor n \rfloor$ es la manera de escribir el número natural n en el lenguaje.

El metalenguaje que se utilizará a todo lo largo de la materia es la teoría de conjuntos habitual de la matemática. Por el permanente uso de funciones matemáticas, muchas de las definiciones serán fácilmente traducibles a lenguajes de programación funcionales. Una diferencia importante entre el metalenguaje y, por ejemplo, Haskell, es que en el metalenguaje todas las funciones son totales.

Nos hemos referidos a dos aspectos de las ecuaciones semánticas, la metacircularidad y el uso de metavariables. Veamos ahora al aspecto principal: en que medida estas ecuaciones definen una función de tipo (*).

3.1. Composicionalidad, dirección por sintaxis. El lector seguramente aceptará que las ecuaciones de arriba definen un único significado para cada frase de tipo $\langle intexp \rangle$. Pero ¿cuáles son las características de tales ecuaciones que garantizan este significado único?

Se puede demostrar (fuera del alcance de este curso) que si se respetan ciertos criterios sintácticos para las ecuaciones, seguro que hay significado único. Cuando hablamos de ecuaciones 'semánticas' estamos dando a entender que siguen algún criterio que garantiza existencia y unicidad. Nosotro adoptaremos el siguiente estilo de definición para garantizar esto.

Un conjunto de ecuaciones es *dirigido por sintaxis* cuando se satisfacen las siguientes condiciones:

- hay una ecuación por cada producción de la gramática abstracta
- cada ecuación que expresa el significado de una frase compuesta, lo hace puramente en función de los significados de sus subfrases inmediatas

El lector podrá comprobar que las ecuaciones dadas para $\langle intexp \rangle$ satisfacen estas condiciones. De esta manera estaremos garantizando la existencia y unicidad del significado.

Aunque este estilo de definición garantiza ciertas propiedades deseables, podemos hablar de propiedades "buenas" de la semántica sin necesidad de que las mismas estén garantizadas mediante un conjunto de ecuaciones dirigidas por sintaxis. Tal es el caso de la *composicionalidad*, que es una propiedad atribuible a las funciones semánticas, independientemente de como estas están dadas.

Se dice que una semántica es *composicional*, cuando el significado de una frase no depende de ninguna propiedad de sus subfrases, salvo de sus significados. Composicionalidad es muy importante ya que implica que podemos reemplazar una subfrase e_0 de e por otra de igual significado que e_0 sin alterar el significado de la frase e.

Composicionalidad no es lo mismo que dirección por sintaxis: composicionalidad habla de una propiedad de la semántica, mientras que dirección por sintaxis habla de la forma en que se definió dicha semántica. Pero por otro lado *dirección por sintaxis garantiza composicionalidad*.

Concluyendo: dirección por sintaxis garantiza existencia y unicidad del significado y también garantiza composicionalidad, todas propiedades deseables. Por ello, insistiremos en que nuestras ecuaciones sean dirigidas por sintaxis.

4. Variables y ligadura

En cualquier lenguaje de programación interesante está presente el concepto de variable, como un mecanismo elemental de abstracción sobre los objetos que el programa maneja. Este concepto obliga a revisar los dominios semánticos, ya que para dar semántica a (por ejemplo) $x + 3$, debemos previamente saber que valor se le "asigna" a x. En los primeros lenguajes que vamos a abordar, tal asignación se efectúa a través del concepto de *estado*, entendido en este contexto como una función que le asigna valores a las variables. [1]

Vamos a abordar el estudia de las problemáticas relacionadas con los conceptos de variable y ligadura a través del Cálculo de Predicados. La elección de este lenguaje (que no es un lenguaje de programación) es por consistencia con el Reynolds, y se fundamenta en el hecho de poder aprovechar la familiaridad que el lector tiene en este tópico, para poder concentrarnos en los asuntos específicos del manejo sintáctico.

4.1. Los conceptos de variable y ligadura. Abajo transcribimos la gramática abstracta. Los puntos interesantes en contraste con la gramática de $\langle intexp \rangle$ que vimos en la sección anterior son:

- Disponemos ahora de 3 tipos de frases, las cuales por supuesto tendrán 3 dominios distintos de significado.
- $\langle var \rangle$ carece de producciones. No especificaremos qué es el conjunto de frases $\langle var \rangle$, pero asumimos que es un conjunto infinito de símbolos. Usualmente utilizaremos las metavariables u, v, w, z para referirnos a elementos de $\langle var \rangle$.
- Aparecen construcciones (para todo, existe) que poseen subfrases de distinto tipo.

[1] Existen dos conceptos en la teoría de lenguajes de programación mediante los cuales se efectúa la tarea de asignar valores a entidades sintácticas: *el estado y el entorno*. El último se concibe inicialmente en el contexto de los lenguajes funcionales, en donde prevalece una forma estática de toma de valor, mientras que el primero es un concepto específico de los lenguajes imperativos, en donde los valores cambian de forma dinámica. El lograr determinar esto conceptos como independientes fue uno de los pasos fundamentales del desarrollo de la teoría de lenguajes de programación en los años 70.

Gramática abstracta para el lenguaje de los predicados

$\langle intexp \rangle ::=$
- $0 \mid 1 \mid 2 \mid ...$
- $\langle var \rangle$
- $-\langle intexp \rangle$
- $\langle intexp \rangle + \langle intexp \rangle$
- $\langle intexp \rangle * \langle intexp \rangle$
- $\langle intexp \rangle - \langle intexp \rangle$
- $\langle intexp \rangle \,/\, \langle intexp \rangle$
- $\langle intexp \rangle \;\%\; \langle intexp \rangle$

$\langle assert \rangle ::=$
- $\mathbf{true} \mid \mathbf{false}$
- $\langle intexp \rangle = \langle intexp \rangle$
- $\langle intexp \rangle < \langle intexp \rangle$
- $\langle intexp \rangle \leq \langle intexp \rangle$
- $\langle intexp \rangle > \langle intexp \rangle$
- $\langle intexp \rangle \geq \langle intexp \rangle$
- $\neg \langle assert \rangle$
- $\langle assert \rangle \vee \langle assert \rangle$
- $\langle assert \rangle \wedge \langle assert \rangle$
- $\exists \langle var \rangle \,.\, \langle assert \rangle$
- $\forall \langle var \rangle \,.\, \langle assert \rangle$

Utilizaremos las convenciones ususales de precedencia de operadores aritméticos, sumadas a la convención de que el alcance de un cuantificador $\forall$ o $\exists$ se extiende hasta el final de la frase, o hasta la aparición de un paréntesis que cierra cuyo alcance contiene al cuantificador.

Definimos el *conjunto de estados posibles* Σ como la familia de todas las funciones totales de $\langle var \rangle$ en $\mathbf{Z}$:

$$\Sigma \;=\; \langle var \rangle \to \mathbf{Z}$$

Podemos decir que el significado de una expresión entera e será una función que dependiendo del estado σ dará un entero que es el valor de e en σ; es decir, una función de Σ en $\mathbf{Z}$:

$$\begin{array}{lcll} [\![\_]\!]^{intexp} & : & \langle intexp \rangle \to \Sigma \to \mathbf{Z} & \\ [\![e]\!]^{intexp} & : & \Sigma \to \mathbf{Z} & \text{(para cualquier expresión } e\text{)} \end{array}$$

Las ecuaciones semánticas para las expresiones enteras fueron dadas (parcialmente) en la sección anterior.

Antes de dar las ecuaciones semánticas correspondientes a las expresiones booleanas, definimos la siguiente operación sobre estados. Sea $\sigma \in \Sigma$ un estado, v una variable y n un entero. Entonces $[\sigma | v : n]$ es un estado que coincide con σ en todas las variables salvo posiblemente en v, donde este nuevo estado tiene asignado n. En símbolos:

$$[\sigma | v : n]w \quad = \quad \text{si } (w = v) \text{ entonces } n \text{ sino } \sigma w$$

Esta misma notación se usa a lo largo de la materia cada vez que se quiere modificar una función en uno solo de sus posibles argumentos. Además, se puede iterar: $[[\sigma | v : n] | w : m]$ es el estado que cuando se aplica a w da m, cuando se aplica a v (suponiendo que $v \neq w$) da n, y cuando se aplica a cualquier otra variable u da σu. Para simplificar se escribe $[\sigma | v_1 : n_1 | ... | v_k : n_k]$ en lugar de $[[...[\sigma | v_1 : n_1] | ...] | v_k : n_k]$.

Por ejemplo, si f es una función real, la función $[f|\pi : 1|4 : 2|6 : 3]$ es idéntica a la función f salvo en los puntos $\pi, 4$ y 6. Observar lo que ocurre si modifico esta última función en 4, recomponiendo su valor original $f(4)$:

$$\begin{array}{rcl} [[f|\pi : 1|4 : 2|6 : 3] \mid 4 : f(4)] & = & [f|\pi : 1|4 : 2|6 : 3|4 : f(4)] \\ & = & [f|\pi : 1|4 : f(4)|6 : 3] \\ & = & [f|\pi : 1|6 : 3] \end{array}$$

Ahora sí, la función y las ecuaciones semánticas más representativas correspondientes a las expresiones booleanas.

$$\begin{array}{lcll} [\![\_]\!]^{assert} & : & \langle assert \rangle \to \Sigma \to \{V, F\} & \\ [\![p]\!]^{assert} & : & \Sigma \to \{V, F\} & \text{(para cualquier predicado } p\text{)} \end{array}$$

Notar que son dos funciones, una para expresiones enteras $[\![\_]\!]^{intexp}$ y otra para predicados $[\![\_]\!]^{assert}$, pero para no recargar la notación usaremos indistintamente la notación $[\![\_]\!]$ para ambas.

$$\begin{array}{lcl} [\![\mathbf{true}]\!]\sigma & = & V \\ [\![\mathbf{false}]\!]\sigma & = & F \\ [\![e = e']\!]\sigma & = & ([\![e]\!]\sigma = [\![e']\!]\sigma) \\ & \vdots & \\ [\![p \wedge q]\!]\sigma & = & ([\![p]\!]\sigma \wedge [\![q]\!]\sigma) \\ & \vdots & \\ [\![\forall v.p]\!]\sigma & = & (\forall n \in \mathbf{Z}.\ [\![p]\!][\sigma|v : n]) \\ [\![\exists v.p]\!]\sigma & = & (\exists n \in \mathbf{Z}.\ [\![p]\!][\sigma|v : n]) \end{array}$$

Es importante observar que la definición es dirigida por sintaxis. Entonces definen el significado de manera única y la semántica resultante es composicional.

Por otro lado, la metacircularidad abunda en esta definición. Acá debemos cuidarnos de no trasladar al lenguaje los vicios del metalenguaje, por ejemplo los problemas relacionados con la división por 0. En esta primera aproximación a la semántica denotacional vamos a asumir que todas las funciones son totales. En particular, la división está siempre definida (como función del dominio semántico), incluso si el divisor es 0. Más adelante abordaremos la manera en que los lenguajes gestionan situaciones de error o no definición.

Ejemplos:

$$
\begin{array}{lcl}
[\![\mathbf{true}]\!]\sigma & = & V \\
[\![\mathsf{x}+0=\mathsf{x}]\!]\sigma & = & ([\![\mathsf{x}+0]\!]\sigma = [\![\mathsf{x}]\!]\sigma) \\
 & = & ([\![\mathsf{x}]\!]\sigma + [\![0]\!]\sigma = \sigma\mathsf{x}) \\
 & = & (\sigma\mathsf{x}+0=\sigma\mathsf{x}) \\
 & = & V \\
[\![\forall \mathsf{x}.\ \mathsf{x}+0=\mathsf{x}]\!]\sigma & = & \forall n \in \mathbf{Z}.\ [\![\mathsf{x}+0=\mathsf{x}]\!][\sigma|\mathsf{x}:n] \\
 & = & \forall n \in \mathbf{Z}.\ V \\
 & = & V
\end{array}
$$

4.2. Variables y metavariables. Ahora que tenemos variables en el lenguaje, es oportuno repasar la noción de metavariable: es una variable del metalenguaje. En las ecuaciones que definen la semántica, ejemplos de metavariables son e y e' (corren sobre expresiones enteras), p y q (corren sobre los predicados), n (corre sobre enteros) y σ (corre sobre estados). Incluso v NO es una variable del lenguaje. Es en realidad una metavariable que corre sobre las variables del lenguaje. Si v fuera una variable del lenguaje, deberíamos hacer otra ecuación para cada una de las otras variables del lenguaje. En vez de eso, se hace una sola ecuación donde v representa a una variable cualquiera. Esto la torna una metavariable. Usaremos u, v, w para metavariables y $\mathsf{x}, \mathsf{y}, \mathsf{z}$ para variables del lenguaje (note que usamos distintos tipos de letra para distinguirlas). En los ejemplos de arriba, x es "la"variable x, o sea una frase de $\langle var \rangle$.

Por ejemplo, en la frase $\forall \mathsf{x}.\ \forall \mathsf{y}.\ \mathsf{y} > \mathsf{x}$, los símbolos x e y son "las"variables x e y. Por lo tanto, sabemos que son variables diferentes. En cambio, en $\forall v.\ \forall w.\ w > v$, tratándose de metavariables no está excluida la posibilidad de que v y w sean iguales, (por ejemplo ambas iguales a x).

4.3. Ligadura. El lenguaje de la lógica, como muchos otros lenguajes expresivos, incorporan la posibilidad de que una variable juegue dos roles claramente diferenciados: la variable puede referirse a un objeto no específico, genérico dentro de un universo, o puede remitir a un objeto particular. Para permitir este fenómeno en la lógica de predicados se diferencia entre *variables ligadas* (o acotadas) y *variables libres*.

Además de composicionalidad, hay otras propiedades deseables que la semántica debería satisfacer. Por ejemplo, el significado de una frase no debería depender del nombre de las variables ligadas, es decir, $\forall \mathsf{x}.\ \mathsf{x}+0=\mathsf{x}$ debería tener el mismo significado que $\forall \mathsf{y}.\ \mathsf{y}+0=\mathsf{y}$.

Para formular dichas propiedades es necesario introducir las siguientes definiciones:

- **Ocurrencia ligadora**: una ocurrencia ligadora de una variable es la que se encuentra inmediatamente después de un cuantificador ($\forall$ o $\exists$).
- **Alcance de una ocurrencia ligadora**: En $\forall v.\ p$ o $\exists v.\ p$, el predicado p es el alcance de la ocurrencia ligadora de v.
- **Ocurrencia ligada**: cualquier ocurrencia de v en el alcance de una ocurrencia ligadora de v es una ocurrencia ligada de v. Dicha ocurrencia ligada puede estar en más

de un alcance de ocurrencias ligadoras de v. Se dice que está ligada por la ocurrencia ligadora de menor alcance.
- **Ocurrencia libre**: una ocurrencia de una variable que no es ligadora ni ligada es una ocurrencia libre.
- **Variable libre**: una variable que tiene ocurrencias libres es una variable libre.
- Una **expresión cerrada** es una que no tiene variables libres.

Ejemplo: En

$$\begin{array}{cccccccccccccccccc} \forall \mathsf{x}. & (& \mathsf{x} & = & \mathsf{y} & \wedge & \forall & \mathsf{y}. & (& \mathsf{x} & = & \mathsf{y} & \wedge & \exists & \mathsf{x}. & \mathsf{x} & + & \mathsf{y} & = & \mathsf{x})) \\ 0 & & 1 & & 2 & & & 3 & & 4 & & 5 & & & 6 & 7 & & 8 & & 9 & \end{array}$$

- ocurrencias ligadoras: 0, 3 y 6
- ocurrencias ligadas: 1 y 4 (ligadas por 0), 5 y 8 (ligadas por 3) y 7 y 9 (ligadas por 6). Observar que 7 y 9 también están en el alcance de 0, pero el alcance de 6 es menor.
- ocurrencias libres: 2.
- variables libres: sólo y es libre.

Observar que x e y son variables libres de $\mathsf{x} + \mathsf{y} = \mathsf{x}$, pero sólo y es variable libre de $\cdot\forall \mathsf{x}.\mathsf{x} + \mathsf{y} = \mathsf{x}$. Si miramos ahora la expresión más grande

$$\forall \mathsf{y}.\ (\mathsf{x} = \mathsf{y} \wedge \exists \mathsf{x}.\ \mathsf{x} + \mathsf{y} = \mathsf{x}),$$

la variable y dejó de ser libre, ahora la variable x es libre.

4.4. Variables libres. Se puede definir el conjunto de variables libres por ecuaciones dirigidas por sintaxis. Para expresiones enteras:

$$\begin{array}{lcl} FV\ \lfloor n \rfloor & = & \emptyset \\ FV\ v & = & \{v\} \\ FV(-e) & = & FV\ e \\ FV(e + e') & = & FV\ e \cup FV\ e' \\ & \vdots & \end{array}$$

y para expresiones booleanas:

$$\begin{array}{lcl} FV\ \mathbf{true} & = & \emptyset \\ FV\ \mathbf{false} & = & \emptyset \\ FV(e = e') & = & FV\ e \cup FV\ e' \\ & \vdots & \\ FV(\neg p) & = & FV\ p \\ FV(p \wedge q) & = & FV\ p \cup FV\ q \\ & \vdots & \\ FV(\forall v.\ p) & = & (FV\ p) - \{v\} \\ FV(\exists v.\ p) & = & (FV\ p) - \{v\} \end{array}$$

Las ecuaciones son dirigidas por sintaxis, esto garantiza que define unívocamente FV.

Preguntas.

1. ¿Cuál de las siguientes frases representa la misma frase abstracta que
$$\forall \mathsf{x}.\ \forall \mathsf{z}.\ \mathsf{x} < t \wedge \forall t.\ t \leq \mathsf{z} \Rightarrow (\exists \mathsf{y}.\ \mathsf{x} \leq \mathsf{y}) \wedge \mathsf{y} < \mathsf{z}?$$
   (1) $(\forall \mathsf{x}.\ (\forall \mathsf{z}.\ \mathsf{x} < t \wedge (\forall t.\ t \leq \mathsf{z})) \Rightarrow (\exists \mathsf{y}.\ \mathsf{x} \leq \mathsf{y}) \wedge \mathsf{y} < \mathsf{z})$
   (2) $(\forall \mathsf{x}.\ (\forall \mathsf{z}.\ \mathsf{x} < t \wedge (\forall t.\ t \leq \mathsf{z}) \Rightarrow (\exists \mathsf{y}.\ \mathsf{x} \leq \mathsf{y}) \wedge \mathsf{y} < \mathsf{z}))$
2. ¿Que diferencia se establece en el texto en relación al uso de los símbolos x, y, respecto de v, w?
3. ¿Qué diferencia hay entre las nociones de *variable libre* y *ocurrencia libre*?

5. Sustitución y el problema de la captura

Las expresiones con variables libres pueden instanciarse sustituyendo sus variables libres por términos. A continuación definimos la operación de sustitución. Comenzamos definiendo sustitución como una función de variables en expresiones. Sea

$$\Delta = \langle var \rangle \to \langle intexp \rangle$$

el conjunto de todas las sustituciones (notar que el lenguaje de la lógica de predicados sólo tiene variables enteras, no hay variables lógicas).

Definimos ahora el operador sustitución, que opera sobre expresiones enteras (términos) y expresiones booleanas (predicados):

Para expresiones enteras, $\_/\_ \in \langle intexp \rangle \times \Delta \to \langle intexp \rangle$ se define mediante las siguientes igualdades:

$$\begin{array}{rcl} 0/\delta & = & 0 \\ 1/\delta & = & 1 \\ \vdots & & \\ v/\delta & = & \delta v \\ (-e)/\delta & = & -(e/\delta) \\ (e+f)/\delta & = & (e/\delta) + (f/\delta)... \end{array}$$

Intuitivamente podemos pensar que una sustitución se propaga por toda la estructura de la expresión entera salvo cuando se encuentra con una variable, en cuyo caso reemplaza según indica la sustitución. Es que las expresiones enteras no tienen cuantificadores, por ello todas las variables de una expresión entera e son libres (aunque pueden no ser variables libres de una expresión booleana mayor que contiene a e como subexpresión) y por ello aplicar una sustitución es sencillo.

Por ejemplo, si $\delta\mathsf{x} = e_0,\ \delta\mathsf{y} = e_1,\ \delta\mathsf{z} = e_2$, entonces

$$\begin{array}{rcl} (\mathsf{x} + \mathsf{y} * \mathsf{z} - 5)/\delta & = & (\mathsf{x}/\delta) + ((\mathsf{y} * \mathsf{z} - 5)/\delta) \\ & = & \delta x + ((\mathsf{y} * \mathsf{z})/\delta) - (5/\delta) \\ & = & e_0 + (\mathsf{y}/\delta) * (\mathsf{z}/\delta) - 5 \\ & = & e_0 + \delta\mathsf{y} * \delta\mathsf{z} - 5 \\ & = & e_0 + e_1 * e_2 - 5 \end{array}$$

Observar que en las ecuaciones que definen la aplicación de una sustitución sólo hay sintaxis. En la ecuación $(-e)/\delta = -(e/\delta)$ el signo - a ambos lados de la ecuación denota lo mismo: el operador - unario. Ninguno de esos operadores es la función opuesto del metalenguaje. De todas formas sigue habiendo metavariables (v, d, e, f).

Para expresiones booleanas construidas desde los operadores usuales la definición es totalmente análoga que las expresiones enteras y valen las mismas reflexiones que hiciéramos ahí.

Cuando consideramos cuantificadores aparece una dificultad. Por ejemplo, supongamos que queremos aplicar una sustitución δ a la frase $\exists \mathsf{x}.\ \mathsf{x} > \mathsf{y}$, que es intuitivamente válida ya que no importa el valor de y, siempre existe un entero mayor x. Una aplicación ingenua de la sustitución, me daría $\exists \mathsf{x}.\ \mathsf{x} > e$ donde $e = \delta\mathsf{y}$. En efecto, la variable x no se toca porque está ligada. Pero ¿qué pasa si e es x?, quedaría $\exists \mathsf{x}.\ \mathsf{x} > \mathsf{x}$ que es falsa.

Otro problema que puede observarse, es que como $\exists \mathsf{x}.\ \mathsf{x} > \mathsf{y}$ por un lado, y $\exists \mathsf{z}.\ \mathsf{z} > \mathsf{y}$ por otro sólo se diferencian en el nombre de la variable ligada, sustituir en uno u otro me debería dar resultados equivalentes. Pero, nuevamente para el caso en que $\delta\mathsf{y} = \mathsf{x} + 1$, en un caso obtenemos $\exists \mathsf{x}.\ \mathsf{x} > \mathsf{x}+1$, y en el otro $\exists \mathsf{z}.\ \mathsf{z} > \mathsf{x}+1$. No son equivalentes (la primera es falsa y la segunda es válida).

El problema es que al sustituir δ en $\exists \mathsf{x}.\ \mathsf{x} > \mathsf{y}$, se está *capturando* la x que era libre en $\mathsf{x} + 1$, ahora pasa a ser ligada. Eso no debería ocurrir.

Una solución es renombrar la variable ligada y luego sustituir. Otra posibilidad es hacer las dos cosas simultáneamente. Si bien es más complicado, es lógico hacerlo así ya que estamos definiendo justamente la sustitución, no deberíamos asumir que sabemos renombrar, ya que renombrar es sustitución de un nombre por otro (en realidad esto es discutible, ya que el renombre es una forma un poco más sencilla de sustitución). Definimos entonces, para $Q = \forall,\ \exists$:

$$(Qv.b)/\delta = Qv_{new}.(b/[\delta|v : v_{new}])$$

donde v_{new} es una variable *nueva* y $[\delta|v : v_{new}]$ ya se definió: sustituye parecido a δ, salvo que a la variable v la reemplaza por v_{new}. Si no elegimos cuidadosamente a v_{new}, el problema de la captura persiste. Volviendo al ejemplo anterior, si en $\exists\ \mathsf{x}.\mathsf{x} > \mathsf{y}$ aplicamos δ, donde $\delta\mathsf{y} = \mathsf{x} + \mathsf{z}$, si sustituimos sin renombrar queda $\exists\ \mathsf{x}.\mathsf{x} > \mathsf{x} + \mathsf{z}$ que captura la x. Si renombramos x por z, nos queda $\exists\ \mathsf{z}.\ \mathsf{z} > \mathsf{x} + \mathsf{z}$ que captura z. Para evitar la captura hay que elegir bien la variable *nueva* v_{new}: la variable no debe ser capturable. Las variables *capturables* son las que pueden aparecer al aplicar la sustitución. O sea:

$$v_{new} \notin \cup_{w \in FV(Qv.p)} FV(\delta w)$$

que es lo mismo que

$$v_{new} \notin \cup_{w \in FVp - \{v\}} FV(\delta w)$$

Para finalizar, entonces, agregamos las ecuaciones

$$\begin{array}{rcl} (\forall v\ .\ b)/\delta & = & \forall v_{new}\ .\ (b/[\delta|v : v_{new}]) \text{ donde } v_{new} \notin \bigcup_{w \in FV(b)-\{v\}} FV(\delta\ w) \\ (\exists v\ .\ b)/\delta & = & \exists v_{new}\ .\ (b/[\delta|v : v_{new}]) \text{ donde } v_{new} \notin \bigcup_{w \in FV(b)-\{v\}} FV(\delta\ w) \end{array}$$

Esto resuelve el problema, pero no está completamente definido ya que puede haber más de una variable v_{new} que satisfaga la condición (de hecho hay siempre una cantidad infinita de variables que la satisfacen). Debe determinarse un criterio para elegir la variable v_{new}: si v mismo satisface la condición, entonces $v_{new} = v$. En caso contrario, se elige v_{new} como la primera variable que la satisface en un orden preestablecido entre las variables.

Preguntas.

1. ¿Qué significa *captura*? ¿Por qué la captura debe ser evitada?
2. Sea δ una sustitución, p un predicado, y $w \in FVp$.
   ¿Puede existir alguna variable libre en δw que no sea libre en p/δ?
3. Caracterice las variables libres de p/δ en función de las variables libres de p y δw (para $w \in FVp$).

5.1. Propiedades de la semántica. Hay dos propiedades importantes que se relacionan con el uso de las variables en los lenguajes que admiten alguna forma de ligadura, y que resultan fundamentales en los lenguajes de programación. La primera, llamada *Coincidencia*, expresa que el significado de una frase no puede depender de variables que no ocurran libres en la misma. La segunda, llamada *Renombre*, asegura que el significado no depende de las variables ligadas de una frase.

Teorema de Coincidencia (TC). Si dos estados σ y σ' coinciden en las variables libres de p, entonces da lo mismo evaluar p en σ o σ'. En símbolos:

$$(\forall w \in FV(p) \;.\; \sigma\ w = \sigma'\ w) \Longrightarrow [\![p]\!]\sigma = [\![p]\!]\sigma'.$$

En particular, TC implica que el valor de un término cerrado es el mismo en cualquier estado, es decir, no depende del estado.

Teorema de Renombre (TR). Los nombres de la variables ligadas no tienen importancia. En símbolos,

$$u \notin FV(q) - \{v\} \Rightarrow [\![\forall u \;.\; q/v \to u]\!] = [\![\forall v \;.\; q]\!]$$

Para la prueba de las mismas, debemos recurrir a propiedades más generales.

Teorema de Sustitución (TS). Si aplico la sustitución δ a p y luego evalúo en el estado σ, puedo obtener el mismo resultado a partir de p sin sustituir si evalúo en un estado que hace el trabajo de δ y de σ (en las variables libres de p). En símbolos:

$$(\forall w \in FV(p) \;.\; [\![\delta\ w]\!]\sigma = \sigma'\ w) \Longrightarrow [\![p/\delta]\!]\sigma = [\![p]\!]\sigma'.$$

Abreviatura. Si consideramos id como la sustitución identidad, que mapea cada variable v en la expresión v, entonces escribiremos $v_0 \to e_0, \ldots, v_{n-1} \to e_{n-1}$ para abreviar $[id|v_0 : e_0| \ldots |v_{n-1} : e_{n-1}]$.

Teorema de Sustitución Finita (TSF, corolario de TS).

$$[\![p/v_0 \to e_0, \ldots, v_{n-1} \to e_{n-1}]\!]\sigma = [\![p]\!]\ [\sigma|v_0 : [\![e_0]\!]\sigma| \ldots |v_{n-1} : [\![e_{n-1}]\!]\sigma]$$

Demostraciones. Primero demostraremos que TSF y TR son corolarios de TS y TC:

Demostración de TSF. Sean la sustitución $\delta = v_0 \to e_0, ..., v_{n-1} \to e_n$ y el estado $\sigma' = [\sigma|v_0 : [\![e_0]\!]\sigma| \dots |v_{n-1} : [\![e_{n-1}]\!]\sigma]$.

TS dice que para demostrar TSF alcanza con comprobar que $\forall w \in FV(p)\ .\ [\![\delta\ w]\!]\sigma = \sigma'\ w$. Demostremos eso entonces. Sea $w \in FV(p)$. Si $w \in \{v_0, \dots, v_{n-1}\}$ (ojo! puede haber repeticiones), sea i el máximo tal que $w = v_i$. Tenemos $[\![\delta\ w]\!]\sigma = [\![\delta\ v_i]\!]\sigma = [\![e_i]\!]\sigma = \sigma'\ v_i = \sigma'\ w$. Si, por el contrario, $w \notin \{v_0, \dots, v_{n-1}\}$, también obtenemos $[\![\delta\ w]\!]\sigma = [\![id\ w]\!]\sigma = [\![w]\!]\sigma = \sigma\ w = \sigma'\ w$.

Demostración de TR. Sea $u \notin FV(q) - \{v\}$. Veamos que $[\![\forall u\ .\ (q/v \to u)]\!] = [\![\forall u\ .\ q]\!]$. Para ello, sea $\sigma \in \Sigma$:

$$
\begin{array}{lcll}
[\![\forall u\ .\ (q/v \to u)]\!]\sigma & = & \forall n \in \mathbf{Z}.\ [\![q/v \to u]\!][\sigma|u : n] & \text{definición de } [\![\ ]\!] \\
& = & \forall n \in \mathbf{Z}.\ [\![q]\!][\sigma|u : n|v : [\![u]\!][\sigma|u : n]] & \text{TSF} \\
& = & \forall n \in \mathbf{Z}.\ [\![q]\!][\sigma|u : n|v : n] & \text{definición de } [\![\ ]\!] \\
& = & \forall n \in \mathbf{Z}.\ [\![q]\!][\sigma|v : n] & \text{TC } \wedge (u \notin FV(q) - \{v\}) \\
& = & [\![\forall v\ .\ q]\!]\sigma & \text{definición de } [\![\ ]\!]
\end{array}
$$

Demostración de TS. Sea

$$\Phi(p, \delta, \sigma, \sigma') = (\forall w \in FV(p).\ [\![\delta\ w]\!]\sigma = \sigma'\ w) \Rightarrow [\![p/\delta]\!]\sigma = [\![p]\!]\sigma'$$

demostraremos que para todo $p, \delta, \sigma, \sigma'$, $\Phi(p, \delta, \sigma, \sigma')$ se cumple, por inducción en p.

Si $p = 0$, se cumple ya que

$$
\begin{array}{lcll}
[\![p/\delta]\!]\sigma & = & [\![0/\delta]\!]\sigma & p = 0 \\
& = & [\![0]\!]\sigma & \text{definición de substituir} \\
& = & 0 & \text{definición de } [\![\ ]\!] \\
& = & [\![0]\!]\sigma' & \text{definición de } [\![\ ]\!] \\
& = & [\![p]\!]\sigma' & p = 0
\end{array}
$$

Lo mismo para las demás constantes numéricas y booleanas.

Si $p = v$, asumimos $\forall w \in FV(p).\ [\![\delta\ w]\!]\sigma = \sigma'\ w$, que equivale a $[\![\delta\ v]\!]\sigma = \sigma'\ v$. Luego, tenemos

$$
\begin{array}{lcll}
[\![p/\delta]\!]\sigma & = & [\![v/\delta]\!]\sigma & p = v \\
& = & [\![\delta\ v]\!]\sigma & \text{definición de substituir} \\
& = & \sigma'\ v & \text{hipótesis} \\
& = & [\![v]\!]\sigma' & \text{definición de } [\![\ ]\!] \\
& = & [\![p]\!]\sigma' & p = v
\end{array}
$$

Si $p = e_0 + e_1$, asumimos $\forall w \in FV(p).\ [\![\delta\ w]\!]\sigma = \sigma'\ w$, que implica para $i \in \{0, 1\}$, que $\forall w \in FV(e_i).\ [\![\delta\ w]\!]\sigma = \sigma'\ w$ y por hipótesis inductiva $\Phi(e_i, \delta, \sigma, \sigma')$ y $[\![e_i/\delta]\!]\sigma = [\![e_i]\!]\sigma'$.

Entonces,

$$\begin{array}{rcll} [\![p/\delta]\!]\sigma & = & [\![(e_0+e_1)/\delta]\!]\sigma & p = e_0+e_1 \\ & = & [\![(e_0/\delta)+(e_1/\delta)]\!]\sigma & \text{definición de substituir} \\ & = & [\![e_0/\delta]\!]\sigma + [\![e_1/\delta]\!]\sigma & \text{definición de } [\![\ ]\!] \\ & = & [\![e_0]\!]\sigma' + [\![e_1]\!]\sigma' & \text{hipótesis inductiva} \\ & = & [\![e_0+e_1]\!]\sigma' & \text{definición de } [\![\ ]\!] \\ & = & [\![p]\!]\sigma' & p = e_0+e_1 \end{array}$$

Lo mismo para los demás operadores unarios y binarios.

Si $p = \forall v\ .\ b$, asumimos $\forall w \in FV(p).\ [\![\delta\ w]\!]\sigma = \sigma'\ w$, que equivale a asumir que $\forall w \in FV(b) - \{v\}.\ [\![\delta\ w]\!]\sigma = \sigma'\ w$. Sean $u \notin \bigcup_{w \in FV(b)-\{v\}} FV(\delta\ w)$ y $n \in \mathbf{Z}$ arbitrarios. Sean $\delta_0 = [\delta | v : u]$, $\sigma_0 = [\sigma | u : n]$ y $\sigma'_0 = [\sigma' | v : n]$. Se puede demostrar que se cumple $\forall w \in FV(b).\ [\![\delta_0\ w]\!]\sigma_0 = \sigma'_0\ w$. En efecto, si $w = v$, entonces

$$\begin{array}{rcll} [\![\delta_0\ w]\!]\sigma_0 & = & [\![\delta_0\ v]\!]\sigma_0 & w = v \\ & = & [\![u]\!]\sigma_0 & \text{definición de } \delta_0 \\ & = & \sigma_0\ u & \text{definición de } [\![\ ]\!] \\ & = & n & \text{definición de } \sigma_0 \\ & = & \sigma'_0\ v & \text{definición de } \sigma'_0 \\ & = & \sigma'_0\ w & w = v \end{array}$$

En caso contrario, $w \neq v$ entonces

$$\begin{array}{rcll} [\![\delta_0\ w]\!]\sigma_0 & = & [\![\delta\ w]\!]\sigma_0 & \text{definición de } \delta_0 \text{ y } w \neq v \\ & = & [\![\delta\ w]\!]\sigma & \text{TC, definición de } \sigma_0 \text{ y } u \notin FV(\delta\ w) \\ & = & \sigma'\ w & \text{hipótesis y } w \neq v \\ & = & \sigma'_0\ w & \text{definición de } \sigma'_0 \text{ y } w \neq v \end{array}$$

Esto prueba $\forall w \in FV(b).\ [\![\delta_0\ w]\!]\sigma_0 = \sigma'_0\ w$. Por hipótesis inductiva, $[\![b/\delta_0]\!]\sigma_0 = [\![b]\!]\sigma'_0$. Luego,

$$\begin{array}{rcll} [\![p/\delta]\!]\sigma & = & [\![(\forall v\ .\ b)/\delta]\!]\sigma & p = \forall v\ .\ b \\ & = & [\![\forall u\ .\ (b/\delta_0)]\!]\sigma & \text{definición de sustituir} \\ & = & \forall n \in \mathbf{Z}.\ [\![b/\delta_0]\!]\sigma_0 & \text{definición de } [\![\ ]\!] \\ & = & \forall n \in \mathbf{Z}.\ [\![b]\!]\sigma'_0 & \text{hipótesis inductiva} \\ & = & [\![\forall v\ .\ b]\!]\sigma' & \text{definición de } [\![\ ]\!] \\ & = & [\![p]\!]\sigma' & p = \forall v\ .\ b \end{array}$$

que es lo queríamos comprobar.

Demostración de TC. TC se utiliza en TS, por lo tanto debe demostrarse sin utilizar TS, ni TSF ni TR. Se propone $\Phi(p, \sigma, \sigma') = (\forall w \in FV(p).\ \sigma\ w = \sigma'\ w) \Rightarrow [\![p]\!]\sigma = [\![p]\!]\sigma'$ y debe demostrarse que para todo p, σ, σ', $\Phi(p, \sigma, \sigma')$ vale por inducción en p. La prueba, más sencilla que la de TS, queda como ejercicio.